# Higher Quality Requirements Specifications through Natural Language Patterns

Christian Denger
*Fraunhofer Institute for Experimental Software Engineering*
*D-67663 Kaiserslautern*
*Germany*
*denger @iese.fhg.de*

Daniel M. Berry
*School of Computer Science*
*University of Waterloo*
*Waterloo, ON N2L 3G1*
*Canada*
*dberry @uwaterloo.ca*

Erik Kamsties
*Software Systems Engineering*
*University of Essen*
*D-45117 Essen*
*Germany*
*kamsties @sse.uni-essen.de*

## Abstract

*In most current industrial software engineering projects, the majority of requirements documents are written almost entirely in natural language. However, specifying the requirements in natural language has one major drawback, namely the inherent imprecision, i.e., ambiguity, incompleteness, and inaccuracy, of natural language. Since the requirements document forms the basis of the whole development process, such defects can have severe consequences for the whole project. Therefore, it is important to deal with these defects in a requirements specification right from the start. This paper presents an approach for reducing the problem of imprecision in natural language requirements specifications with the use of natural language patterns, which allow formulating requirements sentences in a less ambiguous, more complete, and more accurate way. To ensure the applicability of our approach we based our patterns on a metamodel for requirements statements for embedded systems. With this metamodel, we ensure that all forms of requirements statements are described with the patterns. We validated the effectiveness of the patterns by using them to rewrite a substantial, previously written, requirements specification to eliminate its imprecisions.*

**Keywords:** accuracy, ambiguity, authoring, completeness, embedded systems, metamodel, natural language, patterns, precision, quality, requirements specification, rewriting

## 1. Introduction

For a software project to be successful, it is essential that the requirements of a software system reflect the user's needs. Despite the availability of other notations such as tables, diagrams, formal notations, and pseudo-code, most industrial requirements documents are written primarily in natural language. Since the requirements document forms the basis of the whole development process, the requirements document must be at the very least correct, complete, and unambiguous. However, specifying requirements in natural language has one major drawback, namely the inherent *imprecision*, i.e., the inherent ambiguity, incompleteness, and inaccuracy of natural language. We use "imprecision" to mean the union of all three problems, ambiguity, incompleteness, and inaccuracy.

An imprecise requirement can have severe consequences for a project, such as wrongly implemented requirements and, as a result, high costs for rework and for delayed product releases. Often, the different stakeholders of the process are not even aware of the imprecision. Each stakeholder has her own understanding of the requirement and none is aware that the others may have different understandings [8]. Recent research on approaches [6] for overcoming the problems of the use of natural language show that this topic needs more serious consideration in the requirements engineering community. The idea to use requirements statement patterns to improve the quality of requirements specifications is considered also in recent research [9].

Detecting imprecisions after the requirements are written is one approach to analyze the requirements document for imprecisions and other requirements defects. The approach of this paper is to avoid introducing such problems during the authoring process rather than checking for the defects after the requirements are written. The proposal is for the requirements author to use *natural language patterns* that guide her in specifying requirements that are less ambiguous, more complete, and more accurate. Hereinafter, these natural language patterns are usually called simply "patterns". Occasionally, they are called "language patterns" when they need to be distinguished from other kinds of patterns.

With these patterns, the level of imprecision in the requirements specifications is reduced right from the start. Because requirements defects are at least an order of magnitude more expensive to correct when they are undetected until later, implementation stages [16], we believe the approach to be cost effective, even if it doubles the time to write the requirements specifications.

More and more of the systems and devices we use everyday, e.g., automobiles, aircraft, and pacemakers, contain computers controlling the behaviors of these systems and devices. These computers are called *embedded* systems. Failures in embedded systems can be catastrophic because human lives may depend on the systems and devices that contain the embedded systems. Given the importance of the embedded systems domain, it is



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

surprising that there is hardly any advice in the literature on writing natural language requirements specifications in this domain. Therefore, the focus of our approach is on requirements for embedded systems. Accordingly, we developed a metamodel for requirements statements in embedded systems, defining the elements needed to specify requirements in this domain. The metamodel serves as a basis for developing the patterns and ensures their completeness, i.e., all elements that are needed to describe a requirement of an embedded system can be specified.

Section 2 gives a brief overview of related work. Section 3 discusses a metamodel for requirements of embedded systems and some of the patterns based on the metamodel. Section 4 describes the validation of our approach. Section 5 presents the lessons we learned during validation and the impact of the validation on our approach. Finally, Section 6 summarizes the paper and gives some hints about future work.

## 2. Related Work and Our Work

A survey of the literature on the use of natural language in requirements documents shows that there are several approaches to reduce the problems arising from the use of natural language. These approaches may be divided into two main types:

- approaches for detecting imprecision in previously written requirements documents [5, 6, 7, 15, 18], and
- approaches for preventing the introduction of imprecision into requirements documents being written [2, 3, 11, 12, 14].

The aim of the first type of approaches is to improve inspection. Thus, keywords and rules are identified to help a requirements engineer detect potentially ambiguous, incomplete, and inaccurate requirements statements in a previously written requirements specification.

Our approach is of the second type and aims to prevent the introduction of imprecision during authoring. Thus, we consider only the related work about the second type of approaches.

The patterns defined by Rolland and Proix [13, 14], by Ohnishi [11, 12], and by Ben Achour [3] help reduce the level of imprecision of requirements statements in the information system and the database domain. In each approach, its authors define a small set of whole-sentence patterns that may be used to describe and standardize requirements in its domain.

The patterns devised by Barr [2] focus on the specification of time constraints in embedded systems, and not on other aspects of requirements in this domain. Barr's patterns are used to check the completeness of our time-constraint patterns; we made sure that the coverage of our patterns is at least that of Barr's.

Each of the approaches found in the literature has proved to be useful to help make requirements statements more precise. Since our approach aims at reducing the problems of the use of natural language in the embedded system domain, we have converted some of the ideas of verb classes [11] and of cases [13, 14] to this domain.

In contrast to earlier approaches, our approach is based on a requirements-statements metamodel, developed specially for embedded systems. The metamodel serves as a basis for developing patterns for requirements statement regarding an embedded system [10]. Thus, the patterns describe the elements that need to be in such requirements statements. Since the metamodel describes all possible requirements statements for the domain, the completeness of the patterns derived from the metamodel can be ensured.

Konrad and Cheng [9] explore the idea of using a metamodel of a system's requirements to characterize the requirements specifications for the system. Our metamodel focuses on small-scale requirements statements in the context of embedded systems and not on large-scale aspects, such as sensor-actuator dependencies or fault handlers, which are described by Konrad and Cheng's patterns. Their metamodel is on a higher level than ours, which is for single requirements sentences. However, their metamodel should be useful should we ever need to extend our approach with large-scale patterns.

Finally, Smith, Avrunin, Clarke, and Osterweil provide patterns for disciplined natural language descriptions of software properties in the context of formalizing informal requirements [17]. For the simple reason that practitioners, for better or for worse, will not work with formal specifiations regardless of the benefits, our focus is staying with natural language specifications, but making them as clear and unambiguous as possible.

Our approach focuses not on complete-sentence patterns, but on language patterns for sentence parts, e.g., for events, conditions, system reactions, exceptions, etc. These patterns are then combined into sentence patterns describing complete requirement statements. Afterwards, the sentence patterns are combined into scenarios for complete requirements specifications. Thus, the patterns are modular, and thus, they are more flexible than those of approaches that focus on complete sentence patterns. The requirements writer may combine the different elements of a requirements statement, e.g. events, conditions, reactions, and exceptions as needed in a given situation. In addition, the structure of the sentence patterns guides the requirements writer in performing the combination. Thus, our approach gives the author more freedom of expression during authoring while obtaining more precise requirements specifications through standardization. Finally, the metamodel facilitates changing and extending the patterns.

In addition to the patterns, we developed authoring rules to be used in combination with the patterns. These rules describe how to use natural language and the patterns themselves in order to achieve the desired reduction of ambiguity, incompleteness, and poor understandability of requirements. Space limitations prevent presenting these rules in this paper, although we occasionally do justify a textual change by appeal to these here-unstated rules. Fortunately, the rules make sense also in everyday writing. The full set of authoring rules can be found in the first author's Master's thesis [4].

## 3. Metamodel and Language Patterns

This section gives only an excerpt of the metamodel and patterns for requirements statements in the embedded systems domain. Due to space restrictions we could not include the complete metamodel and all of the patterns. To facilitate the explanation, we focus on those parts of the metamodel that are most



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

relevant in describing a *user's* interaction with a system, i.e., user events or actions and system reactions. Only the patterns describing events, reactions, and conditions are presented. The complete metamodel and the full set of patterns can be found in the first author's Master's thesis [4].

## 3.1. Metamodel for Embedded Systems

Devising language patterns requires identifying what must be described with the patterns. The focus of our approach is on patterns for functional requirements of embedded systems. Therefore, we developed a metamodel that describes the elements of statements of user-level functional requirements of embedded systems. This metamodel was developed by analyzing (1) the literature about embedded systems and (2) existing requirements specifications of embedded systems to determine the types of behaviors that are specified for embedded systems and the language elements are needed to specify these behaviors. These elements are then represented in the metamodel.

Some benefits that we gain by using a metamodel to derive the patterns are, in order of increasing importance:

1. Each element of the metamodel describes a language element that is used in specifying the behaviors of embedded systems. Then, each element of the metamodel is covered by at least one pattern. Thus, to the extent that the metamodel captures all relevant aspects of embedded-systems-requirements statements, we know that the derived patterns are complete.
2. Because each pattern is related to a metamodel element, whenever the domain changes and thus the metamodel changes, the requirements engineer can easily identify the patterns that need to be changed.
3. The metamodel gives us modular patterns. The metamodel defines how the elements of a requirements statement are related to each other and how they may be combined into complete requirements sentences. Thus, a requirements writer may combine elements as needed. This ability to combine elements makes our approach more flexible than those that define only sentence patterns whose elements cannot be combined as needed.

Figure 1 (Figures 1 and 2 are after the references.) shows one diagram of the metamodel for requirements statements for embedded systems. Notice the root class Partial Function and its two subclasses Continuous Behavior and Discrete Behavior. These two Behavior subclasses have a number of components, some of them shared. The diagram thus specifies that a requirements statement may be describing what is called a "partial function" and that two types of partial functions are continuous behaviors and discrete behaviors. (We say only "may be", because we are, as explained, not showing the full metamodel, which describes all possible requirements statements.) The componentry of the two Behavior subclasses describe the language elements that are needed to describe the semantics of the Behaviors. For example, the metamodel insists that a requirements statement that describes a Discrete Behavior of an embedded system consist of one or more Events followed by one or more Reactions triggered by those Events.

Some details of the componentry of the Behavior are described in the same diagram, and other details are best given in another diagram, such as that of Figure 2. The division of the metamodel into diagrams is dictated by the need to avoid overly complex and cluttered diagrams. Figure 2's diagram describes additional elements that are needed to completely describe Events and Reactions. For example, the diagram says that there are three types of Events and two types of Reactions. It says also that a Reaction may have a duration that defines how long the Reaction is performed or that it may have an explicitly defined Begin and an explicitly defined Completion.

The following section gives some of the patterns needed to specify a discrete behavior of an embedded system. They are all derived from the portion of the metamodel shown in this section and are sufficient to deal with the example requirement statement given later in the paper.

## 3.2. Selected Language Patterns

Table 1 summarizes the full set of patterns derived from the metamodel. In any row of Table 1, the pattern name in first column serves to describe the situation in which to use the specific patterns listed in the second column. The following abbreviations are used in this table: `SP` = sentence pattern, `EP` = event pattern, `RP` = reaction pattern, `CRP` = conditioned reaction pattern, `ABCP` = abstract boolean condition pattern, `BCP` = boolean condition pattern, `SCP` = state condition pattern, `TCP` = time condition pattern, `CCP` = continuous computation pattern, `PP` = priority pattern, `OP` = order pattern, `EXP` = exception pattern, `ERP` = exception reaction pattern, `RoRP` = realization of reaction pattern, `RoCP` = realization of computation pattern, `CoRP` = completion of reaction pattern, `ADP` = assumption duration pattern, `NFRP` = nonfunctional requirements pattern.

In the descriptions of the patterns, the following notational conventions are used. Each parenthesized all-upper-case Times-Roman term defines the role that the preceding sentence part plays in the containing requirements statement. In the case of a verb, an all-upper-case term specifies the verb-class. A light-faced Courier term represents a pattern class. A bold-faced Courier term represents a constant pattern element. A light-faced Courier oblique term represents a variable pattern element. An angle-bracketed term represents an optional pattern element. Each parenthesized all-lower-case Times-Roman term specifies additional information about the preceding sentence part. Each Helvetica term refers to an element that is defined in the metamodel. A pair of curly braces, "{ }", serves to delimit the scope of the contained "|", meaning "or".

The subset of the patterns that are relevant to the example we use to illustrate our approach is shown in Figures 3 through 6. They cover key portions of patterns for requirements sentences, events, conditions, and reactions. Following the metamodel excerpt, a requirements sentence describes either a discrete behavior or a continuous behavior. Furthermore, the metamodel insists that if a requirements sentence describes a discrete behavior, then the sentence consists of one or more events and one or more reactions. If the sentence describes a continuous behavior, then the sentence consists of a starting event, a computation, and a stopping event. Alternatively, the sentence may



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

| Pattern Content | Pattern Name |
|---|---|
| Functional Requirement Sentence Patterns | SPD, SPC |
| Event Patterns | EP1, EP2, EP3, EP4, EP5, EP6 |
| Reaction Patterns | CRP, RP1, RP2 |
| Computation Pattern | CCP |
| Condition Patterns | ABCP, BCP, SCP, TCP1, TCP2, TCP3, TCP4, TCP5, TCP6 |
| Relationships Patterns | PP, OP1, OP2, OP3, OP4, OP5, OP6 |
| Exception Patterns | EXP1, EXP2, ERP1, ERP2, ERP3 |
| Patterns for Special Aspects | RoRP, RoCP, CoRP, ADP |
| Nonfunctional Requirement Sentence Pattern | NFRP |

Table 1: Overview of Patterns

SPD: *Phrase that contains an event* (EVENT)
**then** *Phrase that contains a reaction* (REACTION)

SPC: *Phrase that contains a start event* (EVENT)
*Phrase that contains a continuous behavior* (COMPUTATION)
{ **for as long as** *Phrase that contains a condition* (CONDITION) |
**until** *Phrase that contains a stop event* (EVENT) }

Figure 3: Functional Requirement Setence Patterns

EP1: <**When** | **If**> (conjunction)
*noun phrase* (VARIABLE) *verb* (VALUE CHANGE)
{*numeral adjective* (VARIABLE VALUE) | *noun phrase* (VARIABLE)}

EP3: <**When** | **If**> (conjunction)
*noun phrase* (ACTOR | RECEIVER) *verb* (ACTION | COMMUNICATION) *noun phrase* (ACTUATOR | OBJECT)

EP5: <**When** | **If**> (conjunction)
*noun phrase* (ACTOR | VARIABLE | STATE OF)
**within** *time* (variable of type Time)

Figure 4: Event Patterns

BCP: **If** (conjunction)
*noun phrase* (VARIABLE) {**is** | **is not**} (CURRENT VALUE)
<*adjective phrase* (comparison statement, e.g. greater than, less than, etc.)>
*comparison complement* (VARIABLE VALUE | VARIABLE) <TCP1 - TCP5> (DURATION)
**then** (conjunction)
<ERP1> (else case of the condition)

TCP1: **for** *time* (variable of type Time)

TCP2: **for at least** *time* (variable of type Time)

TCP3: {**for** <**not**> **more than** | **for at most**} *time* (variable of type Time)

TCP4: TCP2 {**but**|**and**} TCP3

TCP5: {**for as long as** | **while**} ABCP | BCP | SCP

TCP6: **until** {EP1 | EP2 | EP3}

Figure 5: Condition Pattern

CRP: BCP | SCP | TCPx (CONDITION) {RP1 | RP2} (REACTION)

RP1: <EP1 | EP2 | EP3 | EP4> (BEGIN) *noun phrase* (ACTOR) *verb* (ACTION) *noun phrase* (ACTUATOR)
{ <*Phrase that contains a timed condition*> (DURATION) |
<*Phrase expressing a completion time*> (COMPLETION) }
<*Phrase expressing how the reaction is realized*> (REALIZATION)

Figure 6: Reaction Patterns



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

consist of a computation and a condition describing the duration of the computation. Of course, it is possible that an instance of one pattern contains instances of the same pattern or other patterns. For example, an instance of a reaction pattern may contain an instance of an event pattern or of a condition pattern. An instance of a relationship pattern is used to connect two or more sentences and to describe the ordering or the priorities of the connected sentences. An instance of an exception pattern is used to describe a nonroutine system behavior or a deviations from expected behavior.

### 3.2.1. Some Sentence Patterns

Two of the sentence patterns are found in Figure 3. Discrete behavior is described with the `SPD`[1]; and continuous behavior is described with the `SPC`. A discrete behavior specifies a system reaction in response to an event. A continuous behavior specifies a reaction of the system that is started by a certain event and is performed until another event occurs or a certain condition comes true.

### 3.2.2. Some Event Patterns

The metamodel describes several types of events that can occur in the embedded systems domain. Some of the event patterns derived from the metamodel are found in Figure 4.

The `EP1` describes an event in which the value of a variable in the environment of the system is changed. An instance of the `EP1 is`[2]:

> If (conjunction) the water pressure (VARIABLE) rises above (VALUE CHANGE), the maximum value (VARIABLE) ...

The `EP3` describes a specific form of signal event, namely a communication event with or within an embedded system. An instance of the `EP3` is:

> If the system (ACTOR) receives (COMMUNICATION) the signal 'start' (ACTUATOR) ...

The `EP5` describes a time-out event. An instance of the `EP5` is:

> If (conjunction) the system (RECEIVER) does not receive (expression of a timeout) the signal 'temp' (OBJECT) within 4 ms (variable of type Time), ...

These patterns and others standardize the elements that must be contained in an instance of an event specification, and thus help ensure that the event is precisely specified. For example, the `EP3` ensures that when someone is specifying a communication event, she must specify the ACTOR or sender of the OBJECT in the communication, the OBJECT that is transmitted in the communication, and the RECEIVER of the OBJECT in the communication. Furthermore, she gets explicit guidance about which kind of pattern should be used for which kind of event.

### 3.2.3. Some Condition Patterns

The metamodel describes several types of conditions that can be used to constrain events in the embedded systems domain. Some of the condition patterns derived from the metamodel are found in Figure 5.

The difference between an event and a condition must be clarified. An event is a change in the value of a variable or a change in the system state. A condition is a test of the current value of a variable or a test of the current system state. Thus, a condition is a predicate about a variable or the system state.

In the full metamodel, two classes of conditions are differentiated, simple and complex. The `BCP` describes simple conditions. An instance of the `BCP` is: is:

> If (conjunction) the battery value (VARIABLE) is (CURRENT VALUE) less than (comparison statement) 9 V (VARIABLE VALUE) for more than 4 sec (DURATION), then ...

The phrase "If the battery value is less than 9 V, then ..." utilizes the `BCP` for boolean conditions. The noun phrase "the battery value" is the variable that is compared and the verb phrase "is less than" the adjective phrase "less than" that tells how the comparison is to be done. The phrase "9 V" is the value to which the variable is compared. The optional part `TCP1 - TCP5` refers to a time condition pattern. The phrase "for more than 4 sec" is an example of a time condition. If the condition is not fulfilled, the system has to perform a special exception reaction described by the `ERP1`, which cannot be described here due to space limitations.

In general, a time condition is a special form of a simple condition. A time condition represents the duration of a computation, reaction, or condition, and as a result, must always be connected to a phrase describing a computation, reaction, or condition. `TCP1` through `TCP6` are time condition patterns.

### 3.2.4. Some Reaction Patterns

The portion of the metamodel in Figure 2 shows that patterns for at least two types of reactions are needed. These patterns are found in Figure 6.

In normal natural language, the events in a narrative are understood as taking place in the order in which they are written. In a conditioned reaction, the condition must be evaluated before the reaction is performed. Thus, the requirements writer should state all relevant conditions restricting a reaction immediately before the reaction is stated. This rule is realized in the pattern, `CRP`, for conditioned reactions. An instance of the `CRP` is:

> <Event>. If the current speed is greater than 60 km/h (CONDITION), then the system initiates the automatic braking (REACTION).

The `RP1` describes a single, unconditioned reaction that is not a communication action. To specify a communication action, a different pattern, not shown here, should be used.

---

[1]Note that each of the pattern names is an acronym for a phrase that includes the word "pattern". Therefore, it is both incorrect and unnecessary to say "`SPD` pattern"

[2]Whenever an instance of a pattern is given, the roles of components are identified by parenthesized expressions following the components.



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

## 4. Validation

The patterns derived from our metamodel of requirements specifications of embedded system were subjected to validation of their effectiveness in

1. helping the basic writing of requirements specifications for embedded systems, and in
2. helping to avoid the introduction of imprecision in these requirements specifications.

Since the focus of the patterns is the reduction of imprecision in functional requirements statements, the validation focused on functional requirements.

The reader should observe that the goal of the case study was to validate only the structure of the patterns, to prove the concept of the approach. The practical usefulness of the patterns and of the approach in terms of effort to apply the patterns is an open issue that will be dealt with in our future work.

To carry out the validation, we rewrote an existing requirements document using the patterns. During validation, we asked of each requirements statement:

1. Is it possible to use the patterns to completely rewrite the requirements statement?
2 Is it possible to rewrite the requirements statement without lengthening the statement?
3. Is it possible to improve the precision of the requirements statement by using the patterns? This question includes the subquestions:
   a. Is it possible to reduce the ambiguity of the statement?
   b. Is it possible to improve the completeness of the statement?
   c. Is it possible to improve the accuracy of the statement?

These questions served as our evaluation criteria.

We rewrote the requirements specification of the Enhanced Voice Mail System (EMS), a system situated in the field of communication systems. The EMS was developed for a case study on scenario networks [1] and is a multifunction system for telephone voice messaging. The requirements were written jointly by a domain expert and a requirements engineering expert. It must be understood that the requirements of the EMS are not stated in any particular specification form, but that they represent only functional requirements. In total, 42 system requirements were specified. Besides the system requirements, several scenarios are presented to clarify the requirements. These scenarios were not rewritten as part of the validation exercise. However, whenever a requirement was not clear, the related scenario was considered to clarify the requirement. In particular, several conditions are specified only in the scenarios. The relevant snippets of natural language scenario specification were considered requirements specification and were subjected to rewriting.

To rewrite the requirements of the EMS, the requirements statements of the original document were analyzed in order to identify defects. The original requirements were reviewed with the help of the authoring rules we developed in addition to the patterns. Whenever an original requirements statement violated a rule, the nature of the violation was noted. Then, the patterns were used to rewrite the statement into one without the violation. That is, the original statement was classified into its type according to what it is trying to describe, e.g., discrete behavior or continuous behavior. From the type, the relevant patterns were identified. The statement was decomposed into its parts, describing events, conditions, reactions, etc. The closest matching pattern was identified by attempting to match the parts of the statement to parts of the relevant patterns. Once the closest matching pattern was identified, it was possible to see which parts of the pattern had no matching part in the statement, and thus what parts are missing in the statement. Then, the statement was rewritten according to the pattern. Finally, the original and the rewritten statements were compared according to the evaluation criteria given at the beginning of this section.

Sections 4.1 and 4.2 show one representative problematic requirements statement and its rewrite. Space limitations preclude going into more than one example. The complete analysis and rewriting of the EMS requirements specifications can be found in the first author's Master's thesis [4].

### 4.1. Analysis of the Original Requirements

One of the original requirements statements is:

> R3.1.4. Stuttered dial tone: EMS shall support notification by stuttered dial tone played for as long as no key is pressed; that is, EMS shall interact as necessary with other systems so that when the subscriber has one or more new messages, the subscribed phone will give a stuttered dial tone rather than a standard dial tone.

Analysis of this requirement reveals several ambiguous phrases. The first clause, "EMS shall support notification by stuttered dial tone played for as long as no key is pressed", is refined in the second clause, "EMS shall interact as necessary with other systems so that when the subscriber has one or more new messages, the subscribed phone will give a stuttered dial tone rather than a standard dial tone.", following the phrase "that is". The reader who has seen the original requirements specification can see that the phrase "played for as long as no key is pressed" was added to the original requirements specification so that we could apply more of the patterns. The first clause is recognized as matching `RoRP`. The clause contains a condition "when the subscriber has one or more messages" and a reaction "the subscriber's phone plays a stuttered dial tone". The event that triggers the reaction is not specified because the event is implicitly described via the vague phrase "EMS shall interact as necessary with other systems" and the name of the requirement "Stuttered dial tone". This vague phrase contains two sources of imprecision, namely the phrases "interact as necessary" and "other systems". The phrase "as necessary" violates one of our authoring rules, namely the one that is against using phrases that are open to subjective interpretations. The phrase "other systems" is also ambiguous, since it is not clear which other systems are referenced. Finally, the phrase "rather than a standard dial tone" is removed as redundant, since a stuttered dial tone is *not* the standard dial tone.



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

## 4.2. Rewriting the Requirement

The requirement R3.1.4 can be rewritten with the SPD, since R3.1.4 describes a discrete behavior. The event follows the EP3, and the reaction follows the CRP. The reaction within the CRP can be described by the RP1, which has a duration, which in turn can be described by the TCP5. The EMS's interaction "as necessary with other systems" is describable by RP2. The time ordering of the reactions is described by the OP2. The condition of the reaction is described by the SCP, and the exception reaction for when the condition is not fulfilled is described by the ERP2. The rewritten requirements statement is:

> 3.1.4 FR.Stuttered_Dial_Tone: If the subscriber picks up the phone and a message is waiting, then the subscriber's phone plays a stuttered dial tone for as long as no key is pressed. For this purpose, the central office system sends the signal 'has new messages' to the subscriber's phone. Then, if the subscriber's phone state is 'has new', the EMS sends the signal new messages' to the central office. Then, the central office sends a stuttered dial tone to the subscriber's phone for as long as no key is pressed. If the subscriber's phone state is 'has no new', the EMS sends the signal 'no new messages' to the central office. Then, the central office sends a standard dial tone to the subscriber's phone, for as long as no key is pressed.

This example shows how applying the patterns helps eliminate ambiguous, incomplete, and inaccurate requirements statements. Basically, the requirements writer has to supply all essential parts. Thus, the event that triggers the reaction in this example "If the subscriber picks up the phone and a message is waiting" gets specified. Moreover, the implicitly specified exception reaction that is performed when the condition does not hold ends up being explicitly specified. Finally, the vague phrase "other systems" is replaced by the explicit name of the system with which the EMS interacts. When information is missing, it is necessary to learn it, usually from the client or user. In this case, neither the client nor user was readily accessible. Fortunately, the event that triggers the interaction and the interaction itself were described in full detail in the accompanying scenario document.

## 4.3. Evaluation of the Rewriting

After the sources of imprecision in the original requirements statements were identified and the imprecise statements were rewritten according to the patterns, it was necessary to evaluate the results according to the questions raised at the beginning of Section 4. The full evaluation is found in the first author's Master's thesis [4]. Here, we answer these questions for the example requirements statement analyzed and rewritten in Subsections 4.1 and 4.2.

Clearly the patterns were used to completely rewrite the imprecise requirements statement. The resulting statement is clearly more precise than the original; it provides all the information that was missing in the original:

- it provides the triggering event,
- it provides the exception reaction for when the condition does not hold,
- it provides the name of the other system,
- it provides the full details of the interaction with the other system, and
- it provides an explicit ordering of the events, condition testing, and reaction,

that were missing in the original statement. The rewritten statement is less ambiguous, more complete, and more accurate.

However, it was not possible to rewrite the requirements statement without lengthening it. In fact, in most cases, the rewritten statement *is* longer than the original. In this case, the original statement was *missing* information. Adding missing information to any statement is bound to make the statement longer. When information is missing in a statement, making the statement longer by adding the missing information must be regarded as a virtue. If it is necessary to use a longer statement in order to get a more precise statement, so be it.

There is yet another possible drawback of the rewritten statement. It can be regarded as clumsier, less clear, and not as well written as the original. While the increase in length in the rewritten statements can be regarded as a virtue because missing information is provided, the resulting clumsiness of expression cannot be regarded as a virtue. Indeed, a referee of a previous version of the paper said that he or she thought that the rewritten version is harder to read than the original and even suggested a better rewriting, using passive voice:

> R3.1.4. Stuttered dial tone: EMS shall support notification of new messages as follows. When the phone is idle and one or more new messages are waiting, then if the phone senses offhook, it plays a stuttered dial tone for no longer than 0.5 seconds after offhook until a key is pressed or onhook is sensed.

Prompted by the referee's rewriting, we produced yet another rewriting, using active voice:

> 3.1.4 FR.Stuttered_Dial_Tone: The subscriber picks up the phone. If the subscriber has one or more new messages, then the subscriber's phone plays a stuttered dial tone for as long as no key is pressed or the phone is onhook. If the subscriber has no new messages, then the subscriber's phone plays a standard dial tone for as long as no key is pressed or the phone is onhook.

We agree that these rewritings are improvements over our pattern-directed rewriting. Each is at once less clumsy, clearer, shorter, and more to the point than our rewritten statement. In particular, it is less clumsy than the original statement because it replaces each instance of the repeated "the subscriber's phone state is 'XXX'" construction, which smacks of implementation detail, by a clear user-relevant adjective that means the same thing. Moreover, neither rewriting suffers from any of the imprecisions of the original. The comments and suggestions of the referee's report indicate that the referee is an expert in ambiguity in natural language requirements specifications and is very skilled in writing precise requirements specifications. Moreover, for the second rewriting prompted by the referee's rewriting, we were in a more artistic writing mode, not constrained by an obligation to adhere to the patterns.

In defense of our approach, we say only that whenever a



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

skilled, ambiguity-aware, domain-expert writer is available to completely rewrite a requirements specification to eliminate all imprecision, he should be given the job. In the usual case in which a less-than-ideal writer is available, the patterns provide useful guidance to the writer that is available. The results, while not as good as that which can be produced by the skilled writer, should nevertheless be an improvement over what could have been.

On the other hand, part of the length reduction and increased clarity of the non-pattern-directed rewriting was due to a decision on the part of the writer to make the requirements statement more user oriented by leaving out details of the communication between the EMS, the central office, and the telephone. Such a decision requires judgement and cannot be encoded into the patterns. The patterns expose missing information, and it is up to the human writer to decide whether the information is indeed relevant given the audience, purpose, and level of the specification. Observe that if the patterns were to be applied to the referee's or our rewriting, then the missing information would be exposed, and supplying it would make the resulting statement longer.

Table 2 shows the defects detected in the original complete requirements. Note that incomplete requirements and implicit assumptions are related to each other; incomplete requirements cause the reader to make implicit assumptions, and implicit assumptions are symptoms of incomplete requirements:

| | |
|---|---|
| Ambiguous pronoun references | 3 |
| Complex sentences, including nontemporal order | 11 |
| Incomplete requirements | 9 |
| Tacit assumptions or information | 9 |
| Misplaced words | 3 |
| Vague and subjective words and phrases | 5 |
| Use of synonyms and homonyms | 1 |

Table 2: Frequencies of Kinds of Defects

There is yet another threat to our evaluation mentioned in the full report that needs to be mentioned here, even though only a small portion of the case study is reported here. It must be admitted that not all patterns could be validated in the case study. The EMS document does not contain enough requirements to cover all patterns. Therefore, it is recommended that the patterns be used in real industrial projects or in large experimental case studies in order to validate the patterns more fully. Nevertheless, a majority of the patterns were validated by the present case study. It was possible to rewrite each sentence of requirements document of the case study with the patterns.

To conclude this section, overall, applying the rules and the patterns helps to clarify requirements, makes requirements easier to understand, facilitates their interpretation, and eliminates several potential sources of imprecision, i.e., ambiguity, incompleteness, and inaccuracy.

## 5. Observations and Lessons Learned

During the rewriting, we observed that the original requirements document contains only a few subjective and vague phrases and no optional requirements. The requirements statements are almost all specified with short sentences, and complex descriptions are quite rare. Synonyms and homonyms occur in only one requirement. Thus, the original requirements document was actually quite well written, probably because the authors of the EMS requirements are domain and requirements experts with many years of experience.

We observed also that a single requirements statement that is supposed to mention an event, a condition, and a reaction often mentions only a condition and a reaction, leaving the event only implicitly mentioned in the form of the name of the requirement or in the form of a phrase that describes a feature the system allows the user to perform. Such a requirements statement is poorly written by its mixing another kind of requirements statement with its description of a discrete behavior. Adherence to the patterns reduces the incidence of these kinds of defects.

It is sometimes difficult to rewrite requirements with the patterns, since the information that is needed according to the pattern is not available in the original requirement. If the required information is nowhere in the requirements document, then the client and users must be consulted and asked to supply the missing information. On the other hand, the use of the patterns from the beginning helps *avoid* incomplete statements, since the requirements writer must exactly follow the predefined structure. This structure prescribes each necessary part of a requirements statement and, consequently, helps a requirements engineer to ask the right questions to elicit the missing information.

To make the patterns usable in industrial projects, we realized that it is necessary to recommend how to use the patterns. These recommendations help a requirements writer to identify the applicable pattern in any situation and give detailed advice on how to specify a functional requirement of an embedded system. These recommendations are completely described in the first author's Master's thesis [4].

During validation and in several discussions with other scientists, we could see that the patterns approach would be applicable in domains other than embedded systems. We now believe that the patterns approach is applicable for all reactive systems, including those that have heavy interaction with users. We saw also that the patterns can be used for detecting imprecision problems in already written requirements specifications. Of course, these other applications of patterns need to be validated in future research activities.

## 6. Conclusion and Future Work

This paper addresses the problem of improving the precision of natural langauge requirements specifications for embedded systems by reducing ambiguity, incompleteness, and inaccuracy in the statements of the requirements specifications. The approach described in this paper calls for the use of domain-dependent language patterns to help the embedded system requirements writer write more precise natural language requirements statements. The same patterns can help a requirements inspector find the sources of imprecision in already written natural language requirements specifications for embedded systems.

The approach required developing a metamodel for describing all possible requirements statements in the embedded



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

systems domain. From the metamodel, patterns describing all possible embedded systems requirements statements were derived. To the extent that the metamodel completely describes embedded systems requirements statements, so do the patterns.

The metamodel and its patterns were applied in a case study in order to validate their effectiveness in assisting the authors to write precise requirements about an embedded system. This paper has shown snippets from this case study, that is, it has shown portions of the metamodel, some of the patterns, one particular but typical imprecise requirements statement, and a rewriting of that requirements statement according to the patterns into a more precise requirements statement. The portion of the metamodel, the specific patterns, and the particular requirement statement were chosen to complement each other in order to allow the reader to get a flavor of how each is used in the approach. The reader that is interested in the full details should consult the first author's Master's thesis [4].

This paper has shown the part of the evaluation of the effectiveness of the approach that involves the exhibited portions of the metamodel, patterns, and original and rewritten requirements statement. The evaluation concluded that the patterns do allow reduction of imprecision in natural language requirements statements about embedded systems, however not without the possibility of lengthening the specification and not without the possibility of using clumsier language. We accepted that requirements statements are bound to be lengthened as missing information is supplied. Avoiding clumsier language requires writing skill that may not always be available.

There are other issues that need to evaluated:

- The domain of the case study is embedded systems. Does the approach work in domains other than embedded systems?
- The case study involved a previously written, actually quite polished requirements specification produced by few people for a not too large system that was the subject of academic research. Does the approach work in practice in an industrial setting, with large requirements specifications pieced together from the viewpoints of hundreds of stakeholders for a large, highly interacting system?
- The case study demonstrates only that the patterns *can* be used to write more precise specifications. Moreover, the person carrying out the case study was the author of the patterns; there is no question that he understood the approach and the patterns. The case study did not validate that the patterns can be used *effectively* by anyone and that the resulting requirements specification will be precise or at least significantly better than they would have been
  - without the use of the patterns,
  - if other author-guiding approaches had been applied,
  - if other imprecision-avoidance approaches had been applied, or
  - if imprecision-*removal* approaches, e.g., inspection, had been applied?

  Does avoidance of imprecisions in the writing of requirements specifications result in more precise requirements specifications than finding and removing imprecisions during inspection?
- The case study was carried out in a setting in which there were no project deadlines, other than that of finishing the whole Master's thesis. Is the approach cost effective? Does writing requirements according to these patterns and then finding fewer imprecisions during inspection take less time than just writing requirements and finding and correcting more imprecisions during inspection?

It may very well be that because of the effort to learn and use these patterns, our approach does not work in practice without strong tool support. Thus, it is necessary to consider the development of a tool to semi-automate the use of the patterns, perhaps in finding common signs of imprecision in draft statements, in finding applicable patterns, and in prompting for all the required parts of statements being written.

## Acknowledgments

The authors thank the referees of this and previous versions of this paper for their comments.

D.M. Berry's work was supported in part by NSERC grant NSERC-RGPIN227055-00.

## References

[1] Alspaugh, T.A. and Antón, A.I., "Scenario Networks: A Case Study of the Enhanced Messaging System", in *Seventh International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'01)*, Interlaken, Switzerland (2001).

[2] Barr, V., "Identifikation von Spezifikationsmustern im Echtzeitentwurf anhand der Fallstudie Antiblockiersystem", Diplomarbeit der Universität Oldenburg, Fachbereich Informatik (1999).

[3] Ben Achour, C., "Guiding Scenario Authoring", pp. 152–171 in *Proceedings of the Eighth European-Japanese Conference on Information Modeling and Knowledge Bases*, IOS Press, Vamala, Finland (25–29 May 1998).

[4] Denger, C., "High Quality Requirements Specifications for Embedded Systems through Authoring Rules and Language Patterns", M.Sc. Thesis, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany (2002).

[5] Fabbrini, F., Fusani, M., Gnesi, G., and Lami, G., "Quality Evolution of Software Requirements Specifications", in *Proceedings Software and Internet Quality Week 2000 Conference*, San Francisco, CA (2000).

[6] Fantechi, A., Gnesi, G., Lami, G., and Maccari, A., "Application of Linguistic Techniques for Use Case Analysis", in *Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02)*, IEEE Computer Society Press, Essen, Germany.

[7] Götz, R. and Rupp, C., "Regelwerk Natürlichsprachliche Methode", Sophist, Nürnberg, Germany (1999), http://www.sophist.de.

[8] Kamsties, E., "Surfacing Ambiguity in Natural Language Requirements", Ph.D. Dissertation, Fachbereich Informatik, Universität Kaiserslautern, Germany, also Volume 5 of Ph.D. Theses in Experimental Software Engineering, Fraunhofer IRB Verlag, Stuttgart, Germany (2001).

[9] Konrad, S. and Cheng, B., "Requirements Patterns for Embedded Systems", in *Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02)*, IEEE Computer Society Press, Essen, Germany (2002).

[10] Melchisedech, R., "Verwaltung und Prüfung natürlichsprachlicher Spezifikationen", Ph.D. Dissertation, Institut für Informatik,



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

Universität of Stuttgart (2000).
[11] Ohnishi, A., "Customizable Software Requirements Languages", in *Proceedings of the Eighth International Computer Software and Application Conference (COMPSAC)*, IEEE Computer Society, Los Alamitos, CA (1994).
[12] Ohnishi, A., "Software Requirements Specification Database Based on Requirements Frame Model", pp. 221–228 in *Proceedings of Second IEEE International Conference on Requirements Engineering (ICRE'96)*, IEEE Computer Society Press, Colorado Springs, CO (15–18 April 1996).
[13] Rolland, C. and Proix, C., "A Natural Language Approach for Requirements Engineering", pp. 257–277 in *Proceedings of Conference on Advanced Information Systems Engineering, CAiSE 1992*, Manchester, UK (12–15 May 1992).
[14] Rolland, C. and Proix, C., "Guiding the Construction of Textual Use Case Specifications", CREWS Report Series 98–1, Lehrstuhl für Informatik V, Aachen, Germany (1998).
[15] Rupp, C., *Requirements-Engineering und -Management*, Second Edition, Hanser, Munich, Germany.
[16] Schach, S.R., *Object-Oriented and Classical Software Engineering*, Fifth Edition, McGraw-Hill, New York, NY (2002).
[17] Smith, R.L., Avrunin, G.S., Clarke, L.A., and Osterweil, L.J., "PROPEL: An Approach Supporting Property Elucidation", pp. 11–21 in *Proceedings of the 24th International Conference on Software Engineering*, Buenos Aires, Argentina moved to Orlando, FL (2002).
[18] Wilson, W.M., Rosenberg, L.H., and Hyatt, L.E., "Automated Quality Analysis of Natural Language Requirements Specifications", NASA Software Assurance Technology Center, The Software Assurance Technology Center (SATC), NASA Goddard Space Flight Center (GSFC), Greenbelt, MD (1996), http://satc.gsfc.nasa.gov/support/PNSQC_OCT96/pnq.html.

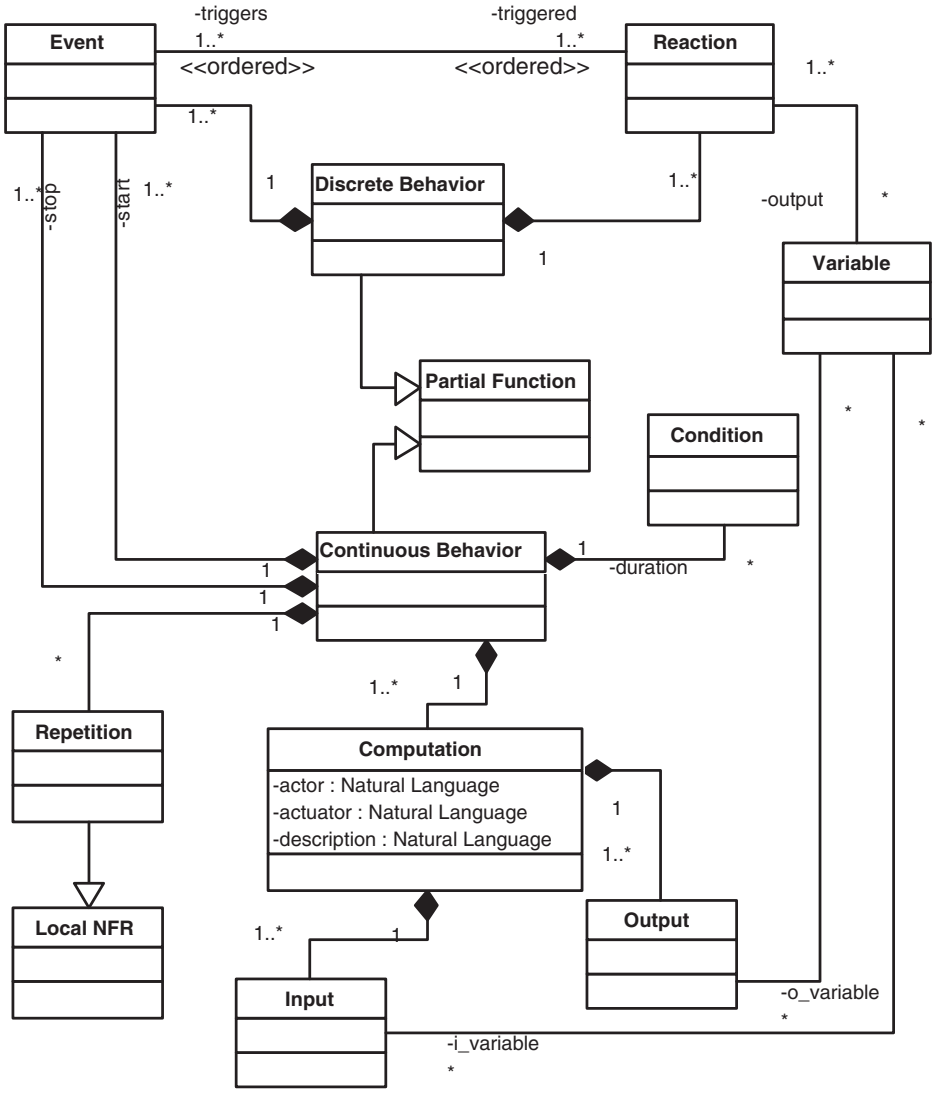


Figure 1: Metamodel for Requirements Statements



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

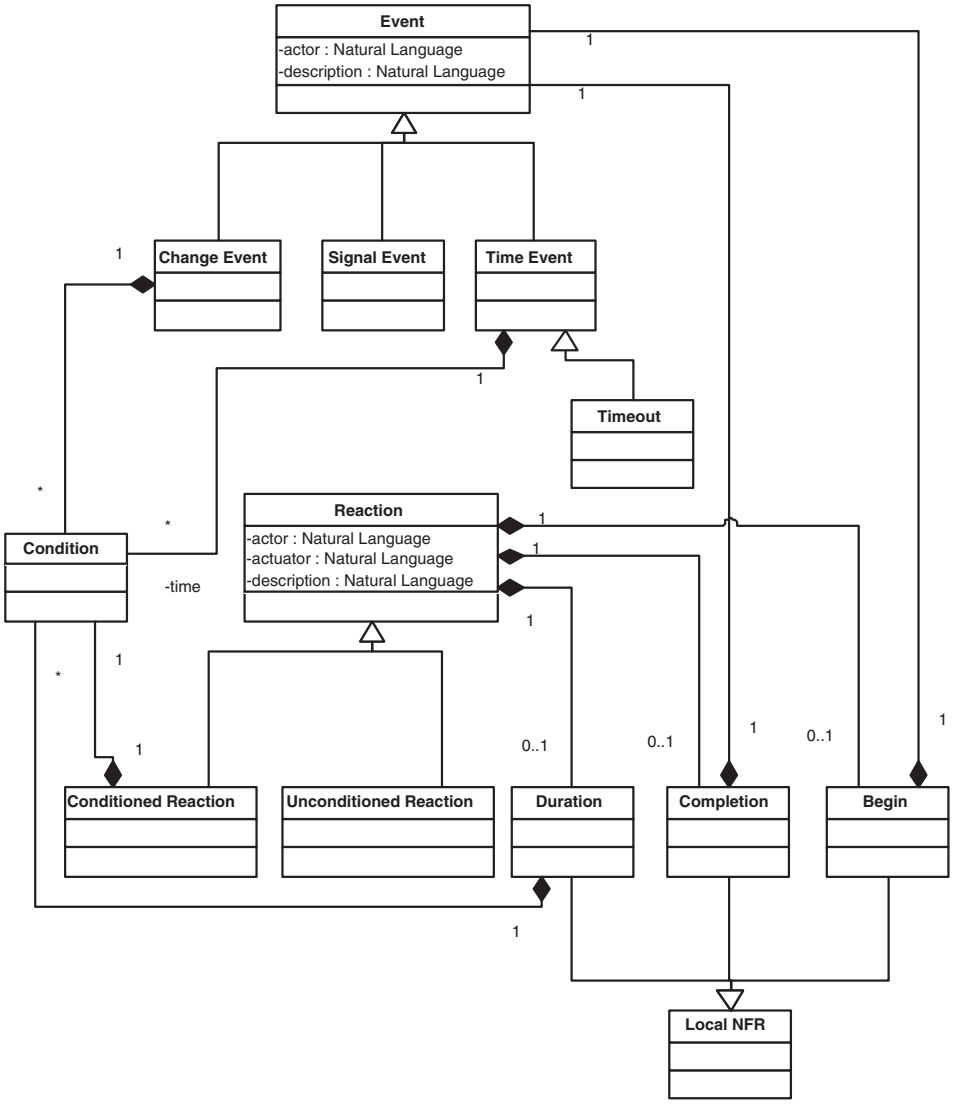


Figure 2: Metamodel for Events and Reactions



Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE