Feedback-aware Requirements Documents for Smart Devices

Burkhard Igel[2], Erik Kamsties[1], Fabian Kneer[1], Bernd Kolb[2], and Markus Voelter[3]

[1] Dortmund University of Applied Sciences and Arts,
Emil-Figge-Str. 42, 44227 Dortmund, Germany
{erik.kamsties,fabian.kneer}@fh-Dortmund.de
http://www.fh-dortmund.de
[2] itemis AG, Germany,
{igel,kolb}@itemis.de
[3] independent/ itemis
voelter@acm.org

Abstract. [Context/ Motivation] A smart device is a software-intensive system, which operates autonomously and interacts to some degree with other systems over wireless connections (e.g., a iRobot Roomba vacuum cleaner). These systems are often faced with uncertainty in the environment. Runtime representations of requirements have recently gained more interested to deal with this challenge and the term *requirements at runtime* was coined. These representations allow to reason about the requirements at runtime and to adapt the configuration of a system according to changes in the environment. **[Questions/Problems]** The research question is how the results of online monitoring of requirements and the system's decisions about changes in the configuration are communicated to the requirements engineer to better understand the environment. There is a gap between the written requirements document and the dynamic requirements model inside the system. This problem is exacerbated by the fact that a requirements document are mostly informal while the dynamic requirements model is formal. **[Principal ideas/results]** This paper introduces an approach to bridge the gap between development time and runtime representations in order to keep them consistent and to facilitate a better understanding. We suggest to weave the feedback from the runtime system into requirements documents using a domain-specific language which keeps the informal nature of requirements. An annotated requirements document helps get a better understanding of the system's actual behavior in a given environment. The approach is implemented using mbeddr, a novel domain-specific language for developing embedded systems, and illustrated in a case study.

Keywords: Smart Device, Embedded System, Domain-specific Language, mbeddr, Requirements at Runtime, Self-Adaptivity

1 Introduction

Runtime representations of requirements have received increased interest in the last years. Runtime representations are the basis for reflection on requirements, that is to understand, explain, and modify requirements at runtime, in order to deal with continuously changing environmental needs [1] – a significant challenge for today's software-intensive systems.

We propose in this paper an approach for relating *runtime* representations of requirements with *development time* representations. The goal is to gain insights into how requirements evolve over time and how the system is actually used from the perspective of a requirements engineer. The focus is on resource-constrained embedded systems. For the development time representation, we use mbeddr[4], a modular domain-specific extension to the programming language C. mbeddr is also capable of dealing with requirements, it stores the artifacts along with code and maintains traceability. That is, mbeddr provides an integrated view on requirements and implementation in C, which are maintained in the same formalism and same tool (see see Fig. 1). Requirements documents can be generated from mbeddr in the usual formats (e.g., HTML, PDF).

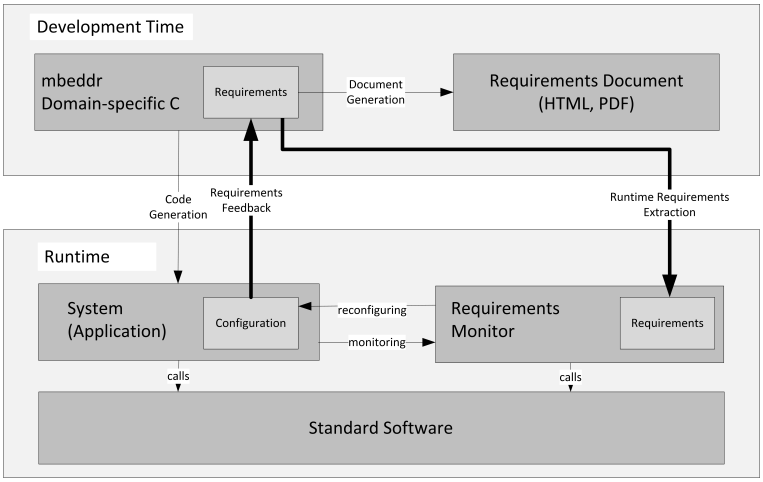

Fig. 1. Bridging the gap between development time and runtime artifacts

For the runtime representation, the system (application) itself is generated from the implementation in mbeddr (see Fig. 1). The system can be reconfigured using a configuration, in its simplest form the configuration is a set of parameters. In order to be able to run the sytem on an embedded hardware platform, some standard software is necessary which abstracts from hardware details (in an automotive project for instance, this is typically the AUTOSAR basic software). A requirements monitor is aware of the

[4] `http://mbeddr.wordpress.com/`

requirements and implements an approach for runtime adaptivity, a goal-oriented approach in our case. The runtime requirements (i.e., the goal model) are extracted from the development time requirements expressed in mbeddr. If environmental changes results in an update of the configuration, then this feedback is weaved into the mbeddr requirements. These changes are highlighted during re-generation also in the PDF/HTML requirements document.

The contribution of this paper is a formal, tool supported link between the development time and runtime representations (bold arrows in Fig. 1). The benefit is two-fold. First, a requirements engineer better understands the adaptations of systems in the field. Second, a user can be better informed about the actual system behavior.

The remainder of this paper is organized as follows. Section 2 outlines the background of our research. Section 3 introduces mbeddr and explains how it is used as an development time representation for requirements (and of course for domain-specific code generation). Section 4 discusses the runtime representation of requirements and the missing link between the two representations. Our approach is illustrated using a detailed case study of a vacuum cleaner. Section 5 reviews the related work. Section 6 concludes with a summary and an overview on our future work.

2 Background

2.1 Embedded Systems

The focus of our work is on smart systems, which are a subclass of embedded systems. Embedded systems impose heavy constraints on software. First of all, as these systems are mass-produced, the capabilities of the hardware are optimized to the purpose of the respective system. That is, the horsepower of the CPU and the memory size are limited. We often see 8-bit microcontrollers running at 16 MHz and offering 256 KB flash memory (e.g., in wireless sensor networks). Embedded Linux systems come with a 32-bit CPU which is clocked at 400 MHz and higher and ca. 512 MB of memory. Requirements about energy consumption prohibit more powerful hardware, since many embedded systems run on batteries. The programming language C is prevalent.

Smart systems in particular collaborate with each other, typically over wireless connections. For example, a warning about an iced bridge may be passed through a sequence of vehicles. The systems are only loosely coupled and a smart system must adapt its behavior to the current situation (e.g., when there are not enough networked cars available).

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

2.2 Self-adaptive Systems

A self-adaptive system has the ability to dynamically and autonomously reconfigure its behavior to respond to changing environmental conditions [1]. We consider a self-adaptive system as consisting of four parts: a requirements model, the system, a monitor,

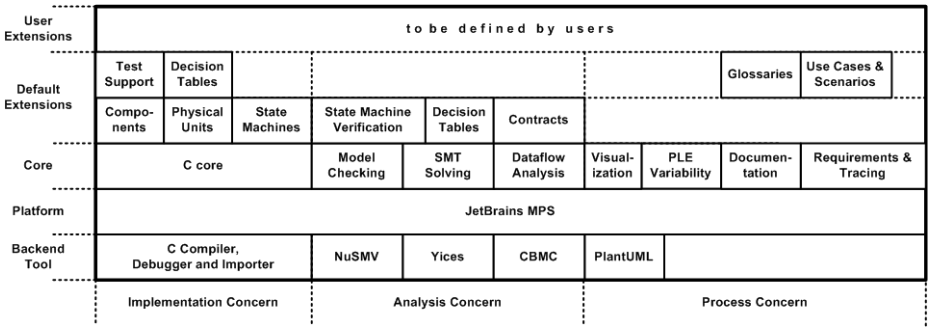

Fig. 2. mbeddr rests on the MPS language workbench. Above it, the first language layer contains an extensible version of the C programming language plus special support for logging/error reporting and build system integration. On top of that, mbeddr comes with a set of C extensions (components, state machines, units) plus cross-cutting support for requirements, traceability, documentation, visualization and variability.

and an impact analyzer. See Fig. 4 for the relation between these parts. Note that (1) monitor, (2) requirements model, and (3) impact analyzer together form the requirements monitor shown Fig. 1 .

The requirements model is a machine-readable representation of the system's requirements, which is the basis for requirements reflection at runtime. Often, a goal-oriented model is used for this purpose (see Related Work at Sec. 5).

The system implements the development time requirements. It is attached to the runtime requirements model using assertions. In the case of a goal-oriented requirements model, an assertion satisfies a soft goal. Assertions refer to parameters, which are monitored inside the system.

If an assertion fails, a requirement may be broken and the impact analyzer is invoked. The question is, which parts are affected and whether a change in the model is really necessary. Sometimes a change is postponed in order to keep an important goal satisfied. If a change is necessary, a new configuration is computed and the system switches eventually to that new configuration.

The next section describes our approach to the development time representation of requirements.

3 Development Time Representation using mbeddr

We selected mbeddr for the development time representation of requirements. mbeddr is an open source project supporting embedded software development based on incremental, modular domain-specific extension of C. It also supports languages, that address other aspects of software engineering such as requirements or documentation. Fig. 2 shows an overview, details are in [16] and [17].

3.1 mbeddr Overview

mbeddr builds on the JetBrains MPS language workbench[5], a tool that supports the definition, composition and use of general purpose or domain-specific languages. MPS uses a projectional editor, which means that, although a syntax may look textual, it is *not* represented as a sequence of characters which are transformed into an abstract syntax tree (AST) by a parser. Instead, a user's editing actions lead *directly* to changes in the AST. Projection rules render a concrete syntax from the AST. Consequently, MPS supports non-textual notations such as tables, and it also supports unconstrained language composition and extension – no parser ambiguities can ever result from combining languages (see [15] for details).

The next layer in mbeddr is an extensible implementation of the C programming language (version C99 - ISO/IEC 9899:1999) in MPS. On top of that, mbeddr ships with a library of reusable extensions relevant to embedded software. As a user writes a program, he can import language extensions from the library into his program. Major extensions include test cases, interfaces and components, state machines, decision tables and data types with physical units. For many of these extensions, mbeddr provides an integration with static verification tools (model checking state machines, verifying interface contracts or checking decision tables for consistency and completeness; see also [10]).

We selected mbeddr to represent requirements at development time because it is geared to the embedded domain, it allows for code generation for the abstractions in the domain (e.g., state machines), and because it supports three important aspects of software engineering: requirements engineering and tracing, product line variability and documentation. We discuss the requirements aspect in the following subsection.

3.2 Requirements in mbeddr

mbeddr exploits language engineering to provide a powerful tool for embedded software engineering: the vast majority of problems is solved by providing domain-specific languages that express different aspects of the overall system. This is also true for requirements. Requirements are captured using an extensible language specific to the *requirements domain*. Like any other requirements management tool, the mbeddr requirements language primarily describes requirements with a short title, a unique ID and a prose description (see Fig. 3). However, it also supports a number of unique features, which we utilize in our approach:

- Extensibility: The mbeddr requirements language can be extended in any direction. That is, we are able to add e.g., goal-oriented modeling (see Sec. 2.2).
- The *right* degree of formality: Most industrial embedded systems are specified using a mixture of formal and informal/semi-formal representations of requirements. This observation imposes a challenge to our goal of requirements feedback: how to feed (formal[6] results from executing a system into the requirements if require-

[5] `http://jetbrains.com/mps/`

[6] We use the term *formal* here in the sense of *machine-readable* as it is usually done in model-driven engineering.

ments are informal? mbeddr supports *partial formalization* by formal concepts that are introduced directly into informal requirements.
– Traceability and consistency: As requirements and code are maintained by mbeddr, traceability is supported between requirements and to other artifacts. If informal requirements are partially formalized (e.g., using parameters), mbeddr maintains consistency between requirements and code such that a change of a parameter in the code changes the value in the requirement and vice versa. We extend traceability in our approach towards runtime representation of requirements.

3.3 Extension of mbeddr to deal with runtime requirements

We extended mbeddr to cover *parameters*, *optional requirements*, and *i* goal models* to establish the missing link between development time and runtime requirements. These extensions are discussed below. As a running example throughout the paper, we use the vacuum cleaner case study, which was originally introduced in [2] and [1].

The use of parameters in functional requirements is a common technique for embedded systems e.g., to enforce adaptability regarding a technical environment, or particular customers. Parameters were added to the original mbeddr requirements language in a way so they can be embedded in the requirements prose description ([14] explains how to do this). Like variables in programs, parameters have a name, a type and an initial value. Fig. 3 shows an example, the Requirement RE3 is completely informal, except for a formal concept *parameter* called `maxSuction` and annotated as `@`param.

Some requirements for an embedded system are considered optional, that is the respective function can be switched on/off at runtime. We assume that these requirements are initially *enabled* and become *disabled* at runtime if a conflict arises between requirements due to change in the environment. RE1 and RE2 in Fig. 3 are optional.

```
1 Clean at night
  RE1 /functional: option
  [ The robot shall clean the apartment at night. ]

2 Clean when empty
  RE2 /functional: option
  [ The robot shall clean the apartment when nobody is inside. ]

3 Minimize noise level
  RE3 /functional: tags
  [ The robot shall work with a reduced suction power (lower than 50%), so
    $param(maxSuction: int32 = 50). ]
```

Fig. 3. Example requirements with parameters and an *option* attribute

We use an i* goal model to resolve possible conflicts at runtime. For this purpose, we developed a new mbeddr language module to describe a goal model and to link it to

the requirements. The goal model is basically a textual description of the classic i* goal model shown in Fig. 5 therefore we do not provide an example.

Finally, the exchange of information between mbeddr at developement time and an embedded system at runtime needs consideration. This is currently realized by exchanging XML files between the host and target system. Listing 1 shows part of the information sent from mbeddr to the embedded system derived from the requirements in Fig. 3 (again, the goal model is omitted).

Listing 1. XML representation with optional requirements and a parameter

```
<parameters>
  <requirement option="true" name="RE1" />
  <requirement option="true" name="RE2" />
  <requirement name="RE3">
    <param name="maxSuction" type="int32">50</param>
  </requirement>
</parameters>
```

4 Runtime Representation of Requirements

This section describes our runtime representation of requirements. The implementation follows the overall structure that also underlies other approaches and consists of four parts, namely a requirements model, a system, a monitor, and an impact analyzer (as discussed in Sec. 2.2 and shown in Fig. 4). The following discussion starts with the requirements monitor, i.e., the requirements model within, and ends with the system.

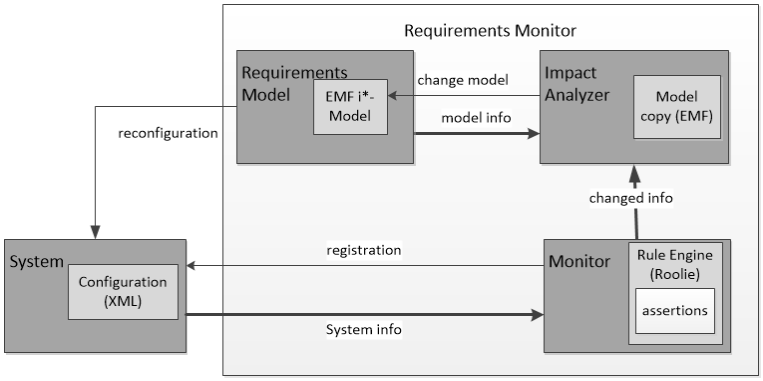

Fig. 4. Concept for runtime representation and monitoring of requirements

4.1 Requirements Model

We decided to use the i* model. A simple implementation of i* is provided by the openOME[7] tool by the University of Toronto. We use its meta model which is defined with the Eclipse Modeling Framework (EMF) [8].

We extended the openOME meta model by a new attribute `Priority` of a goal. The attribute is defined as an enumeration with the values `VeryLow, Low, High, VeryHigh, Unknown`. This extension is needed for the impact analyzer, who must find the goal with the highest priority to decide how to change the requirements model to maintain the satisfaction of this goal. Fig. 5 shows an i* model for the vacuum cleaner. The elements task, goal, soft goal and resource have an attribute `EvaluationLabel` which stands for the satisfaction of that element. Note that in Fig. 5 the goals have a priority.

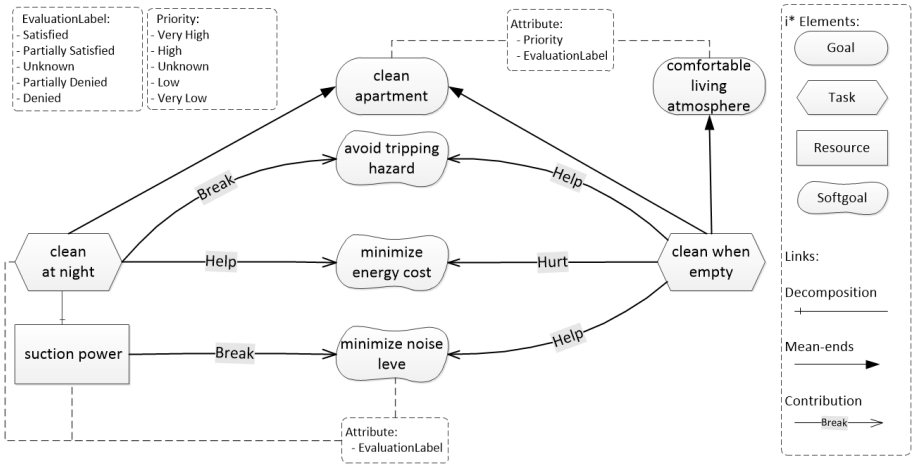

Fig. 5. i* model of the vacuum cleaner

The extended i* EMF model is used as runtime representation of requirements. This model is created in mbeddr at development time, it is imported by the requirements monitor, and can be changed and displayed at runtime. The underlying source code for accessing the runtime requirements is generated by EMF.

4.2 Monitor

As we explained in Sec. 2.2 the requirements model is linked to the system by assertions. An assertion is modeled as a rule in a rule engine. An assertion must be fulfilled by the system to satisfy a soft goal or set a type of a contribution link, see Fig. 5.

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

We defined two kinds of assertions. The first is linked to a soft goal. If the assertion breaks, the satisfaction of the linked soft goal is modified. The second kind of assertion is linked to a contribution link. If such an assertion breaks, the type of the contribution link is modified. Both kinds of assertion refer to parameters.

An assertion is a boolean condition on parameters which is evaluated by the rule engine. See Listing 2, here the parameter is `time`. The rule fails if the current time is not between *timeMax* and *timeMin*. Note that the thresholds, in the example `timeMax`, `timeMin`, can be changed at runtime. Thus, an assertion can be modified to delay changes to the requirements model. The following assertions are defined for the vacuum cleaner i* model:

1. No tripping hazard
2. Lowest energy cost between 22 and 8 o'clock
3. Noise level too high when suction power over 50%

Assertion 1 is assigned the contribution link between *clean at night* and *avoid tripping hazard*. If the assertion breaks, the type of the contribution link changes from *unknown* to *break*. Assertion 2 is important to satisfy the soft goal *minimize energy cost*. It means this soft goal can only be satisfied between 22 and 8 o'clock. Finally, Assertion 3 can be modified by changing the threshold of 50%. Consequently, the change to the requirements model can be delayed to assure the satisfaction of a goal with a high priority.

To implement the rule engine we use Roolie [9]. This framework supports defining, changing and checking rules at runtime. Listing 2 is related to Assertion 2.

Listing 2. Implementation of a rule with Roolie

```
1 boolean passes = time > timeMax || time < timeMin;
```

An assertion is evaluated on the values of parameters. For this purpose, these parameters are observed by the monitor. It is implemented with the observer pattern, this means, the monitor registers itself as an observer to the system and gets a notification whenever one of the parameters change.

4.3 Impact Analyzer

The impact analyzer is called when an assertion breaks. In this case the impact analyzer receives an information about the concerned element (soft goal or contribution link) and computes a new satisfaction of a soft goal or a new type for a contribution link. This change is applied only to a copy of the requirements model to avoid premature changes.

The impact analyzer performs a reverse evaluation of the requirements model based on the model evaluation process for i* [10]. In short, this process starts at the soft goal connected to the broken assertion, traverses all elements of the requirements model, and ends when all satisfactions are recomputed. Fig. 6 shows an example calculation for the vacuum cleaner. Assume Assertion 1 is broken, which is assigned to the contribution

[9] `http://roolie.sourceforge.net/`
[10] `http://istar.rwth-aachen.de/tiki-view_articles.php`

link between *clean at night* and *avoid tripping hazard*. The type of the contribution link changes to *break* (bold link). From the target of this contribution link (*avoid tripping hazard*) we start to compute the satisfactions of the other model elements.

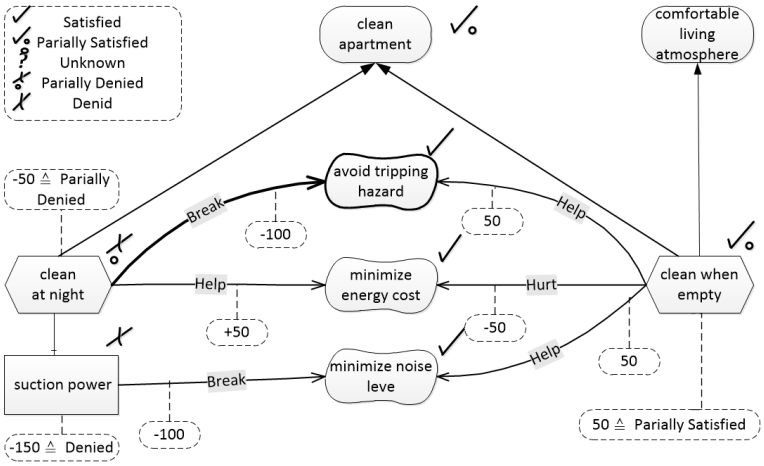

Fig. 6. i* model with calculation values

The satisfaction for a goal is calculated over the mean-end link. This is an *or*-relationship between a goal and one or more tasks. The goal satisfaction is taken from the highest satisfaction of one of the linked tasks. In Fig. 6 the goal (*clean apartment*) gets the satisfaction from the task (*clean when empty*).

A task can have decomposition links to a task, goal, soft goal, or resource. This is an *and*-relationship. Every link element must be satisfied to satisfy the task. Because of the reverse evaluation, all elements get the satisfaction of the task. In Fig. 6, the resource (*suction power*) gets the satisfaction from the task (*clean at night*). The satisfaction of a task or a soft goal that does not get its satisfaction from the environment is computed as sum over of the values of the contribution links, see Fig. 6.

Because the analyzer use a reverse evaluation, the value of the contribution link must be determined from the desired satisfaction of the target and the type of the contribution. The result is the value, the source must have in this relationship to get the desired satisfaction of the target. Fig. 7 shows how to combine a soft goal satisfaction and a contribution link type. The values were derived in a pragmatic fashion. In the example in Fig. 6, the satisfaction *partiallyDenied* is computed for the task *clean at night*. It is the result of the sum of the assigned contribution links. The results are 50 (combination of *help* and *satisfied*) and -100 (combination of *break* and *satisfied*, see Fig. 7)

To simplify the analysis, the analyzer only looks at the number of satisfied goals before and after the change is applied to the copy. After the change, the analyzer checks if the goals with higher priority are still satisfied (this is the reason for our extension of the openOME meta model). If the analyzer gets different numbers or a main goal is no longer satisfied, the analyzer tries to delay the change by modifying the threshold of

	Denied	PartiallyDenied	Unknown	Partially Satisfied	Satiesfied
Break	100	50	0	-50	-100
Hurt	50	50	0	-50	-50
Some-	50	50	0	-50	-50
Unknwon	0	0	0	0	0
Some+	-50	-50	0	50	50
Help	-50	-50	0	50	50
Make	-100	-50	0	50	100

Fig. 7. Combination of soft goal satisfaction and contribution link type

an assertion. If an assertion cannot be modified (because there are no thresholds as in Assertion 1), the analyzer takes the copy as the new original.

Finally, the analyzer generates a new configuration for the system. A configuration is a new assignment of values to parameters and options of the runtime requirements shown in Listing 1. For a requirement assigned to a task with a positive satisfaction, the option is set to *true*. For a requirement assigned to a task with a negative satisfaction, the option is set to *false*. In the example, the *option* attribute of Requirement RE1 is set to *false*, i.e., the function *clean at night* is disabled in the new configuration. RE2 remains *true*.

4.4 System

The system, i.e., the embedded software, reads sensor data and writes to actuators. Two additional interfaces are required. First, the system provides an interface to allow for monitoring parameters. The monitor can register itself with this interface and gets a notification if a parameter changes its value.

Second, the system provides an interface to read a new configuration after a modification of the requirements model. At the beginning the system reads a configuration generated by the requirements monitor. This configuration contains all information on how a goal is achieved and which functions are active under the current environmental situation.

4.5 Requirements Feedback

The information about changes is collected by the requirements monitor and is sent to mbeddr at some point in time (note that we are doing a post-mortem analysis of the systems using mbeddr). This is the last step in our feedback cycle shown in Fig. 1. The information is again represented using XML and contains the concrete parameter values, activated requirements, and the changes to the goal model.

It is important to note that there is a 1-to-many relationship between development time and runtime requirements, i.e., the development time requirements receive feedback from many runtime instances. Thus, a system identifier is added to the feedback, see Listing 3.

Listing 3. XML file with system identification

```
1 <parameters systemID=”42”>
```

Finally, the feedback file is imported by mbeddr. The information about the parameters are stored as child nodes of the parameter in the requirements description, separately for each system ID. In mbeddr, the requirements engineer can use a filter to focus on a specific system and show the values of the parameters directly in the requirements. It is also possible to inspect all parameter values of all systems that provided feedback.

Fig. 8 shows the change of the requirements from Fig. 3. Now the requirements RE1 and RE2 have a value for the option tag. For the system with the ID 42, requirement RE1 is disabled and requirement R2 is enabled.

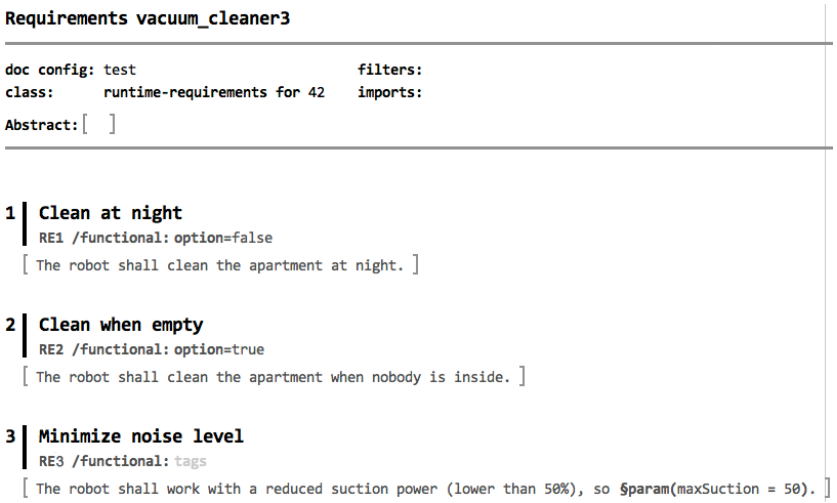

Fig. 8. Requirements with Feedback

mbeddr provides a function for generating requirements documents (e.g., HTML or PDF) out of a mbeddr requirements. This function is modified in order to highlight the changes happened at runtime so that a requirements engineer is able to better understand how systems evolve.

4.6 Scenario

The following scenario illustrates how the system, monitor, impact analyzer, and requirements model interact in a given situation.

The vacuum cleaner cleans different surfaces in an apartment. Depending on the environment and prioritization of the goals, one of the realization strategies is selected. Assume, the robot cleans at night. A person walk throws the apartment. Now, because of a power failure the lights switch off. The following steps are carried out:

1. The monitor notices a change in the apartment and triggers the rule engine.
2. The rule engine checks all assertions and notices that Assertion 1 *no tripping hazard* is broken.
3. The rule engine informs the impact analyzer that the contribution link must be set to *break*.
4. The changes are applied to a copy of the model and the satisfaction of each element is calculated.
5. Both goals are satisfied by the task *clean when empty* (see Fig. 6). This means the copy became the original model.
   (a) The information about the change is stored in a file.
   (b) mbeddr imports this file and weaves the information into the requirements (see Listing 8).
   (c) This definition is used to generated a requirements document in PDF or HTML with highlighted deltas.
6. Out of the new requirements model, the analyzer generated a new configuration for the system *clean at night* is disabled, *clean when empty* is enabled.
7. The system reads the configuration and switches to the new strategy.

5 Related Work

Many approaches have been developed to deal with adaptivity, including neural networks, rule engines, and dynamic decision networks. In the domain of embedded systems, adaptive fuzzy controllers are a typical solution.

Fuzzy control is based on fuzzy logic was first proposed by Zadeh [20]. The application of fuzzy logic to control systems was first accomplished by Mandami [6]. Fuzzy control can be considered as a feedback control system driven by characteristic curves. These characteristic curves are described by a rule based system and so called fuzzy sets. If there are any changes necessary by an adaptive feedback control system, those changes can be directly viewed by changed rules and fuzzy sets.

Fuzzy reasoning results are influenced by the rules and structure of the fuzzy sets. Both rules and fuzzy sets can be considered as requirements for the resulting system behavior. The main issue for fuzzy control in terms of requirements engineering is that fuzzy control is not trying to model the system, but trying to model the behavior of the operator, who is controlling the system. Together with the possible backward reasoning process for the adaption of fuzzy sets, fuzzy control shows possibilities for feedback awareness in requirements.

Fuzzy control is one solution adaptability in embedded systems. Unfortunately, it is restricted to *control systems*, i.e, continuous systems. Discrete system as the vacuum cleaner are not well-addressed.

The following discussion of related work in **adaptivity in RE** is based on the reference model shown in Fig. 4. For the *requirements model* most authors [9], [8], [12], [1], [19], [4] use a goal-orientated model such as KAOS [13], i* [18] or an extension such as Tropos [3] or adaptiveRML [9]. The system is attached to the requirements model using *domain assumptions* [9], [8], [12], [1], [5], *claims* [19], or *assertions* [4]. Assertions are monitored by a monitor systems such as Flea [4], ReqMon [8], [11], or SalMon [8], [7].

Qureshi et al. [9] tries to handle premature changes of the model with assertions that are modified at runtime. Oriol et al. [8] are working on notifications to involve users in decision processes.

Further research aims at improving the decision process by using dynamic decision networks (DDNs). Bencomo et al. [2] transfer an i* goal model into a DDNs to get a more dynamic representation of the requirements. The DDNs chose the tasks with dynamic evolving notes instead of the static contribution links.

The general architecture shown in Fig. 4 is a common ground, but the beforementioned approaches do not explicitly address embedded systems.

6 Discussion

This paper identifies a problem in a RE, which arises when a system is aware of its requirements: how are possible changes to the requirements during runtime communicated to requirements engineers and users?

The goal of our work is from the perspective of a requirements engineer to gain insights into how requirements evolve over time and how the system is actually used. One challenge lies in the representation of requirements at development time in a way they can be easily extracted and used at runtime. The feedback from the system to the development time requirements imposes another challenge, as requirements are represented as a mixture of informal and formal representations. However, the feedback from system is formal.

We proposed an approach to establish the missing link between development time and runtime representations of requirements in the context of embedded systems. Especially smart/embedded systems are interesting, as the human is not in the loop, thus the system must decide autonomously.

The benefits of our approach lie in the ability to communicate changes in the runtime requirements to a requirements engineer. We are currently focusing on parameters. Our future work addresses the communication of the changes to the users and the formalization and monitoring of further aspects of requirements such as conditions and relations between requirements. Further work is also underway on how to automatically resolve conflicts in goal models. Finally, a case study is planned in the context of an industrial research project on automotive software development tool chains.

References

1. 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.
2. Nelly Bencomo and Amel Belaggoun. Supporting decision-making for self-adaptive systems: From goal models to dynamic decision networks. In Joerg Doerr and AndreasL. Opdahl, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7830 of *Lecture Notes in Computer Science*, pages 221–236. Springer Berlin Heidelberg, 2013.
3. J. Brinkkemper, J. Mylopoulos, A. Solvberg, and E. Yu. Tropos: A framework for requirements-driven software development., 2000.

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. E. Mandami and N. Baaklini. Prescriptive method for deriving control policy in a fuzzy logic controller. *Electronic Letters*, 11, 1975.
7. Marc Oriol, Jordi Marco, Xavier Franch, and David Ameller. Monitoring adaptable soa-systems using salmon. In *Workshop on Service Monitoring, Adaptation and Beyond (Mona+)*, pages 19–28, June 2008.
8. Marc Oriol, NaumanA. Qureshi, Xavier Franch, Anna Perini, and Jordi Marco. Requirements monitoring for adaptive service-based applications. In Bjrn Regnell and Daniela Damian, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7195 of *Lecture Notes in Computer Science*, pages 280–287. Springer Berlin Heidelberg, 2012.
9. NaumanA. Qureshi, IvanJ. Jureta, and Anna Perini. Towards a requirements modeling language for self-adaptive systems. In Bjrn Regnell and Daniela Damian, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7195 of *Lecture Notes in Computer Science*, pages 263–279. Springer Berlin Heidelberg, 2012.
10. Daniel Ratiu, Markus Voelter, Bernhard Schaetz, and Bernd Kolb. Language Engineering as Enabler for Incrementally Defined Formal Analyses. In *FORMSERA'12*, 2012.
11. 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.
12. 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.
13. Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley, 2009.
14. M Voelter. Integrating prose as a first-class citizen with models and code. In *7th Workshop on Multi-Paradigm Modelling*, 2013.
15. Markus Voelter. Language and IDE Development, Modularization and Composition with MPS. In *4th Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011)*, LNCS. Springer, 2011.
16. Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Journal of Automated Software Engineering*, 2013.
17. Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.
18. Yiqiao Wang, SheilaA. McIlraith, Yijun Yu, and John Mylopoulos. Monitoring and diagnosing software requirements. *Automated Software Engineering*, 16(1):3–35, 2009.
19. 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.
20. L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338 – 353, 1965.