Specification and Formal Analysis of the Light Control Problem by an integrated formal approach

J. Christian Attiogbé

IRIN - Université de Nantes
2, rue de la Houssinière - 44322 Nantes Cedex 3
email: Christian.Attiogbe@irin.univ-nantes.fr

(Submission to FME'2001 - Category 1)

Abstract: A stepwise analysis of the Light Control Problem is presented using the Configuration Machines Formalism, an integrated approach to deal with data and behavior. The result is a specification of the main parts of the Light Control System to be developed. The specification is a set of communicating machines which correspond to the subsystems identified in the Light Control System. Machines capture both data and behavior of different parts of the system. The data part of each machine is specified using Z like formalism whereas the behavior part is specified as a Labelled Transition System. Communications between machines, between machines and environment are based on synchronous message exchanges. The specification obtained is then used for formal analysis: validation of the specification by proving some properties extracted from the requirements, simulation issue is addressed by presenting a technique to translate the specification into Promela in order to use the Spin environment.

Keywords: Requirements Capture, Formal specification techniques, Configuration Machines, embedded software, Z/Eves, Promela

1 Introduction

We present in this paper a specification of the Light Control Problem (LCP) together with a formal analysis as a result of the works done to capture the requirements given in the informal description of the problem. The paper presents our approach to reach this result and a stepwise description of the way we do it. Formal analysis is done with $Z/EVES$ [Saa97] and Promela/Spin [Hol91]. The complete description of the LCP and some customer's feedbacks are available electronically at [Iss99]. As the informal description is very long (13 pages), we recall here the main ideas to guide the readers and introduce some details in the papers during the specification. The problem is the development of a new light control system to replace an existing one in the fourth floor of the Building 32 of the University of Kaiserlautern. The new system should provide some possibilities to control manually and automatically the lights in the different rooms of the floor and to control the lights with respect to user needs and daylight. We must note according to this previous short description that the system is a reactive embedded system with some particular aspects: high level interaction between physical devices (sensors, convertors, lights, actuators) and the system, importance of the time (some actions must be executed periodically), important interaction between the system and its environment, be it with human (users of the building, facility manager) or with control devices (detectors, actuators, etc), reliability: the function of the light system should be safe for users (no hazardous conditions for persons, inventory, or building are allowed), conviviality: (easy use of the interface, warnings for malfunctions, display of light status, etc). The description given for the floor of the building includes different kinds of rooms: Offices, Hallway sections, Computer Labs, Hardware Labs, Peripherals, Meeting rooms. The equipments of each room are described: ceiling lights, control panel, push buttons, etc. The descriptions given (in [Iss99]) to Computer Lab, Hardware Lab, Meeting Room, Peripheral Room, Hallway Section and Staircase show that, the control procedures of the equipments are quite the same and mainly based on that of Office and Hallway Section. There are in the requirements, some paragraphs expressing user needs (noted U1, U2, U3 etc), Facility Manager needs (FM1, FM2, FM3, etc) and non-functional needs (NF1, NF2, NF3, etc). We introduce some of these paragraphs during the specification. Facility manager is a person responsible of the the control system, he is informed of malfunctions, he can interact with the system.
We restrict our attention, therefore, to the specification of a room controler extended to an office controler and we mention how this specification can be extended to the whole requirements.
We make in the work presented here the demonstration that our approach is suitable for this given problem. The Light Control Problem is a real life case and constitutes by its complexity a substantial case study or benchmark

to put formal methods to the test. We show in this paper that most of the difficulties of the problem can be tackled simply with our approach. We expect that this report of experiment can help the readers and more generally software developers to use our approach in such a real life case study.
Our approach is based on the use of the Configuration Machines Formalism [Att98],[Att00] more as a technique to combine several paradigms of specification than as a really new formalism. Therefore, we consider that any system can be described through the set of configurations by which it transits during its execution. Each configuration is characterized by different information: the objects manipulated by the system, the events on which the system reacts, the rules defining objects manipulation, the environment of the system including graphical aspects and communication devices. The notion of machines is strongly related to that of state machines and abstract machines. A machine captures both data and behavior of part of system. It implies the step by step evolving through time. The notion of configuration is considered as an abstraction of the classical notion of state in state machines. It describes classes of states by their properties. As done with the model of state machines, to describe the behavior of a system, we have to describe its evolution from the initial configuration and for each control event.
There are two aspects, the static one which corresponds to the description of configuration and the dynamic one which is the description of the evolution from one configuration to another. A system can be made of several independent subsystems interacting to achieve the functionalities of the system. The environment in which a system will evolve is a main factor for its specification and development. In our approach we always consider the environment in which any given system should be specified.
The paper is organized as follows: the section 2 gives an overview of the formalism and the methodological issue for requirements capture. In section 3 we give the main guidelines as a control model and some steps for capturing the requirements of the Light Control Problem. The section 4 presents the specifications obtained when capturing the requirements. Section 5 gives a brief specification of the whole system. Section 6 is devoted to formal analysis: correctness of the specification, validation of some requirements by properties proving, translation of the specification into interacting processes. The last section is devoted to an appraisal on the work and some concluding remarks.

2 Configuration Machines Formalism

2.1 Overview

Configuration Machines Formalism [Att98] is intended to specify multifaceted systems (data, functional or process, behavior, graphical view, communication). The formalism is suited for the specification of multifaceted systems. By multifaceted systems we mean systems where the facets such as *data (simple or complex)*, *functional (operations on data)*, *behavior (with concurrency or not, distributed or not)* are present. The formalism combines basic fundamentals concepts, simplicity (access to nonspecialists), readability, expressiveness.
The main construct is that of *Machine* which gathers data part and behavior part. The data part uses Z specification language [Spi92], [BSC94] to describe a *state space* for machines and more generally for systems built from machines. The behavior part is described using semantic rules called *transactions*. The expression of transaction is based on the notation

$$\frac{H \ \vdash \ P}{H \ \vdash \ Q}$$

where $H \ \vdash \ P$ characterizes initial *configuration* which works as a *guard* for the transaction and $H \ \vdash \ Q$ characterizes the *configuration* reached. P and Q are (conjunctions of) predicates. H combines more generally declarations and predicates. An *action* name is used as label for each transaction; it is the action the machine can do to reach the configuration described. The notation for transaction is

$$\left| \frac{H \ \vdash \ P}{H \ \vdash \ Q} \right| [aa]$$

where aa is the action name.
The Z schemas ([Declarations | Predicate]) are used as predicate [BSC94] to describe the configuration in the form $H \ \vdash \ P$. A transaction is enabled if its initial configuration is *true*. If the initial configuration (guard) of a transaction is not met, the transaction is blocking. A machine evolves by doing nondeterministically the action which labels one of the enabled transactions. The formalism permits the expression of *invariant*: transaction for which, if the initial configuration is true the reached configuration should be true. A machine can receive an events ee (the notation is $\xleftarrow{ee}$) or send events (the notation is $\xrightarrow{ee}$). These events are part of configuration description. They can be valued and are introduced as predicates: $\xleftarrow{ee}$ stands for the predicate *the event ee*

is received, $\xrightarrow{ee}$ stands for the predicate *the event ee is sent*. Therefore we can have the quantified forms: $\forall\, i\ \in \{1,2,3\} \bullet \xrightarrow{ee(i)}$ and $\exists\, i \mid \xleftarrow{ee(i)}$. At this language level, the formal interpretation is that there is a type *Event* characterising all events; a dummy event of this type is considered for initialisation purpose ($dummy\_event$). Configuration Machine, as a structuration unit can be compared with *class* in $object - Z$ or $abstract\ machine$ in B. However whereas B uses precondition in operations (resulting in chaos if the pre-condition is not enabled), Configuration Machine and Object-Z use guards (resulting in blocking if pre-condition is not enabled).
Another specification unit introduced latter is that of *system*. Indeed machine types can be instantiated, extended, composed to give *systems*. Also *systems* have in addition the clauses INITIALISATION, PROPERTIES. Other details are introduced during the specification.

2.2 Methodological issues

In general, methods or approaches to specify, analyze, design and develop a software system are related to the kind of system: the nature of the system, the different facets present in the system and their importance. The main methodological approach behind the use of Configuration Machines Formalism is the use of *abstraction* and *composition* relatively to the kind of system we have. Here, for the Light Control Problem (LCP), we have a kind of computer-driven process control system which can be considered as a *reactive system*. We show in figure 1 a generic architecture of process control systems. It will be used as a basis of the analysis of the LCP.

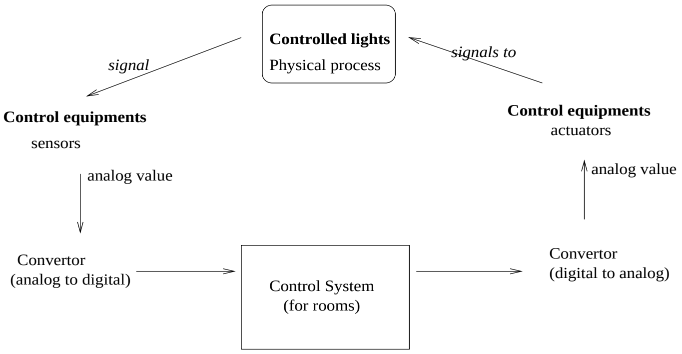

Figure 1: Architecture of process control

Abstraction demands the developer to find the main machines related to the problem (informal specification) to be solved. In the methodological point of view, machines have to be identified to entities or parts of the problem which can be characterized by a state and which evolve by interacting with the environment. Interaction with environment needs the identification of messages or events exchanged between machines. In case of computer-driven system, we have to identify all parts of the problem which match the generic architecture in figure 1. In the specification process presented in the sequel, we use abstraction to distinguish the subsystems which result in the specification of different controlers.
Composition demands the developer to describe the machines in such a way that they can be used or instantiated in different manner, extended, combined and they can communicate with other machines or with their environment. In the sequel composition is used to specify the entire systems as (parallel) composition of various subsystems (office controlers, hallway controlers, etc).

3 The Light Control Problem: requirements and analysis

We assume that the main constituents to drive the physical system presented in the problem description are in place, that means sensors, actuators, convertors from analog to numeric and from numeric to analog. We then consider mainly the software part having as inputs the (numerical) information from sensors via convertors and as outputs (numerical) information addressed to actuators via convertors. Some assumptions will be made for material if necessary.

3.1 Analysis for requirements capture

The approach to capture the essential of the requirements is by trying to build or draw parts of the system which look like the process presented in figure 1. Therefore we have to distinguish, controled equipments, control equipments, and software part. This step of the work should result in the modelisation of some aspects of the system: data abstraction to characterize some parts of the system.
The analysis follows the description given for the constituents of the floor of the building: different kinds of rooms, offices, hallway sections, Computer Labs, Hardware Labs, Peripherals, Meeting Rooms.
The main equipments in the LCP are lights to be controlled. Associated to lights we have actuators and sensors dispatched in different kind of rooms. All rooms are equiped by an important common equipments (lights, control devices), this can lead to the generalisation of some models and some control procedures. From the modelisation point of view, it is clear that some parts are common for modeling different kind of rooms. From the control point of view, there are *common needs* for rooms and *specific needs* for some other rooms. Accordingly, we will have a *common behavior* of control shared with some *room controlers* and *specific behavior* particular to some other rooms controlers. Therefore each (kind of) room will have its own light control system.
Roughly, we abstract a subsystem (as a machine defining a *controler*) from each description where it is possible to have controlled equipments and lights controled. The result is a *room controler*, a *office controler*, a *hallway controler*, etc. The global light control system is the composition of the different control systems of rooms. The result of the analysis can be stated as in figure 2 describing a general *control model* for the rest of the work. In the sequel, we give two machines as the specifications of $RoomControler$ and $OfficeControler$. The latter machine is specified as an extension of the former. The other machines for the control of Hallway section, Computer labs, Hardware Lab and Meeting Rooms are specified in the same manner and we do not list them in the paper.

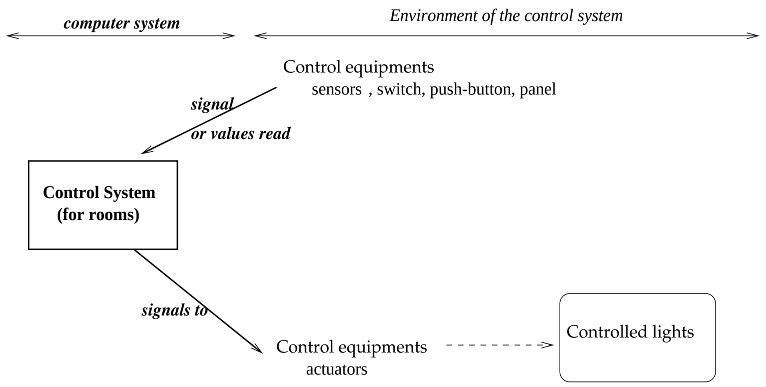

Figure 2: Essential of the control model

3.2 Requirement capture approach

Our requirements capture approach can be summarized by the following steps :

- Use the control model idea to identify the main subsystems
- For each subsystem identified:
  1. Draw the corresponding control model,
  2. Write or complete the Z specification to be used in the data part of this subsystem,
  3. Write a skeleton of the *machine type* corresponding to this subsystem,
  4. Give a complete events alphabet for the machine by checking carefully the components appeared in the drawing obtain in 1,

5. Write the transactions and invariants corresponding to the subsystem needs and complete the skeleton of the machine by the names given to those transactions,

6. Build *Systems* by using and composing *machines types*.

4 Applying the approach to obtain the specification

The specification of the Light Control System follows the steps of the requirements analysis.

4.1 General room control system

Each specific room has its control system with specific control procedures. Here we describe a machine which captures the commonalities of all rooms control procedures and which will be used to specify them. This machine will be the basis for the structuration of the whole system.

Modeling room

We are about to elaborate a model for the control of room. This model will encapsulate the main data for the control of the room and for the information about the room. It gives some elements to be used to describe state space of machines.
The following is the begining of Z specification, which will be used and completed in the rest of the paper. We introduce here some details, from the informal description of the LCP, to comment the specifications.
The given sets are:

$$[DIMVALUE, DOOR, HISTORYITEM]$$

The values used for dimmable lights are if the type $DIMVALUE$. $DOOR$ is used to characterise doors of rooms. $HISTORYITEM$ is used to record all malfunction messages (as items) for diagnosis purpose.

Some type definitions are:

$$CeilLightState ::= On \mid Off \mid Ambient$$

A ceiling light can take three different states(on, off, ambient).

$$BiState ::= On\_1 \mid Off\_0$$

A lot of components of the system can take two different states (on, off), we introduce $BiState$ for this purpose.

$$Occup\_State ::= Occupied \mid UnOccupied$$

In the problem, a room can be occupied (by a person) or not, the type $Occup\_State$ will manage this information.

$$Lighting\_Option ::= Window \mid Wall \mid Both$$

According to the problem description, user of a room can chose a (predefined) light scene for the lights in his or her room. An option is associated to a light scene to indicate the order with which lights are used to achieve a desired light level. This option has one of the three values described by the type introduced $Lighting\_Option$.

$$SettingMode ::= direct\_mode \mid lscene\_mode$$

The problem description states that the ceiling light groups can be set directly or by using a light scene. The control system can determine via the control panel which offers this possibility, how the ceiling light groups are set.

$$KindRoom ::= kRoom \mid kOffice \mid kHallway \mid kHardLab \mid kCompLab \mid kPerRoom \mid kMeetRoom$$

$$DoorState ::= Closed \mid Open$$

The rooms have doors which can be closed or open.

Now we give some abbreviations using the problem description words in order to make the specification more readable:

$$LightState == BiState$$

Some equipements indicate the lights status.

$$Date == \mathbb{N}$$

We need date to handle some time constraints.

$$Delay == \mathbb{N}$$

Time constraints are exposed in the requirements: (U3) reestablish a chosen light scene if a person reoccupies a room that he lives after less than T1 minutes, (FM3) put room ceiling light groups off after a T3 minutes, if it the room is unoccupied.

$$LightLevel == \mathbb{N}$$

A light level is an illumination (amount of light measured in lux) in a room.

$$Light == \mathbb{N}$$

A number is associated to each light in order to distinguish it among the other.

$$StatusLine == BiState$$

It indicates the status of a device.

$$IMD\_State == BiState$$

Motion detector (IMD) has a status (on / off).

$$PB\_State == BiState$$

Push button (PB) has a status.

$$MotionDetector == \mathbb{N}$$

Each motion detector has a unique number.

$$OLS == \mathbb{N}$$

The building is equiped with several outdoor light sensors (OLS). Each outdoor light sensor has a unique number.

$$CeilingLight == \mathbb{N}$$

Each ceiling light has a unique number.

$$DimmerActuator == \mathbb{N}$$

Each dimmer actuator has a unique number.

$$Illumination == \mathbb{N}$$

The illumination is the amount (in lux) of light failing on a surface.

$$PushButton == \mathbb{N}$$

Each push button has a unique number.

$$LsName == \mathbb{N}$$

There are predefined light scenes to set the lights. We associate a unique number to each light scene.
We introduce now a schema to model a light scene. According to the requirements [Iss99] a light scene has a name, a desired ambient light, an option to achieve the desired ambient light.

$$\begin{array}{|l}
\hline
\textit{LightScene} \\
lsname : LsName \\
amb_illum : Illumination \\
lsc_option : Lighting_Option \\
\hline
\end{array}$$

The schema $LightScene$ is the modelisation of light scene.
Now we introduce some axiomatic definitions:

$$\begin{array}{|l}
getclock : Date \\
computeAmbLight : Illumination \rightarrow\!\!\!\!\!\!+\; LightLevel \\
computeLLvl : LightScene \times LightLevel \times KindRoom \\
\qquad\qquad \times DoorState \rightarrow\!\!\!\!\!\!+\; \mathrm{seq}\, DIMVALUE \\
\hline
\end{array}$$

We use a discrete approach of time handling. We assume the existence of a global clock which can deliver a (time) value by a function $getclock$. We assume that the function $getclock$ can be implemented in a given implementation environment, to provide a global clock for system development. We also assume that the function $computeAmbLight$ is used to compute the ambient light in a room according to the illumination given by outdoor sensor. The function $computeLLvl$ has to be developed as a math formula, to ensure the calculation of light status and dimmer settings. This definition is deferred.

$$\begin{array}{|l}
theOffices : \mathbb{F}\mathbb{N} \\
theHallws : \mathbb{F}\mathbb{N} \\
theOls : \mathbb{F}OLS \\
theStaircs : \mathbb{F}\mathbb{N} \\
md_t, cl_t : Date \\
limit_ols_val : \mathbb{N} \\
defaultls : LsName \\
Very_High_Illum : Illumination \\
ols1, ols2, ols3, ols4 : OLS \\
\hline
md_t > 0 \quad \wedge \quad cl_t > 0 \\
defaultls = 0 \\
Very_High_Illum = 100 \\
ols1 = 1; ols2 = 2; ols3 = 3; ols4 = 4
\end{array}$$

$theOffices$, $theHallws$, $theOls$, $theStaircs$ stand for all the offices in the system, all hallways, all outdoor light sensors, all staircases respectively. md_t and cl_t are global variables used as time parameters. md_t is used to periodically check motion detector. cl_t is used to check ceiling light (direct) manipulation. md_t, cl_t can be set for the system initialisation. $limit\_ols\_val$ is used as a limit value to detect a malfunction of an outdoor light sensor. Indeed the requirement is, NF1:*in case of malfunction of an outdoor light sensor, it always deliver the last correct value*; then we make the assumption that if the same value is received $limit\_ols\_val$ times then there is a malfunction. $defaultls$ represents the default light scene. $Very\_High\_Illum$ is a hight value used with the setting option *both* of a light scene. The requirement is that, in case of malfunction the lights are completely on with the setting option $BOTH$. $ols1, ols2, ols3, ols4$ give the number associated to each of the outdoor light sensors.

We introduce now some type definitions to be used in the specification of some particular messages appearing during the control of the system.

$$OLS\_MESG \quad ::= MALFUNCTION\_ols \mid OK\_ols$$

$$
\begin{aligned}
&MDetector\_MESG &&::= MALFUNCTION\_md \mid OK\_md\\
&Input\_Value\_MESG &&::= Unreasonable\_inputs \mid OK\_invalue\\
&Actuator\_MESG &&::= MALFUNCTION\_Actuator \mid OK\_actuat\\
&Sensor\_MESG &&::= MALFUNCTION\_Sensor \mid OK\_sensor
\end{aligned}
$$

$$
\begin{array}{|l}
item\_o : OLS\_MESG \rightarrow HISTORYITEM\\
item\_d : MDetector\_MESG \rightarrow HISTORYITEM\\
item\_iv : Input\_Value\_MESG \rightarrow HISTORYITEM\\
item\_a : Actuator\_MESG \rightarrow HISTORYITEM\\
item\_s : Sensor\_MESG \rightarrow HISTORYITEM
\end{array}
$$

We mainly give two values (OK, MALFUNCTION) for the different types. Of course these types can be completed by other predefined messages. The functions to $HISTORYITEM$ are necessary to build the item to be recorded for each event.

We introduce now some schemas to model some parts of the system.

$$
\begin{array}{|l}
\hline
\textit{RoomControlPanelIndicator}\\
unreason\_input : Input\_Value\_MESG\\
ols : OLS\_MESG\\
actuator : Actuator\_MESG\\
\hline
\end{array}
$$

The schema $RoomControlPanelIndicator$ describes the indicators for each room control panel. For example each control panel has an indicator to issue warnings concerning unreasonable inputs.

$$
\begin{array}{|l}
\hline
\textit{FMControlPanel}\\
RoomControlPanelIndicator\\
roomOls : \mathbb{N} \rightarrow\!\!\!\!\!+ OLS\_MESG\\
roomActuator : \mathbb{N} \rightarrow\!\!\!\!\!+ Actuator\_MESG\\
roomMD : \mathbb{N} \rightarrow\!\!\!\!\!+ MDetector\_MESG\\
hwOls : \mathbb{N} \rightarrow\!\!\!\!\!+ OLS\_MESG\\
hwActuator : \mathbb{N} \rightarrow\!\!\!\!\!+ Actuator\_MESG\\
staircaseSensor : \mathbb{N} \rightarrow\!\!\!\!\!+ Sensor\_MESG\\
\hline
\operatorname{dom} roomOls = thOffices\\
\operatorname{dom} roomActuator = theOffices\\
\operatorname{dom} roomMD = theOffices\\
\operatorname{dom} hwOls = theHallws\\
\operatorname{dom} hwActuator = theHallws\\
\operatorname{dom} staircaseSensor = theStaircs\\
\hline
\end{array}
$$

This schema describes the indicators used for the facility manager (FM) control panel. This panel has indicators about other rooms (each room, each hallway, each staircase), for this reason we introduce partial functions to indicates the value associated to given rooms.

The following schema models rooms. The schema $Room$ is built with the intention to use it in other specific room modeling.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Room \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
lscene : LightScene\\
dftlscene : LightScene\\
lastlscene : LightScene\\
lightlevel : LightLevel\\
ceilstate : CeilingLight \pfun CeilLightState\\
imdstate : MotionDetector \pfun IMD\_State\\
amblightlvl : LightLevel\\
std\_amblightlvl : LightLevel\\
occupation : Occup\_State\\
clg\_setting : SettingMode\\
ls\_maintain\_delay : Delay\\
light\_off\_delay : Delay\\
userin\_t : Date\\
userout\_t : Date\\
ols\_sensor : OLS\\
ols\_value : Illumination\\
nb\_odlval : \mathbb{N}\\
CP\_indicators : RoomControlPanelIndicator\\
mds : \mathbb{F}\, MotionDetector\\
clgps : \mathbb{F}\, CeilingLight\\
clgpshb : PushButton \rightarrow CeilingLight\\
dimacts : \mathbb{F}\, DimmerActuator\\
pushbs : \mathbb{F}\, PushButton\\
sllstate : \mathbb{N} \pfun LightState\\
mdstate : MotionDetector \pfun LightState\\
doorState : DoorState\\
slls : \mathbb{F}\, \mathbb{N}\\
kindRoom : KindRoom\\
hallwdoors : \mathbb{F}\, DOOR\\
\hline
dftlscene.lsname = defaultls\\
\mathrm{dom}\, imdstate = mds\\
\mathrm{dom}\ ceilstate = clgps\\
\mathrm{dom}\ clgpshb = pushbs\\
\mathrm{ran}\ clgpshb = clgps\\
\mathrm{dom}\ sllstate = slls\\
\mathrm{dom}\ mdstate = mds\\
ols\_value \in \mathrm{dom}\, computeAmbLight\\
amblightlvl = computeAmbLight(ols\_value)\\
\hline
\end{array}
$$

Each specific room model will constrain the schema *Room* by including it and defining a predicate part which will characterize more precisely the room as needed.

4.2 Specifying a basic room controler

According to the commonalities of the different room controlers, we have to specify a common basic room controler which gathers the main data and behaviors. The specific controlers for offices, hallway sections, etc will be extensions of the basic room controler. We define a machine type called *RoomControler*. This type provides the main behavior rules of room controlers. Its data model is based on the schema *Room* which is the model of the common part of all rooms.

MachineType RoomControler [$I\!N$]
DATAMODEL
$State \mathrel{\widehat{=}} Room$

$$\begin{array}{|l}
\hline
\textit{InitState} \\
State' \\
\hline
\#hallwdoors' = 1 \\
lscene' = dftlscene' \\
\ldots \\
\hline
\end{array}$$

BEHAVIOR
{set_lscene_mode, set_direct_mode, Inv_Safe_Illimination, set_odl, set_default_lights, mdf_lscene, upd_ls_maintain_delay, upd_light_off_delay,cl_on, cl_off, Inv_chosen_lscene, upd_imdstate, user_in, user_out, Inv_Light_off, upd_ceilstate, modify_clight }
INITCONFIG

$$\begin{array}{|l|}
State \vdash nb_odlval = 0 \\
\quad kindRoom = kRoom \\
\quad lscene = dftlscene \\
\quad CP_indicators.ols = OK \\
\quad CP_indicators.actuator = OK \\
\quad lr_event = dummy_event \\
\quad ls_events = \{\} \\
\quad \ldots
\end{array}$$

COMMCSTRAINTS
every cl_t * * * * * upd_ceilstate *
every md_t * * * * * upd_imdstate *
End RoomControler

That is the syntax for expressing machine types. It introduces the machine name, possibly some clauses for the use of other machines or for the composition of other machines. The BEHAVIOR clause introduces the list of all action names labelling transactions of this machine; the descriptions of transactions themselves are deferred. A INITCONFIG clause introduces the initial configuration of the machine. Two particular variables are employed in the description of configuration in order to manage communication and the associated constraints: lr_event (last received event), ls_events (last sent events). Thus, each event received or sent is registered in such a way that communication constraint (clause COMMCSTRAINTS) can be defined; for example some sequences of events can be forbidden.

The COMMCSTRAINTS clause is used to describe communication constraints for example causality relation between action, time constraints, priorities. Here a time constraints is used with the statement **every**. This statement introduces a *constrained transaction* which is enabled periodically every thousandth of second, hundredth of second, ..., month. A constrained transaction has a initial configuration restricted to (the specific predicate) **constrained** described as *true* (however constrained transactions are enabled only with the statement **every**). It is the case for the transaction $upd\_ceilstate$ and $upd\_imdstate$ in the sequel.

4.2.1 Common events alphabet for rooms control

The behavior of machines are strongly related to their events alphabet. The transactions which will express the behavior of the system and by the way the capture of the user needs, are based on the treatment of events exchanged between machines to achieve the user needs. Here we describe the events and their associated types (according to the data model introduced).

In the following tables we list the events received or sent by the equipment indicated in the first column.

4.2.1.1 Events received from OLS, control panel(CP), FM (Facility Manager)

Equipments	**events list and typing**
Outdoor light sensor	*odl_val* ($\mathbb{N}$; *Illumination*)
light scene device (CP)	*set_lscene (LightScene)*
Ceiling light switch device (CP)	*switch_cl_on*($\mathbb{N}$), *switch_cl_off*($\mathbb{N}$)
T1 setting device (CP)	*set_ls_maintain_delay*($\mathbb{N}$)
Mode modification device (CP)	*switch_lscene_mode* *switch_direct_mode*
FM Control Panel	*set_light_off_delay*($\mathbb{N}$)

The idea for all these events is as the following (considering the second line) : from the control panel (CP), the control system receives the event *set_lscene* with a parameter of type *LightScene*. Furthermore, we implicitely have $set\_lscene : LightScene \rightarrow Event$.

4.2.1.2 Events received from ceiling light groups

Equipment	events list and typing
ceiling light group	*set_ceilstate(CeilLightState)*

The motion detector does not send events to the controler. It has ([Iss99] page 8) a state (`1/0` as described in the requirements) which is read by the office controler.

4.2.1.3 Events sent to Facility Manager via its control panel

Equipment	events list and typing
Control Panel(FM)	*malfunc_sensor(OLS)*

4.2.2 Transactions defining the behavior of room controler

The requirement (U9) indicates: *The room control panel for an office should contain at least (i) a possibility to set each ceiling light group, (ii) a possibility to set the chosen and the default light scene, (iii) a possibility to set T1.* Together with the user need U9, we consider a further possibility to set the modification mode of ceiling light groups (see [Iss99]): they can be set by group (*light scene mode*) or directly (*direct mode*) using U9(i). Therefore we consider the two modes: *lscene_mode, direct_mode* already introduced in the data part. The room attribute used for this is *clg_setting*. The mode can be switched from the room control panel by adressing a message *switch_lscene_mode* or *switch_direct_mode* to the controler.
As transactions are deferred and then separated from the machine type, the machine name is used as a header of each transaction, a parameter (*ii*) is also used to identify the corresponding machine. This parameter helps to indicate the source or target of events.

4.2.2.1 Setting modification mode

MachineType: *RoomControler(ii)*

$$\left.\frac{\begin{array}{l} State \vdash clg\_setting = direct\_mode \\ \qquad \xleftarrow{switch\_lscene\_mode} \end{array}}{\begin{array}{l} State \vdash \\ \qquad lr\_event = switch\_lscene\_mode \\ \qquad clg\_setting = lscene\_mode \end{array}}\right| \; [\; set\_lscene\_mode \;]$$

Note that, $\xleftarrow{switch\_lscene\_mode}$ is a predicate whereas *switch_lscene_mode* in the RHS is used as an expression. The guard which enables the behavior specified by this transaction is that the setting mode (*clg_setting*) has the value *direct_mode* and the control system receives the event *switch_lscene_mode*. In this case, the system evolves into the final configuration described. The *lr_event* is recorded. The *ls_events* variable does not change because there is no event sent in this behavior.

For the rest of the transactions, we do not explicitly write the modification of the variable *ls_events* and *lr_event*. A general invariant enables to easily manage this:
MachineType: *All*

$$\frac{State \vdash \quad \xleftarrow{event\_name}}{\begin{array}{l} State \vdash \\ \quad lr\_event = event\_name \end{array}} \; [\,invariant\,]$$

The transaction associated to the event *set_direct_mode* is specified in the same way. We do not give it here.

4.2.2.2 An invariant for safe illumination in an occupied room

A paragraph of the requirements said (U1) : *If a person occupies a room, there has to be safe illumination, if nothing else is desired by the chosen light scene.*
To exploit U1, we have to set an invariant for the system. Invariants are described just like transactions, their names have the prefix *Inv_*.
MachineType: *RoomControler(ii)*

$$\frac{\begin{array}{l} State \vdash occupation = occupied \wedge \\ \quad lscene.lsname = defaultls \end{array}}{State \vdash lightlevel \geq 14} \; [\,Inv\_Safe\_Illumination\,]$$

4.2.2.3 Dealing with Outdoor light sensor and its malfunction

We use here a counter, to detect the repeated reception of the same value sent by the outdoor light sensor. If the value sent by the sensor is not the last value, then the counter value is reset.
MachineType: *RoomControler(ii)*

$$\frac{\begin{array}{l} State \vdash ols\_value \neq vv \\ \xleftarrow{odl\_val(ii,\ vv)} \end{array}}{\begin{array}{l} State \vdash \\ \quad nb\_odlval = 1 \ \wedge \ ols\_value = vv \end{array}} \; [\,set\_odl\,]$$

Requirement NF1: *In case of malfunction of the outdoor light sensor, the sensor always sends the last (correct) value.*

MachineType: *RoomControler(ii)*

$$\frac{\begin{array}{l} State \vdash ols\_value = vv \\ \xleftarrow{odl\_val(ii,\ vv)} \end{array}}{\begin{array}{l} State \vdash \\ \quad nb\_odlval = nb\_odlval + 1 \end{array}} \; [\,set\_odl\,]$$

Note that two transactions (with different guards) are given for this event. The interpretation is simply an alternative.

Detecting malfunction of an outdoor light sensor

The two following paragraphs of the requirements indicate the needs for malfunction:
NF2: *if any outdoor light sensor does not work correctly, the default light scene for all rooms is that all ceiling light groups are on.*
NF3: *If any outdoor light sensor does not work correctly and a hallway section is occupied, the ceiling light group in this hallway section has to be on.*
According to the assumption we made for detecting malfunction, the behavior of the control system in this case is specified as:
MachineType: *RoomControler(ii)*

$$
\begin{array}{|l|l}
State \vdash nb\_odlval \geq limit\_ols\_val & \\
\hline
State \vdash & [\ set\_default\_lights\ ] \\
\quad lscene.amb\_illum = Very\_High\_Illum & \\
\quad lscene.lsc\_option = BOTH & \\
\quad dimvalues = computeLLvl(lscene, amblightlvl, & \\
\qquad\qquad kindRoom, doorState) & \\
\quad CP\_indicator.ols = MALFUNCTION & \\
\quad \forall\ cli \in clgs \bullet ceilstate(cli) = On & \\
\quad \forall\ cli\ \in\ clgs \bullet \xrightarrow{set\_cldimval(cli,100)} & \\
\quad \forall\ cli\ \in\ clgs \bullet \xrightarrow{pulse(cli)} & \\
\quad \xrightarrow{malfunc\_sensor(ols\_sensor)} &
\end{array}
$$

We assume that the detection happens according to the value of the global parameter $limit\_ols\_val$, given by the Facility Manager and defining the limit of the reception of the same value. In case of the detection, the users of the rooms are informed ($CP\_indicator.ols$) and the lights has to be on ($Very\_High\_Illum$ with the option $BOTH$). The transaction includes the behavioral rule indicated in NF3, because, whatever is the occupation of the room, the transaction will work. Events are sent to ceiling light group to update their status according to the defined behavior. Here the variable ls_events is (implicitly) modified.

4.2.2.4 Modifying light scene and then changing dim value of ceiling lights

Requirement U9(ii) indicates that the light scenes can be determined by using control panel. The idea is that the controler computes the dim values corresponding to each light with respect to a given light scene. This computation should take place whenever the light scene changes.
The ceiling lights are maintained by the control system depending on different light scenes. So, when light scenes change, the controler receives a message from control panel and sends the new values to one of the dimmer-actuators (named lle1, lle2) or to both.
Each ceiling light group is possibly dimmed individually by an actuator. The value used by the actuator comes from the control system and is set by using the control panel.
MachineType: *RoomControler(ii)*

$$
\begin{array}{|l|l}
\xleftarrow{set\_lscene(newls)} & \\
\hline
State \vdash & [\ mdf\_lscene\ ] \\
\quad lscene = newls & \\
\quad dimvalues = computeLLvl(lscene, amblightlvl, & \\
\qquad\qquad kindRoom, doorState) & \\
\quad lightlevel = lscene.amb\_illum & \\
\quad \forall\ cli\ \in\ clgs \bullet \xrightarrow{set\_cldimval(cli,dimvalues(cli))} & \\
\quad \forall\ cli\ \in\ clgs \bullet \xrightarrow{pulse(cli)} &
\end{array}
$$

The controler modifies the dimmer value of the corresponding ceiling light according to the light scene needed (and described with the control panel) and toggles the lights through the actuators. Actuators receive (via the line *dim value*) their dimmer values from the controler. Consequently, the control system (room controler here) will receive from the ceiling light group, the state (*on/off*) of the ceiling light state.

4.2.2.5 *Modifying the parameter* `T1`

`T1` is a delay used to maintain a chosen light scene in a room. It is set by an user.
MachineType: *RoomControler(ii)*

$$\left|\begin{array}{l} \xleftarrow{set\_ls\_maintain\_delay(t1v)} \\ \hline State \vdash \\ \quad ls\_maintain\_delay \;=\; t1v \end{array}\right| [\; upd\_ls\_maintain\_delay \;]$$

The modification of the parameters `T2, T3` (delays for the extinction of light in unoccupied hallway section and room respectively) is specified in the same way (a transaction labelled by $upd\_light\_off\_delay$).

4.2.2.6 *Modifying a ceiling light from the control panel*

Exploiting light scenes

Requirement U5: *For each room, the chosen light scene can be set by using the room control panel.*
From the requirement U5 (and the definition of light scene), we know that there are predefined light scenes which are settings of the light levels and options which should be used to achieve them. Each light has a dim value. The control system computes (function *computeLLvl*), and gives according to the current light scene (U10 [Iss99]), the appropriate value for each ceiling light group and then achieves the desired light scene. Requirement U10 saids that the ceiling lights should be maintained by the control system.

4.2.2.7 *Changing a ceiling light to on from the control panel*

MachineType: *RoomControler(ii)*

$$\left|\begin{array}{l} State \vdash clg\_setting = direct\_mode \\ \quad \exists\, cli \in dom\ dimvalues \;\mid \\ \quad \xleftarrow{switch\_clg\_on(cli)} \\ \hline State \vdash \\ \quad dimvalues(cli) = 100 \wedge \\ \quad ceilstate(cli) = on \\ \quad \xrightarrow{set\_cldimval(cli,dimvalues(cli))} \\ \quad \xrightarrow{pulse(cli)} \end{array}\right| [\; cl\_on \;]$$

Changing ceiling light state to *on* implies the modification of the dim value of the light group. To maintain the chosen light scene (U2), the controler has to compute new values for lights. Messages containing the new values are sent to actuators and then the lights are toggled.
To change the ceiling light to *off*, the specification of the transaction (labelled by cl_off) is obtained with the same manner. Changing to *off* implies the conservation of the dim value of the light group.

4.2.2.8 *Maintaining chosen light scene*

MachineType: *RoomControler(ii)*

$$\left|\begin{array}{l} occupation = occupied \\ \hline lightlevel = lscene.amb\_illum \end{array}\right| [\; Inv\_chosen\_lscene \;]$$

This invariant ensures that at any moment, the light level in a room ($lscene$) is the same as the chosen light scene or the default light scene which is known through $lscene$.

4.2.2.9 Control of motion detector

Motion detectors are passive, their states are read by the controler which accordingly modifies the *occupation state* of the room. We assume the existence of an independent procedure (*scrutinize_md*) connected to the motion detector and which can periodically scrutinize the status of the sensor in order to update for example registers (*mdstate*) read by the controler. The simplest way is to get as is the content of the registers as the states manipulated by the controler. This can be refined properly. We capture this assumption by introducing a constrained transaction which is enabled every *md_t* unit of time.

MachineType: *RoomControler(ii)*

$$\begin{array}{|l|l}
\textbf{constrained} & \\
\hline
State \vdash & [\ upd_imdstate\ ] \\
\quad \forall\, i \in dom\ imdstate \bullet imdstate(i) = mdstatus(i) &
\end{array}$$

If the motion detector state becomes *on* when the room is not occupied, then the room becomes occupied. The *global time* is recorded in order to verify the constraint on `T1` (U3, U4).

MachineType: *RoomControler(ii)*

$$\begin{array}{|l|l}
State \vdash & \\
\quad \exists\, imd \in dom\ imdstate \mid & \\
\quad imdstate(imd) = on \ \wedge occupation = unoccupied & \\
\quad t_i = getclock & \\
\hline
State \vdash occupation = occupied & [\ user_in\ ] \\
\quad inuser_t = t_i & \\
\quad (inuser_t - outuser_t) \geq ls_maintain_delay \Rightarrow & \\
\qquad lscene = lastlscene & \\
\quad \wedge\ (inuser_t - outuser_t) < ls_maintain_delay \Rightarrow & \\
\qquad lscene = dftlscene & \\
\quad dimvalues = computeLLvl(lscene, amblightlvl, kindRoom, & \\
\qquad\qquad doorState) & \\
\quad lightlevel = lscene.amb_illum & \\
\quad \forall\, cli \in clgps \bullet \xrightarrow{set_cldimval(cli, dimvalues(cli))} & \\
\quad \forall\, cli \in clgps \bullet \xrightarrow{pulse(cli)} &
\end{array}$$

We consider that if there is no more movement in the room, it is *unoccupied*. If the motion detector state becomes *off* when the room state is *occupied*, then the room becomes *unoccupied*. The global time is recorded in order to compare the difference with next time an user will enter the room with `T1`.

MachineType: *RoomControler(ii)*

$$\begin{array}{|l|l}
State \vdash \forall\, imd \in dom\ imdstate \bullet & \\
\qquad imdstate(imd) = off \ \wedge occupation = occupied & \\
\quad t_i = getclock & \\
\hline
State \vdash occupation = unoccupied & [\ user_out\ ] \\
\quad outuser_t = t_i & \\
\quad lastlscene = lscene &
\end{array}$$

4.2.2.10 Dealing with facility manager needs (FM3)

According to FM3, the (room) controler must have a possibility to switch off all lights in the office. A part of this requirement is captured by an invariant. The other part is captured by a facility manager action.

MachineType: *RoomControler(ii)*

$$\begin{array}{|l|}
State \vdash \\
\quad \forall\, cli \in clgs \bullet ceilstate(cli) = off \\
\quad \forall\, imd \in \mathrm{dom}\ imdstate \bullet imdstate(imd) = off \\
\quad occupation = unoccupied \\
\quad t_i = getclock \wedge \\
\quad\quad (t_i - outuser_t) \geq light_off_delay \\
\hline
State \vdash \\
\quad \forall\, cli \in \mathrm{dom}\ ceilstate \bullet ceilstate(cli) = off \\
\quad \forall\, cli \in clgs \bullet \xrightarrow{pulse(cli)}
\end{array}\ [\ Inv_Light_off\ ]$$

The light are put *off* without modifying their dim values.

4.2.2.11 Managing the state of lights according to direct manipulations

The control system should be informed of all manual settings. To deal with this requirement, the control system should check periodically, the state of sensors (status lines) and modify accordingly its knowlegde about the controled lights. We assume that independently from the control system, it can exist a procedure which is connected to the sensor and which stores in a register for example, the state of the sensor. Therefore, in the specification we have to ensure that the system periodically reads this register, computes the corresponding status (*on/off*) and puts it in its variable *ceilstate*. In the simplest case, we can assume that the value read is exactly the status.
MachineType: *RoomControler(ii)*

$$\begin{array}{|l|}
\textbf{constrained} \\
\hline
State \vdash \\
\quad \forall\, i \in dom\ sllstate \bullet \\
\quad\quad ceilstate(i) = sllstate(i)
\end{array}\ [\ upd_ceilstate\ ]$$

4.2.2.12 Managing Manual setting of lights

Control of the state of ceiling light group

The control system receives an event from ceiling light group for the notification of the state change under push buttons action. The office controler knows about the state of the ceiling light group by *listening* to the status line of the corresponding dimmable light. Indeed this actuator sends the state of the light to the system [see 8:P19]. For example when, in an office, the push-buttons *pb1* and *pb2* are used to change ceiling lights, the office controler should change the state of the corresponding light when it receives the event *set_ceilstate*. As the push-buttons are not connected to the control system, it is the ceiling light which communicates via *status line*, its status (*on, off*) to the control system. We introduce only one transaction for this requirement assuming that each ceiling light group uses its own *channel* for communication with the control system; it is the justification of the use of *clg* to indicate which ceiling light group.
MachineType: *RoomControler(ii)*

$$\begin{array}{|l|}
\xleftarrow{set_ceilstate(clg,\ vv)} \\
\hline
State \vdash \\
\quad ceilstate(clg) = vv
\end{array}\ [\ modify_clight\ ]$$

That ends the transactions description for the machine type *RoomControler*. In the following, we use this machine type to describe office controler.

4.3 Specifying an office control system

The office controler system is connected on the one hand to lights (ceiling light groups) via actuators (dimmer-actuators) and on the other hand to control equipments: sensors (motion-detector), push-buttons and control-panel which are the sources of events or actions to control the lights; these connections made the environment of the control system. Then the controler, as a machine in our specification, receives and sends events to its environment. The appropriate reactions to these events define its behavior.
The office controler is specified as a machine type named *OfficeControler*. For this purpose, we describe the data model and we list all the events this machine will receive/send from/to his environment. This list of events is elaborated according to the equipment connected to the controler and the lights controled.

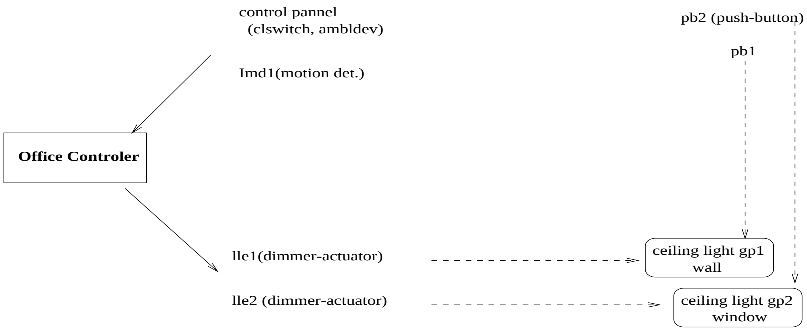

Figure 3: Office controler and its environment

4.3.1 The data model of office controler

To model the data part of the office controler we use the room data model (inclusion of the Z schema *Room*) and add information specific to office: the dim value of each ceiling light, modeled by a partial function from Light to dim values.

$$\begin{array}{|l}
\multicolumn{1}{l}{Office} \\
\hline
Room \\
dimvalues : CeilingLight \pfun DIMVALUE \\
\hline
\#mds = 1 \\
\#clgps = 2 \\
\#slls = \#clgps \\
\mathrm{dom}\, dimvalues = clgps \\
\#hallwdoors = 1 \\
\#dimacts = 2 \\
\#pushbs = 2 \\
\hline
\end{array}$$

The schema $Office$ captures the information for the description of an office. The predicate part of the schema constraints the declaration to precise that there is one motion detector ($\#mds = 1$), two ceiling light groups ($\#clgps = 2$), etc.

4.3.2 Events alphabet for the control

4.3.2.1 Received events

These events come from sensors (motion detector) and control panel. In the case of passive sensors, we will read their states instead of receiving events from them. From control panel we have all events sent to room controler:

4.3.2.2 Sent events

Events sent to ceiling light groups (via actuators) by the office controler are: *pulse, ocs_active, set_cldimval*

– to actuators (lle1, lle2)

Equipment	events list
dimmer-actuators (lle1,lle2)	*set_cldimval($I\!N$; DIMVALUE), pulse, ocs_active*

Now we can build the machine type *OfficeController*.

4.3.3 The office controler

MachineType OfficeControler[$I\!N$]
EXTENDS RoomControler
DATAMODEL
$State \mathrel{\widehat=} Office$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;InitState\;}}\\
State'\\
\hline
kindRoom' = kOffice\\
\ldots\\
\hline
\end{array}$$

BEHAVIOR
{set_lscene_mode, set_direct_mode, Inv_Safe_Illimination, set_odl, set_default_lights, mdf_lscene, upd_ls_maintain_delay, upd_light_off_delay,cl_on, cl_off, Inv_chosen_lscene, upd_imdstate, user_in, user_out, Inv_Light_off, upd_ceilstate, modify_clight, send_ocs_state }
INITCONFIG

$$\left|\; State \vdash kindRoom = kOffice \;\right|$$

COMMCSTRAINTS
Every 60 * * * * * *send_ocs_state* *
End OfficeControler

The *OfficeControler* is an extension of the *RoomControler* (clause EXTENDS). This clause of the specification approach enables one to inherit the description and the behavior of an already defined machine type (here *RoomControler*). Note that in this case the data model (schema $Office$) is an extension (by Z schema inclusion) of the data model (schema *Room*) of the machine extended (*RoomControler*).
The behavior clause lists the transactions (of the extended machine) used for the new machine and completes it, here we have in addition, the transaction labelled by *end_ocs_state*. This transaction defines more precisely the behavior of the *OfficeControler*; it is a constrained transaction used to inform that the control system is active. The INITCONFIG and COMMCSTRAINTS clauses are also completed. These clauses appear in the extension only if there is an update.
The office controler sends, every 60s, a signal "*control system is active*" to each dimmable light.
MachineType: *OfficeControler(ii)*

$$\left|\begin{array}{l}
\textbf{constrained}\\
\\
\hline
State \vdash\\
\quad \forall\, i \in \operatorname{dom} ceilstate \bullet \xrightarrow{ocs_active(i)}
\end{array}\right| [\; send_ocs_state \;]$$

The constraint is expressed in the machine and states that *every 60s this transaction is enabled.*

5 Specification of the Light Control System

We have already specified the subsystems of the light control system. Now we have to combine these subsystems to build the entire control system. There are 22 rooms and 3 hallways in the building which will be controlled. Among the 22 rooms there are 14 Offices (with numbers 415, 416, 429, etc), 1 meeting room, 3 computer labs, 2 Hardware labs and 2 peripheral rooms. The peripheral rooms are not controlled by the system. We have specified the machine types for the control of each kind of rooms. Now we have to use the machine types appropriately and compose them. Configuration Machines formalism provides means to use and compose machine types. The SYSTEM construct enables the use of a machine type to specify a subsystem with particular properties. To specify one of the 14 office controlers we have to express:

System *OfCtrl429* $\widehat{=}$ OfficeControler(429)
INITIALISATION

$$\begin{array}{|l|}
State \vdash ols_sensor = ols1 \\
\cdots
\end{array}$$

EndSystem

Note the construct **System** *OfCtrl429* $\widehat{=}$ OfficeControler(429) used to simply define an instance of $OfficeControler$. The number 429 is one of the number used for offices in the problem description. Thus we can define, using the same machine type, several (numbered) office controlers as (sub) systems. The INITIALISATION clause (specific to each subsystem) completes the INITCONFIG clause (general to all machine of the same type) in order to make more precise the configuration of specific machines. Typically, here we will have 14 different office controlers "running" in parallel and, for example, they have different number (429, 430, etc) and do not use the same outdoor light sensor. The other clauses we can use in the SYSTEM construct are: INVARIANT, PROPERTIES, CONSTRAINTS. They are used to follow the study of the specification: verification of properties, simulation.
In the same way we can build controlers for the control of hallway sections and finally the general control system is a parallel composition (noted ||) of the subsystems.

System *Build32Ctrl*

$$\begin{array}{ll}
OfCtrl429 & \| \\
OfCtrl432 & \| \\
HwCtrl1 & \| \\
HwCtrl2 & \| \\
\cdots &
\end{array}$$

INITIALISATION

$$\begin{array}{|l|}
md_t = \cdots \\
cl_t = \cdots \\
limit_ols_val = \cdots \\
\cdots
\end{array}$$

EndSystem

6 Formal analysis : Z/EVES and Promela/Spin

The formal specification is a starting point for formal analysis. By formal analysis we mean reasoning using well formally defined techniques, for examples proving desired properties, animation, refinement.
Given a formal method, either a specific environment (set of tools) is developed for it, or a bridge is established between this formal method and other well known environment. We use the second approach here. As we use an integrated approach combining data as well as behavior in the specification, we have to use a combination of tools to make the formal analysis. That justifies the use of *Z/Eves* [Saa97] and *Promela/Spin* [Hol91].
We do not undertake here a complete analysis of the light control problem. However we show how formal analysis can be conducted from our approach of specification. First of all the specification itself should be correct. This part

is achieved especially by using the $Z/Eves$ environment. Then the specification is validated again the requirement (functional aspect).
A kind of validation is for example to establish from the specification that (U1) *if a room is occupied then there is a safe illumination if nothing else is desired by the chosen light.*
We also indicate how an animation of the specification can be done by translating it into Promela specification (as interacting processes).

6.1 Using $Z/Eves$ for the data part

The data part of the specification is based on Z specification. Thus the $Z/Eves$ environment is used to check it. The specification has been imported in the $Z/Eves$ tool, and then all the Z parts are checked and proved correct (consistency).

Proving properties

Consider the validation of U1 as a property: *If a room is occupied then there is a safe illumination if nothing else is desired by the chosen light.*

There are two (disjunctive) parts in this property. We have to prove that both are established.
For the first part, we have to prove that *if a person occupies a room with a default light scene then they should be a safe illumination.*
Proof:
i) From the initialisation (InitConfig) of $RoomControler$ we have $State \vdash lscene = dftlscene$.
ii) Thus $State \vdash dftlscene.lsname = defaults$ (from $Room \vdash dftlscene.lsname = defaults$)
iii) If a person enters a room, the transaction labelled $user\_in$ is enabled, then we have $State \vdash occupation = occupied$
iv) Now consider *ii)* and *iii)* and invoke the invariant $Inv\_Safe\_Illumination$,
we get $State \vdash lightlevel \geq 14$ (which describes safe illumination)
□
For the second part, we have to prove that, if a person occupies a room with a chosen light scene the illumination is that corresponding to the chosen light scene.
Proof:
i) If a person enters a room, the transaction labelled $user\_in$ is enabled, then we have $State \vdash occupation = occupied$

ii) If the default light scene is modified by a new light scene ($newls$), then we have (by enabling the transaction labelled by mdf_lscene):
$State \vdash lightlevel = lscene = newls \wedge lightlevel = lscene.amb\_illum$
thus we have the illumination chosen by the user.
□
Therefore, combining the transactions (invariants and others) enable to prove easily some properties.

6.2 Translating transactions to Promela processes

Our approach enables to animate machines behavior by translating them into promela processes, completing the processes and doing formal analysis on the promela processes.
The language PROMELA [Hol91] is closely related to the guarded command languages of Dijkstra [Dij75] and Hoare. PROMELA is intended to specify protocols. However computer networks systems and protocols have some general characteristics shared by a lot of computer systems such as reactive systems. They react to events from their environment. Their sub-systems can work simultaneously and they can communicate. They are hierarchically structured. The PROMELA language is a formalism for the specification and the validation of procedure rules as a partial description of a protocol. This description of protocol is called a *validation model.* It is partial because in PROMELA emphasis is put on the description of an abstract and simple model of protocols in order to study their structures and to verify their properties. The other aspects such as implementation details, message format, message

encoding, etc are not considered. Briefly, a validation model defines interactions of processes as in a distributed system. PROMELA processes are finite state machines. A process specification is done through a *process type* which describes the behavior of the process.
SPIN (Simple PROMELA interpretor) [Hol91] is a toolbox associated to the language PROMELA. It can be presented on several viewpoints. First, it is an interpretor for the PROMELA validation models. Second, it is a verification system (a model checker) for asynchronous concurrent systems. Third, it is a simulator of communicating processes. Each design with PROMELA can be validated within SPIN.
A validation of a design in PROMELA consist to prove that the design is correct according to the correctness criteria: *absence of deadlocks, liveloks and improper terminations.* For this purpose PROMELA offers mechanisms to specify correctness criteria. The SPIN tool includes procedures to verify these specific properties [HP94] [GPVW95].
The mechanism to specify correctness properties are assertion statement (`assert`) or *temporal claims* (`never ...`).
We are not going to deal with all the validation process in this presentation. We focus on the translation of our specifications into Promela processes from which the validation is done.

The approach for the translation is done in two steps: translating data into promela data types, translating behavior into Promela processes.

Translating data

The main data part of machines is defined with Z schemas. This data part is formally analysed within the $Z/Eves$ environment. Accordingly, we abstract from this data part, an essential part needed to make the processes.
We take care in the translation to have specific data for each (instance of) machine. For this purpose, the translation of a Z schema results in an array (from 1 to `N_Schema_Name`) of elements representing the attributes of the schema.

Examples

Schemas of the form $Sch \mathrel{\widehat=} [x : T; y : T]$
we have :

```
#define N_Sch ...
T x[1..N_Sch];
T y[1..N_Sch];
```

Dealing with sets
$S : \mathbb{F} Ts$ gives

```
#define MaxS ...
Ts  S[1..MaxS]
```

Generalisation to schema $S \mathrel{\widehat=} [Ss : \mathbb{F}\ Ts]$
Intuitively each instance of a schema S has its associated array As. Then we have `#define N_S ...` for the number of instances,
The numbers in `1..N_S` are used to distinguish each instance.

We also have `1..MaxSs` elements for each set Ss.
The schema translation is then done as following:

```
/* N instances of S*/
#define N_S ...
short Ss[N_S];
/* a set for all Ss */
#define MaxArraySs N_S*MaxSs
short ArrSs[MaxArraySs];
```

The principle is the same for schema like $S \mathrel{\widehat=} [f : X \rightarrow Y]$.
A stepwise application of this approach enables us to get the data part we need for elaborating processes. The predicate part of the schema is less straightforward.

Translating transactions: algorithm

For this second step, a straightforward procedure enables us to build processes skeleton from transactions.
The principle consists in obtaining promela processes (mainly with guarded statements) from the transactions specified for machines. A machine has several transactions. We have a textual notation for transaction using the simple trivial notation

```
From  [MachineName]
  Initial Configuration
REACH [ActionName]
  Final Configuration
END
```

The translation algorithm is described as follows:

For a machine Mi ,
Create a process with the same name preceded by the string `proctype` *and followed by the interface* `(ii)`.
Create an empty body (`do od`) *for the process type.*
For each transaction of the machine Mi ,
Build a guarded statement `Gti`.
Add `Gti` *to the guards of the body of the process associated to Mi.*
For each event received by the machine ,
Create a channel nammed by the concatenation of the event name and the string `channel`.
For each event sent ,
Create in the same way a channel
Emission/reception of events ,
For an expression like `<--evt(ee)--`
generate `evt_channel[i]?evt(ee)`
For an expression like `--evt(ee)-->`
generate `evt_channel[i]!evt(ee)`

Generation of conditions

For an expression like `(clg_setting = direct_mode)`
generate `(clg_setting[i] == direct_mode)`

Translating simple transaction
Consider the following transaction,

```
FROM [Machinei(ii)]
   State |- clg_setting = direct_mode
   <--switch_lscene(id)--
REACH [transi]
   --ok(id)-->
END
```

The generated code is like

```
proctype Machinei(ii)
{
 do
 /* transi */
 :: (clg_setting[ii] == direct_mode) -> switch_lscene_channel?switch_lscene(id) ->
 ok_channel!ok(id)
 od
 }
```

Based on this algorithm we write a prototype translator for transactions. It helps enough for building the main processes as the core of the system to analyse. However the translation does not cover completely the initial specification in terms of machines. This is due to the intensive use of quantified predicates. Therefore for simple cases the approach is efficient. The drawbacks noted here are not specific to our approach and constitutes the limits of some formal methods. But considering the global requirement and the treatment done, the benefit is significant.

7 Appraisal of this experiment

Our specification can be checked point to point with the problem description. We give here a concise synthesis of the requirements capture. User's needs are captured by one or several transactions or invariants. Although all the

specification is not presented here in the paper, the list presented below shows the different transaction labels and which part of the requirements they cover:
U1 by *Inv_Safe_Illumination*, U2 by *Inv_chosen_lscene*, U3 by *user_in*, U4 by *user_in*, U5 by *mdf_lscene*, U6 by *mdf_lscene*, U7 by *upd_ls_maintain_delay*, U8, NF1, NF2 NF3 by *set_default_light*, U9 by *cl_on, cl_off, mdf_lscene, upd_ls_maintain_delay*, U10 by *user_in, mdf_lscene, set_default_lights*, U13 by *Inv_Safe_Illumination*, U14 by *user_in*, FM1 by *ComputLLvl*, FM2 by *Inv_Light_Off*, FM3 by *Inv_Light_Off*, FM4 by *upd_ls_maintain_delay*, FM5 by *upd_light_Off_delay*, FM6 by *turn_light_off*, FM7 partly by *set_default_light*, FM8 by *the indicators in the FM Control Panel*, FM9 not yet treated, FM10 by *reprt_malfunction*, FM11 by *set_MF_mdetector, set_MF_actuator* , NF4 by *set_default_lights*, NF8, NF9 *not yet done.*

7.1 Related works

The Light Control Case Study was submitted to developers and researchers for treatment. A special issue of the Journal of Universal Computer Science will be published. The work we present here go further the *Requirements Engineering* of the LCP which was originally the main challenge around this case study.
In this paper the emphasis is on the stepwise specification of the LCP with an integrated approach and some characteristics of this approach which facilitates the study of the problem. We underline here some approaches related to the formalism we have used to achieve the case study.
There are a lot of works on combining 'data-oriented' approach and 'behavior-oriented' approach to define powerful specification formalisms: LOTOS [ISO89], SDL[EHS97], Event Calculus [Sto97], Z+CCS [Gal96], [GS97],[Bru91],[TA97], (Object-)Z+CSP [Ben90],[Smi97, Fis97], RAISE [Gro95b],[Gro95a], etc. Many of them use the combination of process algebra (CCS and CSP) and state-based formalism especially Z and its object oriented variant: Z with CSP, Object Z with CSP, Z with CCS, but also algebraic formalism (ACT ONE in LOTOS) [ISO89].
Our work is closely related to those mixing Z and process algebra and defining appropriate transitions rules for the system to be specified (like CSP-OZ). However our work differs from these works in the manner to combine data and behavior and the use of guards. The data part is "separated" from the behavior part which is given as a set of behavior rules but it is impossible to write the behavior part without the data part. Then the behavior part does not use an existing (process algebra) formalism, it is rather founded on inference rules which are the basis for semantics definitions. This approach gives us some freeness in the specification. The behavior rules are inference rules in the way of natural semantics rules [Kah87] and exploit the data part. Moreover in our approach all actions are guarded by a configuration characterization and that can be considered as a fundamental difference with the previous works. However our approach can be compared with the Action Systems [BKs83],[SW95], [But92] which also combines state descriptions and actions on state; more precisely a view of action system which models communication by synchronization of commonly labelled actions.

7.2 Concluding remarks and Future works

This paper presents a treatment of the Light Control Problem as a substantial case study or benchmark to put formal methods to the test. The result of our work is a specification of an important part of the system. Some other parts are not yet specified, they have more to do with the physical part than with the software part. For example the development of a mathematical formula to compute light level according to ambient illumination (a kind of function studied by our colleagues of other disciplines). This example shows that in the specification of substantial real life systems, there is a need of integration of several scientific experiments. Other difficulty of this study is about the description of the problem itself (13 pages plus some annexes indicating some feedbacks from the customers). A lot of assumptions have been made to handle the complexity of the requirements, for example detecting malfunction, avoiding ambiguities with time constraints to put off the lights, etc.
The remaining work includes: the precise description of some mathematical functions assumed to be predefined (light level calculation wrt daylight illumination) and used in the specification.
We also address the validation of the system from the specifications by using already defined environment: Z/Eves and Promela/Spin. We spend a lot of time to mechanise this step of the work instead of doing it especially for this case study. The benefit is that the resulting translating procedures can be used as a bridge between our specification technique and existing ones and thus it is more practical and convincing.
An ongoing work is about a technique of refinement from our technique. For this purpose we study a Failure-Divergence semantic [Ros98] for configuration machines and from this basis we'll exploit the refinement technique of algebraic processes.

References

[Att98] Christian Attiogbé. Configuration Machines: A Simple Formalism For Specifying Multifaceted Systems. Technical Report 181, IRIN, University of Nantes, July 1998.

[Att00] Christian Attiogbé. Le système de contrôle d'accès : une spécification à base des machines à configurations. In *Proc. AFADL'2000*, pages 203–220, Grenoble, FR, Janvier 2000.

[Ben90] M. Benjamin. A message passing system: An example of combining CSP and Z. In J. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 221–228. Springer-Verlag, 1990.

[BKs83] R.J. Back and R. Kurki-suonio. Decentralisation of process nets with centralized control. In *2nd ACM SIGACT-SIGOPS Symposiun on Principles of Distributed Computing*, pages 131–142. ACM, 1983.

[Bru91] G. Bruns. A language for value-passing CCS. Technical report, ECS-LFCS-91-175, University of Edimburg, 1991. (revised 1995).

[BSC94] Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice.* BCS Practitioner Series. Prentice Hall, 94. ISBN 0-13-124934-7.

[But92] M. J. Butler. *A CSP Approach to Action Systems.* PhD thesis, Oxford University, 1992.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[EHS97] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1997.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derick, editors, *FMOODS'97:*, volume 2, pages 423–438. Chapman and Hall, 1997.

[Gal96] Andy Galloway. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-Perspectives Systems.* PhD thesis, University of Teesside - School of Computing and Mathematics, 1996.

[GPVW95] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. PSTC'95 Conference.* Warsaw, Poland, 95.

[Gro95a] The RAISE Language Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall, 95.

[Gro95b] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, 95.

[GS97] A. Galloway and W.J. Stoddart. An operational semantics for ZCCS. In *ICFEM'97:*, volume 4 of *IEEE*, pages 272–282, September 1997.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewoods, Cliffs, NJ07632, 1991.

[HP94] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. FORTE'94 Conference.* Bern, Switzerlang, 94.

[ISO89] ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

[Iss99] J.UCS Special Issue. Problem Description: Light Control System. Technical report, IICM, 1999. http://rn.informatik.uni-kl.de/ recs/problem/.

[Kah87] G. Kahn. Natural semantics. In *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[LOT88] ISO LOTOS. *A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour.* International Organisation for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1988.

[Ros98] A.W. Roscoe. *The Theory and Practice of concurrency.* Series in Computer Science. Prentice-Hall, 1998.

[Saa97] M. Saatlink. The Z/Eves System. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag, 1997.

[Smi97] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent system. In *FME'97:*, IEEE, 1997.

[Spi92] Mike Spivey. *The Z Notation: a Reference Manual.* Series in Computer Science. Prentice Hall, 92.

[Sto97] W.J. Stoddart. An introduction to the Event Calculus. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97*, volume 1212 of *Lecture Notes in Computer Science*, pages 10–34. Springer-Verlag, 1997.

[SW95] Jane Sinclair and Jim Woodcock. Event refinement in state-based concurrent systems. *Formal Aspects of Computing*, 7:266–288, 1995.

[TA97] K. Taguchi and Keijiro Araki. The State-based CCS Semantics for Concurrent Z Specification. In *ICFEM'97:*, IEEE, pages 283–292. IEEE Society Press, September 1997.