Model-based Generation of a Requirements Monitor

Fabian Kneer and Erik Kamsties

Dortmund University of Applied Sciences and Arts,
Emil-Figge-Str. 42, 44227 Dortmund, Germany
`{fabian.kneer,`
`erik.kamsties}@fh-dortmund.de`
`http://www.fh-dortmund.de/`

Abstract. Runtime representations of requirements have recently gained interested to deal with uncertainty in the environment and the term *requirements at runtime* has been established. Runtime representations of requirements support reasoning about the requirements at runtime and adapting the configuration of a system according to changes in the environment. Such systems often called self-adaptive systems. Core part of respective approaches in the field is a *requirements monitor*. That is, an instance which is able to observe the system's environment and to decide whether a requirement is broken, based on assertions.
The problem addressed in this paper is how to generate the application-specific parts of a requirements monitor. Such a monitor consists of some *goal model* for decisions at runtime, *assertions* connected to the goal model, and *parameters* on which assertions are defined. We present in this paper a model-driven approach to enhance requirements documents by goal models, assertions, and parameters in a way which is (1) understandable to requirements engineers and (2) a sufficient basis for generating the requirements monitor. The contribution is an integrated view on requirements for self-adaptive systems and a concept for code generation.

Keywords: Self-adaptive systems, requirements at runtime, requirements monitor

1 Introduction

Today's software-intensive systems are faced with anticipated and unanticipated variations in their operating context. Runtime representations of requirements have recently gained interested to deal with changing end-user requirements, operating context conditions, and resource availability [11].

Reasoning in the presence of uncertainty is a classic field of Artificial Intelligence (AI) research and some RE approaches make use of AI techniques for instance of Fuzzy logic [16]. Yet, the majority of RE approaches seek to extended established RE techniques (e.g., KAOS) to provide a system with a representation of its own requirements [17].

To our experience, the term *adaptivity* in relation to requirements needs explanation. First, the *development time* and *runtime* view of adaptivity should be separated. Considering runtime first, a system typically fulfills a set of *goals*. These goals are related to *functions* of the system. One approach to adaptivity is to enable a system to *learn new functions* from analyzing its usage, for instance by employing machine learning techniques [7], [13].

Another approach is to see changes in the requirements, context, and resources as forming a new problem to be solved [11] with a given solution. Following this idea, the relation between goal and function is augmented by an *assertion*. Such an assertion describes a Boolean condition that evaluates to true if a function fulfills a goal. For example, a sensor needs to deliver valid values. If an assertion breaks, the function has to be adapted, e.g., by changing parameters or switching to a different function. Thus, we view adaptation as monitoring *assertions* and selecting the best possible modification of functions under a given set of assertions each evaluating to true or false. The notion of a *best* possible modification has to be defined by a requirements engineer a priori.

Requirements monitoring has been researched for while. Core part of the approaches is a requirements monitor, which observes the environment, decides whether a requirement is broken, and computes a new system configuration based on runtime requirements (see Section 5).

Problem. We propose in this paper an approach for generating a requirements monitor, because a monitor contains application-specific parts that must be implemented for each new system, e.g. assertions. Thus, such a generator eases the development of self-adaptive systems.

As Figure 1 shows, in our approach a requirements monitor consists of a rule engine, which observes a set of assertions, and an impact analyzer, which analyzes the effect of a broken rule with the help of the runtime requirements (e.g., goal model). Finally, the requirements monitor releases a new configuration of the system with fits best to the changed situation. Probes are added to the application to monitor variables, which are of interest to the assertions. We add a goal model, assertions, and variables to the *development time* representation, in order to be able to generate a *runtime* requirements monitor.

Probes, assertions, and the goal model in the runtime representation are specific to the respective application and thus must be produced by the generator. The application itself results from typical software development activities and is, thus not subject of this work.

Contribution. First, we propose a metamodel which defines the additional requirements artifacts which allow to generate a requirements monitor. Second, we outline a generator for a requirements monitor. The benefit is an integrated view of the different views on the requirements for self-adaptive systems and a significant decrease of effort in building an adaptive system.

Structure. The remainder of this paper is organized as follows. Section 2 outlines the background of our research. Section 3 discusses the metamodel and Section 4 describes the generator. Section 5 reviews the related work. Section 6 concludes with a summary and an overview on our future work.

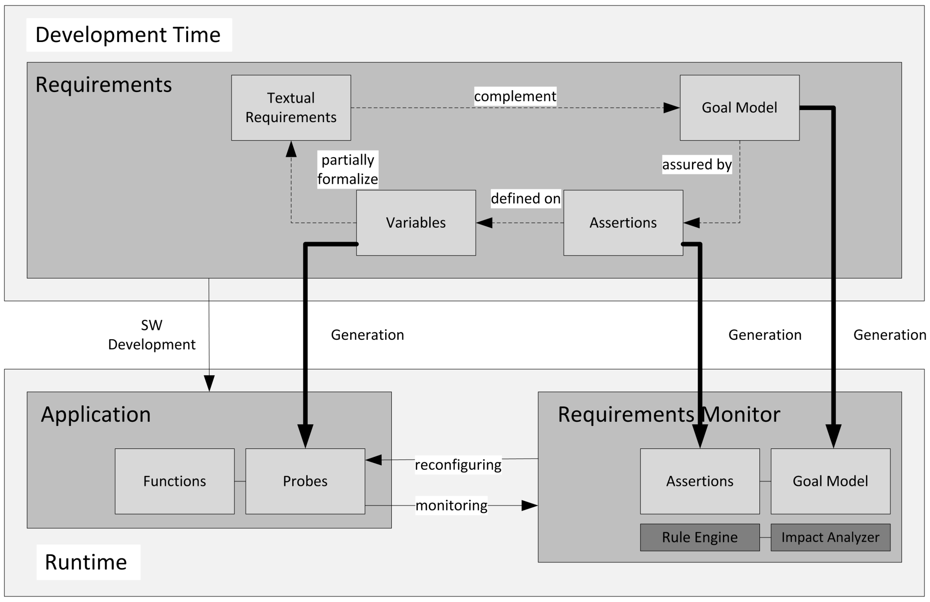


Fig. 1. Generation of a Requirements Monitor based on Development Time Artifacts

2 Background

The focus of our work is in particular on embedded systems, which impose heavy constraints on software. As these systems are mass-produced, the capabilities of the hardware are optimized to the purpose of the respective system. That is, the power of the CPU and the memory size are limited. Constraints on energy consumption prohibit more powerful hardware, since many embedded systems run on batteries.

2.1 Self-adaptive Systems

Many embedded systems act autonomously, that is they make decisions without the confirmation of a human operator. The software cannot be easily maintained or tuned to changing conditions manually. Therefore, there is a need for adaptivity. However, adaptivity conflicts with other design goals such real-time behavior, safety considerations, and the resource constraints mentioned above.

A self-adaptive system has the ability to dynamically and autonomously reconfigure its behavior in order to respond to changing environmental conditions [2]. We consider a self-adaptive system as consisting of two parts: the application and a requirements monitor (see Figure 1). Between the application and the monitor a feedback loop is established. The feedback loop consists of the steps *collect*, *analyze*, *decide*, and *act*, as described by Cheng et al. [3]. The application implements the development time requirements. The requirements monitor contains a requirements model, which is a machine-processable representation of the system's requirements. The requirements model is the basis for computing new configurations at runtime in case of environmental changes. Often, a goal-oriented model is used for this purpose (see Related Work in Section 5).

2.2 DO$_{\text{RE}}$F

For the development time representation, a semi-formal requirements language is required for the generator in order to parse the requirements. DO$_{\text{RE}}$F ("do requirements first") is an education and research framework for requirements engineering [8], it offers such a domain-specific language (DSL) for requirements and modeling. Pretty-printed requirements documents (HTML, PDF) can be generated and parts of the language can be executed for analysis purposes. The idea of DO$_{\text{RE}}$F is to let requirements engineers focus at the content and the semantics of requirements documents rather than layout.

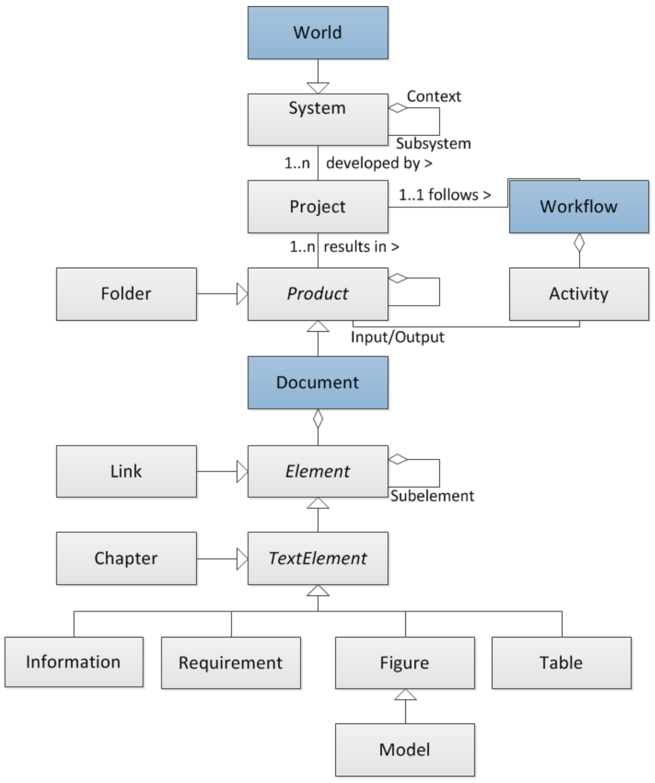


Fig. 2. DO$_{\text{RE}}$F metamodel

The language is

- comprehensive - it addresses products and processes in RE
- method-agnostic - it does not infer a particular RE approach
- extensible by modules - using modules, the framework can be tailored to a particular RE approach (e.g., i*).

All concepts of the language are organized in a single tree. Figure 2 shows the general structure of the tree. The root node is called *World*, it is a system, which composed of subsystems. A *System* is related to a project and a *Project* results in a set of documents from a product perspective. A *Document* contains *TextualElements* like requirements and *Models*. *Activities* are used to describe the RE process.

We extend $\text{DO}_{\text{RE}}\text{F}$ to cover variables, optional requirements, and i* goal models [18] in order to generate a requirements monitor. These extensions are discussed below. As a running example throughout the paper, we use a vacuum cleaner case study that was originally introduced in [2] and [1].

3 Metamodel

In this section we describe the metamodel of our approach, which is a refinement of the development time artifacts shown in Figure 1. The metamodel (cf. Figure 3), shows on the upper side the textual requirements of $\text{DO}_{\text{RE}}\text{F}$, on the lower side a subset of concepts of an i* goal model, and how assertions are connected to the i* model and the textual requirements.

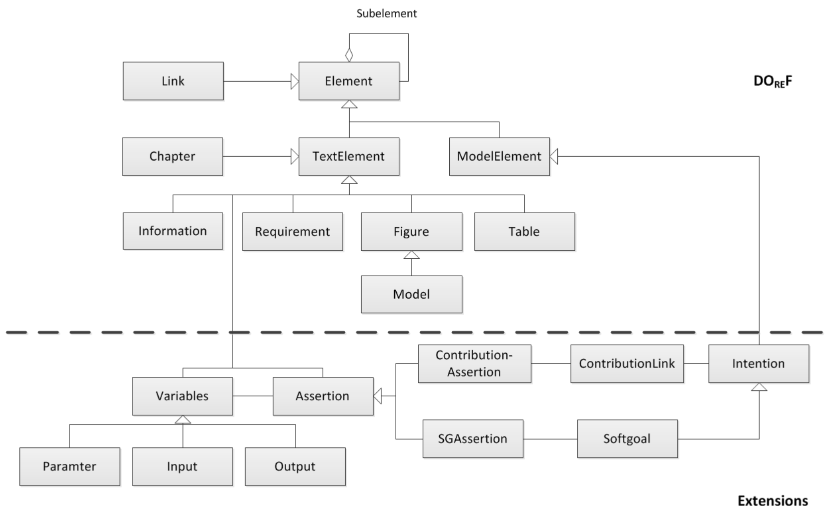


Fig. 3. Metamodel for Development Time Requirements

The goal is to generate the requirements monitor shown in Figure 1. It consists of two components: (1) a rule engine and (2) an impact analyzer. In the following, we explain briefly the interaction of the components (for details see [6]).

The rule engine monitors assertions. *Assertions* are Boolean conditions describing assumptions about the environment, which usually should be fulfilled. If an assertion fails, a requirement may be violated and the *impact analyzer* is invoked. The impact analyzer assesses which parts of the *goal model* are affected and whether a change in the model is really necessary. If a change is necessary, a new *configuration* is computed and the *system* switches to that new configuration eventually.

In the following subsections we describe the parts of the metamodel and their role in generating a requirements monitor.

3.1 Semi-formal Textual Requirements

As mentioned before, the textual requirements are written in DO$_{\text{RE}}$F (see Section 2.2), Listing 1.1 shows an example. Figure 4 shows the generated PDF requirements document. We extended the requirement with an *optional* property, which indicates whether a function that is related to a requirement is active under consideration of the environment and the current configuration of the system. The requirements *Clean at night (SRS-29)* and *Clean when empty (SRS-30)* have an *optional* property. The function associated to *Clean at night (SRS-29)* is not activated in the current configuration.

Listing 1.1. Textual Requirements with DO$_{\text{RE}}$F

```
 1  Req("Clean at night",
 2      "The robot shall clean the apartment at night.",
 3      {'Optional': "False",
 4       'Conflict': ["/*/Clean when empty"]})
 5  Req("Clean when empty",
 6      "The robot shall clean the apartment when nobody is inside",
 7      {'Optional': "True",
 8       'Conflict': ["/*/Clean at night"]})
 9  Req("Power",
10      "The suction power must not exceed ${suction_power}.",
11      {'Priority': 1,
12       'ConceptRefs': ["/*/Concept/*/Suction Power"]})
13  Req("Silence",
14      "The operation of the vacuum cleaner should be as silent as
            possible.",
15      {'Priority': 2})
```

Textual requirements are semi-formalized by means of parameters, input and output variables. These variables are marked with the annotation `${name}` (see Listing 1.1 line 10) in the textual requirements. Every variable has an unique *ID*,

3.3-1 Clean at night (SRS-29). The robot shall clean the apartment at night.
Type: Requirement
Optional: False
Conflict: Clean when empty(SRS-30) on page 6

3.3-2 Clean when empty (SRS-30). The robot shall clean the apartment when nobody is inside
Type: Requirement
Optional: True
Conflict: Clean at night(SRS-29) on page 6

3.3-3 Power (SRS-31). The suction power must not exceed 50.
Priority: 1
ConceptRefs: Suction Power (ID-23) Parameter: maxSuction: Int - 50 input: suction: Int
Type: Requirement

3.3-4 Silence (SRS-32). The operation of the vacuum cleaner should be as silent as possible.
Priority: 2

Fig. 4. Textual Requirements with DOReF

a *type* and a default *value*. The requirement *Power (SRS-31)* has a parameter *maxSuction*.

An *Input* variable describes an environmental quantity, which is monitored by the system. An *Output* variable describes an environmental quantity, which is controlled by the system. A *Parameter* is used to tune a function, e.g., to adapt a particular threshold.

The information for the parameters and variables are identified during requirements elicitation with stakeholders, they result from variable characteristics in the domain of the system. One simple example in the automotive domain is a windshield wiper. It has parameters for calibrating a rain sensor and the speed of the wiper.

On the upper side of Figure 3 the *Requirements* class is shown. Figure 4 shows that it contains a unique *ID*, a *Name* and a short *Summary*. A requirement can have different *Properties*. For example *Type*, *Effort* or a reference like *ConceptRefs* (see Listing 1.1 line 12). For the requirements monitor we add an *Optional* property. The property indicates if the function associated to the requirement is active or not.

3.2 Goal Model

We use the goal-orientated modeling language i* [18] to model the requirements at runtime and to calculate a new configuration for a system.

A simple implementation of i* is provided by the openOME[1] tool. We use its meta model, which is defined based on the Eclipse Modeling Framework[2] (EMF). For a better view on our metamodel (see Figure 3) we only show those parts of this model, which are related to the extension, but not the full i* metamodel.

The relevant parts are the *Intention/ i*-Elements*, the *Softgoal* and the *ContributionLink*. Every Intention (Goal, Task, Resource and Softgoal) can be related to a requirement. This relationship is build over *References* in the properties, for example *RelatedTo* in Listing 1.2 line 8.

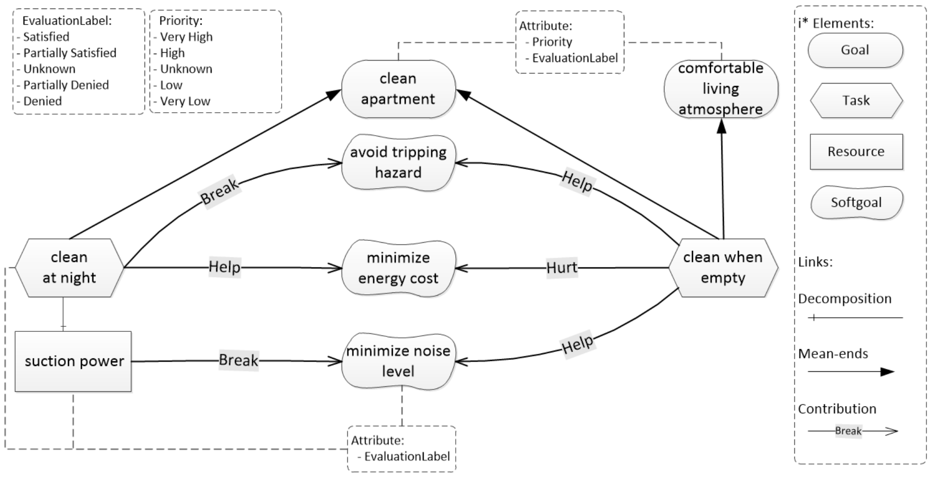


Fig. 5. i* model of the vacuum cleaner

Figure 5 shows an i* model of the previously mentioned vacuum cleaner case study. Listing 1.2 is a textual representation of this i* model using DO$_{\mathrm{RE}}$F. The task *Clean at night* is related to the requirement *Clean at night*, see line 8.

[1] `https://se.cs.toronto.edu/trac/ome/`
[2] `http://www.eclipse.org/modeling/emf/`

Listing 1.2. i* model of the vacuum cleaner in DOREF

```
Actor("vacuum cleaner")
cd("./-")
Goal("Clean Apartment",
  {'Description': "The apartment should be cleaned by the robot.",
   'Priority': 1,})
Goal("Comfortable living atmosphere",
  {'Priority': 2})
Task("Clean at night", {"RelatedTo": "/*/Customer Requirements/*/
     Clean at night"})

Task("Clean when empty", {"RelatedTo": "/*/Customer Requirements
     /*/Clean when empty"})
SoftGoal("Avoid tripping hazard")
SoftGoal("Minimize energy cost")
SoftGoal("Minimize noise level")

MeanEndLink(node("/*/vacuum cleaner/*/Clean at night"),
    node("/*/vacuum cleaner/*/Clean Apartment"))
MeanEndLink(node("/*/vacuum cleaner/*/Clean when empty"),
    node("/*/vacuum cleaner/*/Clean Apartment"))
MeanEndLink(node("/*/vacuum cleaner/*/Clean when empty"),
    node("/*/vacuum cleaner/*/Comfortable living atmosphere"))

ContributionLink(node("/*/vacuum cleaner/*/Clean at night"),
    node("/*/vacuum cleaner/*/Avoid tripping hazard"), "HELP")
ContributionLink(node("/*/vacuum cleaner/*/Clean at night"),
    node("/*/vacuum cleaner/*/Minimize energy cost"), "HURT")
ContributionLink(node("/*/vacuum cleaner/*/Clean at night"),
    node("/*/vacuum cleaner/*/Minimize noise level"), "HELP")
...
```

For the following subsection discuss how an assertion is connected to a *Softgoal* or a *ContributionLink*.

3.3 Assertions

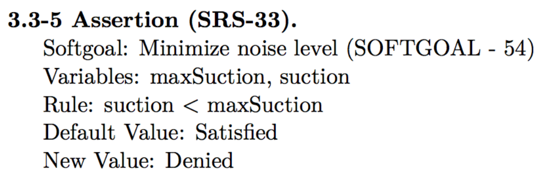
A softgoal is an element whose satisfaction is depending on components or events inside the system and its environment. A contribution link has a type (for example make, help, etc.) which is also depending on the situation of the system and its environment. To determine the satisfaction or the type of a link, the requirements engineer states assertions about the required situation of the system and its environment.

An assertion can be seen as a kind of a test case to prove if a requirement can be fulfilled at runtime. Thus, a requirements engineer derives an assertion from a requirement.

Listing 1.3. Assertion with DOReF

```
1 SGAssertion("/*/Minimize noise level",
2            ["maxSuction", "suction"],
3            "suction < maxSuction",
4            "Satisfied", "Denied")
```

Because a requirements engineer needs to describe assertions, we added a textual description for the assertions to DOReF, see Listing 1.3. Figure 6 shows an assertion which is related to a softgoal with the ID *SOFTGOAL-54* and two variables *suction* and *maxSuction* (Listing 1.3 line 2). The parameter *maxSuction* can be also found in Figure 4.



3.3-5 Assertion (SRS-33).
Softgoal: Minimize noise level (SOFTGOAL - 54)
Variables: maxSuction, suction
Rule: suction < maxSuction
Default Value: Satisfied
New Value: Denied

Fig. 6. Textual assertion

The structure and relationship between an intention and the assertions are shown in Figure 3. We define two kinds of assertions, one to handle the softgoal satisfaction and one for the contribution link type.

An example *SoftgoalAssertion* is shown in Listing 1.3 and Figure 6. The *ID SRS-33* is generated by the framework. The *rule* is a Boolean condition defined over the related *Variables*, which is evaluated by the rule engine inside the requirements monitor. The Boolean condition employs Java Boolean expressions. That is, it evaluates to *false* or *true* and uses Boolean operators, like ==, &, !, etc. If an assertion is broken, the related softgoal satisfaction changes from the *Default Value* to the *New Value*. In example above, it would change from *Satisfied* to *Denied*.

3.4 Consistency Between the Models

As shown in Figures 4 and 6 and Listing 1.1, 1.2 and 1.3 there is a direct connection between the requirements model and the i* model by means of assertions. This allows to define consistency checks on requirements models. We have identified four consistency rules:

- Every task must be related to a requirement.
- Every requirement, which is related to a task, must have an *optional* property.

– For every requirement with variables inside the textual description, at least one assertion must be defined.
– For every root softgoal (i.e., the softgoal only has incoming contribution links) or for one of the incoming contribution links an assertion must be defined.

Rule 1 and 2 are checking the relationship between tasks and requirements. Every goal can be achieved over different alternatives. These alternatives are described by tasks, which reflect possibly conflicting functional requirements. So these requirements can only be fulfilled if the task is satisfied. This requirements need the *optional* property to show if they are fulfilled. Rule 3 ensures that the rule engine checks all defined variables. As mentioned in Section 3.3 the satisfaction of a softgoal depends on the environment and state of the system. Rule 4 ensures that every softgoal can be defined.

4 Generating Requirements Monitor

This section describes the generation of the requirements monitor using the presented metamodel (see Figure 3). Our first step was to identify components, which are application-specific, and thus need to be generated. The impact analyzer is the only component which does not require generation. The analyzer gets its inputs from the goal model and every information for changing and computing the model from the rule engine and the integrated assertions. The evaluation process does not depend on the application. The other components are application-specific and will be described in the remainder of this section.

4.1 Runtime Goal Model

The goal model is the runtime representation of the requirements and describes the alternative realization strategies of the system. This component must be generated. As described in Section 3.2, we use an i* model for representation of the requirements. This model is developed by a requirements engineer using $\text{DO}_{\text{RE}}\text{F}$. Inside the requirements monitor the Eclipse Modeling Framework (EMF) is used to access the i* model. The underlying source code for accessing the model is generated by EMF. The information for creating an i* model can be exported from $\text{DO}_{\text{RE}}\text{F}$ and imported by the requirements monitor.

4.2 Rules

The monitor component owns a rule engine, which checks the assertions. To implement the rule engine we use Roolie[3], a framework that supports defining, changing and checking rules at runtime. The required information for generating the application-specific parts of the rule engine is inside our metamodel.

[3] `http://roolie.sourceforge.net/`

The *Rule* inside the assertion is used for the Boolean condition of the rule inside the rule engine, see Listing 1.4. The rules use rule arguments *RuleArgs* (see line 1) to check the condition. A rule also have the following information of a connected *i*-Element*: *ID*, *satisfaction*, or *type of contribution link*. If a rule breaks, the required information about the *i*-Element* is sent to the impact analyzer, which computes all satisfactions of the *i*-Elements* inside the model.

Listing 1.4. Implementation of a *Rule* with Roolie

```
1 int maxSuction = ruleArgs.getMaxSuction();
2 boolean passes = currentSuction < maxSuction;
```

The assertions are related to variables. The variables are stored as rule arguments (*RuleArgs*) inside the rule engine. If a variable in the system changes, the monitor set the related *RuleArg* with a new value. The rule engine checks every connected assertion. The variable *ID* is used for the *RuleArg*, see Listing 1.5. The default value is used as initial value of the *RuleArg*.

Listing 1.5. Implementation of *RuleArgs* with Roolie

```
public enum ArgField {
         MaxSuction, Suction;
    };
public void setMaxSuction(int MaxSuction) {
        setInt(ArgField.MaxSuction, MaxSuction);
    }
public int getMaxSuction() {
        return getInt(ArgField.MaxSuction);
    }
public void setSuction(int Suction) {
        setInt(ArgField.Suction, Suction);
    }
public int getSuction() {
        return getInt(ArgField.Suction);
    }
```

4.3 Parameter Observer

For checking the assertions the monitor has to communicate with the application and set the current value of the variables/ *RuleArgs* inside the rule engine. The monitor can register at an interface of the application to receive the parameter information. If a connection is established the application sends messages with new Parameter values (`Suction::40`).

The monitor has to receive and analyze this messages and set the *RuleArgs*. Listing 1.6 shows the result of the generation process.

Listing 1.6. Monitor

```
String[] splitMsg =
   ((String) msg).split("::");
if(msg[0].equals("suction"))
{
   ruleEngine.getRuleArgs().setSuction(Integer.parseInt(msg[1]));
   ruleEngine.testSuction();
}
```

The message is split and the parameter *ID* is used to set the right *RuleArg*.

5 Related Work

Requirements engineering for self-adaptive systems has gained a lot of interest as a recently published survey has shown [17]. The authors found that most approaches use goal-oriented modeling techniques.

The environment is usually attached to a goal model using *domain assumptions* [11, 10, 14, 2, 5], *claims* [15], or *assertions* [4]. Assertions are monitored by a monitor system such as Flea [4], ReqMon [10, 12], or SalMon [10, 9]. To our knowledge none of the before-mentioned approaches explicitly address the question of how to *generate* the part of a requirements monitor, which are specific to the adaptive system under consideration.

Because we work in the area of embedded systems we have to handle resource constraints as mentioned in Chapter 2. The approaches in [10],[11], [12], and [15] are focused on service-oriented systems instead of embedded systems.

6 Conclusion

This paper proposes a way of augmenting a requirements specification with additional information so that a requirements monitor can be generated. A concept for such a generator is outlined. The result is geared towards our approach of representing a self-adaptive system described in [6], but can be transferred also to other approaches, which make use of assertions, domain assumptions, or claims.

This is an important step towards our goal of providing feedback to requirements engineers and users about changes made to the requirements during runtime. The key is to be able to build adaptive systems without significant overhead during development time. Especially embedded systems are interesting, as a human operator is often not available to give a confirmation to a system's decision. Thus the system must decide autonomously. Moreover, model-driven development is already widely used in practice.

We suggested in this paper a concept for making adaptivity explicit in development time artifacts. Nevertheless, discrete adaptivity can also be *programmed* directly into an application. The benefits of an *explicit* documentation of adaptivity are similar to the benefits of an explicit documentation of variability in case of software product lines: understanding and communication are improved.

Our future work addresses the communication of the changes to the users and the formalization and monitoring of further aspects of requirements. Regarding embedded real-time systems in particular, execution times, violations of deadlines, etc. are of interest. This information can also be collected at runtime and feed back to engineers.

References

1. N. Bencomo and A. Belaggoun. Supporting decision-making for self-adaptive systems: From goal models to dynamic decision networks. In J. Doerr and A.L. Opdahl, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7830 of *Lecture Notes in Computer Science*, pages 221–236. Springer Berlin Heidelberg, 2013.
2. N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier. Requirements reflection: requirements as runtime entities. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 199–202, 2010.
3. Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.
4. M.S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Software Specification and Design, 1998. Proceedings. Ninth International Workshop on*, pages 50–59, 1998.
5. S. Fickas and M.S. Feather. Requirements monitoring in dynamic environments. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 140–147, 1995.
6. Erik Kamsties, Fabian Kneer, Markus Voelter, Burkhard Igel, and Bernd Kolb. Feedback-aware requirements documents for smart devices. In Camille Salinesi and Inge Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, volume 8396 of *Lecture Notes in Computer Science*, pages 119–134. Springer International Publishing, 2014.
7. Alessia Knauss. Dealing with uncertainty contextual requirements, 2014. Poster presented at the 20th International Working Conference on Requirements Engineering: Foundation for Software Quality, April 7–10, Essen, Germany.
8. Fabian Kneer and Erik Kamsties. Requirements engineering education at academia: A model-based approach. *Softwaretechnik-Trends*, 2015.
9. M. Oriol, J. Marco, X. Franch, and D. Ameller. Monitoring adaptable soa-systems using salmon. In *Workshop on Service Monitoring, Adaptation and Beyond (Mona+)*, pages 19–28, June 2008.
10. M. Oriol, N.A. Qureshi, X. Franch, A. Perini, and J. Marco. Requirements monitoring for adaptive service-based applications. In B. Regnell and D. Damian, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7195 of *Lecture Notes in Computer Science*, pages 280–287. Springer Berlin Heidelberg, 2012.

11. N.A. Qureshi, I.J. Jureta, and A. Perini. Towards a requirements modeling language for self-adaptive systems. In B. Regnell and D. Damian, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7195 of *Lecture Notes in Computer Science*, pages 263–279. Springer Berlin Heidelberg, 2012.
12. W.N. Robinson. Implementing rule-based monitors within a framework for continuous requirements monitoring. In *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*, pages 188a–188a, 2005.
13. Angela Rook, Alessia Knauss, Daniela Damian, Hausi A. Mueller, and Alex Thomo. Integrating data mining into feedback loops for predictive context adaptation. Technical Report DCS-349-IR, University of Victoria, Victoria BC, August 2013.
14. P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 95–103, 2010.
15. K. Welsh, P. Sawyer, and N. Bencomo. Towards requirements aware systems: Run-time resolution of design-time assumptions. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 560–563, 2011.
16. Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *In 17th IEEE International Requirements Engineering Conference RE 2009*, 2009.
17. Zhuoqun Yang, Zhi Li, Zhi Jin, and Yunchuan Chen. A systematic literature review of requirements modeling and analysis for self-adaptive systems. In Camille Salinesi and Inge van de Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, volume 8396 of *Lecture Notes in Computer Science*, pages 55–71. Springer International Publishing, 2014.
18. Eric Siu-Kwong Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, Toronto, Ont., Canada, Canada, 1996. UMI Order No. GAXNN-02887 (Canadian dissertation).