Managing Logical Control Systems through Time

Mats Ronnling

September 25, 2009
Master's Thesis in Computing Science, 30 ECTS credits
Supervisor at CS-UmU: Michael Minock
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

This thesis investigates the feasibility of modeling immediate and future operations in the real world as updates to a database state. Such a capability will in the future enable us to leverage general database interface technology (e.g. Natural Language Interfaces (NLIs) to databases) to control and query devices and switches in the real world. In this approach, operations are modeled as inserts into a bi-temporal relation with a middle-ware process executing operations over a controller API. The approach has been implemented using POSTGRES triggers and a simple JAVA-based middle-ware component. We have experimented with various policies for how to resolve conflicts between orders and have evaluated the system using simulated data over a light controller scenario for the home/factory of the future. Our empirical study, based on large volumes of simulated data, point toward practical applicability of the approach in large systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Typically the information in a database models some aspect of reality and enables other systems or humans to answer questions over this modeled reality for the purpose of analysis and decision making (see figure 1). While there may be many complex efforts and routines to populate such a database, once built, operations on the database do not typically have direct physical effects on the real world. Updates to the database are simply a manner to keep the modeled world of the database consistent with the real world[1]. In this thesis, we consider inverting this traditional view and consider that inserts into the database might actually post operations that are to be performed in the physical world via a logical control or switch system.

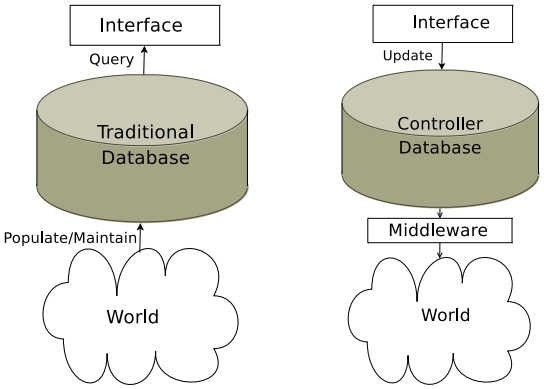

Figure 1.1: Classical databases versus controller databases.

As an example of this, consider a system controlling the lights in a house, factory or city. The physical controller is just a switch offering the API which sets a given light into the on or off state. Consider the two API functions **TurnOnLight(id)** and **TurnOffLight(id)** where calls to these functions have the obvious effects. What makes this simple example interesting is the potentially complex pattern of on/off operations through time. Consider a scenario in which the actions in table 1.1 are performed over the light system.

[1]We shall provide a more nuanced discussion of this in the discussion chapter of this work. For example there are cases in which the copy in the database essentially represents the 'thing' in the real world as in the case, for instance of money.

Time	User	Status or Order
00.00		Light 1 and 2 are off.
05.00	John:	"turn light 1 on from 09.00 to 17.00".
07.00	Kate:	"turn light 2 on from 10.00 to 18.00".
09.00	API CALL:	Light 1 turns on.
09.01	Ron:	"turn light 1 off".
09.01	API CALL:	Light 1 turns off.
10.00	API CALL:	Light 2 turns on.
11.00	John:	"turn light 1 on".
11.00	API CALL:	Light 1 turns on.
13.00	Dave:	"when was light 2 ordered on?" System: "07.00 (by Kate)"
13.01	Dave:	"when was light 2 turned on?" System: "10.00"
18.00	API CALL:	Light 2 turns off.
19.00		Light 1 is on and light 2 is off.

Table 1.1: Lights scenario

To support such a scenario, we need to keep track of the orders made over the light controller as well as the actual states of the lights through time. Additionally, we need to have some type of middle-ware process to make the appropriate API calls at the proper times. Obviously the requests in such a system must be proactive or simultaneous and not retroactive; one can not turn a light on in the past.

The careful reading of the example in table 1.1 indicates that a policy is in effect with respect to how more recent orders interact with prior orders. For example, John's immediate update at 11am overrode his earlier proactive order to turn light off at 17.00. Thus the light remained on past 17.00 in the scenario. This was only true because of the particular policy in place, what we call the splitting case. This thesis proposes and evaluates several alternative policies that resolve conflicts among user orders. In what we call the *clobber case*, user's requests are always performed and override any previous request in the system that is in time conflict. In contrast we have the *blocking case* where any order that is in conflict with a previous order is blocked[2]. Finally we explore a *splitting* policy that accepts orders while accommodating, to the highest possible degree, to earlier orders.

1.1 Organization of this thesis

This thesis is organized as follows: chapter 2 defines the setting and the basic protocols of our approach through fully exploring the lights scenario developed in table 1.1. Chapter 3 describes our implemented system, presenting a general architectural description along with a detailed descriptions of each component. Chapter 4 presents a performance evaluation of the working system. Chapter 5 discusses the work in relation to prior work and points toward future extensions and applications. Chapter 6 summarizes results and concludes.

[2]We did not evaluate hybrid approaches that combine these two policies based on user priories

Chapter 2

Approach

We model the basic operations in the world as the setting of an *attribute* (`attr`) to a *value* (`val`) by a user (`usr`). Such an idea of an operation is quite general and is likely to be adequate for many applications. Of course what is of interest here is *when* such operations take place.

2.1 Bi-temporal databases

At its core, our approach is based on bi-temporal databases [5, 8]. To review we give a brief introduction here. An *event* is an association between a fact (entity, relationship, property, etc.) and a given time value. *Point events* are associated with a single time points (or chronos). *Interval events* are associated with all the time points (or chronos) that lie between the beginning of the interval and an end time point inclusive of the start point, but not the end time[1] There are two common interpretations of the time value associated with a recorded fact: *valid time* or *transaction time*. Valid time represents the time the fact actually holds *in the real world (or modeled reality)*. Transaction time represents the time that the fact holds *in the database*. Some databases use the first notion (*valid time database*), some use the second (*transaction time database*) and some use both time dimensions (*bi-temporal databases*).

Basic tuples of attribute (`attr`), value (`val`), and user (`usr`) are extended with a *valid start time* (`VST`), a *valid end time* (`VET`) as well as *transaction start time* (`TST`) and a *transaction end time* (`TET`). The valid times indicate the time interval over which the attribute (`attr`) should hold the value (`val`) *in the real world*. The transaction times indicate the time interval over which *the database holds the order as true*. In the case that such time intervals have no end time specified, the special symbols `now` and `uc` (until changed) represent the valid end time or transaction end time respectively. An order that has `TET` set to `uc` is called *current*. We say an order is *retracted* when its transaction end time is set to the current time `current`.

2.2 Allen's interval algebra

The presentation of the protocols below conform to the relations of Allen´s interval algebra[4]. In short, this algebra specifies all the relations that can occur between two

[1] By adopting this convention, we may have intervals meet exactly, but not overlap.

intervals. Specifically we use the following notation: The interval x starts at x^- and ends at x^+. $(x^- < x^+)$. Likewise the interval y starts at y^- and ends at y^+. $(y^- < y^+)$

Relation	Symbol	Example	Conditions
x before y y after x	$<$ $>$	`XXX` `YYY`	$x^+ < y^-$
x meets y y met-by x	m $m^\smile$	`XXXX` `YYYY`	$x^+ = y^-$
x overlaps y y overlapped-by x	o $o^\smile$	`XXXX` `YYYY`	$x^- < y^- < x^+$, $x^+ < y^+$
x during y y includes x	d $d^\smile$	`XX` `YYYY`	$x^- > y^-$, $x^+ < y^+$
x starts y y started-by x	s $s^\smile$	`XXXX` `YYYYYYYY`	$x^- = y^-$, $x^+ < y^+$
x finishes y y finished-by x	f $f^\smile$	`XXXX` `YYYYYYYY`	$x^+ = y^+$ $x^- > y^-$
x equals y	$\equiv$	`XXXX` `YYYY`	$x^- = y^-$, $x^+ = y^+$

2.3 Modeling operations

There is one basic factor that must be addressed. It has to do with the fact that at creation time there is some default, background value that an attribute assumes. For example we may assume that lights initially have the value 'off'. This default value, created by the user 'INIT', must always be able to be overridden, no matter what the conflict policy is. This table 2.1 shows the state of our database at the beginning of the lights scenario.

orders(attr	val	usr	VST	VET	TST	TET)
l1	off	INIT	0.00	now	0.00	uc
l2	off	INIT	0.00	now	0.00	uc

Table 2.1: Database state as of 0.00

Now each user operation can be modeled as an insert operation over the attributes `attr`, `val`, `VST` and `VET`. The transaction start time of such an insert is always the current time (`current`) and the end time is always `uc`.
For example at exactly 5.00 the user with account 'John' issued the following request:

```
INSERT INTO orders(ATTR,VAL,VST,VET)
VALUES ('l1','on',9:00,17:00);
```

Under all of our policies, this results in the state of the database as depicted in table 2.2.

orders(attr	val	usr	VST	VET	TST	TET)
l1	off	INIT	0.00	now	0.00	**5.00**
l2	off	INIT	0.00	now	0.00	uc
l1	off	INIT	0.00	9.00	0.00	uc
l1	on	John	9.00	17.00	5.00	uc
l1	off	INIT	17.00	now	0.00	uc

Table 2.2: Database state as of 5.00

A middle-ware process that polls the database then, at 9.00, issues the proper call to the light controller to turn on light 1. At 9.01, Ron turns off the light off with the insert:

```
INSERT INTO orders(ATTR,VAL,VST,VET)
VALUES ('l1','off',9:01,now);
```

Which was an immediate update with an open ended valid end time. This operation makes the remainder of John's earlier order invalid as well as the residual order left over by the INIT process. This is because the system is running in the splitting case presented below. Had the system been running in the blocking case, this order would have been disallowed.

orders(attr	val	usr	VST	VET	TST	TET)
l1	off	INIT	0.00	now	0.00	**5.00**
l2	off	INIT	0.00	now	0.00	**7.00**
l1	on	John	9.00	17.00	5.00	**9.01**
l1	off	INIT	0.00	9.00	0.00	uc
l1	off	INIT	17.00	now	0.00	**9.01**
l2	on	Kate	10.00	18.00	7.00	uc
l2	off	INIT	0.00	10.00	0.00	uc
l2	off	INIT	18.00	now	0.00	uc
l1	off	John	9.01	now	9.01	**11.00**
l1	on	John	9.00	9.01	5.00	uc
l1	on	John	11.00	now	11.00	uc

Table 2.3: Database state as of 19.00

2.4 Conflict policies

As mentioned earlier, we propose three basic policies to handle conflicts between a new order and the old orders that are current in the database. When a new order is inserted, a process is triggered whereby the set of current orders are retrieved that are in conflict with the new order. Conflict occurs between orders that are setting the same attribute (`attr`) in the real world where both the valid time interval and transaction time intervals of the orders overlap. Conflicts are handled differently depending on which protocol is set. However, if the old order has the user `INIT`, the splitting case is always applied to it.

Input: *new* a new order

```
begin
  for old ∈ Conflict(new) do
      if (splitting ∨ old.user = 'INIT')
        then
            if (old.vst < new.vst)
              then
                  add := Copy(old);
                  add.vet = new.vst;
                  Add(add);
            fi
            if (new.vet < old.vet)
              then
                  add := Copy(old);
                  add.vst = new.vet;
                  Add(add);
            fi
            old.tet = current;
      fi
  else if (blocked)
        then
            abort;
      fi
  else if (clobber)
        then
            if (old.vst < new.vst)
              then
                  add := Copy(old);
                  add.vet = new.vst;
                  add.val = initialValue;
                  Add(add);
            fi
            if (new.vet < old.vet)
              then
                  add := Copy(old);
                  add.vst = new.vet;
                  add.val = initialValue;
                  Add(add);
            fi
            old.tet = current;
      fi
  od
end
```

Table 2.4: Pseudo-code for insert trigger.

2.5 Splitting protocol

The splitting protocol overwrites any previous order on the same attribute that is in time conflict with the new order. Should there be any interval of the old order that is not overwritten, the splitting protocol redefines the old order to fit into that interval with its original value and transaction times, but with its interval, the valid times, shortened and adjusted to the new order. This strategy is adopted no matter what the current protocol is, if the old order was given by 'INIT'.

2.6 Clobber protocol

The clobbering protocol, just like the splitting protocol, overwrites any previous order on the same attribute that is in time conflict with the new order. But instead of adjusting the valid times of possible remains of old orders in time conflict with the new order, the clobbering deletes the complete interval of the old order. However, there are two exceptions; 1) should the old order have been given by 'INIT' the splitting policy is applied, and 2) if the old order has already been performed and the current time is in the middle of old's interval, the new order is aborted since one can't make changes in the past. New orders can only clobber old orders whose valid start times are in the future.

2.7 Blocking protocol

The blocking protocol blocks any new order insert that is in time conflict with an old order on the same attribute given by any user except 'INIT'. Orders given by 'INIT' are dealt with using the splitting policy.

Chapter 3

System Implementation

The architecture of our implemented system is depicted in figure 3.1. The interface inserts tuples into the bi-temporal table `ORDERS`. A trigger defined over the `ORDERS` table determines the necessary operations in cases that the currently issued order conflicts with earlier orders. These triggers implement either the clobber, blocked or splitting cases described above. For the sake of performance, orders that are retracted by newer orders are added to the history table (`HISTORY`). The `ORDERS` table represents the currently relevant orders and is scanned at a fixed time granularity by a middle-ware component called the *Scanner*. The Scanner executes the relevant orders via the controller API as they come due and, likewise moves completed orders from the active `ORDERS` table to the `HISTORY` table. In such a way, we maintain only the minimal number of order tuples within the `ORDERS` table.

We will now describe the components in the architecture in more detail.

3.1 The interface

The interface can be any ODBC/JDBC based system. The system has been tested with a standard JAVA[1] application capable of performing updates. As mentioned earlier this interface merely post inserts into the `ORDERS` table to perform immediate or future operations on the real world. It should be noted that the interface also has read access to the history and order tables as well as any other traditional data that resides within the database.

3.2 The database

The controller portion of the database just consists of two relations: `ORDERS`, and `HIST`. The attributes in these relations are depicted in figure 3.1. The type of the `attr` and `val` attributes are string (`VARCHAR(100)`), the type of `VST` (valid start time), `VET`(valid end time), `TST` (transaction start time), `TET` (transaction end time) are `TIMESTAMP`, `id` is of type `SEQUENCE`

The bi-temporal PostgreSQL[2] database has triggers written in PL/pgSQL[7, 3] to implement each of the three protocols defined above. The mechanism uses commonly available row level triggers, meaning that the trigger fires for every tuple that is inserted.

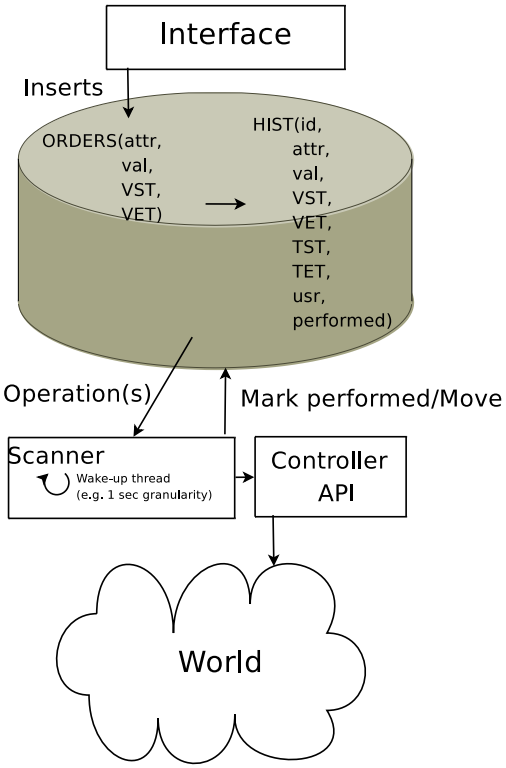

Figure 3.1: View of the architecture

Each time an update is made to the database, the triggers loop through the conflicting past orders and, depending on the protocol, split or move the conflicting orders to the history table. Valid transactions with a `VET` older than the current time are also moved to the history database. That minimizes the number of rows within the active database and makes the update trigger-looping quicker which enhances the performance of the system.

3.3 The scanner

The scanner is written in Java and takes reflects the orders in the database onto the controller API. Once the connection to the database is established, it periodically scans the database looking for current orders that have not been performed and who have a VST that is less or equal to the current time. In such a case it performs the corresponding order over the controller API.

Chapter 4

System Evaluation

Our system evaluation aims at determining the maximum number of switches the system can model, given that lag times are held within a guaranteed bound (e.g. 1 second). Obviously the number of orders given at any moment must also be held fixed. We assume the maximum number of orders given at any moment to be the number of switches in the whole system. Our analysis aims to determine the way that the different protocols influence performance. We have attempted to construct worst case schedules for the protocols to get a pessimistic measure on the maximum number of switches that can be modeled.

4.1 Benchmark scenarios

Rolling Cover

In this scenario, at each time step t an order is issued to turn on the switch at time $t+1$ to $t+2$. When 1 second elapses, another like order is issued. By this protocol, the switch should stay on indefinitely, but always in danger of turning off if there is a lag. See figure 4.1.

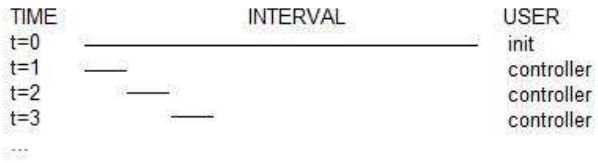

Figure 4.1: Rolling cover

Rolling Flicker

In this scenario, at each time step t an order is issued to turn the switch at time $t+d$ to $t+d+1$, where d is the number of orders issued this far. The resulting orders have a one second gap between them which is covered by init's order (See figure 4.2).

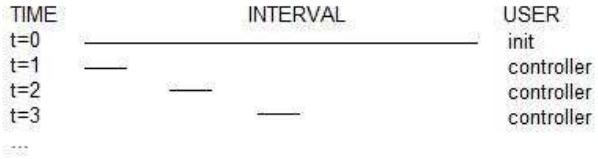

Figure 4.2: Rolling flicker

Rolling Squash

This scenario increases the `VST` of the order by one second and decreases the `VET` by one second per insert. The result is that every user order is during an earlier user order. See figure 4.3.

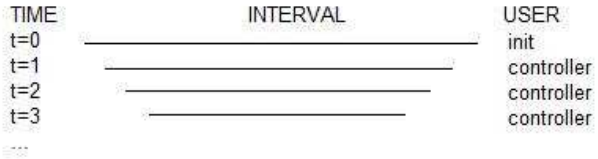

Figure 4.3: Rolling squash

4.2 Tests runs

The test runs were done over a period of 100 seconds each on a public Postgresql server with 2GB RAM, 2x UltraSPARC IIIi 1504MHz and a pair of mirrored 15000 RPM SCSI disks running Sun Solaris 10 update 7. Since the server is public with a large number of users and not dedicated to this task, it is likely that the performance would be improved by just having a dedicated server. Each of the three insert methods were tested over all three protocols to see how much workload each protocol could handle without lagging more than 1% (i.e. 1 second over the whole test period). See figure 4.4.

4.3 Test results

All three protocols handled the *rolling cover* for inserts equally well with 85 switches with one insert each per second before the lag exceeded 1 second. Overall, this scenario gave the best performance since every new order just took the first second of the current order given by 'INIT'. Hence, no splitting was required. The *rolling flicker* method was a bit more burdensome, but all three protocols performed equally well, being able to handle 55 switches. Performance decreased because there are small gaps between every order given by ordinary users, which causes splitting since those gaps need to be filled with the initial order given by 'INIT'. The *rolling squash* scenario gave both the best and the worst performance results. The best performance was achieved when the *blocking protocol* was in effect. Since every order, except the first, was blocked, the database had very little work to do and the system did not lag as much as 1 second even with

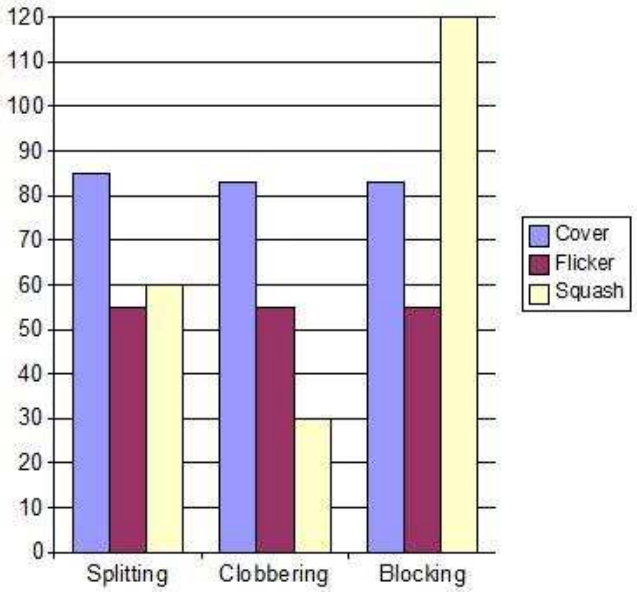

Figure 4.4: Maximum number of switches that each protocol could handle with max 1% (1 second) lag.

200 attributes in the database. The *splitting protocol* managed this scenario decently with 60 switches. Performance was the worst for the *clobbering protocol* as the inserts caused the maximum number of splittings per switch at all times and the gaps caused by the clobbering of earlier orders had to be filled with deleted sections of the initial order given by `'INIT'`.

Chapter 5

Discussion

Relational databases have had a very rich near 40 year history, so it is difficult to say how many ideas presented in this thesis have been explored in the past. Certainly many have set up integrated systems where the state of the database is polled and operations are then carried out in the real world based on the database state. Surprisingly, however, it seems that a general exploration of this idea has not been explored in much depth. One of the only examples we have managed to locate is the work presented in [9] where the authors propose a way to model home appliances as databases to leverage their interface work in NLIs to databases.

In that work the authors leverage a planner to achieve goals posted as tuples in the database. As such they do not in any way deal with temporal aspects of the task, but instead they let the device take care of the temporal control. For example, instead of giving an order to the database saying "20.00 tell the VCR to start recording", they immediately carry out the order on the VCR and tell it to set its timer to 20.00.

A somewhat related idea is complex event processing (CEP). CEP is more of a reactive system which, by monitoring the current events with a large number of sensors, tries find out what complex event that really happened by reasoning with the minor events that just took place as a decision base. A car system using CEP that logs a flat tire and a speed change from 100 to 0 in a very short period of time would think that the car has crashed and would probably set off the alarm. As we all can see, this is no bullet-proof technique. For example, the car could be driven on an icy road when the tire blows, instinctively the driver steps on the break pedal and stops the wheels, hence the speed is zero. But that does not mean that the car has crashed. CEP is a good initiative, but cannot really be compared to a control system based on a bi-temporal database since the CEP is reactive and tries to figure out and approximate what has happened with the help of sensors whereas the control system knows what will happen. Using the control system technology, one can even say some of the things that will happen in the future. Therefore, the fields of application for these two similar, yet different, systems differ.

5.1 Applications

For example, security applications are well-suited for this type of technology. The system would keep track of who enabled the alarm, at what time the order was given and when the system went into that state. Also you could ask the system for how long the alarm

will stay enabled (or disabled). The database log would be of biggest interest in case of a burglary or similar. With this system it would be easy to check if the alarm was turned off and, if so, who turned it off. It would also be possible to see certain trends prior to a crime. Maybe some employee has had a lot of late night activities in the facility lately. Depending on what kind of company that is using the alarm system it could be easier to find a motive. With the growing number of money depot robberies in mind, I think most of us at some point has thought of the possibility that someone working at the depot or the company that owns the depot has been involved in the robbery in one way or another. Times like that it would be really interesting to rummage through the database log to search for certain trends which seem abnormal or suspect.

An addition to the alarm system above would be to use the same system, with minor modifications, to control the gates and doors to the facility and inside it. With both the alarm and the gate system in use the company would have complete logs of which employee that entered the building first and disabled the alarm, and also it would log when the rest of the crew arrived to work. With the gate opening logged it would be possible to check who have been in the building and at what times. If only the alarm system was to be used, anyone with access to the building could go in unnoticed by the system as long as someone has turned off the alarm prior to this persons entry. A combination of the control system handling the gates and the control system handling the alarm would give a pretty solid ground to monitor who have been in certain locations of the facility and when they were there.

With the rising energy prices it is of interest to save energy as far as possible, which is good not only for our wallets but for the environment we live in and depend on. Imagine that you could set the thermostat in your home to lower the heat in the house during daytime on working days. Say that you have ordered the system to lower the temperature inside the house on Mondays through Fridays between 08.00 until 2 hours prior to your homecoming. Then you would save money during day time when nobody is in the house, and the system would turn up the heat before get home so you would not notice any change in your everyday life except the smaller energy bill. To control a thermostat the state, or value, in the virtual model has to have more options than on and off. A range from 0 (off) to 40 degrees Celsius would be required. Having a variety of states instead of just on and off is trivial, but even if the system has ordered the real world (in this case your house) into the state of 20 degrees there is no way for the virtual system to know if 20 degrees is the actual temperature in the real world. In case of an open window in the middle of the coldest winter it would not matter if the system had ordered the real world to be 20 degrees, since the radiators could not possible keep that temperature if icy air flows in through that window. The system would then **think** that the temperature in your home is 20 degrees when in fact it is much lower. The remedy for such events is to install sensors that inform the system of the real world temperature. No sensors have been used in the test scenarios of this thesis due to the limited amount of time, but adding them to the system is simple as the sensors work orthogonally to the actuators.

Chapter 6

Conclusions and future directions

This thesis has conclusively demonstrated the feasibility of modeling logical control systems as bi-temporal databases of orders. Even for very pessimistic schedules, an old, non-dedicated database server with very tight time tolerances can be supported. The system can be guaranteed to support numerous switches (50 switches with a maximum of 1% lag time). Certainly it seems quite feasible that with a more modern, dedicated server operating over less pessimistic real-world schedules, this system is not only practical for the home of the future, but quite likely able to control thousands, if not tens of thousands of switches distributed around a small to medium sized city.

In the future there will be experiments regarding integration of the natural language interface system C-Phrase [6] with the controller system. The linguistic properties of temporal expressions will be handled within C-Phrase and, in the case of giving orders, it will result in inserts of `ORDER` tuples. What will also be interesting is to use views to enable the querying of the system to fully support the types of conversations described in the the introduction. This thesis, being focused as it were in the underlying database and controller mechanism, takes us a long way toward satisfying these types of scenarios. What we now need to reflect on is more about what type of real world practical cases the system can handle.

Chapter 7

Acknowledgements

First of all, I want to thank Michael Minock for good ideas and advice. Secondly, I want to thank Emma Appelqvist and my family for their endless support.

References

[1] *Developer Resources for Java Technology.*
`http://java.sun.com` (visited June 24, 2009).

[2] *PostgreSQL.*
`http://www.postgresql.org` (visited May 20, 2009).

[3] *PL/pgSQL - SQL Procedural Language.*
`http://www.postgresql.org/docs/8.3/interactive/plpgsql.html`
(visited May 21, 2009).

[4] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM. 26/11/1983*, pages 832–843.

[5] R. Elmasri and S. Navathe. *Fundamentals of Database Systems 3rd edition.* Addison Wesley, 2000.

[6] M. Minock. C-phrase: A system for building robust natural language interfaces to databases. *Journal of Data and Knowledge Engineering*, 2009.

[7] B. Momjian. *PostgreSQL: Introduction and Concepts.* Addison Wesley, 2001.

[8] Richard T. Snodgrass. Introduction to tsql2. In *The TSQL2 Temporal Query Language*, pages 19–30. 1995.

[9] Alexander Yates, Oren Etzioni, and Daniel S. Weld. A reliable natural language interface to household appliances. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, 2003.

Chapter 8

Appendix A -source code

SCHEMA
These are the two tables, `orders` and `orders_history`, that are needed in the system. The first intention was to keep everything in the orders `table`, but as the number of rows grew, the performance of the system dropped. We wanted to keep track of the past orders as a log, which would be useful should the system be applied in a security environment, so we were reluctant to delete old or no longer valid orders. The remedy was to create a `history_table` to which those orders were moved. That way the log is kept and the performance sustained. The unique identifier for each row is `id` which is given by the sequence `orders_seq`. To make an insert to `orders`, only `attr`, `val`, `VST` and `VET` are needed. The rest are set by the database.

```
CREATE SEQUENCE orders_seq;

CREATE TABLE orders(
id INTEGER DEFAULT nextval('orders_seq') PRIMARY KEY,
attr VARCHAR(100),
val VARCHAR(100) NOT NULL,
vst TIMESTAMP NOT NULL,
vet TIMESTAMP NOT NULL,
   CHECK (vst <= vet),
tst TIMESTAMP,
tet TIMESTAMP,
usr VARCHAR(40),
performed INT
);

CREATE TABLE orders_history(
id INT ,
attr VARCHAR(100),
val VARCHAR(100) NOT NULL,
vst TIMESTAMP NOT NULL,
vet TIMESTAMP NOT NULL,
tst TIMESTAMP NOT NULL,
tet TIMESTAMP NOT NULL,
```

```
usr VARCHAR(40) NOT NULL,
performed INT NOT NULL
);
```

Splitting protocol

The splitting protocol makes no difference between the users on the database. Each user's order is treated the same way, namely by overwriting any previously given order over that interval and attribute. However, this protocol does not clobber the old orders that the new order is touching, but just redefines their valid start times and/or valid end times if the old orders should cover some interval that is outside of the new order's interval.

```
CREATE FUNCTION intervalchecks () RETURNS TRIGGER AS '
DECLARE
old RECORD;

BEGIN
  IF new.tst is NULL THEN
   new.tst := now();
  END IF;
  IF new.usr IS NULL THEN
   new.usr := USER;
  END IF;
  new.tet := ''9999-12-31 23:59:59'';
  IF new.performed IS NULL THEN
   new.performed := 0;
  END IF;

--  old order(s) during the new order
  LOOP
    SELECT * INTO old
    FROM orders
    WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
    AND (new.vst <= vst AND new.vet >= vet);
    IF NOT FOUND THEN EXIT;
     ELSE
       UPDATE orders SET tet = new.tst WHERE id = old.id;
    END IF;
   END LOOP;

--  new order during the old order(s)
    LOOP
      SELECT * INTO old
      FROM orders
      WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
  AND (vst < new.vst AND new.vet < vet);
    IF NOT FOUND THEN EXIT;
     ELSE
  UPDATE orders SET tet = new.tst WHERE id = old.id;
  INSERT INTO orders(attr,val,vst,vet,tst,tet,usr,performed)
    VALUES(old.attr, old.val, old.vst, new.vst, old.tst, new.tet,
                  old.usr, old.performed);
  INSERT INTO orders(attr,val,vst,vet,tst,tet,usr,performed)
    VALUES(old.attr, old.val, new.vet, old.vet, old.tst, new.tet,
```

```
                  old.usr, 0);
END IF;
END LOOP;

-- new order overlaps or starts old order(s)
LOOP
  SELECT * INTO old
          FROM orders
          WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
    AND (new.vst <= vst AND vst < new.vet AND new.vet < vet);
    IF NOT FOUND THEN EXIT;
ELSE
   UPDATE orders SET tet = new.tst WHERE id = old.id;
  INSERT INTO orders(attr, val, vst, vet, tst, tet, usr, performed)
  VALUES(old.attr, old.val, new.vet, old.vet, old.tst, new.tet,
                old.usr, old.performed);
  END IF;
END LOOP;

-- old order(s) overlaps with new order or new order finishes old order
LOOP
  SELECT * INTO old FROM orders
          WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
  AND (vst < new.vst AND new.vst < vet AND vet <= new.vet);
  IF NOT FOUND THEN EXIT;
  ELSE
    UPDATE orders SET tet = new.tst WHERE id = old.id;
    INSERT INTO orders(attr, val, vst, vet, tst, tet, usr, performed)
    VALUES(old.attr, old.val, old.vst, new.vst, old.tst, new.tet,
                  old.usr, old.performed);
  END IF;
END LOOP;


-- when new order takes place before old order, do nothing.
-- when old order takes place before new order, do nothing.

-- move order with terminated TETs to HISTORY TABLE
-- move those that have already been performed and have VET in past.
LOOP
  SELECT * INTO old
          FROM orders
          WHERE attr = new.attr AND (tet <> new.tet OR (vet < new.tst and
  performed = 1));
  IF NOT FOUND THEN EXIT;
    ELSE
   INSERT INTO orders_history(id,attr,val,vst,vet,tst,tet,
                       usr,performed)
           VALUES(old.id,old.attr,old.val,old.vst,old.vet,old.tst,old.tet,
```

```
                old.usr,old.performed);
   DELETE FROM orders WHERE id = old.id;
  END IF;
END LOOP;
return new;
END;

' LANGUAGE plpgsql;

CREATE TRIGGER intervalchecks BEFORE INSERT
ON orders FOR EACH ROW EXECUTE PROCEDURE intervalchecks();
```

Blocking protocol

The blocking protocol works the same way as the splitting protocol when it comes to overwriting orders given by 'INIT', but it blocks new orders that are in time conflict with any previously given order on the same attribute, if that order has been given by someone else than 'INIT'.

```
CREATE FUNCTION intervalchecks () RETURNS TRIGGER AS '
DECLARE
old RECORD;
BEGIN
  IF new.tst IS NULL THEN
   new.tst := now();
  END IF;
  new.tet := ''9999-12-31 23:59:59'';
  IF new.performed IS NULL THEN
   new.performed := 0;
  END IF;
  IF new.usr IS NULL THEN
   new.usr := USER;
  END IF;



-- old order not given by 'init' in conflict with new, if exists exit
  SELECT id,usr INTO old FROM orders
  WHERE usr <> ''init'' AND attr = new.attr AND ((vst<=new.vst AND new.vst<vet)
  OR (vst < new.vet AND new.vet <= vet) OR (new.vst <=vst AND vet <= new.vet));
  IF FOUND THEN
   RAISE NOTICE ''blocking due to conflict, row: %'', old;
   RETURN NULL;
  ELSE

--  old order(s) during the new order
   LOOP
    SELECT * INTO old
    FROM orders
    WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
    AND (new.vst <= vst AND new.vet >= vet);
    IF NOT FOUND THEN EXIT;
     ELSE
       UPDATE orders SET tet = new.tst WHERE id = old.id;
    END IF;
   END LOOP;

--  new order during the old order(s)
    LOOP
      SELECT * INTO old
      FROM orders
      WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
AND (vst < new.vst AND new.vet < vet);
```

```
IF NOT FOUND THEN EXIT;
     ELSE
  UPDATE orders SET tet = new.tst WHERE id = old.id;
  INSERT INTO orders(attr,val,vst,vet,tst,tet,usr,performed)
    VALUES(old.attr, old.val, old.vst, new.vst, old.tst, new.tet,
                  old.usr, old.performed);
  INSERT INTO orders(attr,val,vst,vet,tst,tet,usr,performed)
    VALUES(old.attr, old.val, new.vet, old.vet, old.tst, new.tet,
                  old.usr, 0);
END IF;
END LOOP;

-- new order overlaps or starts old order(s)
LOOP
  SELECT * INTO old
          FROM orders
          WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
    AND (new.vst <= vst AND vst < new.vet AND new.vet < vet);
    IF NOT FOUND THEN EXIT;
ELSE
   UPDATE orders SET tet = new.tst WHERE id = old.id;
  INSERT INTO orders(attr, val, vst, vet, tst, tet, usr, performed)
  VALUES(old.attr, old.val, new.vet, old.vet, old.tst, new.tet,
                old.usr, old.performed);
  END IF;
END LOOP;


-- old order(s) overlaps with new order or new order finishes old order
LOOP
  SELECT * INTO old FROM orders
          WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
  AND (vst < new.vst AND new.vst < vet AND vet <= new.vet);
  IF NOT FOUND THEN EXIT;
  ELSE
    UPDATE orders SET tet = new.tst WHERE id = old.id;
    INSERT INTO orders(attr, val, vst, vet, tst, tet, usr, performed)
    VALUES(old.attr, old.val, old.vst, new.vst, old.tst, new.tet,
                  old.usr, old.performed);
  END IF;
END LOOP;

-- when new order takes place before old order, do nothing.
-- when old order takes place before new order, do nothing.
-- move order with terminated TETs to HISTORY TABLE
-- move those that have already been performed and have VET in past.

LOOP
  SELECT * INTO old
```

```
          FROM orders
          WHERE attr = new.attr AND (tet <> new.tet OR (vet < new.tst and
  performed = 1));
  IF NOT FOUND THEN EXIT;
    ELSE
   INSERT INTO orders_history(id,attr,val,vst,vet,tst,tet,
                       usr,performed)
           VALUES(old.id,old.attr,old.val,old.vst,old.vet,old.tst,old.tet,
                  old.usr,old.performed);
   DELETE FROM orders WHERE id = old.id;
  END IF;
END LOOP;
return new;

  END IF;
END;

' LANGUAGE plpgsql;


CREATE TRIGGER intervalchecks BEFORE INSERT
ON orders FOR EACH ROW EXECUTE PROCEDURE intervalchecks();
```

Clobbering protocol

The clobbering protocol handles orders given by 'INIT' the same way as the other two protocols, but it treats orders given by other users very differently. Should the new order be in time conflict with any order given by another user (except 'INIT') on the same attribute, the clobbering deletes the complete old order. Should there be any void between the remaining time intervals, this is filled by fetching the original order over that interval given by 'INIT' from the history_table. NOTE! One cannot change the past, and for that reason it is not possible to clobber orders that have already been performed but have VET in the future.

```
CREATE FUNCTION intervalchecks () RETURNS TRIGGER AS '
DECLARE
old RECORD;
initVal VARCHAR(30);

BEGIN
  IF new.tst IS NULL THEN
   new.tst := now();
  END IF;
  IF new.usr IS NULL THEN
   new.usr := USER;
  END IF;
  new.tet := ''9999-12-31 23:59:59'';
  IF new.performed IS NULL THEN
   new.performed := 0;
  END IF;

-- check for conflicts with orders not given by 'init', if exists exit
  SELECT id INTO old FROM orders
  WHERE usr <> ''init'' AND attr = new.attr AND performed = 1 AND
  ((vst <= new.vst AND new.vst < vet )
OR (vst < new.vet AND new.vet <= vet) OR
(new.vst <= vst AND vet <= new.vet));
  IF FOUND THEN
   RAISE NOTICE ''blocking due to performed conflict, row: %'', old;
   RETURN NULL;
  ELSE

--  old order(s) during the new order
   LOOP
    SELECT * INTO old
    FROM orders
    WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
    AND (new.vst <= vst AND new.vet >= vet);
    IF NOT FOUND THEN EXIT;
     ELSE
       UPDATE orders SET tet = new.tst WHERE id = old.id;
    END IF;
   END LOOP;
```

```
--  new order during the old order(s)
    LOOP
      SELECT * INTO old
      FROM orders
      WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
AND (vst < new.vst AND new.vet < vet);
IF NOT FOUND THEN
EXIT;
     ELSE
IF old.usr = ''init''
THEN
  UPDATE orders SET tet = new.tst WHERE id = old.id;
  INSERT INTO orders(attr,val,vst,vet,tst,tet,usr,performed)
    VALUES(old.attr, old.val, old.vst, new.vst, old.tst, new.tet,
                  old.usr, old.performed);
  INSERT INTO orders(attr,val,vst,vet,tst,tet,usr,performed)
    VALUES(old.attr, old.val, new.vet, old.vet, old.tst, new.tet,
                  old.usr, 0);
ELSE -- if o.usr != init
IF old.performed = 1 THEN RETURN NULL;
ELSE
INSERT INTO
orders_history(id,attr,val,vst,vet,tst,tet,usr,performed)
VALUES (old.id, old.attr, old.val, old.vst, old.vet, old.tst, new.tst,
old.usr, old.performed);
DELETE FROM orders WHERE id = old.id;
SELECT val INTO initVal FROM orders_history WHERE id =
(SELECT min(id) AS mid FROM orders_history WHERE attr=old.attr);
               INSERT INTO
orders(attr, val, vst, vet, tst, tet, usr, performed)
VALUES(old.attr, initVal, old.vst, old.vet, old.tst, old.tet,''init'',0);
END IF;
END IF;
END IF;
END LOOP;


-- new order overlaps or starts old order(s)
LOOP
  SELECT * INTO old
         FROM orders
         WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
    AND (new.vst <= vst AND vst < new.vet AND new.vet < vet);
    IF NOT FOUND THEN EXIT;
ELSE
IF old.usr = ''init''
THEN
    UPDATE orders SET tet = new.tst WHERE id = old.id;
   INSERT INTO
```

```
orders(attr, val, vst, vet, tst, tet, usr, performed)
  VALUES(old.attr, old.val, new.vet, old.vet, old.tst, new.tet, old.usr,
  old.performed);
ELSE -- if o.usr != init
IF old.performed = 1 THEN RETURN NULL;
ELSE
INSERT INTO
orders_history(id,attr,val,vst,vet,tst,tet,usr,performed)
VALUES (old.id, old.attr, old.val, old.vst, old.vet, old.tst, new.tst,
old.usr, old.performed);
DELETE FROM orders WHERE id = old.id;
SELECT val INTO initVal FROM orders_history WHERE id =
(SELECT min(id) AS mid FROM orders_history WHERE attr=old.attr);
               INSERT INTO
orders(attr, val, vst, vet, tst, tet, usr, performed)
VALUES(old.attr, initVal, old.vst, old.vet, old.tst, old.tet,''init'',0);
END IF; -- end of performed=0
END IF;  -- end of user != init
  END IF;
END LOOP;


-- old order(s) overlaps with new order or new order finishes old order
LOOP
  SELECT * INTO old FROM orders
         WHERE attr = new.attr AND tet = new.tet AND vet >= new.tst
  AND (vst < new.vst AND new.vst < vet AND vet <= new.vet);
  IF NOT FOUND THEN EXIT;
  ELSE
IF old.usr = ''init''
THEN
     UPDATE orders SET tet = new.tst WHERE id = old.id;
     INSERT INTO orders(attr, val, vst, vet, tst, tet, usr, performed)
     VALUES(old.attr, old.val, old.vst, new.vst, old.tst, new.tet,
                  old.usr, old.performed);
ELSE -- if o.usr != init
IF old.performed = 1 THEN RETURN NULL;
ELSE
INSERT INTO
orders_history(id,attr,val,vst,vet,tst,tet,usr,performed)
VALUES (old.id, old.attr, old.val, old.vst, old.vet, old.tst, new.tst,
old.usr, old.performed);
    DELETE FROM orders WHERE id = old.id;
         SELECT val INTO initVal FROM orders_history WHERE id =
(SELECT min(id) AS mid FROM orders_history WHERE attr=old.attr);
               INSERT INTO
orders(attr, val, vst, vet, tst, tet, usr, performed)
VALUES(old.attr, initVal, old.vst, old.vet, old.tst, old.tet,''init'',0);
END IF; -- end of performed=0
```

```
END IF;  -- end of user != init

  END IF;
END LOOP;

-- when new order takes place before old order, do nothing.
-- when old order takes place before new order, do nothing.

-- move order with terminated TETs to HISTORY TABLE
-- move those that have already been performed and have VET in past.
LOOP
  SELECT * INTO old
          FROM orders
          WHERE attr = new.attr AND (tet <> new.tet OR (vet < new.tst and
  performed = 1));
  IF NOT FOUND THEN EXIT;
    ELSE
   INSERT INTO orders_history(id,attr,val,vst,vet,tst,tet,
                       usr,performed)
       VALUES(old.id,old.attr,old.val,old.vst,old.vet,old.tst,old.tet,
                   old.usr,old.performed);
   DELETE FROM orders WHERE id = old.id;
  END IF;

END LOOP;
return new;
END IF;
END;

' LANGUAGE plpgsql;

CREATE TRIGGER intervalchecks BEFORE INSERT
ON orders FOR EACH ROW EXECUTE PROCEDURE intervalchecks();
```

Chapter 9

Appendix B - user's guide

Scanner
This is straight forward to use. Go to the scanner-folder and type ./start and the scanner is up and running, querying the database for actions to forward to the API. Should you want to perform the orders on any other objects than the simulated lights, just change api.java to perform the appropriate actions. The API receives two strings from the scanner: attribute and value.

Interface
The interface is in the interface-folder. Start it by typing ./start <digit>. If you want to start it without adding any extra attributes to the database, just type ./start. If you want to add more attributes, the following syntax starts the interface and adds 10 new lights: ./start 10. New attributes should only be added by database user 'INIT'. Once the interface is started, you click the attributes you want to give orders, select the valid start time and the valid end time from the top menus. The uc-button sets the valid end time to 9999-12-31 23:59:59. When the wished interval, value and attributes have been chosen, press the set-button to insert the order into the database.

Database
To begin with, you need two database users named init and controller to your PostgreSQL database. Start the database and type \i schema.sql and \i install-splitting.sql if you want to install the splitting protocol. If you wish to use the blocking protocol, type \i install-blocking.sql instead of \i install-splitting. And finally \i install-clobbering.sql if you want to use the clobbering protocol. Remember that only one protocol can be installed at the time.

9.1 System description

No Java code will be included, but the system is fairly simple as these UMLs show. The arrows point toward the owner of the object. No inheritance exists. See figure 9.1 and 9.2.

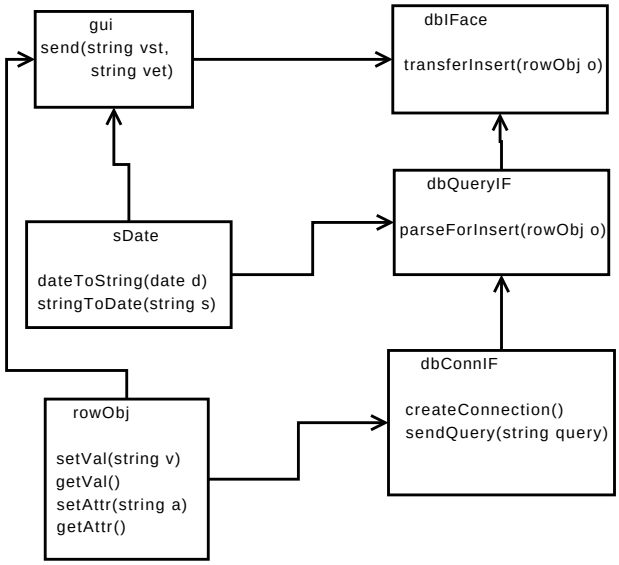

Figure 9.1: UML of the interface

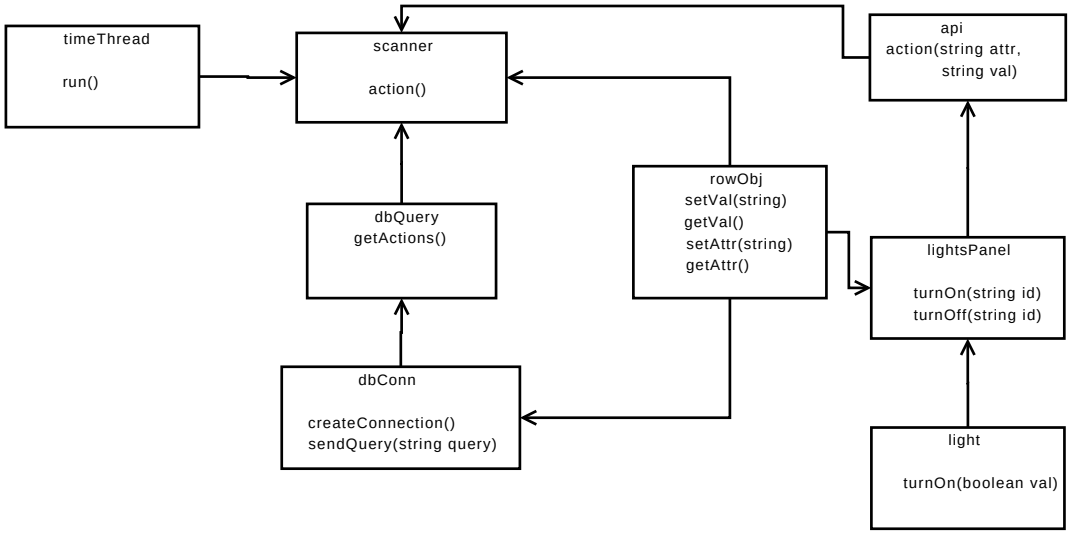

Figure 9.2: UML of the scanner

9.2 Test system

Since it is impossible for a single user to send all the inserts every second to produce worst case scenarios, we had to simulate the inserts. The inserts were generated by a test class which, depending on insert method, gave the inserts different valid time intervals. The insert methods were cover, flicker and squash. Each of them have been presented earlier in this report, but here we will mention more about how the time intervals were generated.

Rolling cover
In the case of rolling cover inserts, each insert follows an earlier order exactly, that is `old.vet = new.vst`. Each order has an interval of 1 second: `new.vet = new.vst`+1, and every new order begins 1 second from now and lasts for 1 more second. This method guarantees that no old orders, except for the initial order, are in time conflict with the new order. It generates a lot of inserts but no splitting since the new order only takes the first chunk of the initial order's time interval. Below is the pseudo code to generate new orders.

```
for each second {
    vst=vet=now();
    vst.setSeconds(vst.getSeconds()+1);
    vet.setSeconds(vet.getSeconds()+2);
    for each attr{
        INSERT INTO orders(attr, val, vst, vet)
            VALUES(obj.name, 'on', vst, vet);
    }
}
```

Rolling flicker
The rolling flicker is more or less the same as the rolling cover, but instead of increasing the valid start time by 1 second per second, it is increased by 2 seconds per second. This method guarantees that there are no previous orders except the initial order that is in time conflict with the new order. It also guarantees a lot of splitting of the initial order since there is a 1 second gap between every new order. Here follows the pseudo code.

```
offset=2;
for each second {
    offset=offset+1;
    vst=vet=now();
    vst.setSeconds(vst.getSeconds()+ offset);
    vet.setSeconds(vet.getSeconds()+ offset+1);
    for each attr{
        INSERT INTO orders(attr, val, vst, vet)
            VALUES(obj.name, 'on', vst, vet);
    }
}
```

Rolling squash
When using the rolling squash insert method, the orders are shaped like an upside down

triangle with the order with the longest interval on top. The second order is 2 seconds shorter than the first, 1 second shorter at the beginning and 1 second shorter at the end of the interval. This is bound to cause a lot of splitting, as long as the blocking protocol is not installed.

```
counter=100
for each second {
    counter=counter-1;
    vst=vet=now();
    vst.setSeconds(vst.getSeconds()+3);
    vet.setSeconds(vet.getSeconds() +(2+(counter*2)));
    for each attr{
        INSERT INTO orders(attr, val, vst, vet)
            VALUES(obj.name, 'on', vst, vet);
    }
}
```

9.3 Screen-shots

Here follow two screen-shots, one of the GUI and one of the simulated lights. There are 100 lights in the system at the moment of the screen-shot and most are turned on.

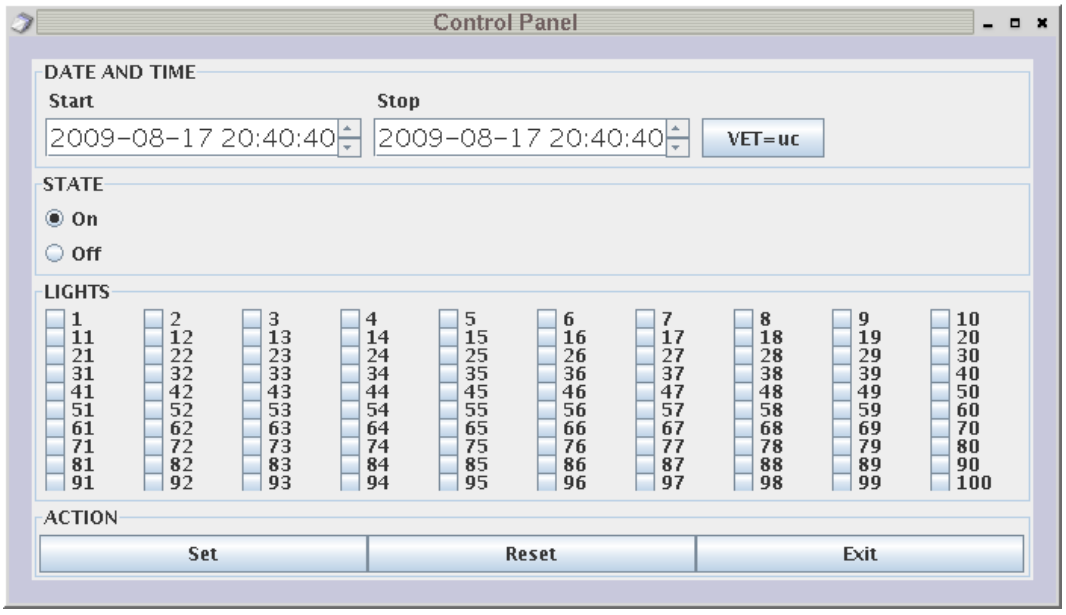

Figure 9.3: Screen-shot of the GUI

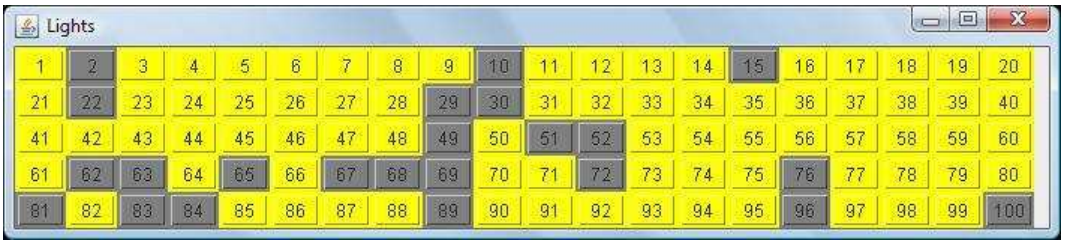

Figure 9.4: Screen-shot of the simulated lights