A Generic Approach to the Formal Specification of Requirements [1]

Christian Peper, Reinhard Gotzhein, Martin Kronenburg
University of Kaiserslautern
67653 Kaiserslautern, Germany
{peper,gotzhein,kronburg}@informatik.uni-kl.de

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. The approach has the potential of reducing the effort to formally specify system requirements.*

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 be 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 (especially for safety critical systems) is advocated since more than one decade, e.g. [10]. Approaches based on the *Four-Variable Model* [12] like SCR [9] or CoRE [3] specify the system's behaviour mainly as a relation *REQ* between *controlled* and *monitored* environment variables. This relation is also an implicit part of the logic based specifications below. In this paper, we address a problem common to these and other requirement specification methods:

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 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. In [4], *design patterns* have been advocated as a promising concept, which is related to other 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.

In Section 2, we present our generic approach to the formal specification of requirements in an FDT- and domain-independent way. This approach is 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. Finally, conclusions and outlook are presented in Section 4.

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

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 (these concepts are not orthogonal). 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

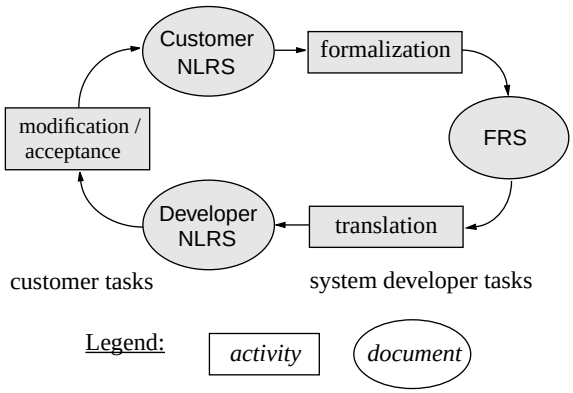
System development usually starts with a *natural language requirement specification* consisting of an initial set of requirements that is supplied by the customer ("Customer NLRS" for short, see Figure 1). As already mentioned, natural language has no unique semantics. For instance, what is the meaning 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.

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 [4].

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. 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.

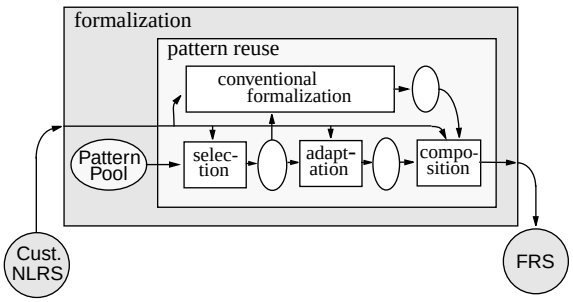
Table 1: Requirement pattern description template

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):

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.
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.
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.

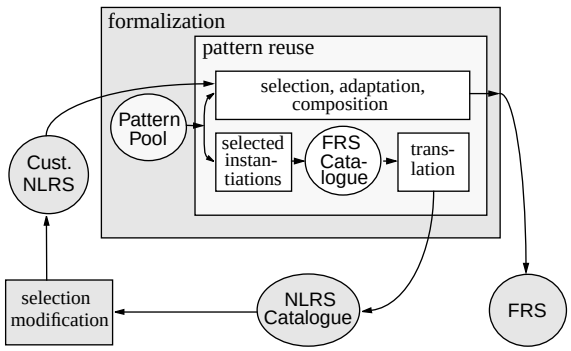
Figure 2: Formalization through *pattern reuse* (cf. Fig. 1)



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 selection 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

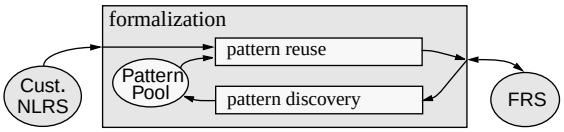


2.3 Discovery of requirement patterns

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



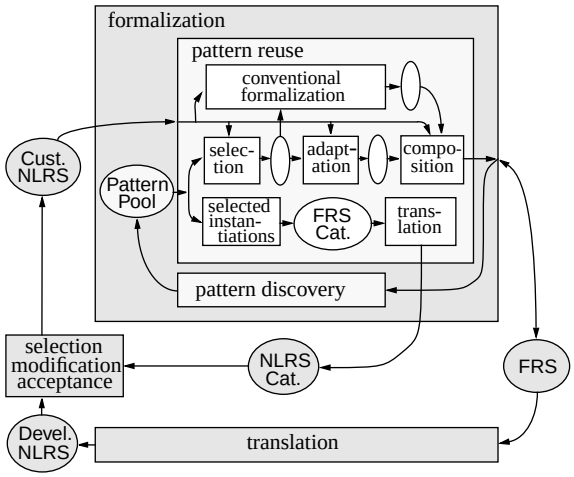
The discovery of new patterns is a difficult and time-consuming process. In general, many requirements have similarities in the way they restrict time bounds, delays and dependences between system states or other domain specific properties. These similarities can be exploited in order to extract the 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 3.2, we report on the discovery of a particular real-time requirement pattern in detail. The discovery of another real-time requirement pattern is presented in [8]. 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 .

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. Section 3.2 then elaborates on pattern discovery. In Section 3.3, a number of requirement patterns are defined. Their reuse is illustrated in Section 3.4.

Figure 5: Complete requirement specification development model



3.1 The system requirement FDT

For reasons that are addressed in [13], 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 the appendix and [11].

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
   - $\neg\varphi, \varphi\wedge\psi, \varphi\vee\psi, \varphi\rightarrow\psi, \varphi\leftrightarrow\psi \in \mathcal{F}$
   - $\Box\varphi, \blacksquare\varphi, \Diamond\varphi, \blacklozenge\varphi, \varphi \boldsymbol{W} \psi, [\varphi] \in \mathcal{F}$
   - $\Box_{\leq\tau}\varphi, \blacksquare_{\leq\tau}\varphi, \Diamond_{\leq\tau}\varphi, \blacklozenge_{\leq\tau}\varphi, \bigoplus^{\tau 1}_{\geq\tau 2} \in \mathcal{F}$
   - $\varphi \mathrel{\dot{\Rightarrow}}_{\leq\tau} \psi, \varphi \Leftrightarrow_{\leq\tau} \psi$
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
- [φ] ("action operator"): is true if φ is true now and was false in the preceding state
- $\bigoplus^{\tau 1}_{\geq\ \tau 2}\ \varphi$ ("accumulated invariance"): is true, if φ is valid for at least time τ2 during the next τ1 time units
- $\varphi\ \diamond\!\!\!\rightarrow_{\leq\tau}\ \psi$ ("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 ¬φ and ¬ψ.

This set of real-time operators is the result of the domain-specific tailoring of the logic's expressiveness. 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 an 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 formluae 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 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 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* , which are introduced to establish a close relationship to the informal problem statements:

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

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 in 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 their conjunction:

Table 4: Conditional Bounded Response and Continuity I

Name
ConditionalBoundedResponseAndContinuity (φ, ψ, *t*)
Example
❑ $(((hazardousCondition \wedge \neg windowClosed) \rightarrow \Diamond_{\leq 30s}[windowClosed \vee \neg hazardousCondition]) \wedge ((hazardousCondition \wedge windowClosed) \rightarrow (windowClosed \; \boldsymbol{W} \; \neg hazardousCondition)))$ 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
❑ $(((\varphi \wedge \neg\psi) \rightarrow \Diamond_{\leq t}[\psi \vee \neg\varphi]) \wedge ((\varphi \wedge \psi) \rightarrow (\psi \; \boldsymbol{W} \; \neg\varphi)))$ 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
❑ $(hazardousCondition \rightarrow \Diamond_{\leq 30s}(windowClosed \wedge windowClosed \; \boldsymbol{W} \; \neg hazardousCondition \vee \neg hazardousCondition))$ 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
❑ $(\varphi \rightarrow \Diamond_{\leq t}(\psi \wedge \psi \; \boldsymbol{W} \; \neg\varphi \vee \neg\varphi))$ 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 outcomes: (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 *DelayedImplication:*

Table 6: Delayed Implication I

Name
DelayedImplication (φ, ψ, *t*)
Example
❑ $(hazardousCondition \rightarrow \Diamond_{\leq 30s}(windowClosed \; \boldsymbol{W} \; \neg hazardousCondition))$ 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
❑ $(\varphi \rightarrow \Diamond_{\leq t}(\psi \; \boldsymbol{W} \; \neg\varphi))$ 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 "$\varphi \rightarrow$ " could be removed from the pattern without any semantic changes. Additionally, the operator $\Rightarrow_{\leq t}$, defined by

$$\varphi \leadsto_{\leq t} \psi \;=_{\mathrm{Df}}\; \Diamond_{\leq t}(\psi \; \boldsymbol{W} \neg\varphi)$$

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* $\leadsto_{\leq 30\ s}$ *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
❑ $(\varphi \leadsto_{\leq t} \psi)$ 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.
Semantic properties
❑ $(\varphi_1 \leadsto_{\leq t} \varphi_2) \wedge$ ❑ $(\varphi_2 \leadsto_{\leq t'} \varphi_3) \;\rightarrow$ ❑ $(\varphi_1 \leadsto_{\leq t+t'} \varphi_3)$ ❑ $(\varphi \leadsto_{\leq t} \psi_1) \wedge$ ❑ $(\varphi \leadsto_{\leq t} \psi_2) \;\leftrightarrow$ ❑ $(\varphi \leadsto_{\leq t} (\psi_1 \wedge \psi_2))$

3.3 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*, *Conditional-Continuity* 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 runningtime:

Table 8: Invariance

Name
Invariance (φ)
Intention
Allows to specify that a certain property must always hold.
Example
❑ *tempActGtZero* Let *tempActGtZero* represent an indoor temperature greater than *0 °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
❑ φ 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, the operator $\varphi \Leftrightarrow_{\leq t} \psi \;=_{\mathrm{Df}} (\varphi \leadsto_{\leq t} \psi) \wedge (\neg\varphi \leadsto_{\leq t} \neg\psi)$ 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
❑ *(hazardousCondition ∧ windowOpen* $\Leftrightarrow_{\leq 30s}$ *warnedUser)* Supposed, the window is only manually operable. Whenever the window is continuously open during a hazardous condition for at least *30s*, 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
❑ $(\varphi \Leftrightarrow_{\leq t} \psi)$ 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
$\Box$ *([windowOpen]* $\rightarrow \Box_{\leq 5\ min}$ *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
$\Box \oplus^{1h}_{\geq 12\ min}$ *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
$\Box \oplus^{T}_{t}\ \varphi$ Within any time interval *T*, φ is true for at least *t* time units.

3.4 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. An example may be found in [8], 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 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 formal description technique, 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. During our work, we have made the following observations:

- In a case study [13], all requirements have been formalized by pattern instantiations. Here, the *Invariance* pattern and the *DelayedImplication* pattern have been used most frequently. However, the *ConditionalBoundedResponse* and *ConditionalContinuity* patterns were not used at all. The reason is that the requirements where these patterns are applicable have been expressed using the *DelayedImplication* pattern.
- 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.
- 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 [5,6]. Also, it may lead to a better traceability of the consequences when modifying requirements of an already developed system. [7] contains a step in this direction.

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

Appendix: 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 [1] is used:

Definition: *(State, timed state sequence, model)*
Let $\mathcal{P}$ be the set of all atomic formulae.

1. A *state* is a function $\sigma: \mathcal{P} \to \{0, 1\}$. The set of all states is denoted as Σ.
2. A *timed state sequence* ρ is a function $\rho: \boldsymbol{R}_0^+ \to \Sigma$ such there exists an interval sequence $I = I_0, I_1, \ldots$ with
   a) $\forall i \in \boldsymbol{N}: I_i = [a_i, b_i)$ with $a_i \in \boldsymbol{R}_0^+, b_i \in \boldsymbol{R}^+ \cup \{\infty\}, a_i < b_i$
   b) $\forall i \in \boldsymbol{N}:$ if $b_i \neq \infty$ then $b_i = a_{i+1}$
   c) $\forall i \in \boldsymbol{N}: \forall t, t' \in I_i: \rho(t) = \rho(t')$
   d) $\forall t \in \boldsymbol{R}_0^+: \exists i \in \boldsymbol{N}: t \in I_i$
   e) If ρ is not constant from any point in time, i.e.: $\forall t_1 \in \boldsymbol{R}_0^+: \exists t_1' \in \boldsymbol{R}_0^+, t_1' > t_1: \rho(t_1) \neq \rho(t_1')$, then: $\forall i \in \boldsymbol{N}: \forall t_2 \in I_i: \forall t_2' \in I_{i+1}: \rho(t_2) \neq \rho(t_2')$
   f) If ρ is constant from a point in time, i.e.: $\exists t_1, t_1 \in \boldsymbol{R}_0^+: \forall t_1' \in \boldsymbol{R}_0^+, t_1' \geq t_1: \rho(t_1) = \rho(t_1')$, then: $\exists n \in \boldsymbol{N}: I = I_0, I_1, \ldots, I_n$ and $\forall i \in \{0, \ldots, n-1\}: \forall t_2 \in I_i: \forall t_2' \in I_{i+1}: \rho(t_2) \neq \rho(t_2')$

   Such an interval sequence 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 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 I fulfilling a) to f). Note that propositional formulae have the same truth value during an interval I_i of I.

Definition: *(Semantics of tRTTL)*

Let $\mathcal{M}$ be a model, $\rho \in \mathcal{M}$ be a timed state sequence, and $I = I_0, I_1, ...$ 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 $\exists t', 0 \le t' < t: \forall t'', t' \le t'' < t: (\rho, t'') \models \neg\varphi)$
4. $(\rho, t) \models \Box\varphi$ iff $\forall\, t', t' \ge t: (\rho, t') \models \varphi$
5. $(\rho, t) \models \Box_{\le\tau}\varphi$ iff $\forall\, t', t \le t' \le t+\tau: (\rho, t') \models \varphi$
6. $(\rho, t) \models \blacksquare\varphi$ iff $\forall\, t', 0 \le t' \le t: (\rho, t') \models \varphi$
7. $(\rho, t) \models \blacksquare_{\le\tau}\varphi$ iff $\forall\, t', t_{low} \le t' \le t:$ $(\rho, t') \models \varphi$ and $t_{low} = \max\{0, t-\tau\}$
8. $(\rho, t) \models \varphi \boldsymbol{W} \psi$ iff $(\rho, t) \models \Box\varphi$ or $(\exists t', t' \ge t: (\rho, t') \models \psi$ and $\forall t'', t \le t'' < t': (\rho, t'') \models \varphi)$
9. $(\rho, t) \models \oplus^T_{\ge\tau}\varphi$ iff $\Sigma_{j \in J}(\min(b_j, t+T) - \max(a_j, t)) \ge \tau$ with $J = \{j \in \boldsymbol{N} \mid t \in I' = [a', b'), a' \le a_j \le t+T, (\rho, a_j) \models \varphi\}$
10. $\Diamond\varphi$ $=_{Df} \neg \Box \neg\varphi$
11. $\Diamond_{\le\tau}\varphi$ $=_{Df} \neg \Box_{\le\tau} \neg\varphi$
12. $\blacklozenge\varphi$ $=_{Df} \neg \blacksquare \neg\varphi$
13. $\blacklozenge_{\le\tau}\varphi$ $=_{Df} \neg \blacksquare_{\le\tau} \neg\varphi$
14. $\varphi \mathrel{\dashrightarrow}_{\le\tau} \psi$ $=_{Df} \Diamond_{\le\tau}(\psi \boldsymbol{W} \neg\varphi)$
15. $\varphi \Leftrightarrow_{\le\tau} \psi$ $=_{Df} (\varphi \mathrel{\dashrightarrow}_{\le\tau} \psi) \wedge (\neg\varphi \mathrel{\dashrightarrow}_{\le\tau} \neg\psi)$
16. $\models_i \varphi$ iff $\forall \mathcal{M}: \forall\, \rho \in \mathcal{M}: (\rho, a_0) \models \varphi$ (initial validity)

References

[1] 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

[2] P.-J. Courtois, D.L. Parnas: *Documentation for Safety Critical Software*. 15th International Conference on Software Engineering, Baltimore, pp. 315-323, 1993

[3] S.R. Faulk, J. Bracket P. Ward, J. Kirby: *The CoRE Method for Real-time Requirements*. IEEE Software 9(5), Sept. 1992

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

[5] B. Geppert, R. Gotzhein, F. Rößler: *Configuring Communication Protocols Using SDL Patterns*. 8th SDL Forum, Paris, France, Sept. 1997

[6] B. Geppert, F. Rößler: *Generic Engineering of Communication Protocols - Current Experience and Future Issues*. 1st IEEE International Conference on Formal Engineering Methods, ICFEM'97, Hiroshima, Japan, Nov. 1997

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

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

[9] C. Heitmeyer, A. Bull, C. Gasarch, B. Labaw: *SCR*: A Toolset for Specifying and Analyzing Requirements*. 10th Annual Conference on Computer Assurance, Gaithersburg MD, June 1995

[10] K. Heninger: *Specifying Software Requirements for Complex Systems: New Techniques and Their Application*. IEEE Trans. on Software Engineering SE-6(1), pp. 2-13, 1980

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

[12] D.L. Parnas, J. Madey: *Functional Documentation for Computer Systems Engineering*. Science of Computer Programming (Elsevier) 25(1), pp. 41-61, Oct. 1995

[13] C. Peper, R. Gotzhein, M. Kronenburg: *Formal Specification of Real-Time Requirements for Building Automation Systems*. SFB 501 Report 01/97, Univ. of Kaiserslautern, Germany, 1997