Formal Specification of Real-Time Requirements for Building Automation Systems

C. Peper, R. Gotzhein, M. Kronenburg

SFB 501 Report 01/97

Formal Specification of Real-Time Requirements for Building Automation Systems

Christian Peper, Reinhard Gotzhein, Martin Kronenburg

{peper,gotzhein,kronburg}@informatik.uni-kl.de

Report 01/97

Sonderforschungsbereich 501

Computer Science Department
University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany

Formal Specification of Real-Time Requirements for Building Automation Systems

Christian Peper, Reinhard Gotzhein, Martin Kronenburg

SFB 501[1]
University of Kaiserslautern

Abstract

A generic approach to the formal specification of system requirements is presented. It is based on a pool of *requirement patterns*, which are related to *design patterns* well-known in object-oriented software development. The application of such patterns enhances the *reusability* and *genericity* as well as the intelligibility of the formal requirement specification. The approach is instantiated by a tailored *real-time temporal logic* and by selecting *building automation systems* as application domain. With respect to this domain, the pattern discovery and reuse tasks are explained and illustrated, and a set of typical requirement patterns is presented. Finally, the results of a case study where the approach has been applied are summarized.

Keywords: requirements engineering, reuse, formal specification, temporal logic, real time, application, reactive systems

1 Introduction

The specification of requirements is among the first tasks of any system development. The requirements document is part of the contract between the customer and the system developer, and will be the basis for the acceptance of the final implementation. To avoid later disagreements, it is important that the requirements are stated completely and precisely, while still being intelligible for both parties. Generally, both sides are likewise interested in a strict limitation of the bilateral duties.

In practice, requirements are often stated unprecisely - due to the use of natural language - and incompletely - due to the inherent difficulty to perceive all essential aspects of the problem to be solved. This could lead to disagreements during subsequent development stages including the acceptance of the final product by the customer. Therefore, the use of formal description techniques (FDTs) for the specification of requirements is being advocated.

Especially for large and complex systems, the investment to obtain a "good" requirement specification is substantial. To reduce this effort, it may be possible to benefit from earlier system developments by reusing parts of already developed products. While reuse has been well studied for systems' *design* - for instance, by using object-oriented techniques - less research is available

[1] This work was supported by the *Deutsche Forschungsgemeinschaft* (DFG) as part of the *Sonderforschungsbereich* (SFB) 501: "Development of Large Systems with Generic Methods".

on how to apply this principle to *FDT-based requirements engineering*. Reuse has the potential of reducing the effort to specify system requirements. Furthermore, reuse in the requirements phase may have a positive impact on subsequent development stages by an increased reuse of designs and implementations.

In general, a prerequisite for successful reuse is that components and systems to be developed are in some sense "similar". Such similarities may be expected, for instance, if the focus is restricted to a certain application area. In this paper, we address requirements occurring in building automation systems, in particular, real-time requirements.

The reuse of predesigned solutions for recurring design problems is an important topic in object-oriented software development. Recently, *design patterns* [Ga+95] have been advocated as a promising concept, which is related to other well-known approaches such as *frameworks*, or *toolkits*. Different from our method, these approaches are directed towards the design and implementation phases, and are not based on FDTs.

The remainder of the paper is organized as follows. In Section 2, we present our generic approach to the formal specification of requirements in an FDT- and domain-independent way. This approach is then instantiated in Section 3 by selecting a tailored temporal logic as FDT and building automation systems as application domain. With respect to this domain, the pattern discovery and reuse tasks are explained and illustrated, and a set of typical requirement patterns together with pattern instantiations is presented. In Section 4, the results of a case study where the approach has been applied are summarized. Finally, we draw conclusions and give an outlook in Section 5.

2 A Generic Approach to the Formal Specification of Requirements

Genericity is an important general (software) engineering concept applying both to products (requirements, design, implementation) and development processes. Genericity of products is supported by concepts such as compositionality, adaptation, parameterization, and reusability[1]. Genericity of the development process is additionally supported by the concepts of synthesis and generation. In the following, we focus on the generic development of requirement specifications.

2.1 A Requirement Specification Development Model

System development usually starts with a *natural language requirement specification* consisting of an initial set of requirements that is supplied by the customer ("Customer NLRS" for short, see Figure 1). As already mentioned, natural language has no unique semantics. For instance, what is the meaning of "In case of a hazardous condition, the windows must be closed"? Does "must be closed" describe a state or an action? If it is an action, when must it be taken? Therefore, these requirements should be formalized by the system developer, yielding a *formal requirement specification* (FRS).

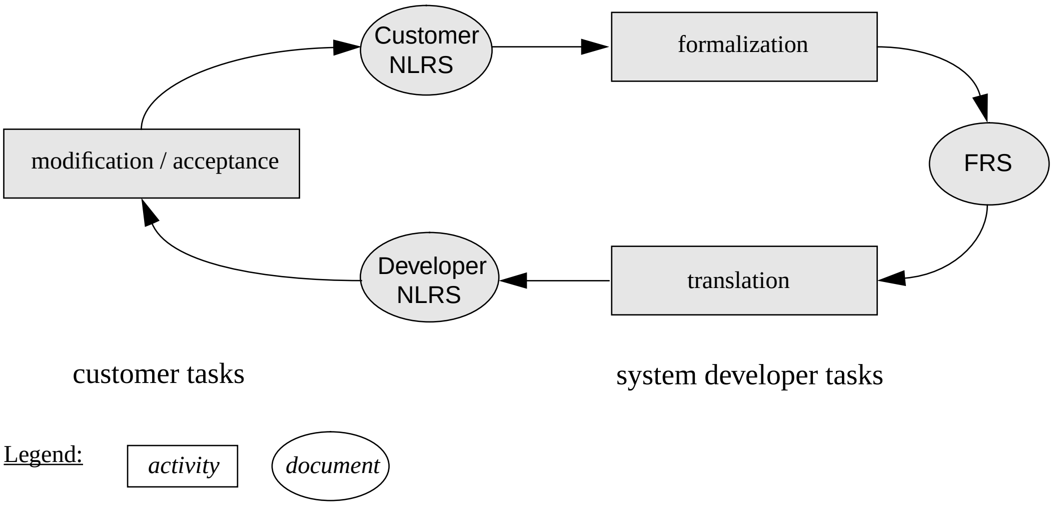
As a result of this formalization, existing ambiguities of the natural language description are resolved in one particular way, which may differ from the original intentions of the customer. Therefore, the customer needs to check whether his intentions are correctly expressed in the FRS. Since the customer may not have a background in FDTs, we propose to translate the FRS back to natural language resulting in a further document called "Developer NLRS" (see Figure 1). Since this natural language description is directly translated from a formal specification, we assume that it is more precise than the original Customer NLRS. The Developer NLRS may now serve as a basis for customer and developer to reach agreement on the system requirements.

[1] These concepts are not orthogonal.

If agreement is reached, the Developer NLRS replaces the previous Customer NLRS and serves as the basis for the acceptance of the final implementation. As another benefit, the new Customer NLRS already has a corresponding formalization, namely the FRS, which can be used as the starting point for subsequent development steps.

If agreement is not reached, the customer supplies a modified Customer NLRS based on the Developer NLRS, and another cycle of the requirement specification development is started. Thus, the development of a requirement specification is an iterative process:

Figure 1: A requirement specification development model



2.2 Reuse of Requirement Patterns

As we have argued, the described approach to the development of requirement specifications has a number of benefits. However, the effort to produce the FRS and its derived natural language description usually is substantial, especially if large and complex systems are to be characterized. To reduce this effort, we propose a *generic approach* to the formalization of requirements. Our approach is based on a *pool of requirement patterns*. By *requirement pattern*, we refer to a generic description of a class of domain-specific requirements. Requirement patterns are related to *design patterns*, a well-known concept of object-oriented software development [Ga+95].

To describe requirement patterns, we propose the format shown in Table 1, called *requirement pattern description template*. Instantiations of this template are termed *requirement patterns*, which, itself instantiated, form the constituents of a requirement specification. The actual contents of the template will depend on the application area and the FDT used to specify patterns and their semantic properties. Criteria for the selection of an appropriate FDT are discussed in Section 2.4. Requirement patterns for a particular application domain and a particular FDT are defined in Section 3.

Table 1: Requirement pattern description template

Name	The name of the requirement pattern
Intention	An informal description of the kind of requirements addressed by this pattern.
Example	An example from the application area illustrating the purpose of the requirement pattern.
Definition	The pattern is described both formally, using a suitable FDT, and in natural language. The formal description is the basis for subsequent development steps finally leading to the requirement specification (pattern selection, adaptation, and composition). The description in natural language will serve the translation of instantiated patterns of the FRS into informal requirements of the NLRS. Furthermore, the description provides some information about possible instantiations.
Semantic properties	Properties that have been formally proven from the pattern. By instantiating these properties in the same way as the requirement pattern, proofs can be reused, too.

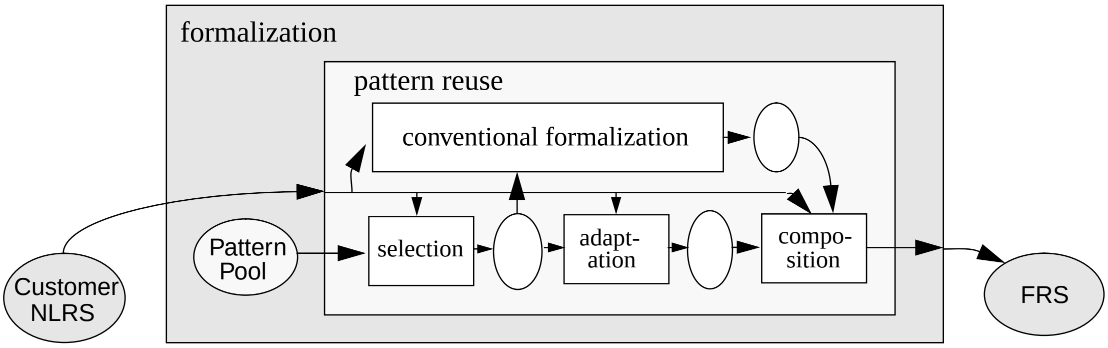
Based on the pattern pool and the Customer NLRS, the formalization of requirements through pattern reuse consists of the following steps (see Figure 2):

Step 1. Requirement patterns are *selected* from the pattern pool. This selection is supported by information provided by pattern descriptions such as intention, definition, and semantic properties.

Step 2. The selected patterns are *adapted* by suitable instantiations. The same kind of adaptation is applied to the semantic properties. Already at this stage is it possible to formally reason about single requirements.

Step 3. The adapted patterns are *composed* to yield the requirement specification. During this composition, it may turn out that there exist conflicts between individual requirements. These conflicts may be resolved, for instance, by exploiting existing precedence relationships between requirements.

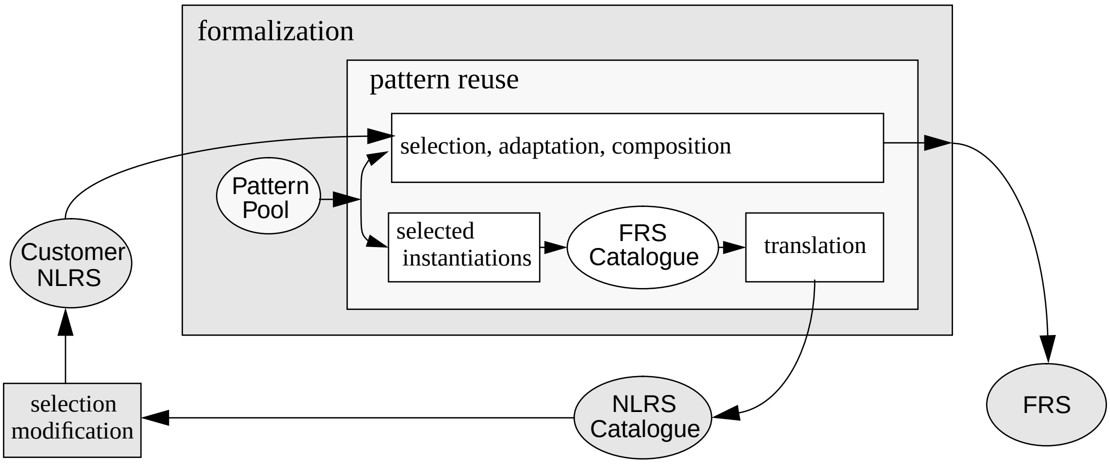
Figure 2: Formalization through *pattern reuse* (cf. Fig. 1)



The degree to which the formalization of requirements can be achieved through pattern reuse depends both on the Customer NLRS and the contents of the pattern pool. If the structure of an informal requirement follows a requirement pattern that is already contained in the pool, then its formalization can be achieved directly by instantiating the pattern. If the structure is different, then either transformations and/or modifications of the informal requirement (cf. Section 2.1) may lead to a structure that is already supported by the pattern pool, or the formalization must be done in the conventional way, i.e. without reuse (see Figure 2). The pool of requirement patterns should evolve over time. As a consequence, the portion of requirements that is developed from requirement patterns will increase, reducing the overall effort of requirement specification.

The existence of a pool of requirement patterns can be exploited further to reduce the effort for the specification of requirements. Based on previous experience, a set of pattern instantiations may be selected, forming a *catalogue* of formal requirements (FRS catalogue, see Figure 3). Translation of these requirements leads to a NLRS catalogue that can be used by customers to state informal requirements during the entire requirements development process. On the one hand, this provides some guidance to customers on how and what requirements may be stated. On the other hand, the formalization of these requirements through pattern reuse, if not yet available, becomes straightforward, since the corresponding pattern is already contained in the pattern pool.

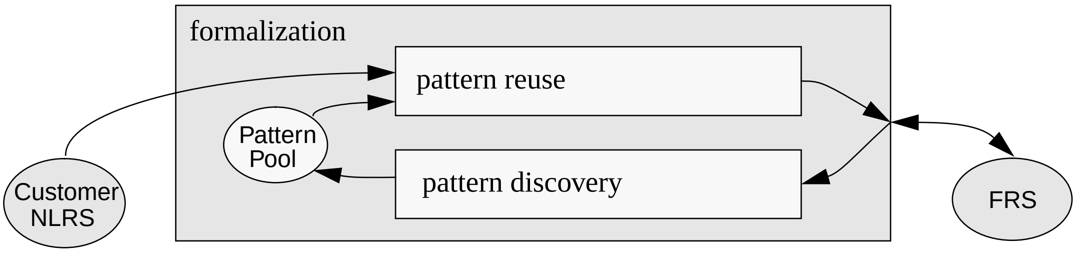
Figure 3: Adding a catalogue



2.3 Discovery of Requirement Patterns

So far, we tacitly assumed the existence of a pattern pool containing a set of already known domain specific patterns, where each entry follows the template defined in Table 1. The main difficulty here is that it is by no means obvious a priori what patterns will be useful later on, as this depends on the application domain as well as on the requirements to be specified. Therefore, building up the pattern pool will be an iterative process itself. This *pattern discovery* task can be modeled as a sub-cycle in the specification development model. Typically, each external specification development cycle triggers one or more internal pattern discovery/reuse cycles affecting both pattern pool and FRS, since each FRS modification can lead to new patterns or the improvement of existing patterns.

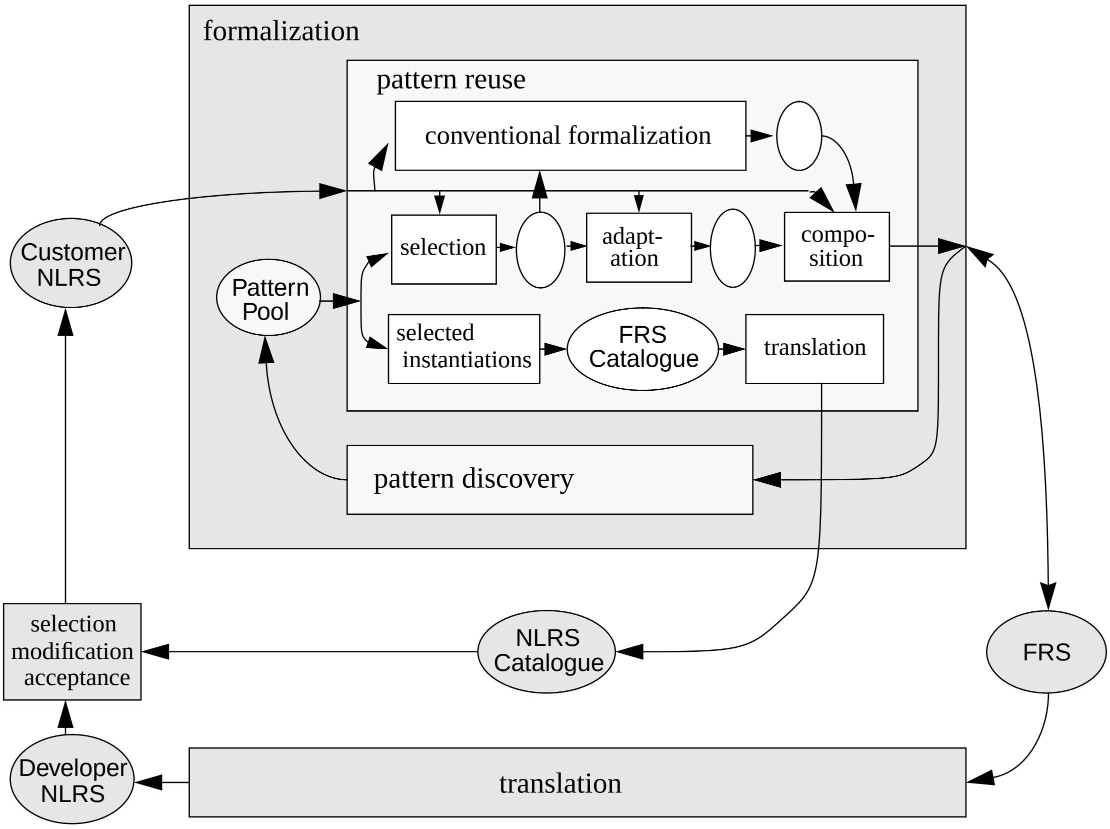
Figure 4: Integration of the pattern discovery task



The discovery of new patterns is a difficult and time-consuming process. In general, many requirements have similarities in the way they restrict time bounds, delays and dependences between system states or other domain specific properties. These similarities can be exploited in order to extract the underlying patterns. Based on a proper FDT, it is often not difficult to find lexically identical or at least similar sub-specifications of requirements. But testing the applicability of a generalized pattern and checking its semantic properties are quite complex tasks, since the meaning of requirements has also to be taken into account. This is supported by the formal semantics of the FDT. In Section 3.3, we report on the discovery of a particular requirement pattern in detail. The discovery of another real-time requirement pattern is presented in [GoKrPe96]. From these examples, it becomes evident that pattern discovery indeed is a substantial investment that will only pay off through extensive pattern reuse.

When we put Figures 1 to 4 together, we obtain the development model shown in Figure 5:

Figure 5: Complete requirement specification development model



2.4 FDT Selection Criteria

The proposed requirement specification development can be facilitated if its subtasks are properly supported by the applied FDT. The following aspects should be considered when selecting a formal technique (see Section 3.1):

The FDT should allow a *domain-specific tailoring* of its expressiveness: on the one hand it must be able to express all necessary functional and non-functional requirements (such as (real-)time in the building automation domain), on the other hand one should not be forced to overspecify in order to express all relevant aspects.

The temporary co-existance of an *incomplete* formal and a non-formal description supports a gradual development of requirements. For this purpose, the FDT's abstraction/refinement mechanism should allow the combination with other description techniques (including natural language).

Furthermore, there are some criteria concerning the pattern pool usage. The selection of a pattern is based either on a natural language input or on a formally specified input. In the first case, a lot of human intelligence is involved in this task. In the second case, the selection could be supported by some kind of syntactical pattern matching. In the same way, the pattern discovery task is based either on semantical or on syntactical similarities in the FRS. Consequently, the FDT syntax should support the development of such *syntactical comparison* operations.

Especially for the adaptation and discovery tasks, the presence of abstraction/refinement and *parameterization* mechanisms is necessary. For the composition task, the *composability* of specifications is essential. The translation task is simplified, if a close correspondance between the informal and the formal requirements can be established. The FDT should therefore have a *property-oriented* style, if the natural language requirements are also given in a property-oriented way.

3 An Instantiation for Building Automation Systems

In this section, we instantiate the generic approach to the formal specification of requirements by selecting a tailored real-time temporal logic as FDT and building automation systems as application domain. In Section 3.1, the tailored real-time temporal logic is sketched. In Section 3.2, the role of the physical environment and its relationship to the requirement specification is highlighted. Section 3.3 then elaborates on pattern discovery. In Section 3.4, a number of requirement patterns are defined. Their reuse is illustrated in Section 3.5.

3.1 The System Requirement FDT

For reasons that are addressed below, we have chosen a tailored real-time temporal logic (tRTTL) as FDT. In this section, we give a short overview of the logic. For further details, in particular, its formal semantics, see Appendix A and [KrGoPe96].

The set $\mathcal{F}$ of correct tRTTL formulae is given by the following formation rules:

1. $\mathcal{P} \subseteq \mathcal{F}$, where $\mathcal{P}$ is the set of propositional atomic formulae

2. Let $\varphi, \psi \in \mathcal{F}$, and $\tau, \tau 1, \tau 2 \in \boldsymbol{R_0}^+$. Then

   - $\neg\varphi, \varphi\wedge\psi, \varphi\vee\psi, \varphi\rightarrow\psi, \varphi\leftrightarrow\psi \in \mathcal{F}$
   - $\Box\varphi, \blacksquare\varphi, \Diamond\varphi, \blacklozenge\varphi, \varphi\boldsymbol{W}\psi, [\varphi] \in \mathcal{F}$
   - $\Box_{\leq\tau}\varphi, \blacksquare_{\leq\tau}\varphi, \Diamond_{\leq\tau}\varphi, \blacklozenge_{\leq\tau}\varphi, \oplus^{\tau 1}{}_{\geq\tau 2} \in \mathcal{F}$

    - $\varphi \Rightarrow_{\leq\tau} \psi$, $\varphi \Leftrightarrow_{\leq\tau} \psi$

3. $\mathcal{F}$ is minimal with 1. and 2.

The informal meaning of the operators is the following:

- $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ are the usual propositional operators (negation, conjunction, disjunction, implication, and equivalence).
- $\Box\varphi$ ("always"): is true, if φ is true now and always in the future. The indexed version $\Box_{\leq\tau}\ \varphi$ is true if φ is true now and during the following τ time units.
- $\blacksquare_{(\leq\tau)}\ \varphi$ ("always in the past"): is true, if φ is true now and has always been true in the past (τ time units)
- $\Diamond_{(\leq\tau)}\ \varphi$ ("eventually"): is true, if φ is true sometimes in the future (τ time units)
- $\blacklozenge_{(\leq\tau)}\ \varphi$ ("sometimes in the past"): is true, if φ has been true sometimes in the past (τ time units)
- $\varphi\ \boldsymbol{W}\ \psi$ ("waiting for"): is true, if φ is true at least until ψ becomes true
- $[\varphi]$ ("action operator"): is true if φ is true now and was false in the preceding state
- $\oplus^{\tau 1}_{\geq \tau 2}\ \varphi$ ("accumulated invariance"): is true, if φ is valid for at least $\tau 2$ time units during the next $\tau 1$ time units
- $\varphi \Rightarrow_{\leq\tau} \psi$ ("delayed implication"): If φ holds permanently for τ time units, ψ holds by then and will hold at least as long as φ.
- $\varphi \Leftrightarrow_{\leq\tau} \psi$ ("delayed equivalence"): If φ holds permanently for τ time units, ψ holds by then and will hold at least as long as φ; analogously for $\neg\varphi$ and $\neg\psi$.

This set of real-time operators is the result of the domain-specific tailoring of the logic's expressiveness, as demanded by the criteria in Section 2.4. The refinement of predicates in terms of another description technique is straightforward. For instance, the predicate *hazardousCondition* could be refined either in natural language ("heavy rain or storm") or in terms of a environment description (*rain > 50 mm/h or wind > 80 km/h*). This may result in conflicts between requirements that become visible only after the refinement. Detection and resolution of conflicts is outside the scope of this paper.

Parameterization of tRTTL formulae is restricted to predicates and time constants. For instance, the specification of the requirement pattern "φ will lead to ψ within τ time units" is parameterized with the formulae φ and ψ and reaction time τ. Composition of requirement patterns can be done by logical conjunction. Tests for syntactical similarity should be feasible. The property-oriented logical description style has turned out to be suitable for the translation into natural language.

3.2 Some remarks on the physical environment

A building automation system acts and reacts in a certain physical environment. A complete description of the tasks of such a reactive system must therefore include some assumptions about the behaviour of the surrounding physics. It is beyond the scope of this report to suggest a proper physical model. Nevertheless, we need to state assumptions about the environment in order to specify the correlations and interactions between the system and its environment.

The temporal logic introduced before is not sufficiently expressive w.r.t. the physical phenomena of the environment, such as temperature or air flow. To describe these phenomena, different formalism including, for instance, differential equations, may be needed in addition. In general, the meaning of predicates that are considered atomic in a temporal logic description can be defined:

1. using the temporal logic itself

   This is the standard concept for refinement and abstraction in a propositional logic. It replaces atomic predicates by more complex formulae. The refined predicate is no longer atomic. It is not possible to leave the restricted world of the logical description in this way.

2. using another description technique

   Since the logical description is interpreted in a certain environment described in another technique, it is necessary to combine both descriptions. The definition of an atomic predicate allows this transition between both models and a specification of correlations. The following environment descriptions typically appear:

   a) in terms of the physical model

      The relationship between the physical model and the logical requirement specification is established by a refinement of the atomic formulae in terms of environmental properties.

   b) in terms of natural language

      It is also possible to postpone the formal definition of an atomic predicate to a later version of the specification. E.g., *hazardousCondition* in Section 4 is not yet defined, but an intuitive meaning is given. Nevertheless, the abstract requirement specification can be given without looking into the precise definition of *hazardousCondition*.

   c) in terms of an existing implementation

      There are often diverse preconditions for an implementation, e.g., a set of already installed sensors that have to be used by the control system provider. This definition option allows to derive the meaning of a predicate from a certain presupposed implementation. E.g., a physical description of "there is no person in the room" would become quite complicated. So, one could pragmatically agree on: *"roomEmpty* iff the motion sensor returns 1".

3.3 Pattern Discovery

In this section, we illustrate the process of pattern discovery. Starting point is an initial Customer NLRS containing statements such as "In the case of hazardous conditions, the windows have to be closed to secure the member's and group's possessions", "Avoidance of damage to windows in regard to weather conditions" and "Close windows in case of possible wind or water damage because of open windows or attempts to open windows".

A first formalization is based on predicates *hazardousCondition* and *windowClosed* modeling the essential states:

- $\Box((\mathit{hazardousCondition} \wedge \neg\, \mathit{windowClosed}) \rightarrow \Diamond_{\leq 30s}\, [\mathit{windowClosed} \vee \neg\, \mathit{hazardousCondition}]\,)$
- $\Box((\mathit{hazardousCondition} \wedge \mathit{windowClosed}) \rightarrow (\mathit{windowClosed}\ \boldsymbol{W}\ \neg \mathit{hazardousCondition}))$

Since this type of state dependences appeared more than once in the first version of the FRS, the patterns underlying these formulae, termed *ConditionalBoundedResponse* and *ConditionalContinuity*, were extracted and inserted in the pattern pool (see Tables 2 and 3; all shaded tables were included in the initial pattern pool). The original usage was kept in the "Example" field. The fields "Intention" and "Semantic properties" of the pattern description template are omitted is some cases for brevity.

Table 2: Conditional Bounded Response

Name	*ConditionalBoundedResponse* (φ, ψ, t)
Example	$\Box((\mathit{hazardousCondition} \wedge \neg\, \mathit{windowClosed}) \rightarrow \Diamond_{\leq 30s}\, [\mathit{windowClosed} \vee \neg\, \mathit{hazardousCondition}]\,)$
	Whenever a hazardous condition is detected, but the window is not closed, then the window will be closed within *30* seconds, or the hazardous condition ceases within this time interval.
Definition	$\Box((\varphi \wedge \neg\psi) \rightarrow \Diamond_{\leq t}\, [\psi \vee \neg\varphi])$
	Whenever φ is true, but ψ is false, then ψ becomes also true within *t* time units, or φ ceases within this time interval.

Table 3: Conditional Continuity

Name	*ConditionalContinuity* (φ, ψ)
Example	$\Box((\mathit{hazardousCondition} \wedge \mathit{windowClosed}) \rightarrow (\mathit{windowClosed}\ \boldsymbol{W}\ \neg \mathit{hazardousCondition}))$
	Whenever a hazardous condition is detected and the window is closed, then the window will remain closed at least as long as the hazardous condition is true.
Definition	$\Box((\varphi \wedge \psi) \rightarrow (\psi\ \boldsymbol{W}\ \neg\varphi))$
	Whenever φ and ψ are both true, then ψ remains true at least as long as φ is true.

Next, it was observed that both patterns were often used together with the same parameter values. This led to another pattern termed *ConditionalBoundedResponseAndContinuity*, formed by the conjunction of the patterns shown in Table 2 and 3:

Table 4: Conditional Bounded Response and Continuity I

Name	*ConditionalBoundedResponseAndContinuity* (φ, ψ, t)
Example	$\square(((\mathit{hazardousCondition} \wedge \neg\mathit{windowClosed}) \rightarrow \lozenge_{\leq 30s}[\mathit{windowClosed} \vee \neg\mathit{hazardousCondition}]) \wedge ((\mathit{hazardousCondition} \wedge \mathit{windowClosed}) \rightarrow (\mathit{windowClosed}\ \boldsymbol{W}\ \neg\mathit{hazardousCondition})))$
	Whenever a hazardous condition is detected, but the window is not closed, the window will be closed within *30 s*, or the hazardous condition ceases within this time interval; and whenever a hazardous condition is detected and the window is closed, then the window will remain closed at least as long as the hazardous condition is true.
Definition	$\square(((\varphi \wedge \neg\psi) \rightarrow \lozenge_{\leq t}[\psi \vee \neg\varphi]) \wedge ((\varphi \wedge \psi) \rightarrow (\psi\ \boldsymbol{W}\ \neg\varphi)))$
	Whenever φ is true, but ψ is false, then ψ becomes also true within t time units, or φ ceases within this time interval; and whenever φ and ψ are both true, then ψ remains true at least as long as φ is true.

This pattern has a considerable syntactical complexity, which makes it difficult to read. Furthermore, it restricts the system behaviour such that it may be impossible to develop implementations in a distributed environment. For instance, if a hazardous condition has just been detected and the window is already closed, it must remain closed. To achieve this, the current value of hazardous condition must be known instantaneously in the corresponding parts of the building automation system, which is a strong limitation.

Therefore, the second version of the previous pattern has reduced system restrictions and a shortened syntax. Nevertheless, the original natural language requirements are not really touched, since their lack of precision leaves enough room for such semantic modifications:

Table 5: Conditional Bounded Response and Continuity II

Name	*ConditionalBoundedResponseAndContinuity* (φ, ψ, t)
Example	$\square(\mathit{hazardousCondition} \rightarrow \lozenge_{\leq 30s}(\mathit{windowClosed} \wedge \mathit{windowClosed}\ \boldsymbol{W}\ \neg\mathit{hazardousCondition} \vee \neg\mathit{hazardousCondition}))$
	Whenever a hazardous condition is detected, then, within *30 s*, the window is closed and will stay closed at least as long as the hazardous condition is true, or the hazardous condition releases.
Definition	$\square(\varphi \rightarrow \lozenge_{\leq t}(\psi \wedge \psi\ \boldsymbol{W}\ \neg\varphi \vee \neg\varphi))$
	Whenever φ is true, then, within t time units, ψ becomes also true and stays true at least as long as φ, or φ releases.

An inspection of the improved pattern's properties produced two results: 1) Suppose φ becomes true and stays so for a longer time period ($>t$). In the time interval t following on this change of

φ, ψ has also to turn to true and it must not fall back before φ - even in this opening time interval. As before, it is not in conflict with the natural language requirement to allow ψ to be in any state during the opening time interval. But afterwards it must be coupled to φ. 2) The new pattern could be expected to be (temporally) transitive, which is not the case for the above definition.

Due to these observations, the pattern definition is modified to allow a "fluttering" of ψ during the mentioned opening time interval and to support the transitivity property. The improved pattern expresses the time delayed implication of two predicates and is therefore termed *Delayed-Implication*:

Table 6: Delayed Implication I

Name	*DelayedImplication* (φ, ψ, *t*)
Example	❑ (*hazardousCondition* → $\Diamond_{\le 30s}$ (*windowClosed* ***W*** ¬*hazardousCondition*))
	Whenever a hazardous condition holds continuously for at least *30 s*, then eventually within this time span, the window is closed and remains closed at least as long as the hazardous condition continues.
Definition	❑ ($\varphi \to \Diamond_{\le t} (\psi\ \boldsymbol{W} \neg\varphi)$)
	Whenever φ holds continuously for at least *t* time units, then eventually within this time span, ψ is true and remains true at least as long as φ.

In a final step, it was found that the premise "$\varphi \to$ " could be removed from the pattern without any semantic changes. Additionally, the operator ➪$_{\le t}$, defined by

$$\varphi ➪_{\le t} \psi \quad =_{\mathrm{Df}} \quad \Diamond_{\le t} (\psi\ \boldsymbol{W} \neg\varphi)$$

was added to the logic and used to abbreviate the modified pattern. The resulting *DelayedImplication* pattern can be used to specify time-displaced dependences between two states:

Table 7: Delayed Implication II

Name	*DelayedImplication* (φ, ψ, *t*)
Intention	ψ is depending on φ with time delay *t*.
Example	❑ *(hazardousCondition* ➪$_{\le 30\ s}$ *windowClosed)*
	Whenever a hazardous condition holds continuously for at least *30 s*, then eventually within this time span, the window is closed and remains closed at least as long as the hazardous condition continues.
Definition	❑ (φ ➪$_{\le t}$ ψ)
	Whenever φ holds continuously for at least *t* time units, then eventually within this time span, ψ is true and remains true at least as long as φ.
Semantic properties	❑ (φ_1 ➪$_{\le t}$ φ_2) ∧ ❑ (φ_2 ➪$_{\le t'}$ φ_3) → ❑ (φ_1 ➪$_{\le t+t'}$ φ_3) ❑ (φ ➪$_{\le t}$ ψ_1) ∧ ❑ (φ ➪$_{\le t}$ ψ_2) ↔ ❑ (φ ➪$_{\le t}$ ($\psi_1 \wedge \psi_2$))

3.4 The Requirement Pattern Pool for Building Automation Systems

In this section, further patterns contained in the initial pool for building automation systems are listed - with the exception of *ConditionalBoundedResponse*, *ConditionalContinuity* and *DelayedImplication*, which have been presented before. These patterns are the result of several case studies.

The invariance pattern is used for the specification of properties that shall hold during the system's running time:

Table 8: Invariance

Name	*Invariance* (φ)
Intention	Allows to specify that a certain property must always hold.
Example	$\Box$ *tempActGtZero* Let *tempActGtZero* represent an indoor temperature greater than $0\ ^{o}C$. Then the indoor temperature is always greater than zero. This formula requires a "no frost" condition (typically to prevent freezing of water pipes, etc.).
Definition	$\Box\ \varphi$ φ is always true.

The delayed equivalence is a bilateral delayed implication with the same time bound t, meaning that ψ is a time displaced copy of φ. For conciseness, operator $\varphi \Leftrightarrow_{\leq t} \psi\ =_{\mathrm{Df}}\quad (\varphi \Rightarrow_{\leq t} \psi) \wedge (\neg\varphi \Rightarrow_{\leq t} \neg\psi)$ was added to the logic.

Table 9: Delayed Equivalence

Name	*DelayedEquivalence* (φ, ψ, *t*)
Intention	ψ is a time displaced copy of the truth value of φ.
Example	$\Box$ (*hazardousCondition* $\wedge$ *windowOpen* $\Leftrightarrow_{\leq 30\ s}$ *warnedUser*) Supposed, the window is only manually operable. Whenever the window is continuously open during a hazardous condition for at least *30 s*, then eventually within this time span, the user is warned and remains warned at least as long as the precondition is true. And conversely, whenever there is a closed window or no hazardous condition for at least *30 s*, then eventually within this time span, the user warning is suppressed and remains suppressed at least as long as this precondition holds.
Definition	$\Box\ (\varphi \Leftrightarrow_{\leq t} \psi)$ Whenever φ holds continuously for at least *t* time units, then eventually within this time span, ψ is true and remains true at least as long as φ. And conversely, whenever φ is continuously false for at least *t* time units, then eventually within this time span, ψ is false and remains false at least as long as φ.

If the validity of the argument φ is only required for a certain time and not for the system's complete running time, this invariance may be limited:

Table 10: Limited Invariance Pattern

Name	*LimitedInvariance (φ, t)*
Intention	Suppression of the "fluttering" of φ, i.e. the fast change of φ's validity. In a certain sense, this pattern is a kind of low pass filter enabling only slow state changes.
Example	$\Box\ ([\mathit{windowOpen}] \rightarrow \Box_{\leq 5\ min}\ \mathit{windowOpen})$
	Each time the window is open, it will stay open for at least *5* minutes. If a lower bound for the close time is given in the same manner, the frequency for window state changes is limited by $1/(2 \cdot 5\ min) = 1.67 \times 10^{-3}\ Hz$.
Definition	$\Box\ ([\varphi] \rightarrow \Box_{\leq t}\ \varphi)$
	Each time φ becomes true, it will stay true for at least t time units.
Semantic properties	$\Box\ ([\varphi \wedge \psi] \rightarrow \Box_{\leq t}\ \varphi \wedge \psi) \rightarrow \Box\ (([\varphi] \rightarrow \Box_{\leq t}\ \varphi) \vee ([\psi] \rightarrow \Box_{\leq t}\ \psi))$ $\Box\ (([\varphi] \rightarrow \Box_{\leq t}\ \varphi) \vee ([\psi] \rightarrow \Box_{\leq t}\ \psi)) \rightarrow \Box\ ([\varphi \vee \psi] \rightarrow \Box_{\leq t}\ (\varphi \vee \psi))$

Another kind of invariance does not request the system to be in a certain state for a continuous time. Instead, it suffices if the accumulated time in this state does not fall short of a given limit.

Table 11: Accumulated Invariance Pattern

Name	*AccumulatedInvariance (φ, T, t)*
Intention	The system must satisfy a property φ at least for a certain time, but the exact points of times are unimportant. Note, that the time *t* could also be replaced by a ratio *t/T* to allow a percental specification.
Example	$\Box\ \bigoplus^{1h}_{\geq 12\ min}\ \mathit{windowOpen}$
	Within any hour the window is open for at least *12* minutes. I.e., the window is open *20%* of the total time to enable sufficient ventilation.
Definition	$\Box\ \bigoplus^{T}_{t}\ \varphi$
	Within any time interval *T*, φ is true for at least *t* time units.

3.5 Pattern Reuse

With the initial requirement pattern pool being available, requirements may now be formalized as described in Section 2.2. This means that given a problem statement of the Customer NLRS, a suitable requirement pattern can be selected from the pool, adapted by setting the pattern parameters, and later composed with further requirements.

As an example, consider the problem statement "If the room is not in use for at least 10 minutes, it must be assured that the doors are locked". An inspection of the pattern pool shows that this statement is close to the *DelayedImplication* (φ, ψ, t) pattern, which is therefore selected. In order to formalize the problem statement, two predicates *roomUsed* and *doorsLocked* are introduced. By suitable naming, we get a close correspondance to the natural language description. Of course, it still remains to be defined how these predicates are related to the physical environment (see Section 3.2). An example may be found in [GoKrPe96], where, starting from the natural language description of the Customer NLRS, a non-trivial formally specified refinement of the predicate *roomUsed* is derived. With the predicates being determined, the requirement pattern can now be adapted by setting the parameters $\varphi = \neg\, roomUsed$, $\psi = doorsLocked$ and $t = 10\ min$, yielding $\Box\, (\neg\, roomUsed \ \Rightarrow_{\leq 10\ min}\ doorsLocked)$.

The translation of the formalized requirement into natural language according to the description of the *DelayedImplication* pattern (see Table 7) yields: "Whenever the room is not used continuously for at least 10 minutes, then eventually within this time span, the doors are locked and remain locked at least as long as the room is not used". Note that this is more precise than the original problem statement. The statement derived from the formalization is then included into the Developer NLRS (see Figure 1), which may now serve as a basis for customer and developer to reach agreement on the system requirements.

As discussed in Section 2.2, the pool of requirement patterns can be further exploited by building up a requirements catalogue. In a first step, the FRS Catalogue is formed by selecting requirement patterns from the pool, and by (partially) instantiating them. For instance, *DelayedImplication* $(\neg roomUsed, doorsLocked, t)$ and *DelayedImplication* $(hazardousCondition, windowsClosed, t)$ may be inserted into the FRS Catalogue. Translation of these formal requirements according to the pattern description (as in the previous example) then leads to the NLRS Catalogue, which can provide some guidance to the customer on what kind of requirements to state, and how to do so. This has the advantage that (some) customer requirements already have a form that is supported by the logical operators, and that can immediately be related to patterns of the pool, thus simplifying the formalization process.

4 Results of a Case Study

The typical starting point for the development of a building automation system comprises a building with some pre-installed (at most partially automized) devices, such as windows and radiators, and a description of the automation system's desired influence on environmental quantities, such as temperature, etc. (usually specified in a natural language).

The starting point for the following formal requirement specification was an informal problem description provided by a project group of the SFB 501. The transition process from the informal to the formal description is illustrated for one selected aspect in [GoKrPe96]. A restricted version of this system contains one sample room for which the system control capabilities had to be fixed precisely, whose complete requirement specification is presented in this section.

4.1 The Physical Environment

In the underlying case study, the use of building automation requirements was restricted to a single room. The most relevant elements of this reference room are:

- two window sections consisting of an upper and a lower sash, respectively
  The upper sash can be automatically driven by an actuator, the lower sash is only manually operable.
- a radiator for each window section

Due to this restriction, the formalization of the physical world may be reduced to a few variables and functions. These variables and functions represent the system's view of the environment. The types Temp and Time can be regarded as real numbers.:

- Temp temp_{Ext}
  the current external (outdoor) temperature
- Temp temp_{Act}
  the current room temperature
- Temp temp_{Rad}
  the current radiator temperature
- deltaTime : Temp x Temp → Time
  the time deltaTime (temp1, temp2) to heat up the room from temperature temp1 to temp2.

Additionally, there are also some variables that are directly influenced by the user. They can be regarded as system input:

- Temp $\text{temp}_{\text{Target}}$: The currently valid target room temperature.
- Temp $\text{temp}_{\text{Comfort}}$, $\text{temp}_{\text{Standby}}$, temp_{Off} : The target room temperature settings for the system states *comfortMode*, *standByMode* and *offMode*.
- $\delta_{\text{Target+}}$: The currenltly valid upper deviation from $\text{temp}_{\text{Target}}$.
- $\delta_{\text{Comfort+}}\delta_{\text{Standby+}}\ \delta_{\text{Off+}}$
- $\delta_{\text{Target-}}$: The currently valid lower deviation from $\text{temp}_{\text{Target}}$.
- $\delta_{\text{Comfort-}}\delta_{\text{Standby-}}\delta_{\text{Off-}}$
- In terms of the physical model some temperature intervals can be defined:

$$\begin{aligned}
&\text{comfortInterval} &&=_{\text{Df}} && [\text{temp}_{\text{Comfort}} - \delta_{\text{Comfort-}}, \text{temp}_{\text{Comfort}} + \delta_{\text{Comfort+}}]\\
&\text{standByInterval} &&=_{\text{Df}} && [\text{temp}_{\text{Standby}} - \delta_{\text{Standby-}}, \text{temp}_{\text{Standby}} + \delta_{\text{Standby+}}]\\
&\text{offInterval} &&=_{\text{Df}} && [\text{temp}_{\text{Off}} - \delta_{\text{Off-}}, \text{temp}_{\text{Off}} + \delta_{\text{Off+}}]
\end{aligned}$$

4.2 Predicates

Most of the atomic predicates (either true or false) for the requirement specification refer to certain states in the environment of the system:

$\mathcal{P}$ = {*extGtActGtTarget, extLtActLtTarget, extLtTargetLtAct, hazardousCondition, heatingPeriod, heatUpPossStandbyComfort, lowerSashClosed, personExpected, roomEmpty, targetEqComfort, targetEqOff, targetEqStandby, tempActGtZero, tempIntervalInclusion, tempRadGtZero, upperSashClosed, upperSashManualClosed, upperSashManualOpen, upperSashOpen, valveControl, warnedUserLowerSash* }

We start with the definition of the propositional predicates in terms of this physical model:

- *extGtActGtTarget* iff $\text{temp}_{\text{Ext}} > \text{temp}_{\text{Act}} > \text{temp}_{\text{Target}}$
- *extGtTargetGtAct* iff $\text{temp}_{\text{Ext}} > \text{temp}_{\text{Target}} > \text{temp}_{\text{Act}}$
- *extLtActLtTarget* iff $\text{temp}_{\text{Ext}} < \text{temp}_{\text{Act}} < \text{temp}_{\text{Target}}$
- *extLtTargetLtAct* iff $\text{temp}_{\text{Ext}} < \text{temp}_{\text{Target}} < \text{temp}_{\text{Act}}$

  The four previously defined predicates respresent certain relations between the current room temperature, the current external temperature and the target room temperature.
- *targetEqComfort* iff
  $\text{temp}_{\text{Target}} = \text{temp}_{\text{Comfort}} \wedge \delta_{\text{Target+}} = \delta_{\text{Comfort+}} \wedge \delta_{\text{Target-}} = \delta_{\text{Comfort-}}$
- *targetEqOff* iff
  $\text{temp}_{\text{Target}} = \text{temp}_{\text{Off}} \wedge \delta_{\text{Target+}} = \delta_{\text{Off+}} \wedge \delta_{\text{Target-}} = \delta_{\text{Off-}}$
- *targetEqStandby* iff
  $\text{temp}_{\text{Target}} = \text{temp}_{\text{Standby}} \wedge \delta_{\text{Target+}} = \delta_{\text{Standby+}} \wedge \delta_{\text{Target-}} = \delta_{\text{Standby-}}$

  The three preceding definitions determine the temperature bounds for the system states *offMode, standbyMode* and *comfortMode*.
- *valveControl* iff $\text{temp}_{\text{Act}} \in [\text{temp}_{\text{Target}}-\delta_{\text{Target-}}, \text{temp}_{\text{Target}}+\delta_{\text{Target+}}]$

  The valve control is regulating the current room temperature in the specified range.
- *heatUpPossStandbyComfort* iff

  deltaTime $(\text{temp}_{\text{Standby}}, \text{temp}_{\text{Comfort}}) \leq$ timeHeatUpStandbyComfort

  It is possible to haet up the room from $\text{temp}_{\text{Standby}}$ to $\text{temp}_{\text{Comfort}}$ in timeHeatUpStandby-Comfort time units (or faster).
- *tempActGtZero* iff $\text{temp}_{\text{Act}} > 0$
- *tempRadGtZero* iff $\text{temp}_{\text{Rad}} > 0$

  The "no frost" condition for the room and especially the radiator.
- *tempIntervalInclusion* iff comfortInterval $\subseteq$ standbyInterval $\subseteq$ offInterval
  The user setttings for the respective temperature ranges are tested for consistency.

The subsequent predicates are given in natural language. They are partially candidates for a later physical refinement:

- *hazardousCondition* iff "a dangerous situation caused by storm and/or heavy rain"
- *heatingPeriod* iff "the current time die aktuelle Zeit liegt innerhalb der Heizperiode
- *lowerSashClosed* iff "the lower window sash is closed"
- *roomEmpty* iff "there is no person in the room"
- *upperSashClosed* iff "the upper sash is closed"
- *upperSashOpen* iff "the upper sash is open"

There are also some candidates for a possible implementational refinment:

- *upperSashManualClosed* iff "the user has manually closed the upper sash"

- *upperSashManualOpen* iff "the user has manually opened the upper sash"

  "Manual" operation means only overriding the systems settings, i.e., the system is informed by a user that the sashs should be closed.
- *warnedUserLowerSash* iff "the user is receiving a warning that the upper sash must be closed"
- *personExpected* iff "a person is expected in the room"

  A later definition could be derived from a typical user behaviour as far as predictable by the control system.

Besides the presented atomic predicates some derived predicates turn out as helpful in the specification. As mentioned they are defined as TL formulae::

- *offMode* iff $\neg$ *personExpected* $\wedge \neg$ *roomUsed*
- *standByMode* iff *personExpected* $\wedge \neg$ *roomUsed*
- *comfortMode* iff *roomUsed*
- *roomUsed* iff

  $\blacksquare_{\leq T1} \neg \mathit{roomEmpty} \vee \langle * [\blacksquare_{\leq T1+T2} \neg \mathit{roomEmpty}] \Rightarrow \rangle \blacksquare \blacklozenge_{\leq T3} \neg \mathit{roomEmpty}$

  This definition of roomUsed requires at least one of the following conditions:

  - there was permanently a person in the room during the last *T1* time units or
  - since the last time when there was a person permanently in the room for at least *T1+T2* time units, the room was never empty for more than *T3* time units.

4.3 Behaviour Requirements

Based on the set $\mathcal{P}$ of atomic predicates the complete system description *Req* can be composed as a conjunction of 23 specific requirement formulae:

$$Req = \bigwedge_{i=1,\ldots,23} R.i$$

Each of these properties below is extended by a translation of the logic expression to its (less precise) natural language representation. These comments significantly improve the intelligibilty of the formal specification. Wherever a requirement is given by a pattern instantiation, its translation was directly derived from the pattern description’s translation:

(R.1) $\Box \bigoplus^{\mathit{timeVentilation1}}_{\geq \mathit{timeVentilation2}} \mathit{upperSashOpen}$

Within any time interval *timeVentilation1* the upper sash is open for at least *timeVentilation2* time units.

(R.2) $\Box$ *tempRadGtZero*

The current radiator temperature is always greater than 0^{o} C
(to prevent freezing of water pipes).

(R.3) $\Box$ *tempActGtZero*

The current room temperature is always greater than 0^{o} C.

(R.4) $\Box$ (*offMode* $\rightarrow$ *valveControl*)

Whenever the system is in state *offMode* the current room temperature is kept in the range $[\mathrm{temp}_{\mathrm{Off}} - \delta_{\mathrm{Off-}}; \mathrm{temp}_{\mathrm{Off}} + \delta_{\mathrm{Off+}}]$.

(R.5) ❑ (*standByMode* → *heatUpPossStandbyComfort*)

Whenever the system is in *standByMode*, the potential change from standby temperature to comfort temperature takes not more than $\mathrm{timeHeatUp}_{\mathrm{StandbyComfort}}$ time units.

(R.6) ❑ $(\mathit{standByMode} \Rightarrow_{\leq \mathit{timeHeatUpOffStandby}} \mathit{valveControl})$

Whenever the system is continously in *standByMode* for at least *timeHeatUpOffStandby* time units, then eventually within this time span, the current room temperature $\mathrm{temp}_{\mathrm{Act}}$ is in the range $[\mathrm{temp}_{\mathrm{Target}}\text{-}\delta_{\mathrm{Target-}}, \mathrm{temp}_{\mathrm{Target}}+\delta_{\mathrm{Target+}}]$ and remains so at least as long as the system is in *standByMode*.

(R.7) ❑ $((\mathit{heatingPeriod} \wedge \mathit{comfortMode}) \Rightarrow_{\leq \mathit{timeHeatUp}} \mathit{valveControl})$

Whenever the system is continously in *comfortMode* during the *heatingPeriod* for at least *timeHeatUp* time units, then eventually within this time span, the current room temperature is regulated and remains regulated in the proper range at least as long as the precondition is true.

(R.8) ❑ (*offMode* → *targetEqOff*)

(R.9) ❑ (*standByMode* → *targetEqStandby*)

(R.10) ❑ (*comfortMode* → *targetEqComfort*)

Whenever the system is in *offMode* (*standByMode, comfortMode*) the target temperature range is set to the predefined settings for *offMode* (*standByMode, comfortMode*).

(R.11) ❑ $(\mathit{hazardousCondition} \Rightarrow_{\leq \mathit{timeSafety}} \mathit{upperSashClosed})$

Whenever a hazardousCondition arises, the upper sashs are closed within *timeSafety* time units.

(R.12) ❑ $(\mathit{hazardousCondition} \wedge \neg\, \mathit{lowerSashClosed} \Leftrightarrow_{\leq \mathit{timeSafety}} \mathit{warnedUserLowerSash})$

Whenever the lower sash is continously open during a hazardous condition for at least *timeSafety* time units, then eventually within this time span, the user is warned and remains warned at least as long as the precondition is true. And conversely, whenever there is a closed lower sash or no hazardous condition for at least *30 s*, then eventually within this time span, the user warning is suppressed and remains suppressed at least as long as the new precondition holds.

(R.13) ❑ *tempIntervalInclusion*

The three possible temperature intervals are always restricted in the following way: the *offMode* interval includes the *standByMode* interval that includes the *comfortMode* interval.

(R.14) ❑ $(\neg\, \mathit{roomUsed} \rightarrow \neg\, \mathit{upperSashManualClosed})$

(R.15) ❑ $(\neg\, \mathit{roomUsed} \rightarrow \neg\, \mathit{upperSashManualOpen})$

If the room is not used any "manual operation" of the upper sash is not accepted.

(R.16) ❑ $(\mathit{upperSashManualOpen} \Rightarrow_{\leq \mathit{timeOpen}} \mathit{upperSashOpen})$

(R.17) ❑ $(\mathit{upperSashManualClosed} \Rightarrow_{\leq \mathit{timeClosed}} \mathit{upperSashClosed})$

Whenever the user continously demands that the upper sashs should be open (close) for at least *timeOpen (timeClosed)* time units, then eventually within this time span, the sashs are open (close) and remain pen (close) at least as long as the user command is valid.

(R.18) ❑ (*extGtTargetGtAct* ➪$_{\leq timeEnergy}$ *upperSashOpen*)

Whenever the current room temperature is lower than the external temperature and target temperature for at least *timeEnergy* time units, then eventually within this time span, the upper sashs are open (to heat the room)and remain open at least as long as this precondition is true.

(R.19) ❑ (*extGtActGtTarget* ➪$_{\leq timeEnergy}$ *upperSashClosed*)

Whenever the current room temperature is lower than the external temperature but higher than the target temperature for at least *timeEnergy* time units, then eventually within this time span, the upper sashs are close (to avoid a heat-up) and remain close at least as long as this precondition is true.

(R.20) ❑ (*extLtTargetLtAct* ➪$_{\leq timeEnergy}$ *upperSashOpen*)

Whenever the current room temperature is higher than the external and higher than the target temperature for at least *timeEnergy* time units, then eventually within this time span, the upper sashs are open and remain open at least as long as this precondition is true.

(R.21) ❑ (*extLtActLtTarget* ➪$_{\leq timeEnergy}$ *upperSashClosed*)

Whenever the current room temperature is higher than the external but lower than the target temperature for at least *timeEnergy* time units, then eventually within this time span, the sashs are close and remain close at least as long as this precondition is true.

(R.22) ❑ ([*upperSashClosed*] → ❑$_{\leq timeConstant}$ ¬ *upperSashClosed*)

(R.23) ❑ ([*upperSashOpen*] → ❑$_{\leq timeConstant}$ ¬ *upperSashOpen*)

Each time the upper sashs are closed (open), they will stay closed (open) for at least *timeConstant* time units.

4.4 Discussion

For the assessment of the pattern reusability, the respective usage frequencies for the specification of the case study are listed in

Table 12: Pattern usage frequencies

Pattern Name	Syntactical Definition	appears in	freq.
Invariance	❑ φ	R.2-5, R.8-10, R.13-15	10
DelayedImplication	❑ (φ ➪$_{\leq t}$ ψ)	R.6, R.7, R.16-21	9
LimitedInvariance	❑ ([φ] → ❑$_{\leq t}$ φ)	R.22, R.23	2
AccumulatedInvariance	❑ $\bigoplus^{T}_{t}$ φ	R.1	1
DelayedEquivalence	❑ (φ ⇔$_{\leq t}$ ψ)	R.12	1
ConditionalBoundedResponse	❑ ((φ∧¬ψ) → ◊$_{\leq t}$ [ψ∨¬φ])	-	0
ConditionalContinuity	❑ ((φ∧ψ) → (ψ ***W*** ¬φ))	-	0

As the table shows, the invariance pattern is the one mostly used. Note that all patterns of the initial pool can be seen as instantiations of the invariance pattern. However, this is not considered in the count. Interestingly, the delayed implication pattern with its more complex semantics is applied quite often. This indicates that this pattern expresses very elementary relations among system states. Therefore, it can be expected that this pattern could also be reused in other application domains. The *ConditionalBoundedResponse* and *ConditionalContinuity* patterns are not used at all. The reason is that the requirements where these patterns are applicable have all been expressed using the *DelayedImplication* pattern.

Looking back at the case study, most of the definitions of the atomic predicates are performed by references to certain physical situations. The resulting list of interactions between the control system and its environment leads straightforwardly to a number of necessary sensors and actuators to implement these interaction facilities (see Table 13). So far, the notion "sensor" could still mean a complex system of several physical sensors to increase reliability or to compute an average value to reduce potential measuring faults.

Table 13: Necessary sensor equipment

atomic predicate(s)	physical variables	sensors
roomEmpty		motion detector
extGtActGtTarget, ExtGtTargetGtAct, ExtLtActLtTarget, ExtLtTargetLtAct, tempActGtZero	temp_{Act}	room air temperature sensor
tempRadGtZero	temp_{Rad}	radiator temperature sensor
upperSashOpen, *upperSashManualOpen*		upper sash open contact
upperSashClosed		upper sash closed contact
lowerSashClosed		lower sash closed contact
extGtActGtTarget, ExtGtTargetGtAct, ExtLtActLtTarget, ExtLtTargetLtAct	temp_{Ext}	outdoor air temperature sensor
hazardousCondition		wind speed sensor
		wind direction sensor
		rain sensor

In some cases it is now possible to replace the intuitive predicate definition directly by the results of a sensor measurement, thus postulating a certain implementation equipment.

5 Conclusion and Outlook

We have presented a generic, pattern-based approach to the formal specification of system requirements. Starting from a pool of requirement patterns, patterns are selected, adapted and composed to obtain a formal requirement specification. The approach has been instantiated by using a tailored real-time temporal logic as FDT, and choosing building automation as application domain. A set of patterns and pattern instantiations, most of them stating real-time properties, is presented, and the process of pattern discovery is illustrated. Furthermore, results of a case study are listed and discussed. During our work, we have made the following observations:

- In our case study, all requirements have been formalized by pattern instantiations.
- Pattern instantiation may be understood as an incremental process. For instance, all requirement patterns presented in this paper are instantiations of the *Invariance* pattern. Also, partial instantiations are possible, resulting in less generic requirement patterns.
- By translating formalized requirements into natural language, a better basis for discussions with the customer was achieved, as this translation could be performed in a uniform way.
- The discovery of "good" requirement patterns is a very time consuming task. However, this seems not to be unusual, since the same experience has been made in different contexts, too.
- The pattern-based approach is scalable in the sense that more patterns can be added to the pool when needed.
- The pattern-based development of requirement specifications can lead to a substantial degree of reuse. With a set of "good" patterns being available, the formalization of matching requirements is straightforward, reducing the overall effort and leading to improved readability.

An important aspect, which is not addressed in this paper, is the detection and resolution of conflicts between requirements. Such conflicts may lead to inconsistencies when requirements are composed, impeding the development of a correct implementation. We are currently investigating criteria in order to detect conflicts, and methods in order to resolve them.

We expect that the pattern-based formalization of requirements may lead to an increased reuse of design decisions and solutions in subsequent development stages, as far as these decisions can be related to the application of particular requirement patterns. Also, it may lead to a better traceability of the consequences when modifying requirements of an already developed system. In [GeGoRö97], we make a step in this direction by tailoring communication protocols based on SDL patterns. We see this as a fertile field for future research.

Acknowledgements. We thank the members of project D1 and Team 2 of the SFB 501, who were involved in the requirement collection and improvement.

References

[AlHe91] R. Alur, T.A. Henzinger: *Logics and Models of Real Time: A Survey* in J.W. de Bakker, C. Huizing, W.P. de Roever, G. Rozenberg (eds.), Real-Time: Theory and Practice, LNCS 600, 1991

[Ga+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[GeGoRö97] B. Geppert, R. Gotzhein, F. Rößler: *Configuring Communication Protocols Using SDL Patterns*, accepted for the 8th SDL Forum, Paris, France, September 1997

[Go93] R. Gotzhein: *Open Distributed Systems - On Concepts, Methods, and Design from a Logical Point of View*, Vieweg, 1993

[Go+96] R. Gotzhein, B. Geppert, C. Peper, F. Rößler: *Generic Layout of Communication Subsystems - A Case Study,* SFB 501 Report 14/96, University of Kaiserslautern, Germany, 1996

[GoKrPe96] R. Gotzhein, M. Kronenburg, C. Peper: *Specifying and Reasoning about Generic Real-Time Requirements,* SFB 501, Technical report 15/96, University of Kaiserslautern, Germany, 1996

[KrGoPe96] M. Kronenburg, R. Gotzhein, C. Peper: *A Tailored Real-Time Temporal Logic for Building Automation Systems,* SFB 501, Technical report 16/96, University of Kaiserslautern, Germany, 1996

Appendix A: The Semantics of tRTTL

To define the semantics of formulae of tRTTL, a model for real-time systems that is based on the one proposed in [AlHe91] is used:

Definition: *(State, timed state sequence, model)*

Let $\mathcal{P}$ be the set of all atomic formulae.

1. A *state* is a function $\sigma\colon \mathcal{P} \rightarrow \{0, 1\}$. The set of all states is denoted as Σ.

2. A *timed state sequence* ρ is a function $\rho\colon \mathbf{R}_0^+ \rightarrow \Sigma$ such there exists an interval sequence $I = I_0, I_1, \ldots$ with

   a) $\forall\, i, i \in \mathbf{N}\colon I_i = [a_i, b_i)$ with $a_i \in \mathbf{R}_0^+$, $b_i \in \mathbf{R}^+ \cup \{\infty\}$, $a_i < b_i$

   b) $\forall\, i, i \in \mathbf{N}\colon$ if $b_i \neq \infty$ then $b_i = a_{i+1}$

   c) $\forall\, i, i \in \mathbf{N}\colon \forall\, t, \forall\, t', t, t' \in I_i\colon \rho(t) = \rho(t')$

   d) $\forall\, t, t \in \mathbf{R}_0^+\colon \exists\, i, i \in \mathbf{N}\colon t \in I_i$

   e) If ρ is not constant from any point in time, i.e.: $\forall\, t_1, t_1 \in \mathbf{R}_0^+\colon \exists\, t_1', t_1' \in \mathbf{R}_0^+$, $t_1' > t_1\colon \rho(t_1) \neq \rho(t_1')$, then: $\forall\, i, i \in \mathbf{N}\colon \forall\, t_2, t_2 \in I_i\colon \forall\, t_2', t_2' \in I_{i+1}\colon \rho(t_2) \neq \rho(t_2')$

   f) If ρ is constant from a point in time, i.e.: $\exists\, t_1, t_1 \in \mathbf{R}_0^+\colon \forall\, t_1', t_1' \in \mathbf{R}_0^+, t_1' \geq t_1\colon$

$\rho(t_1) = \rho(t_1')$, then: $\exists\, n \in \mathbf{N}: \mathcal{I} = I_0, I_1, \ldots, I_n$ and $\forall\, i, i \in \{0,\ldots,n-1\}: \forall\, t_2, t_2 \in I_i: \forall\, t_2', t_2' \in I_{i+1}: \rho(t_2) \neq \rho(t_2')$

Such an interval sequence $\mathcal{I}$ is called *compatible with* ρ.

3. A *model* $\mathcal{M}$ is a set of timed state sequences.

Condition a) excludes *singular* intervals, i.e. intervals of type $[c,c]$, and other kinds of intervals, e.g. (a_i,b_i); b) guarantees that two neighboring intervals I_i and I_{i+1} are *adjacent*; c) guarantees that the state is invariant during each single interval I_i ; Condition d) (together with c)) excludes *Zeno-sequences* of states, i.e. an infinite number of different states during a finite period of time is not allowed. Conditions e) and f) guarantee that each interval I_i of the sequence $\mathcal{I}$ is a *maximum* interval in the sense that it ends, if and only if the state changes. Due to these two conditions, there is for each timed state sequence at most one interval sequence $\mathcal{I}$ fulfilling a) to f). Note that propositional formulae have the same truth value during an interval I_i of $\mathcal{I}$.

Definition: *(Semantics of tRTTL)*

Let $\mathcal{M}$ be a model, $\rho \in \mathcal{M}$ be a timed state sequence, and $\mathcal{I} = I_0, I_1, \ldots$ be an interval sequence compatible with ρ and $I_i=[a_i, b_i)$. Further, let $\varphi, \psi \in \mathcal{F}$, and let τ, T, t, t', t' range over $\boldsymbol{R_0}^+$. Then the satisfaction relation |= is defined as follows:

1. $(\rho, t) \models \varphi$ iff $\rho(t)(\varphi) = 1$ if $\varphi \in \mathcal{P}$
2. $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ are interpreted as usual.
3. $(\rho, t) \models [\varphi]$ iff $(t=0$ and $(\rho,0) \models \varphi)$ or $(t>0$ and $(\rho, t) \models \varphi$ and $\exists\, t', 0 \leq t'< t: \forall\, t'', t'\leq t''< t: (\rho, t'') \models \neg\varphi)$
4. $(\rho, t) \models \Box\varphi$ iff $\forall\, t', t' \geq t: (\rho, t') \models \varphi$
5. $(\rho, t) \models \Box_{\leq\tau}\, \varphi$ iff $\forall\, t', t \leq t'\leq t+\tau: (\rho, t') \models \varphi$
6. $(\rho, t) \models \blacksquare\varphi$ iff $\forall\, t', 0 \leq t'\leq t: (\rho, t') \models \varphi$
7. $(\rho, t) \models \blacksquare_{\leq\tau}\, \varphi$ iff $\forall\, t', t_{low} \leq t'\leq t: (\rho, t') \models \varphi$ and $t_{low} = \max\,\{0,t-\tau\}$
8. $(\rho, t) \models \varphi\, \boldsymbol{W}\, \psi$ iff $(\rho, t) \models \Box\varphi$ or $(\exists\, t', t' \geq t: (\rho, t') \models \psi$ and $\forall\, t'', t \leq t''<t': (\rho, t'') \models \varphi)$
9. $(\rho, t) \models \oplus^{T}_{\geq\tau}\, \varphi$ iff $\sum_{j \in J} (\min\,(b_j, t+T) - \max\,(a_j,t)) \geq \tau$ with $J=\{j \in \mathbf{N} \mid t \in I'=[a',b'), a' \leq a_j \leq t+T, (\rho, a_j) \models \varphi\}$
10. $\Diamond\varphi$ $=_{\mathrm{Df}} \neg \Box \neg\varphi$
11. $\Diamond_{\leq\tau}\, \varphi$ $=_{\mathrm{Df}} \neg \Box_{\leq\tau} \neg\varphi$
12. $\blacklozenge\varphi$ $=_{\mathrm{Df}} \neg \blacksquare \neg\varphi$
13. $\blacklozenge_{\leq\tau}\, \varphi$ $=_{\mathrm{Df}} \neg \blacksquare_{\leq\tau} \neg\varphi$
14. $\varphi \Rightarrow_{\leq\tau} \psi$ $=_{\mathrm{Df}} \Diamond_{\leq\tau} (\psi\, \boldsymbol{W} \neg\varphi)$
15. $\varphi \Leftrightarrow_{\leq t} \psi$ $=_{\mathrm{Df}} (\varphi \Rightarrow_{\leq \mathrm{t}} \psi) \wedge (\neg\varphi \Rightarrow_{\leq \mathrm{t}} \neg\psi)$
16. $\models_{\mathrm{i}} \varphi$ iff $\forall \mathcal{M}: \forall\, \rho \in \mathcal{M}: (\rho, a_0) \models \varphi$ (initial validity)