Journal of Universal Computer Science, vol. 6, no. 7 (2000), 731-757
submitted: 30/12/99, accepted: 20/4/00, appeared: 28/7/00 © Springer Pub. Co.

Requirements Capture and Evaluation in NIMBUS: The Light-Control Case Study [1]

Jeffrey M. Thompson
University of Minnesota
Department of Computer Science and Engineering
Minneapolis, MN 55455
thompson@cs.umn.edu

Michael W. Whalen
University of Minnesota
Department of Computer Science and Engineering
Minneapolis, MN 55455
whalen@cs.umn.edu

Mats P.E. Heimdahl
University of Minnesota
Department of Computer Science and Engineering
Minneapolis, MN 55455
heimdahl@cs.umn.edu

Abstract: Evaluations of methods and tools applied to a reference problem are useful when comparing various techniques. In this paper, we present a solution to the challenge of capturing the requirements for the Light Control System case study, which was proposed before the Dagstuhl Seminar on *Requirements Capture, Documentation, and Validation* in June of 1999.

The paper focuses primarily on *how* the requirements were specified: what techniques were used, and what the results were. The language used to capture the requirements is RSML^{-e}; a state-based specification language with a fully specified formal denotational semantics. In addition, the NIMBUS environment – a toolset supporting RSML^{-e}– is used to visualize and execute the high-level requirements.

Keywords: light control system, specification-based prototyping, formal requirements modeling, state-based specification languages, requirements execution and simulation.

Category: D2.1 Requirements Specifications – Languages

1 Introduction

In this paper we present a solution to the challenge of capturing the requirements for the Light Control System (LCS) case study described in [PD00] [2].

[1] This work has been partially supported by NSF grants CCR-9624324 and CCR-9615088, and University of Minnesota Grant in Aid of Research 1003-521-5965.

[2] The case study is available at http://www.rn.informatik.uni-kl.de/~recs.

The solution in this paper is captured in a fully formal, executable, state-based modeling language called RSML^{-e} (Requirements State Machine Language without events). We will demonstrate how the language is used to capture the *system requirements* for the Light Control System and how our *requirements engineering environment*, NIMBUS (based around RSML^{-e}), is used to dynamically validate and evaluate the system requirements. Furthermore, we will show how RSML^{-e} is used to refine the system requirements to *software requirements* and how NIMBUS can assist in this process—an approach we call *specification-based prototyping* [Thompson *et al.*, 1999].

The paper's focus is on *how* the requirements specification effort was completed and what was learned about the original informal requirements specification in the process. Our goal is to give the reader a basic understanding of the capabilities of RSML^{-e} and the NIMBUS environment. The completed formal requirements specification and the models of the environment (sensors, actuators, and process) are too lengthy to include in this report. The full specification is nevertheless available for review on-line[3].

In the next section we will present our general view of control systems of the type presented in the Light Control System case study. We will discuss our view of the system and software requirements as well as a proposed structuring of the requirements specification. In [Section 3] we present the high-level structure of our view of the Light Control System and the system scope. [Section 4] and [Section 5] present the requirements of the Light Control System in RSML^{-e} and demonstrate how the requirements can be dynamically evaluated in the NIMBUS environment. The process of refining the system requirements to software requirements, and how to dynamically evaluate the refinement steps in NIMBUS, is discussed in [Section 6]. [Section 7] contains an evaluation of the project and our approach as applied to the Light Control System, and provides some recommendations and discuss directions for future research.

2 The General Modeling Approach

The primary application domain for RSML^{-e} and the NIMBUS environment is safety critical applications; that is, applications where malfunction of the software may lead to death, injury, or environmental damage. Most, if not all, such systems are some form of a process control system where the software is participating in the control of a physical system. In this section we will provide a general overview of our modeling approach and describe what information must be captured in the system requirements specification and the software requirements specification.

2.1 Control Systems

A general view of a software controlled system can be seen in [Fig. 1]. This model consists of a process, sensors, actuators, and a software controller. The process is the physical process we are attempting to control. The sensors measure physical quantities in the process. These measurements are provided as input to the software controller. The controller makes decisions on what actions are needed and commands the actuators to manipulate the process. The

[3] The full LCS specification as well as information on our tools are available at www.cs.umn.edu/crisys/lcs/

goal of the software control is to maintain some properties in the physical process. Thus, understanding how the sensors, actuators, and process behave is essential for the development and evaluation of correct software. The importance of this *systems* view has been repeatedly pointed out in the literature [Parnas and Madey, 1991],[Leveson *et al.*, 1994], [Heitmeyer *et al.*, 1996].

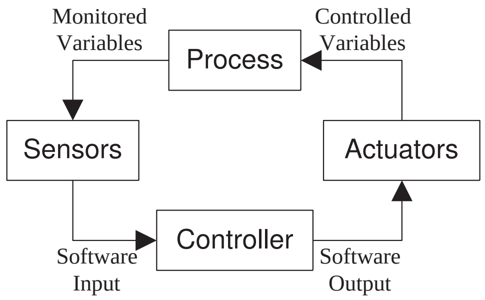

Figure 1: Traditional feedback process control model

To reason about this type of software controlled systems, David Parnas and Jan Madey defined what they call the four-variable model (outside square of [Fig. 2]) [Parnas and Madey, 1991]. In this model, the monitored variables (MON) are physical quantities we measure in the system and controlled variables (CON) are physical quantities we will control. The requirements on the control system are expressed as a mapping (REQ) from monitored to controlled variables. For instance, a requirement may be that *"when a room is occupied, there must be safe illumination."* Naturally, to implement the control software we must have sensors providing the software with measured values of the monitored variables (INPUT), for example, an indication if there is a person in the room. The sensors transform MON to INPUT through the IN relation; thus, the IN relation defines the sensor functions. To adjust the controlled variables, the software generates output that activates various actuators that can manipulate the physical process, for instance, a means to vary the illumination level in the room. The actuator function OUT maps OUTPUT to CON. The required behavior of the software controller is defined by the SOFT relation that maps INPUT to OUTPUT.

The requirements on the control system are expressed with the REQ relation; the system requirements shall always be expressed in terms of quantities in the physical world. To develop the control software, however, we are interested in the SOFT relation. Thus, we must somehow refine the *system requirements* (the REQ relation) into the *software requirements* (the SOFT relation).

2.2 Structuring SOFT

The IN and OUT relations are determined by the sensors and actuators used in the system. For example, to measure the light level in a room we may use a photo resistor coupled with an A/D converter that provides us an estimate of the light level as an integer. Similarly, to control the light level we may use

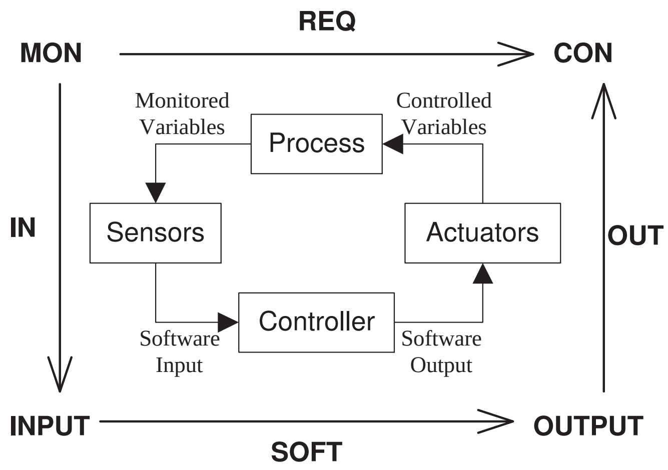

Figure 2: The four variable model for process control systems

dimmers and the light fixtures in the room. Equipped with the REQ relation (mapping MON to CON), the IN relation (mapping MON to INPUT), and the OUT relation (mapping OUTPUT to CON) we can derive the SOFT relation. The question is, how shall we create SOFT and what is the best way to structure SOFT in a language such as RSML^{-e}?

As mentioned above, the system requirements should always be expressed in terms of the physical process. These requirements are likely to change over the lifetime of the controller (or family of similar controllers). The sensors and actuators are likely to change independently of the requirements as new hardware becomes available or the software is used in subtly different operating environments; thus, all three relations; REQ, IN, and OUT, are likely to change over time. If any of the REQ, IN, or OUT relations change, the SOFT relation must be modified. To provide a smooth transition from system requirements (REQ) to the software specification (SOFT) and to isolate the impact of requirements, sensor, and actuator changes, Steven Miller at Rockwell Collins has proposed to structure the software specification SOFT based heavily on the relations in the four variable model [Miller, 1999],[Thompson *et al.*, 1999].

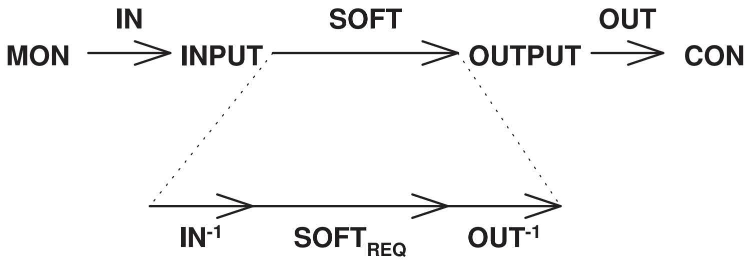

Figure 3: The SOFT relation can be split into three composed relations. The $SOFT_{REQ}$ relation is based on the original requirements (REQ) relation.

Miller proposed splitting the SOFT relation into three pieces, IN^{-1}, OUT^{-1}, and SOFT_{REQ}([Fig. 3]). IN^{-1} takes the measured input and reconstructs an estimate of the physical quantities in MON. The OUT^{-1} relation maps the internal representation of the controlled variables to the output needed for the actuators to manipulate the actual controlled variables. Given the IN^{-1} and OUT^{-1} relations, the SOFT_{REQ} relation will now be essentially isomorphic to the REQ relation and, thus, be robust in the face of likely changes to the IN and OUT relations (sensor and actuator changes). Such changes would only affect the IN^{-1} and OUT^{-1} portions of the software specification. Thus, the structuring approach outlined in this section will isolate the impact of system changes in the software specification SOFT.

In the rest of this paper we illustrate how this framework for requirements specification and refinement is used. We will demonstrate how the REQ relation is captured in RSML^{-e} and how it can be validated through execution and simulation in NIMBUS. We will also demonstrate how the REQ relation is refined to the SOFT relation and how the NIMBUS environment supports dynamic evaluation of the various models created throughout the refinement process. The result of this process will be a formal specification of SOFT that, in NIMBUS, also serves as a prototype of the control software.

3 The Light Control System

The informal requirements for the Light Control system were provided by the problem description included in the call for papers [PD00]. This description provided us with a rough idea of the functionality of the system, but left many areas of the system underspecified, or, more seriously, specified in a way that contradicted other requirements in the system. What follows is our interpretation of the requirements of the system, with a few changes and additions to provide a complete and consistent description of the system. First, we will clarify the physical structure of the components of the system. Then, we can set the system boundaries and identify the monitored (MON) and controlled (CON) variables.

3.1 System Structure

The physical structure of the system and the control software boundaries are outlined in [Fig. 4]. The control software is in the center of the diagram. The Light Control System can be thought of as a set of small control systems, each of which control the light level for a given room or hallway. Because the behaviors of the rooms and hallways are independent of one another, we can model each system separately and then combine the specifications into a complete control system for a floor or a building.

We view a *system* as a collection of *components* connected by communication *channels*. A graphical representation of the collection of system components and communication channels for a single room can be seen in [Fig. 5]. The components are connected to the channels through *interfaces* and can send *messages* over the channels. A message is a collection of *fields* holding the atomic pieces of information communicated between the components. The only information flow between the components is through the unidirectional channels.

Dividing up the specifications in this way allows us to focus on small parts of the system and makes the task of analyzing these pieces simpler. In general, the

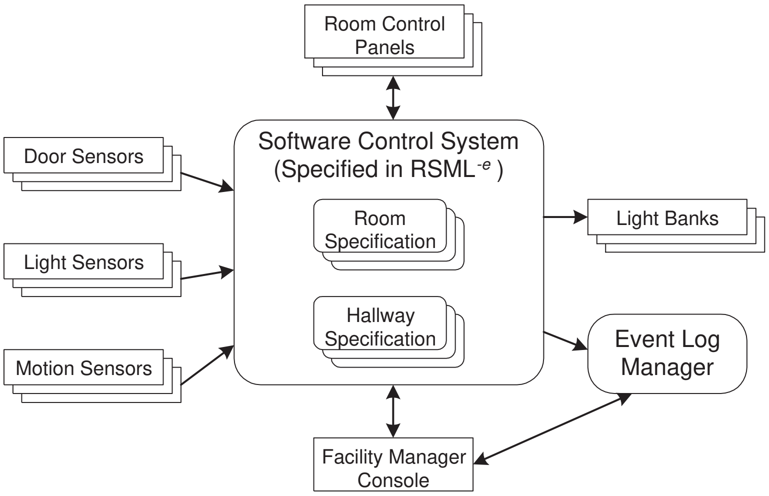

Figure 4: The software control system boundaries for the Light Control System.

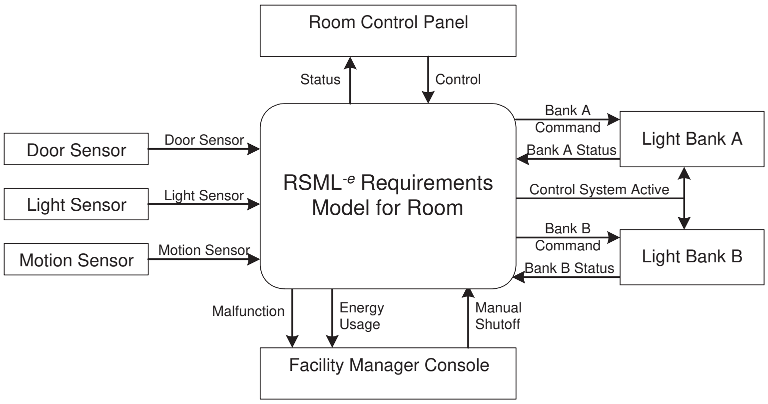

Figure 5: The room model and its interconnections.

hallway can be viewed as providing a subset of the functionality provided by the rooms. For this reason, as well as space concerns, the remainder of this paper will focus on the specification of the system and software requirements for the rooms only (the full specifications for rooms and hallways are available on-line).

3.2 Monitored and Controlled Variables

The first step in a requirements modeling project is to define the system boundaries and identify the monitored and controlled variables; the monitored and controlled variables exist in the physical system and act as the interface between the proposed controller (software and hardware) and the system to be controlled.

In this paper we will not go into the details of how to scope the system requirements and identify the monitored and controlled variables—guidelines to help identify monitored and controlled variables are covered in, for example, [Miller, 1999],[Faulk *et al.*, 1992], [Jackson, 1995].

In the case of the Light Control System, we identified, for example, the presence of a person in a room as a monitored variable and the light level in the room as a controlled variable. Both are clearly concepts in the physical world, and thus suitable candidates as monitored and controlled variables for the requirements specification. A complete list of the monitored and controlled variables we identified in the LCS for a room are defined in [Tab. 1].

4 Modeling the REQ Relation

After we have determined the scope of our system we are ready to capture REQ. Since our work is based around a modeling language called RSML^{-e}, we start the section with a short introduction to the notation before we discuss the Light Control System requirements.

4.1 Introduction to RSML^{-e}

RSML^{-e} is based on Requirements State Machine Language (RSML) developed by the Irvine Safety Research group under the leadership of Nancy Leveson [Leveson *et al.*, 1994]. RSML^{-e} was developed as a requirements specification language specifically for embedded control systems. One of the main design goals of RSML^{-e} was readability and understandability by non-computer professionals such as users, engineers in the application domain, managers, and representatives from regulatory agencies. RSML^{-e} is based on hierarchical finite state machines [Harel and Pnueli, 1985], [Harel, 1987], [Harel *et al.*, 1990] and state-based dataflow languages [Heitmeyer *et al.*, 1996], [Heitmeyer *et al.*, 1995a]. RSML^{-e} supports parallelism, hierarchies, and guarded transitions. The main differences between RSML^{-e} and RSML are the addition in RSML^{-e} of rigorous specifications of the interfaces between the environment and the control software, and the removal of internal broadcast events [Leveson *et al.*, 1999], [Heimdahl *et al.*, 1998].

An RSML^{-e} specification consists of a collection of *state variables*, *I/O variables*, *interfaces*, *functions*, *macros*, and *constants*, which will be briefly discussed below.

In RSML^{-e}, the state of the model is the values of a set of *state variables*, similar to mode classes in SCR [Heitmeyer *et al.*, 1995b]. These state variables

Monitored Variables:

Name	Range	Description
System Variables:		
Light_Level	0..10000 Lux	The amount of light in the room
Occupied	Boolean	TRUE if room is occupied
Light_Level_Undetectable	Boolean	Used for light sensor failure.
Occupied_Undetectable	Boolean	Used for motion or door sensor failure.
Window_Light_Bank_Intensity	0..100	Percent intensity of Window Light Bank
Wall_Light_Bank_Intensity	0..100	Percent intensity of Wall Light Bank
Operator Inputs:		
Chosen1_LS_Button_InVar	Boolean	Chooses/Replaces light scene 1
Chosen2_LS_Button_InVar	Boolean	Chooses/Replaces light scene 2
Chosen3_LS_Button_InVar	Boolean	Chooses/Replaces light scene 3
Default_LS_Button_InVar	Boolean	Chooses/Replaces default light scene
Set_LS_Button_InVar	Boolean	If TRUE and another LS button is pressed, replaces the light scene with the current light scene.
T1	1..1440 minutes	Timeout to reestablish default light scene
T3	1..1440 minutes	Timeout to shut off lights
FacM_Shutoff	Boolean	Allows fac. man. to remotely shut off lights.

Controlled Variables:

Name	Range	Description
System Variables:		
Con_Window_Light_Bank_Intensity	0..100	Intensity of Window Light Bank
Con_Wall_Light_Bank_Intensity	0..100	Intensity of Wall Light Bank
Outputs to Operator:		
Failed	Boolean	TRUE if system detects component failure.

Table 1: The monitored and controlled variables for the Light Control System in a room.

can be organized in parallel or hierarchically to describe the current state of the system. Parallel state variables are used to represent the inherently parallel or concurrent concepts in the system being modeled. Hierarchical relationships allow *child* state variables to present an elaboration of a particular *parent* state value. Hierarchical state variables allow a specification designer to work at multiple levels of abstraction, and make models simpler to understand.

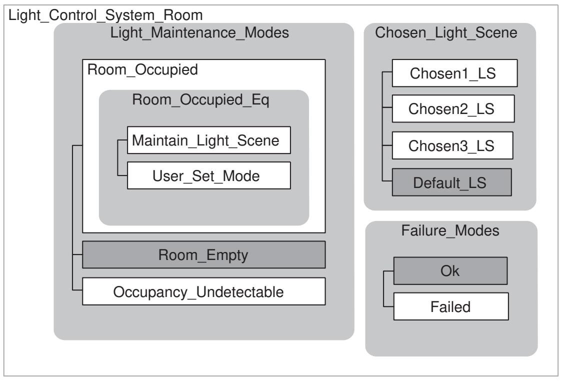

Figure 6: The state machine for the requirements model of the Light Control System in an individual room

For example, consider the Light Control System for an individual room. The state variable hierarchy used to model the requirements on this system could be represented as in [Fig. 6]. This representation includes both parallel and hierarchical relationships of state variables. *Light_Maintenance_Modes, Chosen_Light_Scene* and *Failure_Modes* are three parallel state variables, and *Room_Occupied_Eq* is a child state variable of *Light_Maintenance_Modes*

Next state functions in RSML^{-e} determine the value of state variables. These functions can be organized as *transitions* or *condition tables*. Condition tables describe the conditions under which a state variable *assumes* each of its possible values. Transitions describe the condition under which a state variable is to *change* value. A transition consists of a source value, a destination value, and a guarding condition. A transition is taken (causing a state variable to change value) when (1) the state variable value is equal to the source value, and (2) the guarding condition evaluates to true. The two next state function types are logically equivalent and mechanized procedures exist to ensure that types of next state functions are complete and consistent [Heimdahl and Leveson, 1996].

The state functions are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. State variables are data-dependent on any other state variables, macros, or I/O variables that are named in their transitions or condition tables. If a variable is a child variable of another state variable, then it is also dependent on its parent variable. The value of the state variable must be computed after the items on which it is data-dependent have been computed. For example, the value of the *Room_Occupied_Eq* state variable would be computed after the *Light_Maintenance_Modes* state variable, because its value is dependent on whether or not *Light_Maintenance_Modes* is in the *Room_Occupied* state.

Conditions are simply predicate logic statements over the various states and variables in the specification. The conditions are expressed in disjunctive normal form using a notation called AND/OR tables [Leveson *et al.*, 1994] (see [Fig. 7], [Fig. 8], etc.). The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated columns. An asterisk denotes "don't care."

I/O Variables in the specification are the quantities that are available to the interfaces of the specification. Input variables allow the analyst to record the monitored variables (MON) or values reported by various external sensors (INPUT) which are received in a message. Output variables provide a place to capture the controlled variables (CON) or the values of the outputs (OUTPUT) of the system prior to sending them out in a message.

Interfaces, discussed briefly in [Section 3.1], encapsulate the boundaries between the RSML^{-e} specification and the external world.

To further increase the readability of the specification, RSML^{-e} contains many other syntactic conventions. For example, it allows expressions used in the predicates to be defined as functions (e.g., TotalIntensity()), and familiar and frequently used conditions to be defined as macros (e.g., OccupancyUndetectable()). *Functions* in RSML^{-e} are mathematical functions that are used to abstract complex calculations. A *macro* is simply a named AND/OR table that is used for frequently repeated conditions and is defined in a separate section of the document.

The LCS has a number of different timing properties which must be recorded. RSML^{-e} views time as a continuously increasing input variable with a user defined granularity. From this formal definition, we have built a number of different time expressions (for example, the analyst can get the time that various variables were assigned or changed value). This allows timing properties to be easily expressed. The specific time expressions used in the LCS will be introduced as they are used. For more information on the formal semantics of RSML^{-e}, please see [Whalen, 2000].

4.2 Overview of REQ

Our choice of the various monitored and controlled quantities places certain constraints on what can, and cannot, be specified in the REQ relation. If the monitored and controlled variables are chosen appropriately, then the specification of the REQ relation will be focused on the issues which are *central* to the requirements on the system.

In the Light Control System, some of the informal needs from the problem description will *not* be represented in the REQ relation or they will be represented in an abstract form. For example, the problem description states that *"If any outdoor light sensor or the motion detector of a room does not work correctly, the user of this room has to be informed"* (U8) [PD00]. The REQ relation does not include any notion of a motion detector nor of the outdoor light sensors. Instead, we use the monitored quantities for the light level in the room and whether the room is occupied.

Often when starting to construct the REQ relation, it is helpful to examine the controlled variables of the system. It is necessary to determine what conditions partition the value of a particular controlled variables (i.e., what modes effect the controlled variable) and under what scenarios various outputs should be generated. In this light control system, the controlled variables are the intensity of the window and wall light groups in the office and the failure indication operator output. For now, we will focus on the window and wall intensity variables.

From the problem description, it becomes clear that there are two main activities which affect the way that the REQ relation must determine the values of the controlled variables. First, the REQ relation must be able to capture and use the various user set points (U6, U7, and U9 [PD00], page 9). Second, the control system should somehow maintain the light level in the rooms (FM1 on page 10 and U2 on page 9). Furthermore, it would appear that these two tasks occur concurrently. The following two subsections consider these two tasks.

4.3 User Functionality

In this section, we will consider the maintenance of the various set points by the user and the way in which they are modeled in our version of the REQ relation. This is the simplest of the two main tasks of the REQ relation. Nevertheless, by constructing a formal specification, several issues with the definition of a light scene were exposed.

In the problem description, the room control panel (RCP) is described by U7, U9, and U12. U12 states that the RCP is a mobile, stand-alone device; therefore, a reasonable design for such a device seemed to be to keep the controls as simple as possible. Our design of the RCP is shown in [Fig. 12].

We envisioned the user selecting a light scene by simply pressing its associated button and setting a light scene by first adjusting the light in the room using the top two control groups and then pressing the "set" button while pressing one of the light scene buttons (i.e., similar to the functionality of a typical car radio).

This seems a simple, and intuitive solution; nevertheless, the definition of a light scene given in the problem description states that a name is associated with each light scene. In order to allow the user to enter a name, a more complex interface would seem to be in order. However, this would (1) add significantly to the cost and complexity of the RCP and (2) probably not add much, if anything, in terms of functionality and user satisfaction. Therefore, we decided to simplify the definition of the light scene so that the light scene is simply selected with a button on the RCP (*Scene 1, Scene 2, Scene 3*).

To model the light scenes, we use three state variables to capture the light level (measured in Lux). [Fig. 7] shows the RSML^{-e} definition for the state variable for the first light scene. As part of the definition of any non-enumerated type state variable in RSML^{-e}, the expected minimum and maximum of the variable are given. This, and other criteria, were outlined in [Jaffe *et al.*, 1991].

There are two possible assignments to the *Chosen1_LS_Light_Level* variable. The first covers the case that the user is pressing the "set" button and the button for light scene 1. This condition is determined by checking whether the button in pressed (*Chosen1_LS_Button_InVar* is the input variable for the button, it is equal to kPressed if button is pressed). Similarly, the *Set_Light_Scene_Button_InVar* must also equal kPressed. If these two predicates are true, then the state variable records the value of the monitored variable for the current light level in the room (*Light_Level_InVar*). Otherwise, the variable's value must stay the same. In RSML^{-e} this must be explicitly specified (the second case in [Fig. 7]). The other light scenes (chosen2, chosen3, and default) all have similar definitions for the capture of the desired light level.

State Variable

Chosen1_LS_Light_Level

Type: INTEGER

Units: lux

Expected Min: 0

Expected Max: 10000

:= Light_Level_InVar **IF**

Chosen1_LS_Button_InVar = ButtonPressType::kPressed	T
Set_Light_Scene_Button_InVar = ButtonPressType::kPressed	T

:= PREV_STEP(Chosen1_LS_Light_Level) **IF**

Chosen1_LS_Button_InVar = ButtonPressType::kPressed	T	F	F
Set_Light_Scene_Button_InVar = ButtonPressType::kPressed	F	T	F

Figure 7: Capturing the Light Level in a Light Scene

The light scene definition in the problem description states that the scene consists of the light level and an option: window, wall, or both. Furthermore, if the selection is both, then both light banks are used equally to obtain the desired light level. The control system can detect whether or not both light banks are on and thereby determine the value of this option when the set button was pressed. Nevertheless, we reasoned that the user would almost *always* have both light banks on to some degree or other.

Imagine what would happen if the user (1) adjusted the lights as desired, for example, 40% window and 60% wall; (2) set this to the first light scene; and (3) pressed the first light scene button. Clearly, this light scene uses both the wall and the window light groups. But, if the light scene stores the "both" option, then when the users presses the button for the first light scene, the window light group will be raised in illumination and the wall light group will be lowered. This behavior does not seem intuitive. Therefore, we expanded the definition of a light scene so that it consists of the light level in the room plus intensity of the window light group (0 - 100) and the intensity of the wall light group. Using this information, we can maintain the proportion (window/wall) that the user has selected and thereby, in our opinion, provide a better control behavior for the system.

4.4 Maintaining the Light Level

After the user has chosen the light level and distribution(between window and wall) for the room, this light scene must be maintained. Also, there are a number of requirements related to the user leaving the room and whether or not room occupancy is detectable that need to be considered when computing the values of the controlled variables. This section discusses the control behavior for maintaining the light level and light distribution in the room.

The control system clearly behaves differently depending on the occupancy of the room; if there is a person in the room, the control system must maintain the light level in the room. If there is no one in the room, the control system must determine whether or not to shut off the lights.

State Variable

Light_Maintenance_Modes

Location: Light_Control_System_Room

:= Room_Occupied **IF**

Occupied_InVar = TRUE	T
Occupied_Detectable_InVar = TRUE	T

:= Occupancy_Undetectable **IF**

Occupied_Detectable_InVar = TRUE	F

:= Room_Empty **IF**

Occupied_InVar = TRUE	F
Occupied_Detectable_InVar = TRUE	T

Figure 8: Light Maintenance Modes in the REQ Relation

[Fig. 6] shows this partitioning of the Light_Maintenance_Modes. Each mode has certain conditions under which it is active. These conditions are specified with a state variable definition as shown in [Fig. 8]. These modes depend on two monitored variables (see [Tab. 1]) *Occupancy_Detectable* and *Room_Occupied*, which determine whether we can detect the occupancy status of the room, and if so, whether or not the room is occupied. Note that because this is a specification of REQ, the monitored variables are actually the input quantities. Thus, in [Fig. 8], the monitored quantities have the suffix _InVar (this is a naming convention that we commonly use in RSML^{-e} specifications, not something which is enforced by the tool). Also, we have adopted the convention for boolean variables of writing the more lengthy expression "X_var = TRUE" rather than simly "X_var" which would be a valid boolean expression by itself. NIMBUS, of course, allows either convention to be used.

The way that we have chosen to describe the control of the light level in the room is as follows: (1) the light level in the room is compared with the light level required by the current light scene, (2) if light level is not equal to the light level specified in the current light scene, the light intensity of the window/wall light banks are adjusted proportionally up or down by a small increment. Then, the system will poll the light level again within a short amount of time and eventually, the light in the room will comply with the selected light scene.

There is an issue, however, with the fact that it is *not* desirable to have the control system chage the light intensity at the same time as the user attempts to adjust it; that is, the control system should not fight the user for control over the lights. Thus, it is necessary to partition the Room_Occupied mode into two sub-modes: one where the system is receiving user input and should produce no control actions and one where the system is responsible for maintaining the light level in the room. This partition can be seen in [Fig. 6].

The current light scene is the basis of the control of the light in the room. First, it is computed and then it is used to determine the values of the controlled variables. The current light scene, like any other light scene, consists of a light level (in lux) and the intensity of the window and wall light banks.

[Fig. 9] shows the state variable definition for the light level of the current light scene. On the right, the cases are labeled to clarify the presentation in this paper; this labeling is not a part of RSML^{-e}. The first case in the definition simply states that the light level will be updated to the current light level in the room if the user is setting the controls for the room. This ensures that as the user makes changes to the lights, the changes are maintained, not reset, by the system.

The second group of cases in [Fig. 9] (cases 2-5) handles the user pressing one of the light scene buttons on the RCP. If the user presses one of these buttons, the light level associated with the selected light scene is used as the current light scene and will thus be maintained by the system.

The sixth case determines the light level in the room if the room is unoccupied. The lights are shut off (the light level set to zero) if the room is empty and either T3 has passed or the facility manager has issued the shutoff command. T3 is measured from the time that the Light_Maintenance_Modes state variable assumes the value Room_Empty (the TIME_ENTERED part of line 2). If the facility manager issues a shutoff command, this is indicated by the reciept of a message at the FacM_Shutoff interface. The MESSAGE_AT expression in line 3 of the condition table is true if this is the case.

The specification determines whether the room has been reoccupied by examining the *Light_Maintenance_Modes* state variable. In order to detect a *change* in the variable, the specification must be able to reason about previous values of the variable. RSML^{-e} allows this through the use of the PREV_STEP expression, which returns the value its sub-expression had at the close of the previous computation of the RSML^{-e} specification. In case 7, the room is reoccupied if the *Room_Occupied_Eq* state variable has the value *Maintain_Light_Scene* and in the previous step, the *Light_Maintenance_Modes* state machine did not have the value *Room_Occupied* [4]. When the room is reoccupied, then the light level is determined by whether or not T1 has passed (case seven). The function *Reoccu-*

[4] Note that the ".." notation in the figure before the state variable names is used to indicate that the RSML^{-e} parser should search through the state variable tree to find the given state variable. This notation avoids having to specify full path names within the specification, as duplicate names are allowed in the tree.

Output Variable

Current_LS_Light_Level

Type: INTEGER

Units: lux

Expected Min: 0

Expected Max: 10000

:= Light_Level_InVar **IF**

..Room_Occupied_Eq **IN_STATE** User_Set_Mode	T

Case 1: The user is setting the light level

:= Chosen1_LS_Light_Level **IF**

Chosen1_LS_Button_InVar = ButtonPressType::kPressed	T
Set_Light_Scene_Button_InVar = ButtonPressType::kNotPressed	T

:= Chosen2_LS_Light_Level **IF**

Chosen2_LS_Button_InVar = ButtonPressType::kPressed	T
Set_Light_Scene_Button_InVar = ButtonPressType::kNotPressed	T

:= Chosen3_LS_Light_Level **IF**

Chosen3_LS_Button_InVar = ButtonPressType::kPressed	T
Set_Light_Scene_Button_InVar = ButtonPressType::kNotPressed	T

:= Default_LS_Light_Level **IF**

Default_LS_Button_InVar = ButtonPressType::kPressed	T
Set_Light_Scene_Button_InVar = ButtonPressType::kNotPressed	T

Case 2-5: The user has pressed one of the light scene buttons on the room control panel.

:= 0 **IF**

..Light_Maintenance_Modes **IN_STATE** Room_Empty	T	T
TIME >= ..Light_Maintenance_Modes **TIME_ENTERED** Room_Empty + T3_InVar	T	*
MESSAGE_AT(FacM_Shutoff)	*	T

Case 6: The room is empty and the lights should be turned off

:= Reoccupied_Light_Level() **IF**

..Room_Occupied_Eq **IN_STATE** Maintain_Light_Scene	T
PREV_STEP (..Light_Maintenance_Modes **IN_STATE** Room_Occupied)	F

Case 7: The room is reoccupied.

:= PREV_STEP(Current_LS_Light_Level) **IF**

..Room_Occupied_Eq **IN_STATE** Maintain_Light_Scene	T
PREV_STEP (..Light_Maintenance_Modes **IN_STATE** Room_Occupied)	T

Case 8: Otherwise, the light level should remain constant

Figure 9: Current Light Scene Light Level in the REQ relation

pied_Light_Level returns the default light level if T1 has passed and the chosen light level otherwise. Due to space constraints, this simple function will not be shown.

Finally, the light level will remain the same if the user is not making changes to the light level and room has not been recently reoccupied.

Once the light level is computed, it is compared with the monitored light level in the room. The actual sampling of the light level is performed by the Light_Level_InInterface. If the light needs to be increased/decreased, it is done so while maintaining the proportion of window light to wall light specified in the current light scene. The details of this part of the specification will not be included in this report due to space considerations.

5 Execution of the REQ Relation in NIMBUS

Now that we have an initial version of the REQ specification, it is possible to analyze it and simulate it in a realistic environment using the NIMBUS tools for requirements modeling. The NIMBUS tools provide support for analysis, simulation, and specification construction. NIMBUS is currently being used at the University of Minnesota, Massachusetts Institute of Technology, NASA, and other sites.

Assurance that the requirements specification (system or software) possesses desired properties can be achieved through (1) manual inspections, (2) formal verification of the desired properties, or (3) simulation and testing of the specification. To achieve the high level of confidence in the correctness required in a safety-critical system, all three approaches must be used in concert. One environment, called NIMBUS, under development at the University of Minnesota provides support for all these activities [Thompson *et al.*, 1999], [Thompson and Heimdahl, 1999]. This is the environment which we have used to capture and evaluate the required behavior of the Light Control System.

The three V&V techniques fill complementary roles within the validation and verification process. Manual inspections and visualization provide the specification team, customers, systems engineers, and regulatory representatives the means to informally verify that the behavior described formally in the specification matches the desired "real world" behavior of the system. Formal analysis is helpful to determine if the specification possesses desirable properties, for instance, if the specification is complete and consistent, and whether unsafe states are reachable. Simulation and testing are necessary to provide additional assurance that the specification captures the desirable behavior and is free of faults. In this report we will focus on the capabilities for execution and simulation available in NIMBUS.

5.1 The NIMBUS Environment

The execution and simulation capabilities NIMBUS environment are based on the ideas that (1) the engineers would like to have an executable specification of the system early in the project and that (2) as the specification is refined it is desirable to integrate it with more detailed models of the environment to enable accurate validation of the requirements specification. Other work has been done in this area, for example, other tools, such as Statemate from i-Logix, provide visualizations of the state of the specification. In NIMBUS, however, we also provide a flexible way to integrate the visualizations of the specification with visualizations of the embedding environment. An effort with similar goals was presented in [Heitmeyer *et al.*, 1995a]; however, the effort was mostly a proof of concept and no real tool support for this kinds of visualizations was provided. NIMBUS allows a vast array of visualizations to be constructed using industry standard tools in a quick, cost effective, and flexible manner.

NIMBUS also contains a logging facility, capable of recording all inputs and outputs to the simulation. These logs can later be played back and analyzed, allowing step by step analysis of earlier real-time simulations. The logging facility can also be used to easily create a library of specific test scenarios.

In the initial stages of the project, we want the executions to take their input from simple models of the embedding environment, for example, text files or user input. As the specification is refined, the analyst can add more detailed models of how the controlled system behaves, for example, additional RSML^{-e} specifications or software simulations of the system. As the requirements specification (REQ) is refined to a software specification (SOFT), models of the sensors and actuators can be incorporated into the executions. In order to have a closed loop simulation, a model of the process can be added between the sensor and actuator models. Finally, when the specification has been refined to the point of defining the software inputs and outputs (INPUT and OUTPUT), the analyst can execute it directly with the hardware. This hardware-in-the-loop simulation closes the gap between the prototype and the actual hardware. These ideas are illustrated in [Fig. 10].

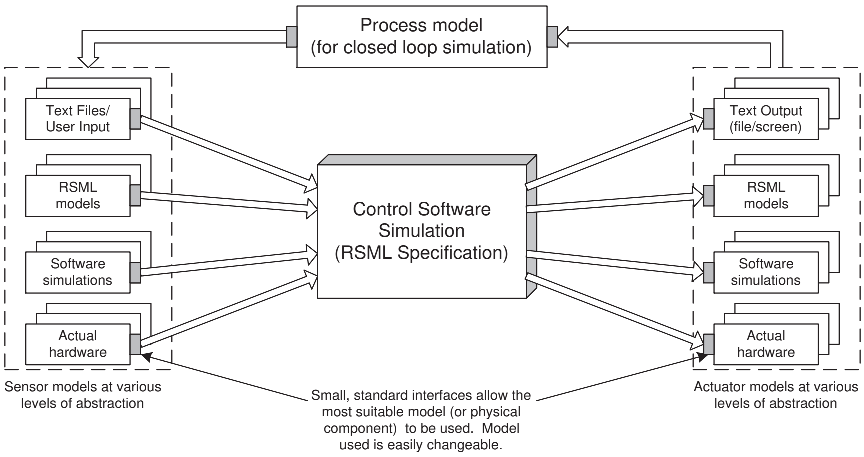

Figure 10: The NIMBUS *Environment*

The flexibility to quickly and easily connect different models of the components in a system provides new opportunities when creating and validating formal specifications. An environment such as ours can be used to aid in requirements based prototyping, in refinement of the specification as well as the environmental models, the evaluation of the operator interface, and in testing both the specification and the implementation derived from it. The detailed discussion of the various capabilities of our environment is beyond the scope of this paper and the interested reader is referred to [Thompson *et al.*, 1999].

5.2 Executing the Requirements

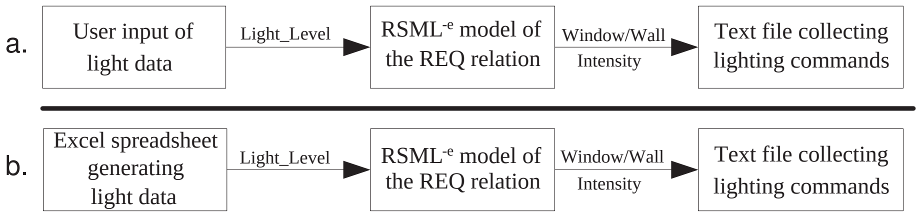

Figure 11: The REQ relation can be evaluated using text files or user input (a) or interacting with a simulation of the environment (b).

The NIMBUS environment allows us to execute and simulate the model we discussed in [Section 4] using input data representing the monitored variables and collect output representing the controlled variables. Input data could come from several sources. The simplest option for input is, of course, to have the user specify the values (either interactively, or by putting the values into a text file ahead of time). This scenario is illustrated in [Fig. 11](a).

Unfortunately, it is often difficult to create appropriate input scenarios since the physical characteristics of the environment enforce constraints and interrelationships over the monitored and controlled variables. For example, if we increase the light output from one of the light groups, how much will the illumination in the room increase? Thus, to create a valid (i.e., physically realistic) input sequence, the analyst must have a model of the environment. Initially, this model may be an informal mental model of how the environment operates. As the evaluation process progresses, however, a more detailed model is most likely needed. Therefore, in this stage of the modeling we may develop a simulation of the physical environment. The NIMBUS architecture lets us easily replace the inputs read from text files with a software simulation emulating the environment. This refinement can be done without any modifications to the REQ specification.

For the Light Control System, we created a spreadsheet in Microsoft Excel to emulate the behavior of light groups and the illumination level in the room ([Fig. 11](b)) This simple environmental model allows us to interactively modify the traffic through the room, the external light available, etc., and to easily explore many possible scenarios.

Developing a specification of the room control panel is another way to enhance the simulation of the REQ specification. There are a number of reasons why input from a mockup of the room control panel (RCP) is better than that of manually-created text files.

First, the RCP provides a significant amount of data to the REQ specification: The user settings of the window and wall intensity, the values of T1 and T3, and the selection and setting of four different light scenes. There are many different combinations of input sequences that can be generated from the RCP. Manually creating test files for even a small number of them early in a modeling effort would be time-consuming and error-prone.

Second, constructing an interface mockup provides valuable opportunities to evaluate the control system with the intended users. For example, in our case we constructed a mockup of the RCP similar to the one pictured in [Fig. 12]. When we tried to use this mockup, however, it became clear that the user would desire a separate control to set the hours and minutes so as to more easily enter the times T1 and T3 into the system and we modified the RCP accordingly. Our mockup was done using Visual Basic and is pictured in [Fig. 12]. In the figure, both the window and the wall light groups have been turned on and the user has selected a value for T1 of 1 hour and 30 minutes and a value for T3 of 5 minutes.

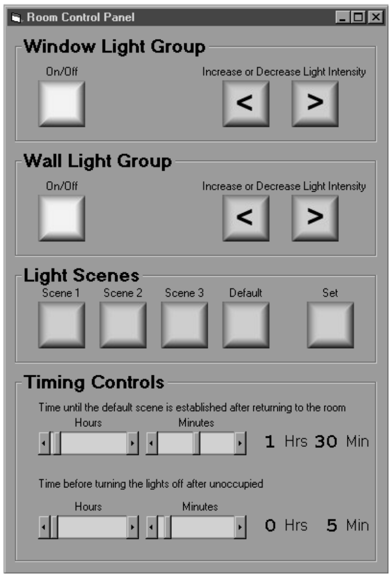

Figure 12: The Room Control Panel with both light banks on.

[Fig. 13] shows data flow between the applications used to do the system simulation of the REQ relation. The values for the room occupancy and the facility manager shutoff signal are still represented by user inputs because in our opinion, representing how occupancy is determined is not part of the REQ relation. The values for light level and the RCP are represented as described above. Note that the user settings of the window and wall intensity must be passed both to the REQ specification and to the light model of the room.

5.3 Results

Simulating the requirements specification provides the opportunity to discover conceptual errors in the requirements specification as well as gain a greater understanding into the requirements themselves. In some cases, the problems were simple modeling errors on our part. More interesting are errors which can be

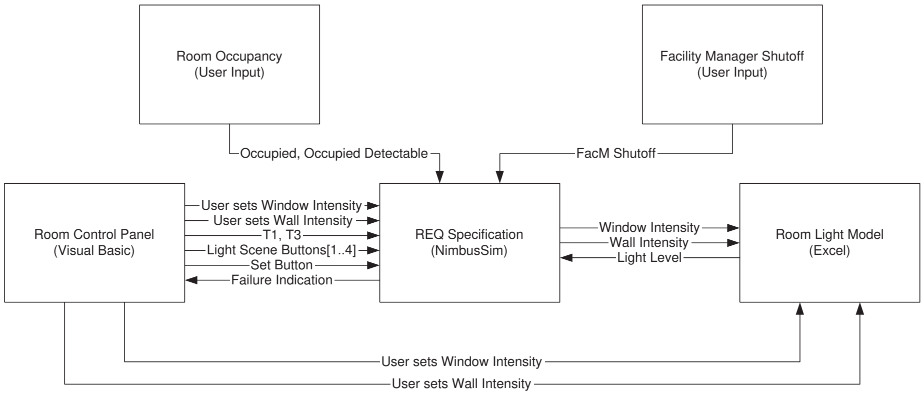

Figure 13: Overview of the REQ simulation

traced back to inconsistencies or underspecification of the original requirements document.

Consider the case where the user has set the values of T1 and T3 so that T3 is greater than T1. In other words, suppose the user has adjusted the lights to a chosen light scene and then left the room. The user is out of the room long enough for T1 to pass, i.e., when the user re-enters the room (according to [PD00], Page 9:U4) the default light scene has to be established. Because the user did not turn out the lights and T3 has not passed yet the lights in the room will still be on as the user left them. Nevertheless, when the control system detects that the room is occupied it *will* change the lighting in the room to comply with the default scene. This is the behavior that is specified in the problem description, but it might not be the behavior that users expect.

We have found that simulation of the high-level requirements in a *realistic* environment is valuable for finding these and other types of conceptual errors. Experimentation in this fashion provides a specification of REQ that is a solid foundation from which to refine a specification of the SOFT relation, a topic which is addressed in the following section.

6 Refining System Requirements to Software Requirements

Once the model of REQ has been thoroughly simulated and analyzed, it is ready to be extended to a model of SOFT. This section discusses how to extend REQ to SOFT and how this was done for several monitored variables in the light control system.

6.1 Refine REQ to SOFT$_{REQ}$

In the real system, the monitored and controlled variables are not directly available; they must be approximated using sensors and actuators. Thus, when refining REQ to SOFT, variables such as *Occupied* will not be directly available from the environment. At this early stage, we may not know exactly what hardware will be used for sensors and actuators; however, we do know that we must use something and we may as well prepare for it early. By encapsulating the monitored and controlled variables we can get a model that is isomorphic to the requirements model; the only difference is that this model is more suited for the refinement steps that will follow as the surrounding system is completed.

In our case, using a macro, *IsOccupied*(), instead of the monitored variable *Occupied* will shield the specification from possible changes in how the final software will determine that a room is occupied (See [Fig. 14]). By performing this encapsulation for all monitored and controlled variables we refine REQ to SOFT_{REQ}, a mapping from estimates of the monitored variables to an internal representation of the controlled variables.

Macro

IsOccupied

Parameters: NONE

Condition:

..Occupied_In **IN_STATE** Occupied	T

Figure 14: The IsOccupied() macro from the refined light control system

6.2 IN, OUT, IN^{-1}, and OUT^{-1}

As the hardware components of the system are defined (either developed in house or procured), the IN and OUT relations can be rigorously specified. The IN and OUT models represent our assumptions about how the sensors and actuators operate.

With the information about the sensor (IN) and actuator (OUT) relations, we can start adding pieces of the IN^{-1} and OUT^{-1} relations to move towards SOFT. To achieve this, we create the IN^{-1} and OUT^{-1} relations in our model. In the following sections, we will focus our attention on the sensors needed to determine if a room is occupied and to detect the light level in the room.

6.2.1 Refining Occupied

The control system must compute whether or not the room is occupied based on the input from the motion and door sensors. For simplicity, we assume that the door sensors are wired as one input to the system, so that if any of the doors are open, the door sensor indicates “open”, and if all the doors are closed, the door sensor reads "closed". When computing whether or not the room is occupied, it is necessary to have some state information (i.e., we must know whether or not the room was occupied in the previous instance in time); therefore, a state variable needs to be added so that IN^{-1} can be properly computed.

The refined state machine can be seen in [Fig. 15]. Instead of assuming that Occupancy is a direct input to the system, as it would be in REQ, the specification now takes the input from one motion detector and the door sensors. Thanks to the structuring of the SOFT relation, this refinement could be done with minimal changes to the $SOFT_{REQ}$ relation.

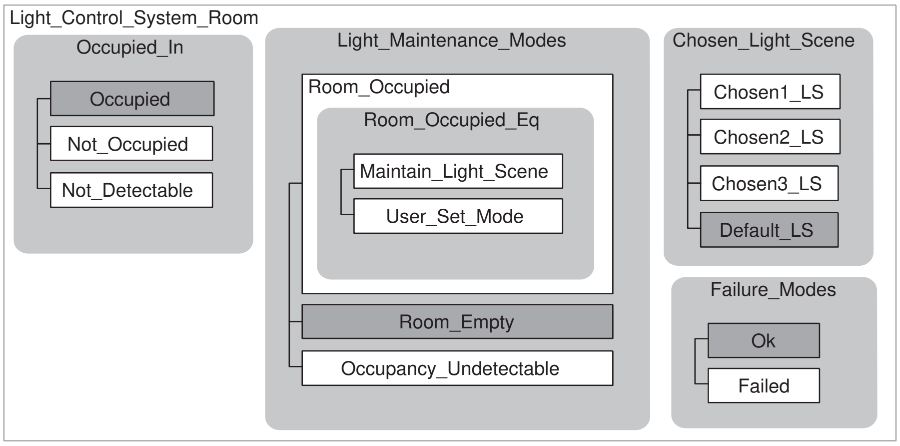

Figure 15: The state machine from [Fig. 6] refined to include the IN^{-1} portions for Occupied

The computation of the occupied quantity is shown in [Fig. 16]. The first two cases determine whether or not the room is occupied based on the value of the motion detector. The last case, the condition to be in *Not_Detectable*, defines the conditions under which there may be a sensor failure or malfunction. If the room was not occupied and the doors have remained closed and then motion is detected, there must be a problem with the sensors.

When attempting to complete this refinement, we discovered conflicts in the problem statement. For instance, consider the following elements from the problem description: (1) "If any motion detector of a room or hallway section does not work correctly, the control system should behave as if the room or hallway section were occupied" (NF4); (2) the motion detector can detect even small motions within the room and it covers the whole room (sensor description); and (3) the occupancy of a room cannot change when the doors are closed (customer feedback 25).

On the surface, these all seem reasonable statements. The first two statements imply that the control software should attempt to detect sensor failures. The last statement says that if the doors in the room are closed, it doesn't matter what the reading from the motion detector is, the occupancy of the room will be unchanged. However, if this were added to, for example, the condition to be in

State Variable

Occupied_In

Location: Light_Control_System_Room

:= Not_Occupied **IF**

Motion_Detected_InVar = FALSE	T

:= Occupied **IF**

PREV_STEP(DoorSensor_InVar = DoorSensorType::kClosed)	*	*
PREV_STEP(..Occupied_In **IN_STATE** Not_Occupied)	T	F
Motion_Detected_InVar = TRUE	T	T
DoorSensor_InVar = DoorSensorType::kClosed	F	*

:= Not_Detectable **IF**

PREV_STEP(DoorSensor_InVar = DoorSensorType::kClosed)	T
PREV_STEP(..Occupied_In **IN_STATE** Not_Occupied)	T
Motion_Detected_InVar = TRUE	T
DoorSensor_InVar = DoorSensorType::kClosed	T

Figure 16: The definition for the Occupied_In *state variable*

Not_Occupied then it would overlap with the condition to be in *Not_Detectable* and an inconsistency is introduced in the model.

6.2.2 Refining Light Level

In the REQ specification, we assume we know the correct level of light in the room (*Light_Level*). In reality, this is not the case; we must add sensing capabilities to determine an approximation of the light level in the room.

The specification states that we should attempt to compute the light level given an outdoor light sensor and the amount of illumination from the two light banks (in a feed-forward type fashion). However, this is very difficult because of several factors:

- In most office buildings, offices are equipped with blinds or curtains. If the user closes the blinds, then little sunlight will enter the room. Even if there are no blinds, if the light coming into the room bothers the occupant he or she will most likely find some way to cover the windows (e.g., by putting up paper). Thus, transmission of light through the glass cannot be assumed to behave according to some set function.
- Users may have desk lamps or other sources of illumination. The light-level algorithm will not be able to account for these alternate light sources.

- The amount of illumination measured by the outdoor light source depends on the angle of the sun relative to the sensor, as well as the intensity of the light. The amount of light actually entering a room depends on the position of the window relative to the sun. All these factors vary with the time of day, weather conditions, and time of year. Providing an accurate calculation of the light level in the room based on the light level at an external sensor would be prohibitively difficult.
- The algorithm must assume that all filaments in the light banks are functioning correctly (i.e., no burned out filaments). This assumption may not be valid.

Because these concerns make the light-level computation error-prone at best, we have chosen to introduce a light sensor into each room or hallway that we monitor. Using this approach, we can directly measure the light level in the room. The monitored variable *Light_Level* can be viewed as a state variable whose value is a scaling function of the light sensor value. For now, we assume that the light sensor is similar to type as described for the outdoor light sensor, so the scaling factor is 1.

In the specification, we introduce a new input variable *Light_Sensor_Level* that records the raw sensor value. Then we (trivially) convert it to the *Light_Level* monitored variable. Although in this case *Light_Sensor_Level* is always the same as *Light_Level*, both are useful, because they decouple the REQ relation from the particular sensors. If we change the sensors, we just have to change the definition of the *Light_Level* variable, without impacting the rest of the model.

Given that we have a light sensor in the room, one problem is that we only know the light level at the location of the light sensor. Therefore, where the light sensor is placed in the room is important. If the light sensor is obscured, or if it is placed very close to one of the light banks, then the light level of the room may be inaccurately measured. Depending on how accurate we required the light level to be, we could create an environment model of the room in which we could move the the light sensor around, then connect it to a RSML^{-e} simulation to investigate the behavior of the system.

6.3 Refining the Simulations

When evaluating RSML^{-e} specifications in NIMBUS, the analyst has great freedom in how he or she models the environment. When we evaluated the REQ specification in [Section 4], we used a user or a software simulations to provide the RSML^{-e} specification with monitored variables and to evaluate the controlled variables. As the IN^{-1} and OUT^{-1} relations are added to the RSML^{-e} specification, the data provided (and consumed) by the model of the embedding environment must be refined to reflect the software inputs and outputs (INPUT and OUTPUT) instead of the monitored and controlled variables (MON and CON).

This can be achieved in two ways; (1) refine the model of the physical process to produce INPUT and consume OUTPUT (incorporate sensors and actuators into the model of the environment), or (2) add explicit and separate models of the sensors and actuators to the simulation. In reality, the refinement of the environmental model and the SOFT relation progress in parallel and is an iterative process. The sensor and actuator models may be added one at a time and the interaction with different components may merit different refinement strategies.

Nimbus naturally allows any combination of the approaches mentioned above to be used.

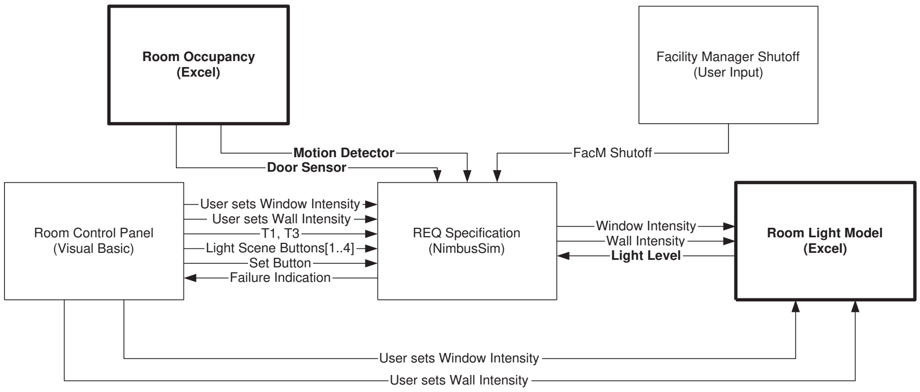

Figure 17: The simulation with the refined notion of occupied

[Fig. 17] shows the refined simulation overview for the light control system. We have replaced the user input for the room occupancy monitored variable with an Excel spreadsheet to model the motion detectors and the doors in the room. This spreadsheet now supplies the required motion detector and door status inputs to the specification. Also, the light model spreadsheet has been refined.

Nimbus provides a flexible framework in which a software specification expressed in RSML^{-e} can be executed while it interacts with various models of the other components in a proposed system. Nimbus supports the refinement of the REQ relation to a SOFT relation by allowing easy interchange of components in the environment. It is important to recognize the difference between models which are good for representing the physical process versus models, like RSML^{-e} specifications, which are good for modeling the software control of the process. Modeling the process itself accurately may require complex numerical functions and simulations. These types of functions are not and should not be within the scope of RSML^{-e}. However, an accurate model of the process is key to the success of specification-based prototyping. In addition, Nimbus provides the abilitiy to perform hardware-in-the-loop simulations. This flexibility allows Nimbus to provide an environment for realistic evaluation of the system.

7 Evaluation and Discussion

In this report we have summarized our experiences with using RSML^{-e} and the Nimbus environment to model the required control behavior of the Light Control System. Our experiences were generally positive and the modeling effort went by without any major complications.

The model was developed by two graduate students over approximately three weeks time (part time). Both students were very familiar with the language, its formal semantics, and the Nimbus environment. Unfortunately, we did not compile any accurate data on the effort required to complete the specification.

The complete formal specification is approximately 50 pages in length. This may seem like a large specification for such a simple problem, but the specification is formatted for readability (a lot of white space and page breaks to make it visually appealing) as well as informal English descriptions of the various parts of the specification.

The main problem we encountered during the specification effort was the incompleteness of the informal requirements provided in the problem description. The formal specification we developed forced resolution of many issues that might otherwise have been passed over until the detailed design or implementation stages. The simulation and execution allowed us to evaluate different panel designs and helped clarify the requirements. In addition, the detailed nature of an RSML^{-e} model forces early adoption of a control strategy. In this effort we found the notion of an external light sensor wholly unacceptable and modified the requirements accordingly. A less detailed modeling approach may defer this issue until later and end up requiring a behavior that cannot be realized in a physical system.

Naturally, the effort exposed some areas where our modeling approach needs improvements.

7.1 Issues for Future Work

The main issue raised during the modeling effort was the lack of an array construct (similar to what is available in Statecharts) in RSML^{-e}. The light scenes are concepts that are naturally modeled as an array of identical models (the three user programmable light scenes and the default light scene are identical). In our specification, however, we had to explicitly include four sets of variables to model the light scenes.

There is no technical reason why arrays have not been included in RSML^{-e}. We originally developed the tool support for RSML^{-e} to prototype and evaluate various static analysis procedures, for instance, completeness and consistency checking, reachability analysis, etc. Since arrays do not add any modeling power—they are simply a syntactic nicety—but add considerable effort when implementing a tool supporting the language, we deemed them superfluous for the more theoretical work we were involved with at the time; to make the tool development easier we omitted arrays from the language. Our subsequent experiences, however, have convinced us that arrays are an absolute necessity in practical modeling and we are currently extending our tool to support arrays.

References

[Faulk *et al.*, 1992] S. Faulk, J. Brackett, P. Ward, and J Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5), September 1992.

[Harel and Pnueli, 1985] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[Harel *et al.*, 1990] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[Harel, 1987] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[Heimdahl and Leveson, 1996] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, pages 363–377, June 1996.

[Heimdahl *et al.*, 1998] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.

[Heitmeyer *et al.*, 1995a] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[Heitmeyer *et al.*, 1995b] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[Heitmeyer *et al.*, 1996] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions of Software Engineering and Methodology*, 5(3):231–261, July 1996.

[Jackson, 1995] Michael Jackson. The world and the machine. In *Proceedings of the 1995 Internation Conference on Software Engineering*, pages 283–292, 1995.

[Jaffe *et al.*, 1991] Matthew S. Jaffe, Nancy G. Leveson, Mats P.E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[Leveson *et al.*, 1994] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, pages 684–706, September 1994.

[Leveson *et al.*, 1999] Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing specification languages for process control systems: Lessons learned and steps to the future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume LNCS 1687, pages 127–145, September 1999.

[Miller, 1999] Steven P. Miller. Modeling software requirements for embedded systems. Technical report, Advanced Technology Center, Rockwell Collins, Inc., 1999. In Progress.

[Parnas and Madey, 1991] David L. Parnas and Jan Madey. Functional documentation for computer systems engineering (volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.

[PD00] The Light Control Case Study: Problem Description, *Journal of Universal Computer Science, Special Issue on Requirements Engineering* (This Volume).

[Thompson and Heimdahl, 1999] Jeffrey M. Thompson and Mats P.E. Heimdahl. An integrated development environment prototyping safety critical systems. In *Tenth IEEE International Workshop on Rapid System Prototyping (RSP) 99*, pages 172–177, June 1999.

[Thompson *et al.*, 1999] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume LNCS 1687, pages 163–179, September 1999.

[Whalen, 2000] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.