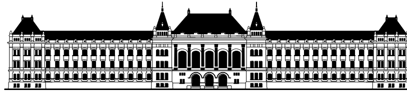

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Pattern-based Formalization of Safety Requirements

Author
György Bártfay

Advisor
Dr. István Majzik

2015

CONTENTS

Kivonat 4
Abstract 5
1. Introduction 6
2. Related Work 7
3. Requirement specification workflow 9
4. The Formalisms 11
  4.1. The features to be supported 11
    4.1.1. Timing extensions 12
    4.1.2. Context related extensions 12
  4.2. Metamodel for the requirements and patterns 13
  4.3. Description of the additional parts 19
  4.4. Concrete syntax (graphical representation) 19
  4.5. Output formalism: CaTL 21
    4.5.1. The precise syntax of CaTL 22
5. The Patterns 26
  5.1. Description and classification of the patterns 26
  5.2. Scopes description 27
6. Composition of patterns 29
  6.1. Rules of the pattern composing 29
  6.2. Mapping between the formalisms 30
7. Tool design 33
  7.1. EMF 33
  7.2. Sirius 33
8. Use cases supported by the tool 35
  8.1. User Interface 35
  8.2. New requirement 36
  8.3. Use existing pattern from store 37
  8.4. Parameterize elements 37
  8.5. Make contexts 37
  8.6. Save requirement as a pattern 38

8.7. Generate output 38
9. Conclusions 39
9.1. Results 39
9.2. Future work 39
References 40
Appendix 41
Metamodel in high resolution 41

Kivonat

A követelmények megfogalmazása igen fontos fázist jelent a szoftverfejlesztésben, különösképpen a biztonságkritikus alkalmazások esetén. Ez egy komplex feladat, továbbá a helyes és konzisztens követelmények megalkotása specifikus tudást és gyakorlatot is igényel.

A szokásos, természetes nyelven leírt követelmények gyakran nem elég precízek és könnyen félreérthetőek. Ugyanakkor a matematikai formalizmusok (például automaták, logikai nyelvek, stb.) használatakor a leírás bonyolulttá válik, ezért nehéz elkészíteni, megérteni illetve módosítani.

A dolgozatban ismertetett megoldás célja a biztonsági követelmények összeállításának egyszerűbbé és könnyebbé tétele, a helyesség és precízség megtartásával. A megoldás kihasználja, hogy a tapasztalatok szerint a biztonsági követelmények jellemzően sémákra épülnek. Erre alapozva kidolgozható egy módszer, melynek segítségével az összegyűjtött minták komponálhatóak és paraméterezhetőek, így összeállítva a komplex biztonsági követelményeket.

Ehhez elméleti kutatásként szükséges volt a minták leírásához használható formalizmus kidolgozása, a paraméterezési módszer definiálása, valamint a komponálhatóság szabályrendszerének megalkotása. A formalizmus támogatja a temporális logikai követelmények, a kontextusfüggő viselkedés és az időzítések, határidők megadását. A mintákból összeállított összetett követelményekhez leképezést definiáltunk egy precíz, formális nyelvre. Az így formalizált követelmények felhasználhatók tervek ellenőrzésére vagy futásidőbeli verifikációhoz monitorok szintézisére.

A módszer alkalmazására készített eszköz tartalmazza a leggyakoribb minták gyűjteményét valamint lehetővé teszi saját minták készítését és beillesztését. A kidolgozott formalizmusok és szabályok alapján biztosítja a minták grafikus megjelenítését, valamint ezek könnyű paraméterezhetőségét és grafikus formában történő összeállítását. Az eszköz beleillik az Eclipse környezetbe, így könnyen tanulható és használható a fejlesztés során.

Abstract

Specification of requirements is a very important phase in software development, especially in case of safety critical systems. This is a complex task; moreover, writing correct and consistent requirements requires specific knowledge and experience.

Requirements described in natural language are often imprecise and easy to misunderstand. However, using precise mathematical formalisms (like automata, logic languages, etc.) has the risk that the expressions become complicated, so it is difficult to write, understand, and change them.

The goal of the solution described in this paper is to make the specification of safety requirements simpler and easier, while preserving the preciseness of formal techniques. Experience shows that safety requirements are typically based on patterns. Based on this fact a method can be developed which helps to specify complex requirements by composing and parameterizing requirement patterns collected in a repository.

To achieve this goal, it was necessary to define a formalism to describe the patterns, work out a method to parameterize them, and construct a set of rules for the composition of the patterns. The proposed formalism supports the specification of temporal properties, context-dependent behaviour, and timing constraints (deadlines). We defined a mapping from complex requirements (composed using the patterns) to expressions of a precise formal language. The requirements constructed and formalized this way can be used to verify designs or synthesize monitors for run-time verification.

The tool designed and implemented to support this method contains a collection of the most often used patterns and also provides the possibility to create and integrate custom patterns. Based on the defined formalism and the set of composition rules, it allows the representation, parameterization, and composition of the patterns in a graphical way. The tool is implemented in the Eclipse environment in order to make it easy to learn and use in the development process.

1. Introduction

Specification of requirements is a very important phase in software development, especially in case of safety critical systems. This is a complex task; moreover, writing correct and consistent requirements requires specific knowledge and experience.

Requirements described in natural language are often imprecise and easy to misunderstand. However, using precise mathematical formalisms (like automata, logic languages, etc.) has the risk that the expressions become complicated, so it is difficult to write, understand, and change them. The goal is to make this workflow easier without decreasing the precision.

Inspecting the typical (manually written) requirement suggests an idea. Some expression occurs frequently, such as “The system never reaches the error state.” or “If <conditions hold> then <specific state shall be reached>“. This discovery implicates the idea, that we can identify some frequently recurring patterns. If there are patterns (schemes) matching to lots of requirements, then we can make the formalization process of new requirements easier by providing a formalization for the schemes. This approach looks promising in the field of context-aware autonomous systems.

The goal of the research described in this report is to create a methodology and a supporting tool based on this approach. The methodology offers a formalism to describe the patterns and a classification and description of widely used patterns. Then it defines a method to parameterize the collected patterns to prepare concrete requirements.

Finally, we present the developed tool based on the Eclipse environment, which contains the pattern collection, provides a graphical interface to parameterize and compose patterns and provides the possibility to generate formal expression from the composed pattern.

2. Related Work

The papers [1] and [2] pay attention to the requirement patterns that frequently occur in the field of the formalization of safety requirements. These patterns will be described in detail later in the Chapter 5. Here we recall the most important conclusion that it is possible to identify a set of often used schemes of requirements. These will be referred as *patterns* from now. Moreover we can make a classification too.

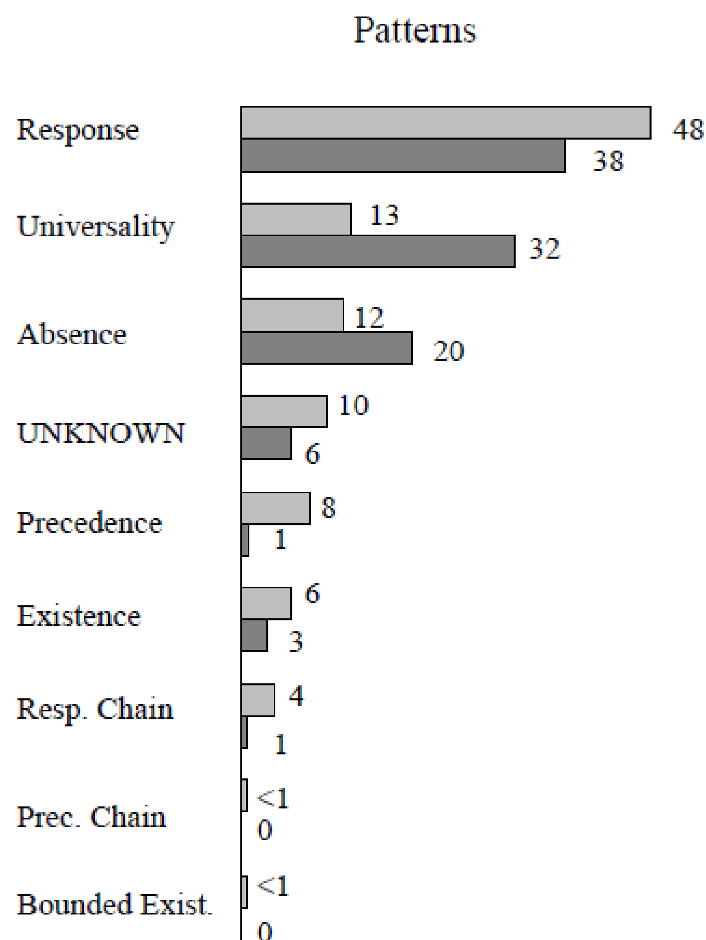

Figure 2.1:The distribution of pattern usage in % [1]

In paper [1] they present the result of a survey, what they made. They observed developers, who actually worked in the field of safety critical system development. The result

(Figure 2.1) shows, that there are several patterns that are often used. It means that we can cover the majority of requirements with 5-6 patterns. Accordingly, it’s an opportunity to create a supporting tool based on built-in schemas for these patterns, as it would make faster the safety requirement formalization process.

The two bars in Figure 2.1 indicate two groups. One of them is formed by the people working in a given project, while the people in the other group are outsiders. It turned out that the difference between the two groups is not significant.

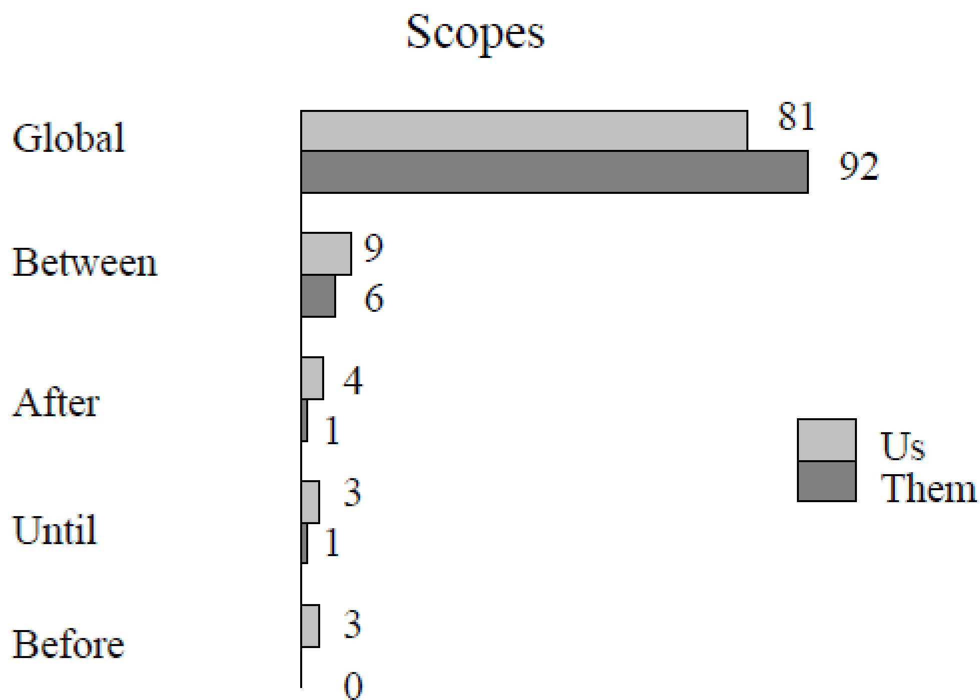

Figure 2.2: The distribution the scope usage in % [1]

Figure 2.2 is from the same survey. It is not a big surprise that the usage of the scopes of the patterns shows a similar result. Note that the detailed description of the scopes can be found later in the Chapter 5. Here we can make a same conclusion as before: most of the usages can be covered with a few scopes.

3. Requirement specification workflow

The requirement specification workflow using our methodology is presented in Figure 3.1. The red highlighted area is what can be supported by our tool. The pattern and context metamodel define the underlying formalisms that are built-in. Right now the context metamodel is presented together with the requirement pattern metamodel, but it can be separated and attached (read-in) in runtime.

The Pattern Store is a collection of the predefined patterns. The mostly used ones are provided with the tool, but the user can also save to the store the requirements made by him as patterns. Through the composing process the user can anytime use patterns from the store by inserting it to the requirement (expression) which is under construction.

To specify a requirement, the user can work from (1) the patterns stored, (2) so-called basic elements (like atomic propositions) from a palette, and (3) context fragments. The expression and the context fragments are displayed in the same view, but they are separated. From the expression the contexts can be referenced by parameterizing the composed elements.

When composing and parameterizing is finished, the result is the requirement represented graphically. In the background it is stored in an object model (based on the metamodel of the supported requirements). This means it can be stored in the file system (like a normal programming project), but it offers even more possibility. It can be saved as a pattern, so it will be the part of the pattern store for later reuse. The other opportunity is to generate a formal expression from the requirement in the form of a temporal logic CaTL. (The CaTL in the picture is an output formalism, it will be described later in Chapter 4.)

With the generated formal temporal logic expression, one can use existing tools to verify system designs or synthesize monitors for runtime verification. The CaTL output is saved in a text file in the file system and provided in property view inside the composing tool.

The composed requirements are more than a simple LTL expression. It can contain temporal logic, Boole algebra and atomic expressions such as time constraints, and context-dependent requirements.

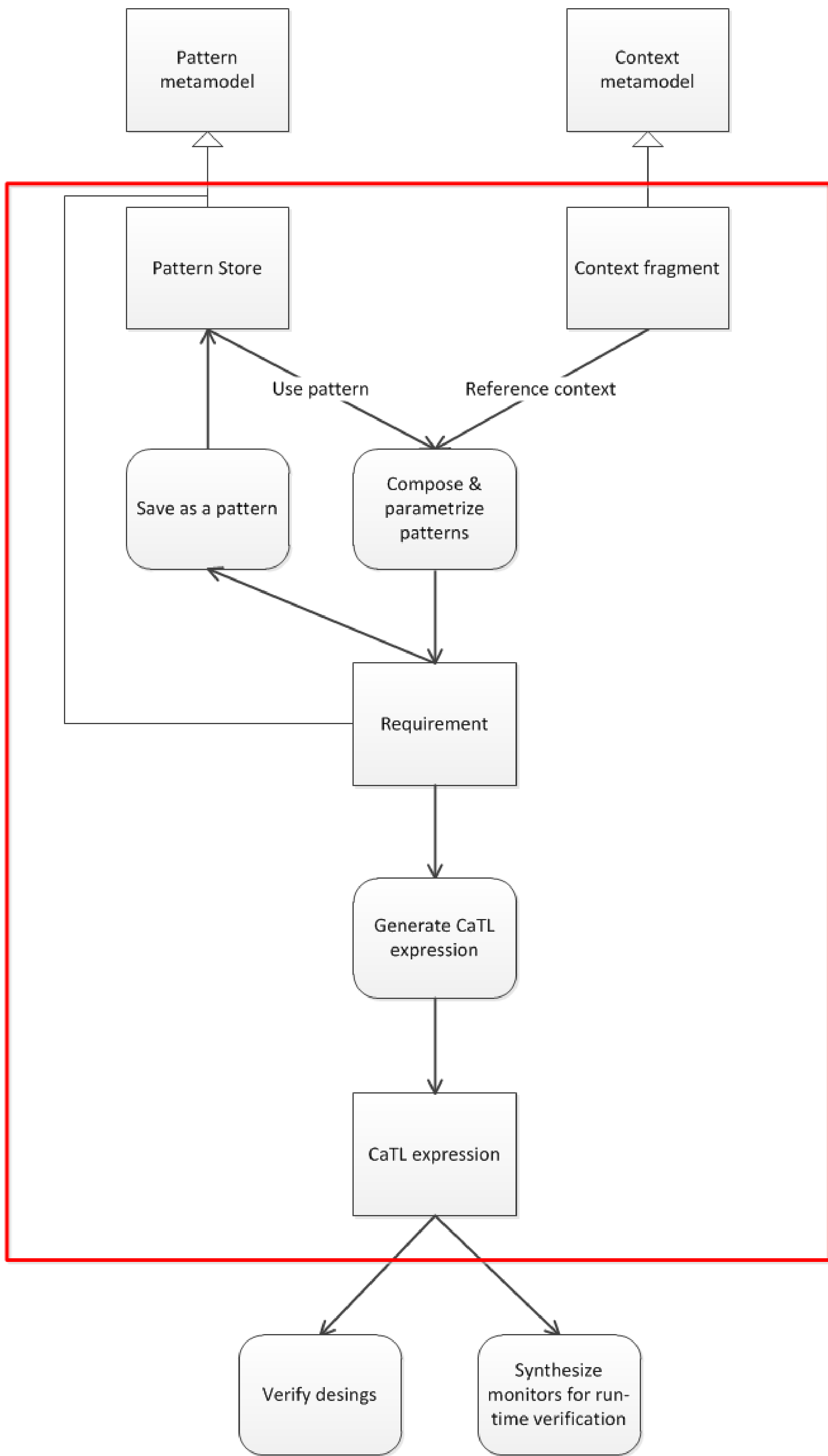

Figure 3.1: The complete workflow

4. The Formalisms

In this chapter we describe our expectations towards the requirements to be specified, then define the metamodel that is able to capture both the patterns and the (complex) requirements composed using these patterns. Finally, we introduce the formalism that is the output of the requirements composer tool.

4.1. The features to be supported

Using our methods, we would like to support the formalization of requirements of systems that are characterized by real-time, event-driven and context-aware behaviour. Examples of such systems include autonomous robots or vehicles.

Basically, the requirements shall capture safety properties ("something bad never happens") and liveness properties ("something good will eventually happen"). These modalities can be formalized using the temporal operators of the Propositional Linear Temporal Logic (PLTL). PLTL expressions can be defined on a trace of steps, in which each step can be characterized by atomic propositions, i.e., local characteristics of the step. In the following we call these atomic propositions in general as events, and the trace of steps is the trace of events. The concept of event includes all elements of a required execution that are relevant from the point of view of the specified properties: input/output signal, sent/received message, function call/return, started/expired timer, entered/left state, change of context, change of configuration, predicate on the value of a variable etc.

Besides the usual Boolean language operators, basic PLTL has the following temporal operators:

- X: "Next" (X P means that the next step in the trace shall be characterized by event P).
- U: "Until" (P U Q means that a step characterized by the event Q shall eventually occur, and until that occurrence all steps of the trace shall be characterized by event P).
- G: "Globally" (G P means that each step in the trace shall be characterized by P).

- F: “Future” or “Eventually” (F P means that eventually a step shall occur in the trace that is characterized by P).

To support the expression of context dependence and real timing, the following extensions are needed:

- Explicit context definitions: Context may appear in the properties typically as condition for a given behaviour. For example, in context of a nearby obstacle a slowdown command is required.
- Timing: Timing related criteria shall also be expressed. For example, the slowdown command shall occur in 10 time units after the observation of the nearby obstacle.

In the following subsections these basic ideas related to time- and context-related extensions are detailed.

4.1.1. Timing extensions

The basic PLTL cannot specify real-time requirements as it is interpreted over models which retain only the temporal ordering of the events (without precise timing information). Therefore PLTL cannot specify requirements like “An alarm must be raised, if the time difference between two successive steps is more than 5 time units”. To tackle this issue, various approaches can be found in the literature. We apply the so-called *timeout based extension of PLTL* that uses an explicit global clock and both static and dynamic timing variables. Using this extension, the above example property is expressed as $G((x = t_0) \rightarrow X((x > t_0 + 5) \rightarrow alarm))$, where x is the clock variable and t0 is a timing variable.

4.1.2. Context related extensions

In our approach, the context is captured in form of a *context model* that describes the environment that is perceived and internally represented in the specified system to influence its behaviour. The static part of the context model supports the representation of the environment objects, their attributes and relations, this way a scene of the environment (e.g., the furniture of a room with their colours, sizes and positions). The objects are modelled using a type hierarchy. The dynamic part of the context model includes the concepts of changes with regard to objects as well as their properties and relations. Changes are represented as context events (e.g., appears, disappears, moves) that have

attributes and relations to the changed static objects and their relations (depending on the type of the context event).

The abstract syntax of the context model is defined in form of a *context metamodel* (note that the type hierarchy of this metamodel can be systematically constructed on the basis of existing domain ontologies).

An example context metamodel of a household robot is presented in Figure 4.1

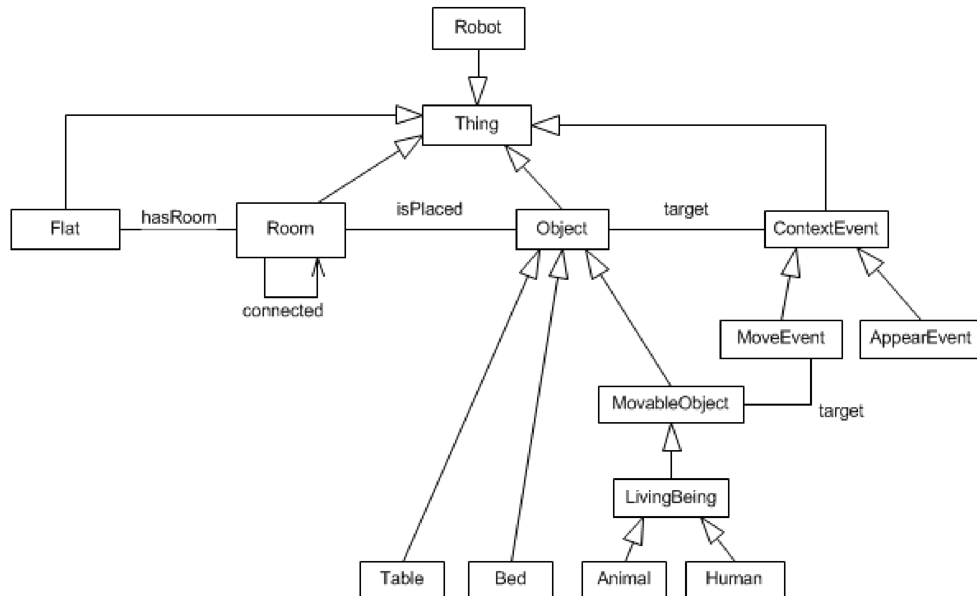

Figure 4.1: The context metamodel

In requirements, the contextual condition is referenced in the form of context fragments which are (partial) instance models of the context metamodel.

4.2. Metamodel for the requirements and patterns

The goal of the formalism is to define the inner structure of the workspace contents, such as composed requirements, stored patterns or contexts in order to make possible to represent and save them for later usage without any loss. The advantage of this approach is the possibility to change the output formalism: it only requires a new mapping from the metamodel to the new formalism.

The whole metamodel is presented in Figure 4.2 (the high resolution version is in the Appendix section). The details are described in the following part.

Figure 4.2: The metamodel

The main parts of this metamodel are the following:

- ‘A’ section: the base of the structure.
- ‘B’ section: temporal logic operators.
- ‘C’ section: Boole operators.
- ‘D’ section: the atomic formulas.
- ‘E’ section: the context part.

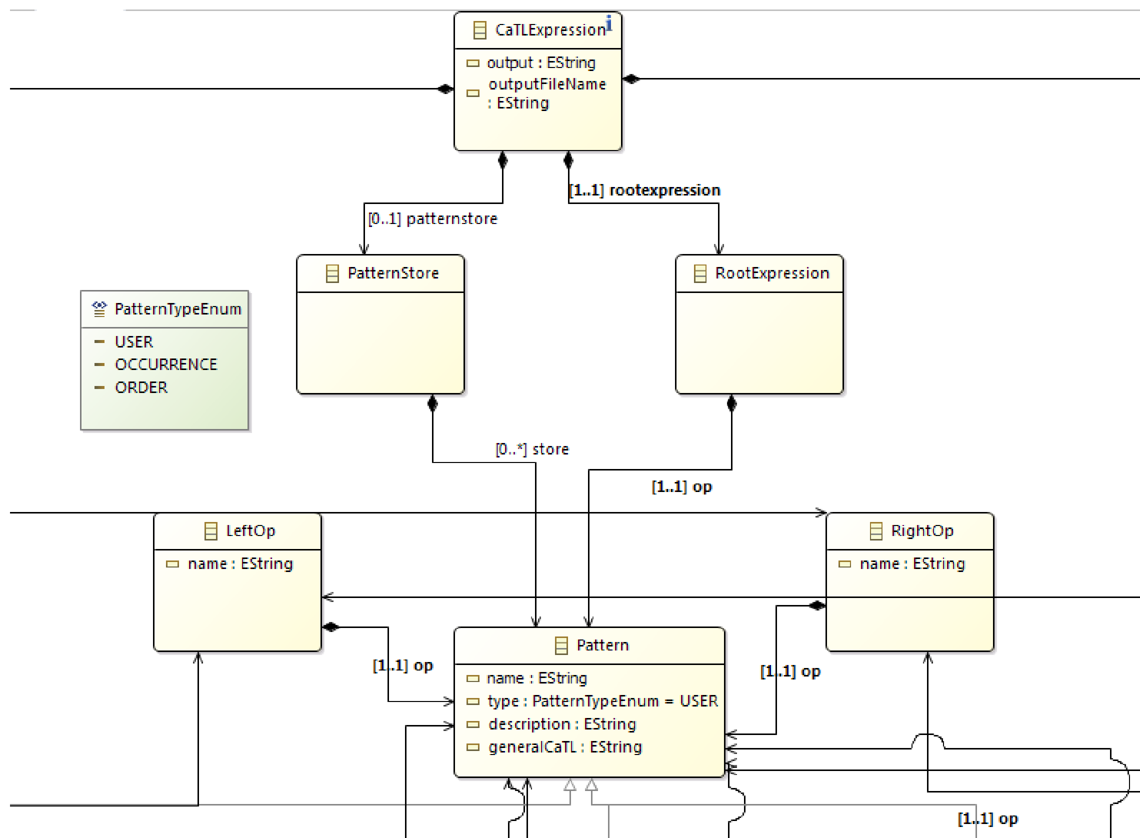

Figure 4.3: The ‘A’ section of the metamodel

Many parts of the metamodel presented in Figure 4.2 is caused by the implementation technology’s requirements. For example, the tons of compositions are exactly this kind of details. Now I will describe the important and relevant parts.

Figure 4.3 shows the base of the whole structure. The CaTLExpression is the root. It will contain everything. The most important element is the Pattern. It’s a high level abstraction in order to handle together the different elements. The root contains one Pattern as the developed requirements, and it can contain more of them as the elements of

the Pattern Store. The Pattern Store is a collection of composed and parameterized patterns to improve the reusability.

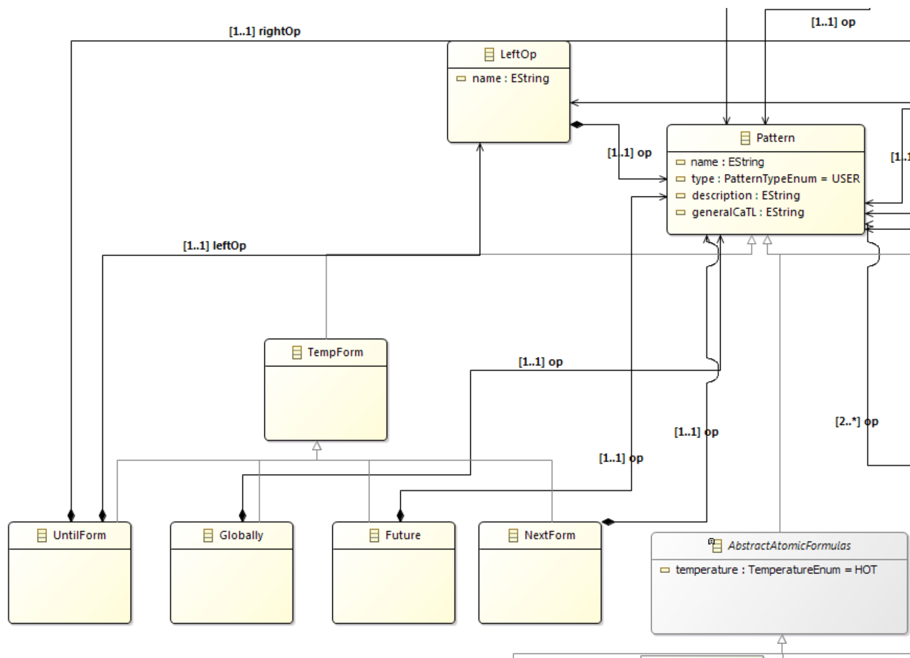

Figure 4.4: The ‘B’ section of the metamodel

Figure 4.4 shows, that the Pattern can be a Temporal Logic formula. But these formulas can contain other Patterns realizing the nested expressions. Next, Future and Globally can contain one inner expression, but the Until has two arguments. Moreover these two should not be swapped, because of the behaviour of the Until formal expression. That’s why there is special element name LeftOp. The RightOp is presented in Figure 4.5.

Figure 4.5: The ‘C’ section of the metamodel

In Figure 4.5 the Boole formulas are presented. It is similar to the previously discussed part of the metamodel. A Boole formula can be a *Negation*, an *Implication*, an *Or*, or an *And*. The Negation can contain one inner expression, the *Implication* is similar to the *Until*, it has two separated reference. The Last two formulas (*Or*, *And*) are different from the others. These can contain more than 2 nested expressions.

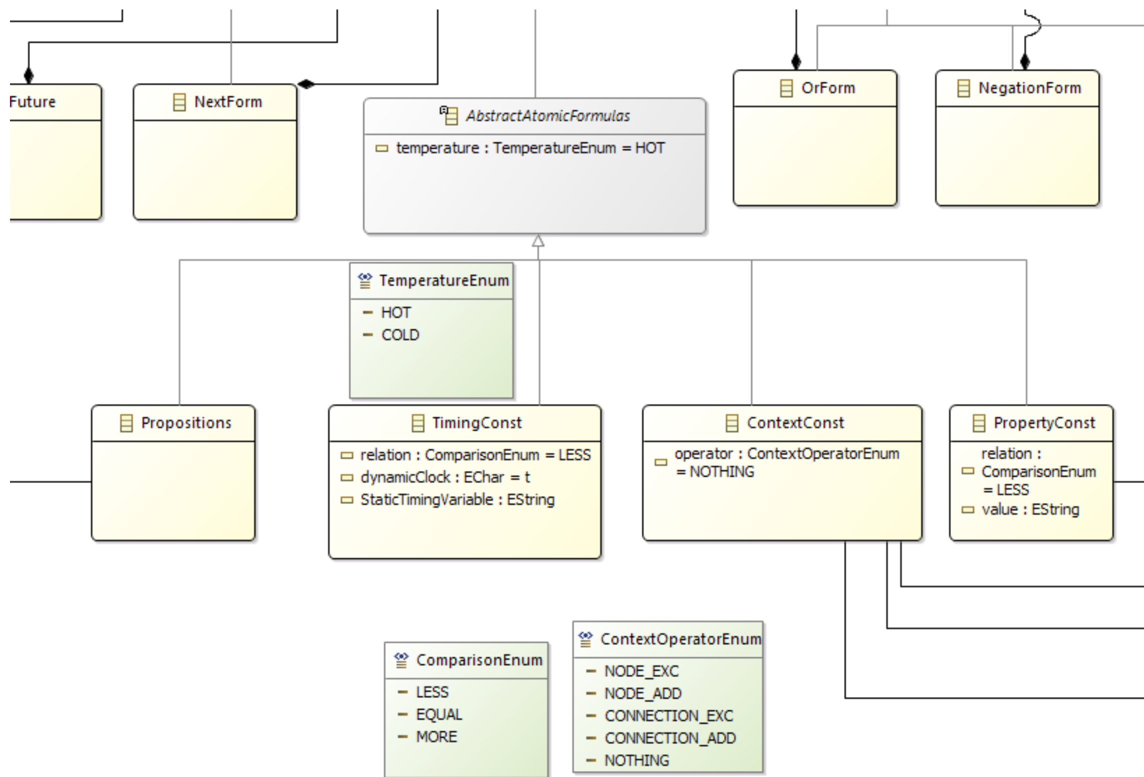

Figure 4.6: The ‘D’ section of the metamodel

In Figure 4.6 the most interesting part is presented. A Pattern can be a Temporal Logic, a Boole formula or an Atomic Formula. This part is where the point is stored.
Every *Atomic formula* has a *temperature*. It can be *hot* or *cold*. Hot means, that the requirement written by that element is mandatory, colt means it’s optional.
An Atomic formula can be one of the four element type. All of them has different content, because these used to express different kind of requirements. The detailed description of these elements can be found later in this paper.

4.3. Description of the additional parts

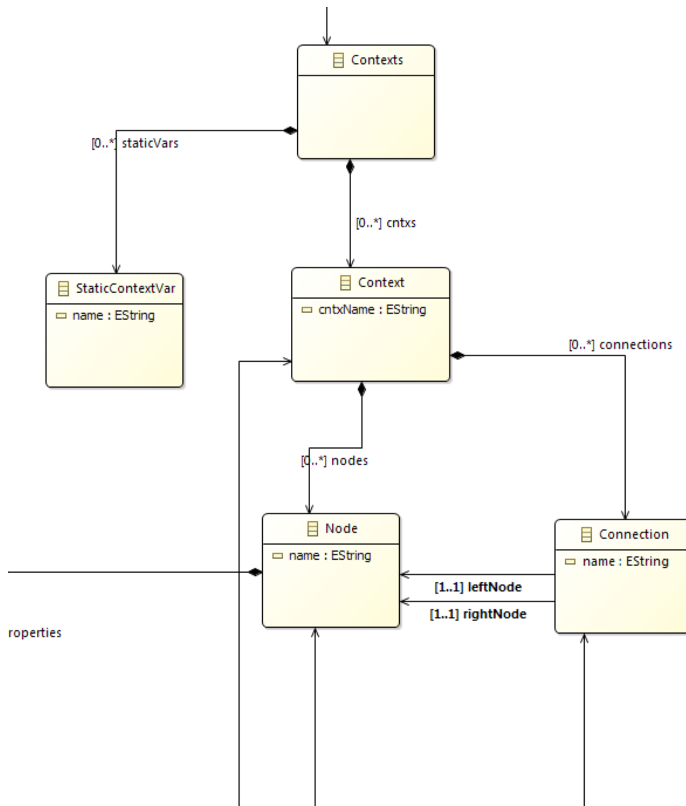

Figure 4.7: The ‘E’ section of the metamodel, part1

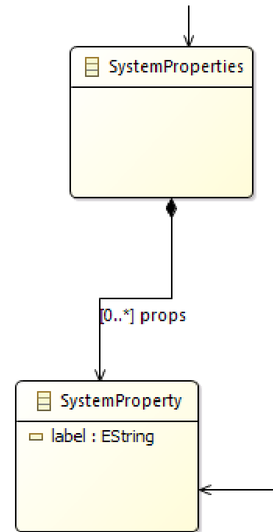

Figure 4.8: The ‘E’ section of the metamodel, part2

The last two parts of the metamodel contains the additional information. These are not used the same way as the other parts. The goal of these is to provide contexts (in Figure 4.7) or system level properties (in Figure 4.8). These are not contained in the concrete requirement, but the atomic formulas reference them. When the output expression is generated, these won't be mapped into that as other elements, but the information they contains will be written into the final formal sentence.

4.4. Concrete syntax (graphical representation)

Model element	Graphical representation	Comments
RootExpression	white rectangle	Root element of the whole structure

Propositions	square	Hot → red frame Cold → blue frame
TimingConst	circle	Hot → red frame Cold → blue frame
ContextConst	triangle	Hot → red frame Cold → blue frame
PropertyConst	diamond	Hot → red frame Cold → blue frame
NextForm	light blue rectangle	
Future	light yellow rectangle	
Globally	light orange rectangle	
UntilForm	green rectangle	Has two separated inner space
OrForm	light red rectangle	
NegationForm	turquoise rectangle	
AndForm	light brown rectangle	
ImpForm	grey rectangle	Has two separated inner space
SystemProperties	light grey rectangle	With label
SystemProperty	white square with black frame	
Contexts	light grey rectangle	With label

Context	grey rectangle	With label
Node	white circle	
NodeProperty	white square	
Connection	Edge between nodes	
PatternStore	light grey rectangle	With label and three separated inner space

Some of the elements has a same graphical element, but isn't a problem, because they can be identified by the place, where they are. The rules of this will be described later in Chapter 6.

4.5. Output formalism: CaTL

To use the composed expressions for verification and monitor synthesis we need a formalism, which supports the used extensions such as context- and time dependent requirements. This formalism is the "Context-aware Timed Propositional Linear Temporal Logic", called CaTL. [3]. It's already used by the Department of Measurement and Information Systems for verification and monitor synthesis using the requirements represented in CaTL.

The basic vocabulary of CaTL consists of a finite set of propositions, static timing variables and static context variables (these static variables are implicitly quantified with a universal quantifier). The value of a context variable is an instance of the context metamodel (e.g., a context fragment, which contains objects with unique identifiers, attributes and links). Moreover, two dynamic variables are used that represent the current time (clock variable) and the current context (observed context). The set of atomic formulas consists of the following elements:

- Propositions are labels, referring to events in the observed trace (each step may include multiple events). Each proposition can be evaluated to true or false in each step.

- Property constraints are predicates over properties of a context object.
- Timing constraints are defined as inequalities on the timing variables, constants and the dynamic clock variable.
- Context constraints are defined using a compatibility relation between context definitions and the current context. Context definitions can be created from context variables and operators as object exclusion, object addition, relation exclusion and relation addition. A context definition e1 context is compatible with the current context e2 (denoted as e1 ≈ e2) if and only if there exists a bijective function between the two object sets e1 and e2, which assigns to each object in e1 a compatible object from e2. Two objects are compatible, if and only if both have the same type and have the same relations to other objects.

To form CaTL expressions, these atomic formulas can be connected by using Boolean operators and PLTL temporal operators. Note that for each atomic formula, a modality can be assigned, where hot means a mandatory formula, and cold means an optional one.

In summary, PLTL atomic propositions are extended with context constraints (to be evaluated with respect to the observed context) and timing constraints (evaluated with respect to the current clock). These expressions are evaluated with respect to a particular step, without affecting the evaluation of the temporal operators.

Let us demonstrate the use of CaTL by the following examples from [3]:

- It is always true, that if the system is in the connected state, then it will eventually become disconnected: $G(connected \rightarrow F(disconnected))$
- It is always true, that if the system is connected, then it will be disconnected in 5 time units: $G(connected \wedge t_0 = t \rightarrow F(disconnected \wedge t < t_0 + 5))$
- It is always true, that if the system is connected and it is in the e1 context, and it will be disconnected in the next step, then eventually it will be in the e2 context: $G(connected \wedge e_1 \approx e \wedge X(disconnected) \rightarrow F(e_2 \approx e))$

4.5.1. The precise syntax of CaTL

The vocabulary consist of a finite set P of propositions, a finite set T of static timing variables and a finite set CM of static context variables.

Each $ci \in CM$ is an instance of M context metamodel ($ci \propto M$). A context metamodel is defined as a 2-tuple $M = (N,R)$, where N represents the set of classes in the metamodel and R represents the relations (i.e., association or generalization) in the model. An $ni \in N$ is a class, which has a set of properties. Each property has a name and a type (e.g., Boolean or string). One can create an EM set of predefined contexts, where $ei \propto M$ for all $ei \in EM$. The context variables and the predefined contexts contain instances of the classes (objects) from M. Each object has a unique identifier and an $ni \in N$ class. The values of the properties can be defined by property constraints (defined later), which refer to the objects by the unique identifiers given in the variables. It is important, that if two context variables contain two objects with the same identifier, then that two objects must be equivalent.

In addition, one can use two dynamic variables: t, which represents the clock and e, which represents the context of the system. M must be defined in such a way, which ensures that e is always a valid instance of M ($e \propto M$).

Af is the set of atomic formulas, which consists of propositions from P, atomic timing constraints, context constraints and property constraints.

- Propositions are labels, referring to properties of a system. Each proposition can be evaluated to true or false in each state of the system.
- The timing constraints are defined in the following form: $t \sim u$, where t is the dynamic clock variable, $\partial \sim \{<, >, =\}$, $u \in \{ti + c, c\}$, $ti \in T$ and $c \in N$.
- The context constraints are defined in the following form: $x \approx y$, where $x \in EM \cup VM$ and $y \in CM \cup \{e\}$. In this notation $\approx$ is a compatibility relation (meaning x is compatible with y) and VM is a set of context definitions. Context definitions are instances of the M metamodel. A context definition can be one of the followings:
    - a static context variable ($ci \in CM$), or
    - a new context, created from a static context variable, with one of the following operators:

      Node exclusion: z - v, where z is a context definition and v is a present class instance of z,

Node addition: z + w, where z is a context definition and w is an instance of the classes of M,

Connection exclusion: z - - a, where z is a context definition and a is a present connection in z,

Connection addition: z + + b(c, d), where z is a context definition and b is connection between c and d, compatible with M.

- The property constraints are expressions over properties of an object. The following syntax is defined to unambiguously select a p property: context.object.p. The syntax of the property constraints is: $p \sim v$, where p is a property, v is a value, which has to be from the same type as the property, and $\sim$ is a comparison operator, which can be evaluated to a Boolean value.

For each atomic formula, Υ assigns the modality of that atomic formula (a so-called "temperature"): $\Upsilon : Af \rightarrow \{hot, cold\}$. An atomic formula with hot temperature is a mandatory, while cold formulas are optional. The notation of the modality is the following. If no additional notation is given, then the modality of the atomic formula is hot (mandatory). The cold (optional) modality of the af atomic formula is written like < af >.

A ϕ CaTL formula can be one of the following:

- Atomic formula: $af \in Af$
- Disjunction: $\phi \vee \phi$
- Negation: $\neg\phi$
- "Next" operator: $X\ \phi$
- "Until" operator: $\phi1\ U\ \phi2$

All static variables used in a formula are implicitly quantified with a universal quantifier. Additional operators can be defined with the previously defined ones as syntactical abbreviations. The most commonly used abbreviations are defined as follows:

- Conjunction: $a \wedge b = \neg (\neg a \vee \neg b)$
- Implication: $a \rightarrow b = \neg a \vee b$
- "Eventually" operator: $F\ \phi$ = true $U\ \phi$, where "true" denotes the Boolean true value

- "Globally" operator: $G\phi = \neg(F \neg\phi)$
- "Weak until" operator: $\phi1 \ W \ \phi2 = (G \ \phi1) \vee (\phi1 \ U \ \phi2)$

The precise semantics of CaTL is described in [3].

5. The Patterns

5.1. Description and classification of the patterns

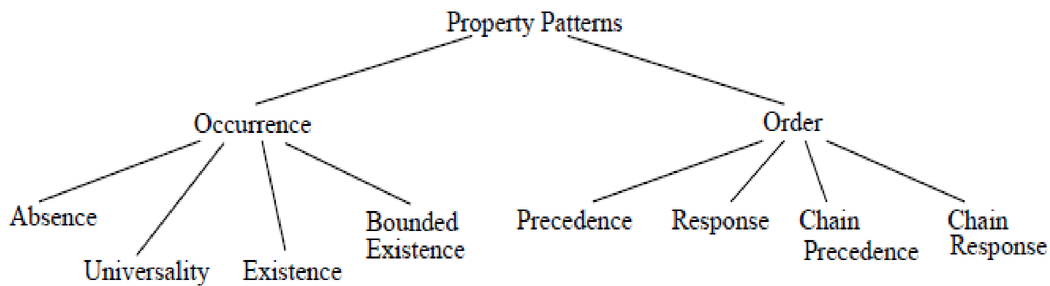

Figure 5.1: Classification of patterns [1]

Occurrence Patterns are used to express properties related to the existence or to the lack of existence of certain states/events in the pattern scope. They have been classified into four subtypes:

- Absence, also known as never happens. The event will never occur within the scope.
- Universality, also known as henceforth. The event will always occur within the scope.
- Existence, also known as eventually. The event may occur at least one time within the scope.
- Bounded existence. The event has to occur a fixed number of times within the scope. Variations of this pattern may be defined replacing the fixed counting of events with ‘‘at least” or an ‘‘at most” construct.

Order Patterns are used to express requirements related to pairs of states/events during defined scopes. There are two order related patterns:

- Precedence. P event has always to precede Q event within the scope.
- Response, also known as Follows, Leads-To. P event has always to be followed by Q event within the scope.

- Chain Precedence. A sequence of Pi events has always to precede a sequence of Qi events within the scope. It can be regarded as a generalization of the Precedence pattern.
- Chain Response. A sequence of Pi events has always to be followed by a sequence of Qi events within the scope. It can be regarded as a generalization of the Response pattern.

In the above classification, the Chain Precedence and Chain Response patterns can be considered as specific cases (a specialization) of Precedence and Response patterns. [2]

5.2. Scopes description

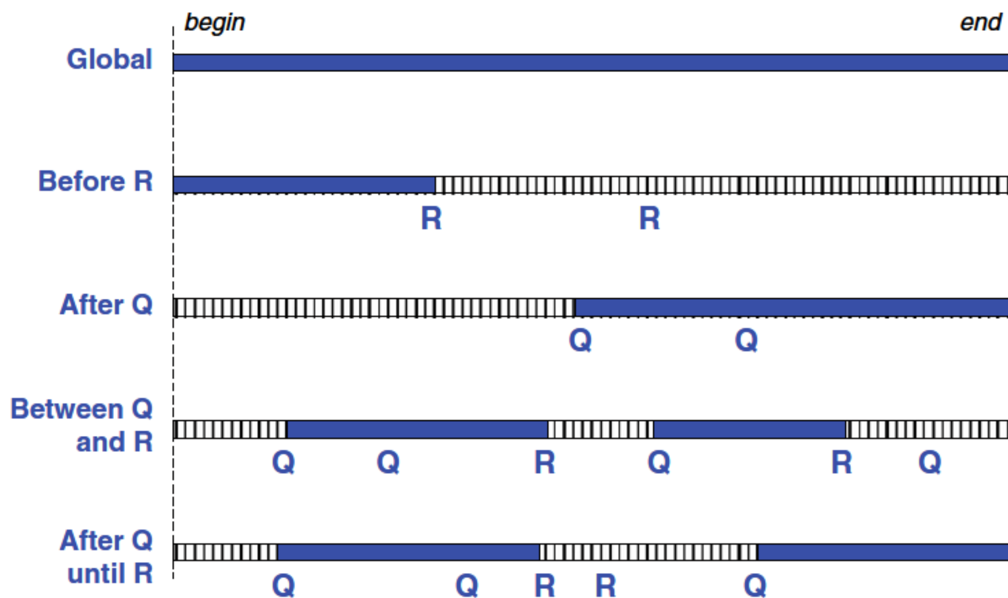

Figure 5.2: Scopes [2]

Based on studies [1], [2] five basic kinds of scopes should be discussed, as shown in Figure 5.1: Classification of patterns:

- Global – the property has to hold for the entire execution.
- Before R – the property has to hold up to the occurrence of state/event R.
- After Q – the property has to hold after the occurrence of state/event Q.

- Between Q and R – the property has to hold in every interval having state/event Q on left and state/event R on right. Please note that multiple overlapped intervals having the same end point are included in the scope, see Figure 5.2: Scopes, for this scope and interval covering Q–Q–R sequence.
- After Q until R – the property has to hold in every interval having state/event Q on left and state/event R on right or no ending event; this means that this property holds even when the interval is not closed by R.

6. Composition of patterns

6.1. Rules of the pattern composing

There are 3 general types of composing rules:

- How many instances can exists from an element type?
- How many Pattern can an element contains?
- Where is it allowed to put an element?

We can identify two groups based the number of allowed instances. The first group's members are allowed to exist in as many instances as they want. The other group is, where the members can only exist in one instance. The members of this group:

- CaTLExpression
- PatternStore
- RootExpression
- SystemProperties
- Contexts

We can see that all of them are a system level element, so this restriction is acceptable.

The second classification is based on the number of nested Patterns they can contain. (Here we are talking about top level nested elements, deeper they can contain more.)

In this classification only the subclasses of Pattern are mentioned. The others are irrelevant in this topic.

- More than two: Or, And
- Exactly two: Until, Implication
- One: Globally, Future, Next, Negation
- None: Atomic formulas

Every element has an own place in the structure. Most of them are very strict, but the subclasses of Pattern are replaceable with each other (with their potential children together). This freedom was one of the main goal in the planning stage.

6.2. Mapping between the formalisms

It is not necessary to use the same metamodel for the composed requirement model and the output formalism. It's sufficient if there exists a precise mapping between them. One of the advantages achieved by this, for example, that a formal language does not have to support all operators of the temporal logic, because by using some of them, the others can be expressed. This also allows changing the output formalism to another one with a different metamodel, if there is a mapping from the requirement metamodel to that too.

Here is an example for the mapping. You can see a graphically composed and parameterized requirement, and the context information.

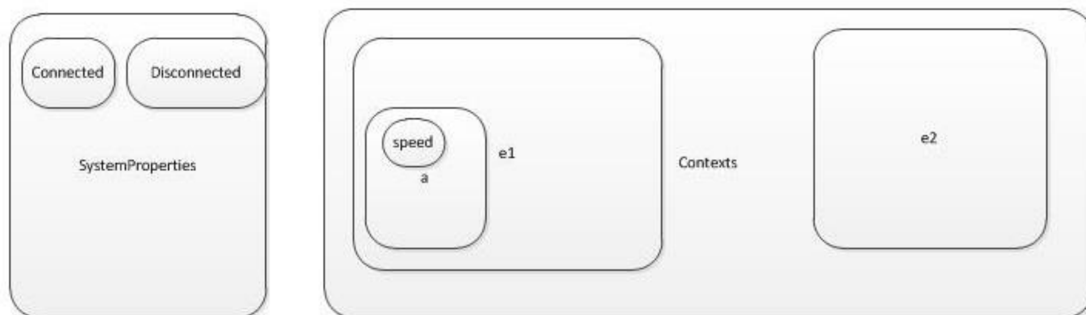

Figure 6.1: Context fragments and System Properties

The context explanation is already presented in the earlier section. Now we will use it together with the requirement.

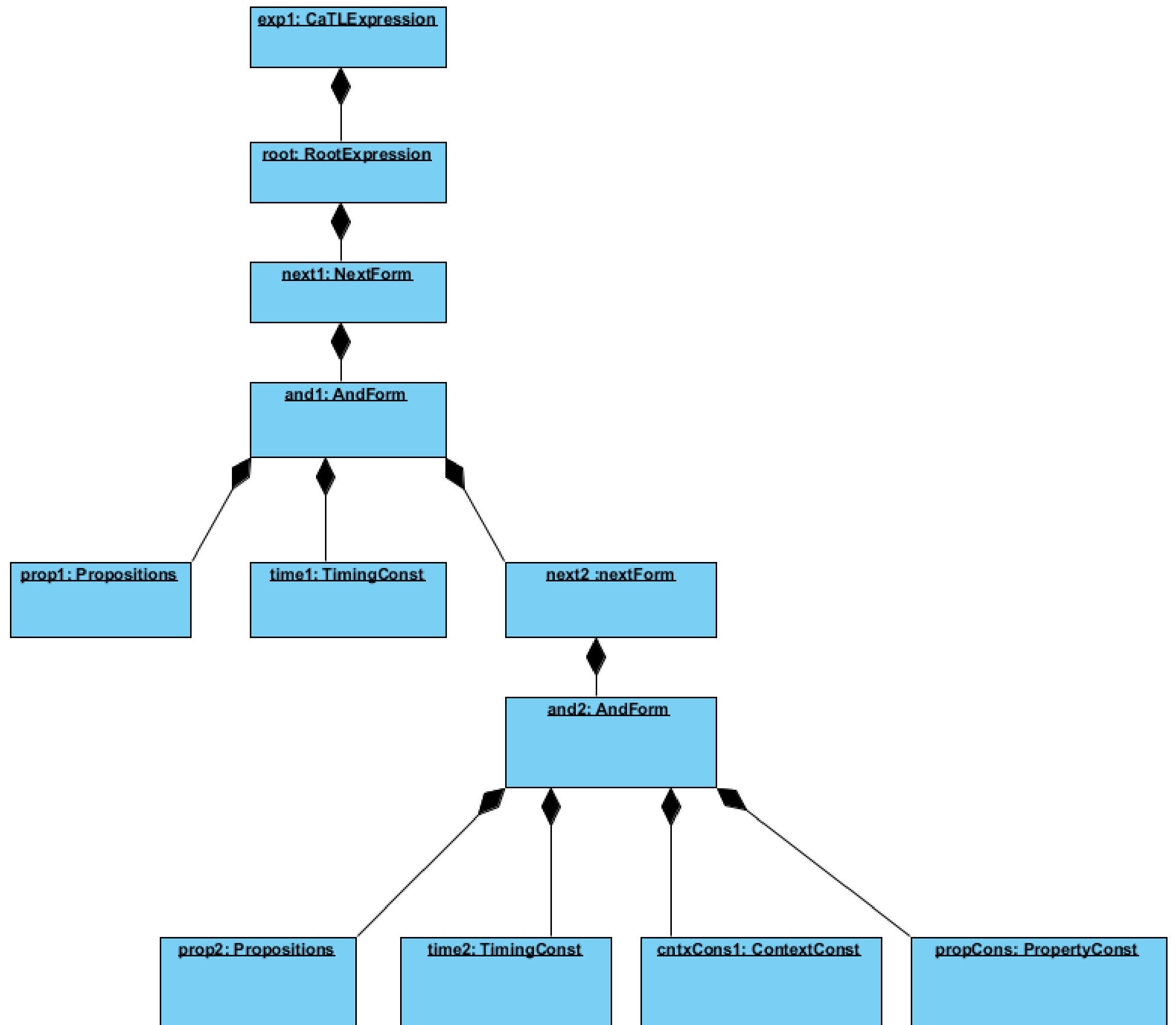

Figure 6.2: Object diagram of a requirement

Figure 6.2 is mapped to the formal CaTL sentence presented in Figure 6.3. (The references between Figure 6.1 and Figure 6.2 are not occurring here, but from the metamodel presented earlier the structure of them can be guessed.)

$X(Connected \wedge t = t0 \wedge X(Disconnected \wedge t < t0 + 5 \wedge e1 \rightsquigarrow e \wedge e1.a.speed < 10))$

Figure 6.3: CaTL expression

The two top level box is needed from implementation reasons.

The third box labelled ‘next1’ is mapped to X(…). Inside this there is an ‘AND’ element, which contains the following elements:

- Property ‘connected’, which is a System Property.
- Time dependent expression: ‘$t_0 = t$’.
- Next temporal logic. It’s a complex element contains an ‘AND’:
    - System Property ‘disconnected’.
    - Time dependent expression: ‘$t < t_0 + 5$’.
    - Context expression: ‘e_1 is compatible with e’.
    - Context Property: ‘e_1.a.speed < 10’.

7. Tool design

For the implementation I used Eclipse-based technologies. It's very practical, because the target audience (designers) is typically familiar with the Eclipse environment. A tool developed in this environment is easy to learn and use.

7.1. EMF

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. [4]

Figure 7.1: The logo of EMF

I used it to design the abstract syntax and to create a base for the developed tool.

7.2. Sirius

Sirius is an Eclipse project which allows the users to easily create their own graphical modelling workbench by leveraging the Eclipse Modeling technologies, including EMF and GMF.

Sirius has been created to provide a generic workbench for model-based architecture engineering that could be easily tailored to fit specific needs.

Based on a viewpoint approach, Sirius makes it possible to equip teams who have to deal with complex architectures on specific domains.

A modelling workbench created with Sirius is composed of a set of Eclipse editors (diagrams, tables and trees) which allow the users to create, edit and visualize EMF models.

The editors are defined by a model which defines the complete structure of the modelling workbench, its behaviour and all the edition and navigation tools. This description of a Sirius modelling workbench is dynamically interpreted by a runtime within the Eclipse IDE.

For supporting specific need for customization, Sirius is extensible in many ways, notably by providing new kinds of representations, new query languages and by being able to call Java code to interact with Eclipse or any other system. [5]

Figure 7.2: The logo of Sirius

With Sirius I defined the concrete graphical syntax, the mapping between the graphic and model elements, the rules of composing and the supporting features.

The following advantages of Sirius were especially useful:

- Fast graphical language developing.
- Cooperation with EMF.
- Validation of the composed expressions can be run-time with the ordinary Eclipse warning, error and quick fix features.
- Drag and drop support in the composing process.
- Possibility to create custom views to improve the understandability.

8. Use cases supported by the tool

8.1. User Interface

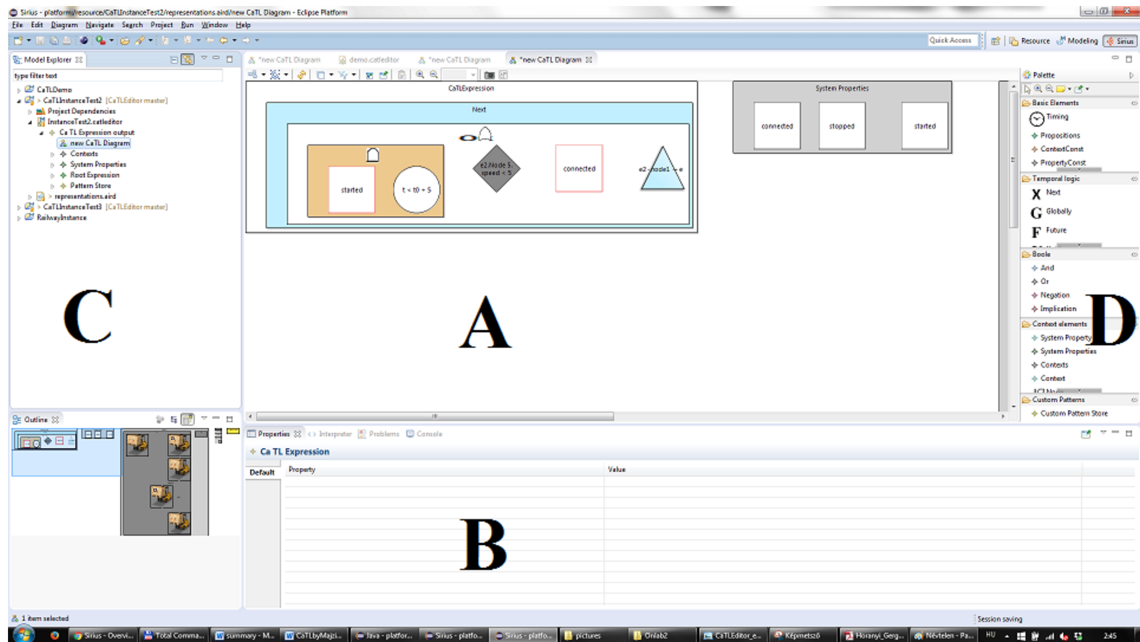

Figure 8.1: The UI of the tool

Figure 8.1 presents the user interface of the tool. It has four main part indicated with capital letters. ‘A’ is the most important. The goal of this area is to realize the composing process. It supports some useful helping feature, for example the ‘Arrange All’ function. It can be used with the button in the top left corner of the area. It makes the whole working area more ordered by resizing and relocating the elements. It is recommended to use it after every bigger modification.

The elements displayed in this working area are separated into three layers (CaTL layer, Context layer, Store layer). These layer can be turned on and off in runtime. It helps to organize the contents. Turning off a layer just hides the element associated to the layer from the graphical working area, they will remain in the object structure.

‘B’ is the properties view. Here can be seen and modificated all of the details of the elements. For example here can be set the name of a named element, or a reference. It always shows the properties of the selected element.

‘C’ is the area where you can see the underlying object structure, such as projects, object and representation files, etc.

‘D’ is the ‘Palette’ section. Form this area you can drag-and-drop elements to the working area (‘A’). Or if you just select an element from the *Palette*, and move the cursor above the working area, the shape of the cursor will give you hint about where you can place the selected element. The *Palette* contains the earlier presented elements of the graphical syntax. It has five groups:

- Basic Elements: atomic formulas (timing constraint, propositions, etc.)
- Temporal logic elements: next, globally, etc.
- Boole logic elements: and, or, etc.
- Context elements: context, node, connection, etc.
- Pattern store element.

8.2. New requirement

To start a new requirement developing process you should create a new project. It contains your composed requirement (in object and graphical representation too), the pattern store with your own patterns and the context fragments together.

It’s not necessary to create a new project for every requirement; it is fine to store more requirement model file in one project.

To start composing your own requirement you should follow these steps:

- Start a new *Modeling Project*.
- In this project create *CaTLEditor Model*. It uses an own file extension; ‘.catleditor’.
- When the wizard asks for a Model Object, choose ‘CaTLExpression’ from the list.
- Right click on the project. In the *Viewponts selection*: check the catleditor line.
- Right click on CaTLExpression (it is under the *.catleditor file, you just created). Here choose the *New Representation*: new CaTL Diagram option.

After this the graphical editor is going to open automatically.

8.3. Use existing pattern from store

In the *PatternStore* you can find the built-in and your own earlier saved patterns. Every pattern is represented by a white square. To use in your requirements you just have to drag-and-drop one of the patterns from the store into your requirement. Make sure that there is a suitable place for the pattern in the destination. For example you can't move it into a *Next* formula if it is not empty. Always keep in mind the composing rules described in Chapter 6.

The used pattern is going to appear in the destination fully detailed, but it is also remains in the store for later usage.

8.4. Parameterize elements

To customize a pattern in order to make it a perfect requirement you have to parameterize some of the elements. For example set a *Proposition's* reference, or add a value to a timing variable.

In the *Properties* view select the *Semantic* tab. After just select an element is the working area. Now you can see and change the selected element's properties. Where is it possible you can choose from a list (for example: make reference to another element). This helps to reduce the typing errors. Keep in mind, that you can only reference an existing element.

8.5. Make contexts

In order to define context fragments, first you have to make sure that there is exactly one *Contexts* element in the working area. You can't make a second one, if already exits one. If isn't any you can create one using the *Palette*.

Inside this *Contexts* element you can create almost infinite amount of context fragment. Inside the fragments you can create nodes, connection between nodes and properties to the nodes.

8.6. Save requirement as a pattern

To save a composed requirement (or a part of it) as a pattern, there is a possibility to add it to the *Pattern Store*. Just right click on the element, which you would like to make the root of the saved pattern. From the appeared menu choose the *Save Pattern* option.

The selected element, and the whole structure contained by it will be saved as a pattern to the Store. The new pattern element will appear in the Store User category. Naturally, the original elements will remain in the requirement, because it's just a copy placed into the Store.

The default behaviour of the tool is to put every pattern saved by the user into the User category, but you can move it to another category, by changing the 'Type' value of the created pattern element in the properties view.

When a pattern is saved, the tool generates a simplified CaTL expression into its properties. It is used as a tooltip to make easier to choose the right pattern from the store. Moreover the user can add a name and a description to the pattern elements in the store. These are also used as a tooltip.

8.7. Generate output

To generate the CaTL expression from the composed requirement, just double click on one of the graphical elements. It will generate the output, and store in two different place.

One of them is inside the object structure, it's one of the properties of the root element. It can be seen if you click on the background of the working area (not on the elements). Then it's in the *Properties* view.

Here can be edited the filename of the output file. The generated formal expression will appear in a text file with the specified filename. It will appear in the user's home folder. It works under Windows and Linux operating systems.

9. Conclusions

9.1. Results

First I was reading about safety requirements, patterns and some different point of view about the topic of the formalization of safety requirements. It's an interesting filed, there are difficulties, but there are possibilities too to achieve something.

The next step was to make plans and define a syntax. The defined graphical syntax is not the final version, but at first it was enough to start the project. To create useful equipment it was necessary to choose an output formal language, which is used in real projects and suitable to express a wide range of safety requirements.

Now the planned tool is implemented with the chosen technologies. There were some difficulties, for example in the middle of the implementation a new major release arrived for Eclipse and Sirius, but it was just a little bump.

Now the tool is working. Sometimes there can be found a bug, but these problems are not so serious. But before a tool can be presented to the public it should be developed further.

9.2. Future work

There are a lot of possibilities to go forward. The main plans are the following:

- Modify the context editing process. Now it can be very huge and confusing if we have lot of context fragments. The plan is to separate the currently edited context fragment from the others.
- The graphical elements should be redesigned. The colours and the shapes are not the best, and in this filed we can achieve a big gain.
- The context metamodel right now is a static part of the whole tool's metamodel. The plan is to separate it, so it could be read in runtime. Then the tool could work with other context model. It could improve the usability.

References

References

[1] M. B. Dwyer, G. S. Avrunin and J. C. Corbett, “Patterns in Property Specifications for Finite-State Verification,” 1999.

[2] P. Bellini, P. Nesi and D. Rogai, “Expressing and organizing real-time specification patterns via temporal logics,” *The Journal of Systems and Software,* 2009.

[3] G. Horányi, “Monitor synthesis for runtime checking of context-aware applications,” 2014.

[4] M. Koegel and J. Helming, “EclipseSource Blog,” [Online]. Available: http://eclipsesource.com/blogs/tutorials/emf-tutorial/. [Accessed 2015].

[5] “Sirius website,” [Online]. Available: http://www.eclipse.org/sirius/overview.html. [Accessed 2015].

Appendix

Metamodel in high resolution

CaTLExpression
output : EString
outputFileName : EString

[1..1] sysProp
[0..1] context
[0..1] patternstore
[1..1] rootexpression

PatternStore

RootExpression

PatternTypeEnum
USER
OCCURRENCE
ORDER

[0..*] store
[1..1] op
[1..1] rightOp

LeftOp
name : EString

RightOp
name : EString

[1..1] leftOp
[1..1] op
[1..1] op

Pattern
name : EString
type : PatternTypeEnum = USER
description : EString
generalCaTL : EString

Contexts

[1..1] leftOp
[1..1] op
[1..1] rightOp
[0..*] staticVars
[0..*] cntxs
[2..*] op

TempForm

BooleForm

[1..1] op
[1..1] op
[1..1] op
[2..*] op

StaticContextVar
name : EString

Context
cntxName : EString

[0..*] connections

SystemProperties

UntilForm

Globally

Future

NextForm

AbstractAtomicFormulas
temperature : TemperatureEnum = HOT

OrForm

NegationForm

AndForm

ImpForm

[0..*] nodes

Node
name : EString

Connection
name : EString

[1..1] leftNode
[1..1] rightNode

[0..*] props

TemperatureEnum
HOT
COLD

[0..*] properties

SystemProperty
label : EString

Propositions

TimingConst
relation : ComparisonEnum = LESS
dynamicClock : EChar = t
StaticTimingVariable : EString

ContextConst
operator : ContextOperatorEnum = NOTHING

PropertyConst
relation : ComparisonEnum = LESS
value : EString

[1..1] usedProp

Property
name : EString

[1..1] prop
[1..1] cont
[0..1] usedNode
[0..1] usedConnection

ComparisonEnum
LESS
EQUAL
MORE

ContextOperatorEnum
NODE_EXC
NODE_ADD
CONNECTION_EXC
CONNECTION_ADD
NOTHING