Research by Experimentation
for Dependability on the
Internet of Things

D-3.1 – Report on Specification Language

Grant Agreement no: 317826
www.relyonit.eu

Date:	November 6, 2013
Author(s) and affiliation:	Luca Mottola (SICS), Thiemo Voigt (SICS), Felix Jonathan Oppermann (UZL), Kay Römer (UZL), Koen Langendoen (TUD)
Work package/task:	WP3, Task 3.1 – Specification Language
Document status:	Final
Dissemination level:	Public
Keywords:	Dependability, Requirement Specification, Language Integration

Abstract This deliverable presents an integrated dependability specification language that enables the user to define dependability requirements for different operation states of the program. Together with the protocol and environment model, this dependability specification forms the basis for the protocol selection and configuration. Two strategies to integrate the dependability specification with different WSN programming languages are proposed. The first integration target is a Java-like, high-level macroprogramming language developed in the 7th framework program project makeSense. The second, more generic integration combines the specification with standard C code for the Contiki platform. Both solutions build on a common XML-based specification language.

Disclaimer

Contents

List of Figures

Executive Summary

In this deliverable we propose an integrated dependability specification language that enables the user to define dependability requirements for different operation states of the program. Together with the protocol and environment models developed in WP1 and WP2, this specification forms an important input for protocol selection and configuration to be developed in Task 3.2 and the run-time adaption of protocol parameters to be developed in Task 3.3.

Existing requirement specification languages are either overly complex and generic or lack commonly agreed formal semantics. Consequently, a dedicated analytical framework is expected to best serve the requirements of the RELYonIT framework. The proposed specification language is based on previous work conducted by Zimmerling et al. [23]. We focus on three key application properties: lifetime, data yield, and latency. As an extension of the original concept, the specification targets application-level performance metrics rather than merely the operation of MAC protocols, and all constraints are associated with a quantified probability of violation.

Individual requirements can be specified per protocol class and may change throughout the application lifetime. A language integration enables to switch between these operation phases at run-time and by this to activate different dependability profiles. In this document we propose two integration strategies for two sensor network programming languages from rather different points of the spectrum in terms of language abstraction level. The first integration target is a high-level macroprogramming language (MPL) developed in the context of the 7th framework program project makeSense. It consists of a stripped-down version of Java that has been extended with constructs to facilitate the development of resource-constrained networked embedded devices. The integration employed existing abstraction concepts found in MPL to seamlessly enable run-time transition between different operational states. The second, more generic integration target is C code for the Contiki platform. Both solutions build on a common XML-based specification language.

Individual advantages and disadvantages of both solutions are discusses, yet the final decision to adopt either approach will be postponed until more detailed requirements of the protocol selection and optimization process (Task 3.2) are available.

1 Introduction

This document presents a specification language for dependability requirements to use in RELYonIT applications. Our design allows sensor network programmers to enforce different dependability requirements throughout the system lifetime. Together with the protocol and environment models developed in WP1 and WP2, this specification forms an important input for protocol selection and configuration to be developed in Task 3.2 and the run-time adaptation of protocol parameters to be developed in Task 3.3.

Dependability is a general concept that may entail different concerns. Because of this, we start in Chapter 2 by presenting preliminary concepts about dependability, making the scope of the project more precise in this respect. Next, we state the requirements that our language design is to accomplish, and in light of those we survey existing approaches. We conclude the chapter with a discussion about pros and cons of the solutions found in the existing literature.

Based on such discussion, Chapter 3 defines a dedicated analytical framework to specify dependability requirements in RELYonIT. The proposed specification language is based on previous work conducted by Zimmerling et al. [23]. We focus on three key application properties: lifetime, data yield, and latency. As an extension of the original concept, the specification targets application-level performance metrics rather than merely the operation of MAC protocols, and all constraints are associated with a quantified probability of violation. We also present our initial thoughts on what specific optimization problem such formulation may entail.

Chapter 4 illustrates two approaches to blend the analytical framework with a concrete sensor network programming language. In doing so, we explore two extremes in a spectrum of language abstraction levels. On the one hand, we present a design to embed the RELYonIT dependability requirement specifications within the makeSense macroprogramming language (MPL): the results of efforts within the FP7 EU-funded project makeSense [6]. On the other hand, we also design a language embedding for Contiki/C, the specific dialect of the C language used in programming for the Contiki [5] sensor network operating system.

We conclude the deliverable in Chapter 5 by providing initial thoughts about what factors may influence the choice of embedding the dependability requirement specification in either sensor network programming language.

2 Preliminaries

In this chapter we provide brief background information useful for motivating the design choices illustrated in the rest of the deliverable.

2.1 Considerations on Dependability

The notion of "dependability" in networked embedded systems may take different forms and refer to different concepts. Therefore, here we seek to briefly recap the possible semantics associated to the notion of dependability and how they play out in the context of the project.

Figure 2.1 reports a schematic view of the dependability taxonomy by Avizienis et al. [2]. The concept of dependability includes four major aspects:

- *availability*, which measures the readiness for correct service of the system, namely, how prompt are the system's reactions when a service is requested;
- *reliability*, which measures the continuity of correct service of the system, namely, to what extent the system can be relied upon to provide the required services;
- *safety*, which measures the absence of catastrophic consequences on the user(s) and the environment due to the system not meeting the application requirements;
- *security*, which measures the system's ability to avoid improper alterations of its behavior in case of (intentional or accidental) malfunctioning.

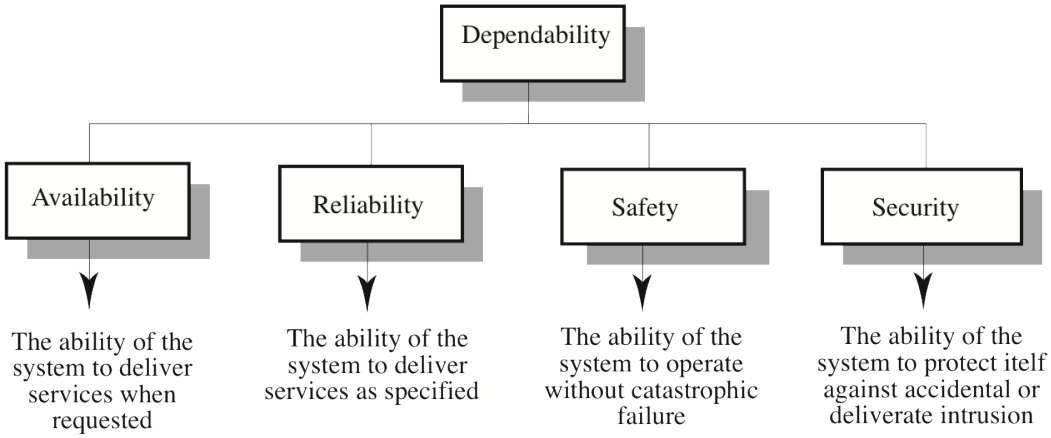

Figure 2.1: Taxonomy of dependability concepts - taken from [2].

The application use cases we consider in the project, described in Deliverable 4.1, exhibit dependability requirements that are mostly related to reliability concerns. For example, the ventilation on-demand scenario requires the system to meet given constraints in data loss, packet latency, and system lifetime. All these aspects, and especially the latter, determine the extent to which the system can meet the application requirements. Similar considerations apply, for instance, to the outdoor parking management scenario as well.

Based on these observations, we mainly consider dependability requirements dealing with reliability concerns. Aspects of availability, safety, and security, although important, are however not in the focus of the project's work.

2.2 Requirements

The dependability specification language we design for RELYonIT must meet some key requirements, such as:

- it must empower developers with ways to express dependability requirements on the underlying networking protocols;
- it must provide inputs for protocol selection and parameterization (Task 3.2), thus being amenable to automated processing by dedicated optimization tools;
- it must allow embedding within existing programming languages to allow the specification of different requirements based on an application's execution state.

Approaches to specifying dependability requirements exist in the most disparate domains. In the following, we briefly discuss those that appear most related to the project's objectives, highlighting what features we may borrow and what aspects would rather prevent their immediate application in the context we consider.

2.3 Existing Approaches

Specifying dependability requirements is a specific instance of the general problem of *non-functional* requirement specification [12, 15]. To this end, the existing literature includes several solutions based on diverse formalisms. In most such cases, however, the different approaches may be conducted to the seminal work about interactions between machine and surrounding environment proposed by Jackson [11].

Approaches to requirement specification at the design stage rely on variations and extensions of existing notations, such as UML [8], or are based on dedicated formally-defined frameworks. For example, Lou et al. [16] propose a pattern-based approach to the specification of non-functional requirements, and integrate such design into existing functional UML models. The pattern-based nature of the approach favors re-use of the requirement specifications across applications. Alloy [10] defines a language framework for requirement specification that offers a syntax compatible with graphical object models, and a set-based formula syntax powerful enough to express complex constraints and yet is amenable to a fully automatic semantic

analysis. In turn, Alloy conceptually builds on the more general Z specification language [20] and the underlying semantics.

Various forms of temporal logic are also often used to specify non-functional requirements [21]. In most cases, these are given as input to model checking tools that probabilistically assess whether a performance objective specified in temporal logic eventually holds in the system, as in the work by Cysneiros and Praido [4]. A different example is that of Aburub et al. [1], who use a custom temporal logic to specify non-functional requirements in business processes. These are then translated into performance objectives and corresponding configurations of the underlying business process execution engine.

At the implementation stage, for example, *software contracts* [18] are a key concept in design-by-contract software engineering methodologies. It prescribes that software designers should define formal, precise, and verifiable interface specifications for software components. Such specifications extend the ordinary definition of abstract data types with preconditions, post-conditions, and invariants. These specifications are in fact referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

In the telecom and networking domains, *service-level agreement*s (SLAs) are negotiated agreements between two or more parties, where one is the user and the others are providers. Such agreements can be a legally binding formal or informal contract. A SLA entails a common understanding about services, priorities, responsibilities, guarantees, and warranties. Each area of service has an associated "level of service" defined. The SLA may specify, for example, the levels of availability, performance, operation, and billing. Such levels of service can also be specified to different degrees, e.g., in terms of a minimum or expected level of service. This generally allows customers to be informed what to expect at the minimum, while providing a measurable average target value. In terms of internal structuring, SLAs commonly include segments to address a definition of services, performance measurement techniques, a problem management schema, customer duties, warranties, disaster recovery, and possible termination of agreement. To ensure that SLAs are consistently met, these agreements are often designed with specific lines of demarcation and the parties involved are required to meet regularly to create an open forum for communication.

SLAs have been applied in disparate domains. In enterprise-level service-oriented architectures, for example, SLAs are used to obtain, monitor, and enforce Quality of Service guarantees between customers and service providers, and to establish the monetary penalties should the guarantees be violated [17]. Similar applications of SLAs are found also in more open environments. Keller et al. [14], for example, define a framework to use SLAs for web services for automatically establishing a billing system depending on the enforced performance guarantees. At infrastructure level, SLAs are often used to specify the service guarantees of backbone networks. Fawaz et al. [7], for example, propose SLA definitions applied to optical networks to manage the bandwidth capacity offered by such network technologies. They define parameters that could be included in their SLA, as well as their values for four classes of services. Different service types—from leased wavelength to bandwidth on demand—are also distinguished.

As opposed to the techniques above, *analytical frameworks* are often employed to quantitatively study the performance of computer systems [13]. Generally speaking, the purpose of an analytical framework is to give the study a disciplined methodology allowing a systematic evaluation of the data at hand. Similar approaches are vastly used especially in networking, where a body of work already exists leveraging analytical frameworks to study the performance

of diverse network technologies, from traditional land-line telecommunication systems to modern wireless networks. For example, Bai et al. [3] define an analytical framework to study the impact of mobility on ad-hoc network routing protocols. The performance of supply chains is analytically studied by Gunasekaran et al. [9]. Instead, C-meter [22] is an analytical framework to study and compare the performance of cloud computing systems.

However, using an analytical framework only defines a very generic methodology, which then needs to be customized to the specific needs of the applications and the systems at hand.

2.4 Discussion

The example approaches discussed above identify very different methodologies, each with its pros and cons w.r.t. possible use in RELYonIT.

Approaches to requirement specification at design and implementation stage, for example, while enjoying a well-specified underlying semantics and being generally amenable to automated processing, appear overkill for the problem at hand, in that, for example, they lend themselves to the specification of arbitrary mechanisms, both functional and non-functional. Specifying dependability requirements in RELYonIT, however, is expected to account for a limited and well-defined set of performance metrics and associated requirements.

SLAs, on the other hand, do match closely the application domains we target in RELYonIT. However, their specification often lacks a commonly agreed-upon semantics and can not always be automatically processed, a key requirement in the project in that the dependability specification shall serve running applications and be used to dynamically adapt their behavior.

It appears that a dedicated analytical framework can best serve the purpose at hand. We will address its specific instantiation in the context of RELYonIT in the next chapter.

3 Specifying Dependability Requirements

To define an analytical framework for specifying dependability requirements, we borrow from Zimmerling et al. [23], who defined an analytical framework for run-time adaptation of MAC protocol parameters in low-power wireless networks. We revise their formulation in two key aspects: *i)* we adapt the framework to account for application-level requirements independent of what protocol layer is optimized, as opposed to the work of Zimmerling et al. [23] who only targeted the MAC level; and *ii)* we extend their formulation so that possible performance constraints are associated to a quantified probability of violation, unlike the generic "soft constraint" concept originally used [23].

3.1 Formulation

We consider as example three performance metrics found in the use case applications of Deliverable 4.1: system lifetime T, data yield R, and latency L. These will be a function of a vector $\mathbf{c}$ that includes the chosen protocol among those available and its operating parameters. We call $\mathbf{c}$ a *protocol configuration* for short, and term the functions $T(\mathbf{c})$, $R(\mathbf{c})$, and $L(\mathbf{c})$ the specific values of T, R, and L when the system applies protocol configuration $\mathbf{c}$.

We treat all but one performance objective as constraints. This way, a dependability requirement entails maximizing or minimizing one performance objective subject to constraints on the remaining metrics, as in

$$\begin{array}{lllllll} \text{Maximize/Minimize} & M_1(\mathbf{c}) & & & & & \\ \text{Subject to} & M_2(\mathbf{c}) & \geq,\leq & C_1 & \textit{probability} & 1-P_{c1} \\ & M_3(\mathbf{c}) & \geq,\leq & C_2 & \textit{probability} & 1-P_{c2} \end{array} \tag{3.1}$$

where each M_i is one among $\{T, R, L\}$ and $\{C_1, C_2\}$ are constraints to be *eventually* satisfied with probability $1 - P_c$, where P_c is the user-provided maximum tolerance for violating the constraint.

Such formulation allows to analytically express the dependability requirements arising in most low-power wireless applications, notably including the motivating use cases in RELYonIT described in Deliverable 4.1. For example, the dependability requirements in the ventilation on demand use case can be formulated as:

$$\begin{array}{lllllll} \text{Maximize} & T(\mathbf{c}) & & & & & \\ \text{Subject to} & R(\mathbf{c}) & \geq & 75\% & \textit{probability} & 1 \\ & R(\mathbf{c}) & \geq & 90\% & \textit{probability} & 0.8 \\ & R(\mathbf{c}) & \geq & 95\% & \textit{probability} & 0.5 \\ & L(\mathbf{c}) & \leq & 5min & \textit{probability} & 1 \\ & L(\mathbf{c}) & \leq & 1min & \textit{probability} & 0.8 \end{array} \tag{3.2}$$

The example also demonstrates how it is possible to express different levels of severity in possibly violating the given constraints. These are specified by expressing multiple constraints on the same performance metric associated to different probabilities of violation. In the example, data yield shall be *guaranteed* to be above 75%; no violations of such constraints are tolerated. Differently, the additional constraints also state that data yield shall possibly be above 90% in at least 80% of the application lifetime, and shall be above 95% in at least half of the application lifetime.

The specification of a performance metric as the single optimization objective is also not mandatory. The dependability requirement arising in the outdoor parking management scenario of Deliverable 4.1, for example, may be expressed only as a set of constraints:

$$\begin{array}{lllll} \text{Maximize/Minimize} & \emptyset & & & \\ \text{Subject to} & R(\mathbf{c}) \geq 90\% & \textit{probability} & 1 & \\ & R(\mathbf{c}) \geq 95\% & \textit{probability} & 0.8 & \\ & R(\mathbf{c}) \geq 99\% & \textit{probability} & 0.5 & (3.3) \\ & L(\mathbf{c}) \leq 30\textit{sec} & \textit{probability} & 1 & \\ & L(\mathbf{c}) \leq 10\textit{sec} & \textit{probability} & 0.8 & \\ & T(\mathbf{c}) \geq 6\textit{months} & \textit{probability} & 1 & \end{array}$$

With this formulation, the requirement is met as long as a protocol's performance satisfies the given constraints during the application's lifetime.

3.2 Optimization Problem

The problem of finding a protocol configuration $\mathbf{c}$ according to the formulation above takes different forms depending on how the constraints are possibly satisfied.

As long as all constraints are satisfied for a protocol configuration $\mathbf{c}$, network designers may not care about what specific value is taken by the performance metrics involved in the constraint definitions. In such a case, *any* protocol configuration $\mathbf{c}$ that maximizes/minimizes the one performance objective $M_1(\mathbf{c})$ in equation (3.1) and satisfies all related constraints is acceptable. The problem thus becomes a *single-objective* optimization problem subject to multiple constraints.

On the other hand, network designers may further consider how a given protocol configuration $\mathbf{c}$ actually satisfies the stated constraints. For example, they may express a specific preference between two solutions that both maximizes/minimizes the one performance objective $M_1(\mathbf{c})$ and satisfy all related constraints, but with different values for $M_2(\mathbf{c})$ and $M_3(\mathbf{c})$ in equation (3.1). In such a case, the problem becomes a *multi-objective* optimization problem (MOP). There may indeed exist a set of solutions that are optimal in the sense that no other solution is superior in *all* objectives, while keeping every constraint satisfied. These are known as *Pareto-optimal* solutions and represent different optimal trade-offs among the involved metrics.

Task 3.2 and 3.3 in the project will explore the trade-offs involved in tackling these different forms of optimization problem. While the latter formulation allows more sophisticated optimization of the network performance, the processing overhead to generate all possible solutions to the multi-objective optimization problem might be overkill.

4 Language Embedding

We present two designs to embed the dependability requirement specification discussed above in a concrete programming language. In doing so, we explore two extremes in a spectrum of language abstraction levels. On the one hand, we present a design to embed the RELYonIT dependability requirement specifications within the makeSense macroprogramming language (MPL). This is the results of efforts within the FP7 EU-funded project makeSense [6], whose goal was to provide programming support for integrating business processes with wireless sensor networks. Nonetheless, sensor network programmers may also use the makeSense MPL as a stand-alone high-level programming language. On the other hand, we also design a language embedding for Contiki/C, the specific dialect of the C language used in programming for the Contiki [5] sensor network operating system.

Both solutions, however, share the basic building blocks of the corresponding tool-chain and a common specification of dependability requirements, which we describe next. Moreover, our designs are based on the following assumptions:

- Dependability requirements are applied to a specific protocol; indeed, the semantics of performance metrics as well as the specific constraints that the developers may impose are, in most cases, protocol-specific, or at minimum related to a homogeneous class of protocols. For example, in a data collection protocol the data yield most often refers to the net packet delivery at a single data sink, whereas for a data dissemination protocol it relates to the percentage of packets delivered to multiple destinations.

- Dependability requirements may change over the application execution; the motivating application scenarios in Deliverable 4.1 already exhibit time-varying dependability requirements, for example, depending on unpredictable emergency situations or drastic changes in the environmental conditions. More generally, the class of applications we target in the project are often characterized by different modes of operations, which naturally correspond to different dependability requirements.

- Dependability requirements apply system-wide, namely no two nodes in the network may have different dependability requirements for the same protocol at the same time. Two aspects relate to this assumption. On the one hand, allowing different dependability requirements at different nodes greatly complicates the optimization problem, likely to the point of making it intractable [23]. On the other hand, allowing different dependability requirements at different nodes entails the possibility that the optimization step indicates different protocols or different parameters for the same protocol at different nodes. Such mixed configurations can rarely actually work, as most existing protocols provide no inter-operability and assume a homogeneous parameter setting.

The assumptions above are key to designing the language embedding of dependability requirements. Moreover, they impact both the conceptual framework where the protocol selection

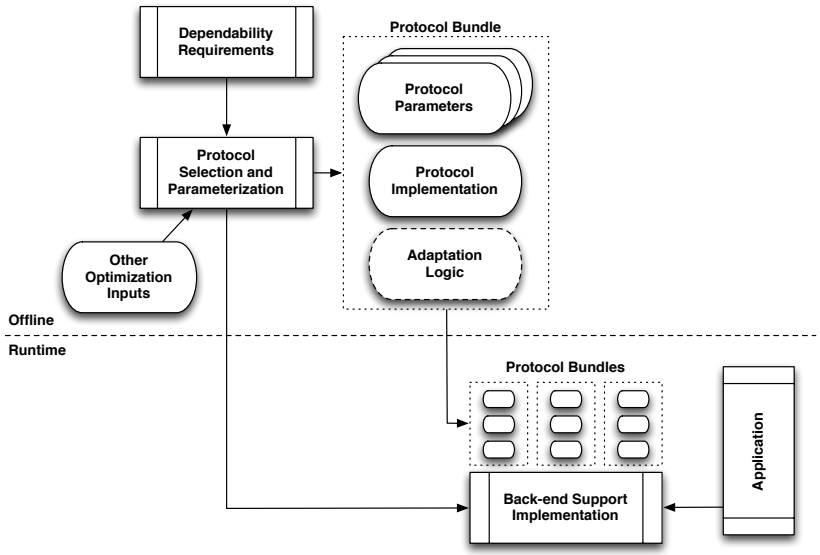

Figure 4.1: Design sketch of RELYonIT tool-chain.

and parameterization functionality (Task 3.2) will operate, and the underlying system support. In particular, based on the third assumption above, the system will need to be equipped with functionality to ensure that when a switch between different protocols occurs, or a change of parameters for the same protocol is triggered, such operation is carried out consistently throughout the network. This can be implemented either at the application level or within the individual protocols. We choose the latter as such mechanisms are, most likely, protocol-specific and hence require intimate knowledge of a protocol's operation.

4.1 Tool-chain

Although the design of the RELYonIT tool-chain depends on how protocol selection and parameterization concretely occurs and in what forms the corresponding solutions are presented, designing a proper language embedding requires to identify the basic building blocks going from the specification of dependability requirements to obtaining running code.

Figure 4.1 shows a design sketch for such a tool-chain. A protocol selection and parameterization tool, developed as part of Task 3.2, takes a machine-readable specification of dependability requirements as input, described in Section 4.2, together with other information useful to determine the most appropriate protocol to use and its parameters; for example, estimates on the network topology characteristics.

The output of the protocol selection and parameterization tool is one or more *protocol bundles*. A single bundle includes the chosen *protocol implementation* and a predefined set of *operating parameters* for the protocol that are known to satisfy, based on the optimization inputs, the dependability requirements at hand. Multiple such sets of optimized operating parameters may be obtained from the protocol selection and parameterization tool for the same

protocol, corresponding to different dependability requirements. When the application changes the current dependability requirement, the individual protocols will switch among the sets of optimized parameters. Optionally, the protocol bundle may also include an adaptation logic component, used to tune the parameters at run-time as the network conditions mutate outside of what was foreseen during the off-line optimization step; for example, whenever the topology characteristics drastically change. These adaption logic components are subject of Task 3.3.

The protocol bundles are deployed together with the executable application code and a thin software layer that implements back-end support functionality to manage protocol bundles. This contains a language-specific encoding of the dependability requirements considered during the optimization process, and is in charge of factoring out the monitoring functionality needed to trigger protocol adaptations based on changing environmental conditions. The back-end support interface and the way programmers access the requirement encoding are language-specific, as we discuss next.

The application interacts with the back-end support for setting the current dependability requirement. As these or the network conditions change, the back-end support may: *i)* switch between different protocol bundles, taking care of the necessary shut-down of currently running protocols and of the start-up of new protocols; *ii)* re-configure a protocol based on the pre-defined sets of parameters included in the bundle, possibly helping with managing the necessary distributed coordination required; or *iii)* execute the adaptation logic, if available, should no predefined parameters be at disposal for the specific situation.

4.2 Dependability Requirement Specification

We use an XML file to specify the dependability requirements input to the protocol selection and parameterization tool in Figure 4.1. The corresponding XML schema is reported in Listing 4.1. This maps to the analytical framework defined in Chapter 3, equation (3.1).

The schema in Listing 4.1 additionally includes a name (supposed to be unique within the application) for the specific dependability requirement, and an indication as to what *protocol class* this requirement applies to. This may take values `COLLECTION`, `DISSEMINATION`, or `P2P`. The former two represent the one-to-many and many-to-one interaction patterns commonly used in sensor networks to enable communication between a node issuing an action and a set of nodes where the action is executed [19]. Collection protocols are used, for example, for a node to gather data from other nodes. Dually, dissemination protocols are used to request the execution of a set of actions on other nodes, e.g., to issue commands to actuators. In contrast, P2P protocols do not focus on a special node, but rather enable a global network behavior and are executed cooperatively by the entire sensor network through many-to-many communication.

A concrete XML example file based on such schema is reported in Listing 4.2, corresponding to the specification of the dependability requirement in the ventilation on-demand scenario of equation (3.2) in Section 3. The requirement is named `Sample Requirement` (line 3), and applies to protocols in the `Collection` class. Note that every possible performance metric has a default binary operator when used as a constraint. For example, in lines 11-13 the data yield R is implicitly required to be *above* the threshold of 0.75, as it would be meaningless otherwise. Similarly, in line 26-28 the latency L is required to be *below* 5 seconds.

Listing 4.1: XML schema for Contiki/C embedding.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="dependabilityReq" type="dependabilityReq"/>

  <xs:complexType name="dependabilityReq">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="protocol_class" type="protocol_class"/>
      <xs:element name="objective" type="metric" minOccurs="0"/>
      <xs:element name="criteria" type="criteria" minOccurs="0"/>
      <xs:element name="constraints" type="constraint"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="protocol_class">
    <xs:restriction base="xs:string">
      <xs:enumeration value="COLLECTION"/>
      <xs:enumeration value="DISSEMINATION"/>
      <xs:enumeration value="P2P"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="metric">
    <xs:restriction base="xs:string">
      <xs:enumeration value="YIELD"/>
      <xs:enumeration value="LIFETIME"/>
      <xs:enumeration value="LATENCY"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="criteria">
    <xs:restriction base="xs:string">
      <xs:enumeration value="MAX"/>
      <xs:enumeration value="MIN"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="constraint">
    <xs:sequence>
      <xs:element name="metric" type="metric" minOccurs="0"/>
      <xs:element name="probability" type="xs:float"/>
      <xs:element name="threshold" type="xs:float"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing 4.2: Concrete XML instantiation of dependability requirement.

```
<?xml version="1.0" encoding="UTF-8"?>
<dependabilityReq xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:noNamespaceSchemaLocation="schema.xsd">
  <name>Sample Requirement</name>
  <protocol_class>COLLECTION</protocol_class>
  <objective>LIFETIME</objective>
  <criteria>MAX</criteria>
  <constraints>
    <constraint>
      <metric>YIELD</metric>
      <threshold>0.75</threshold>
      <probability>1.0</probability>
    </constraint>
    <constraint>
      <metric>YIELD</metric>
      <threshold>0.9</threshold>
      <probability>0.8</probability>
    </constraint>
    <constraint>
      <metric>YIELD</metric>
      <threshold>0.95</threshold>
      <probability>0.5</probability>
    </constraint>
    <constraint>
      <metric>LATENCY</metric>
      <threshold>5</threshold>
      <probability>1</probability>
    </constraint>
    <constraint>
      <metric>LATENCY</metric>
      <threshold>1</threshold>
      <probability>0.8</probability>
    </constraint>
  </constraints>
</dependabilityReq>
```

4.3 Embedding within makeSense MPL

The makeSense macroprogramming language (MPL) is a high-level extensible programming language for sensor networks. Extensibility here refers to the ability to integrate existing programming abstractions with makeSense by embedding custom language constructs in the programming framework. In the following, we briefly describe MPL's key concepts, and illustrate our design for embedding the specification of RELYonIT dependability requirements in makeSense.

4.3.1 MPL Overview

To properly identify the units of functionality, reuse, and extensions, makeSense defines a notion of *meta-abstraction*, implemented through different "concrete" abstractions. These provide the key concepts enabling interaction with the sensor network. Their composition can be achieved by using common control flow statements, provided by a core language that serves as the "glue" among macroprogramming abstractions. The core language is a stripped-down version of Java with extensions to support resource-constrained embedded systems.

Figure 4.2 shows a UML meta-model for the MPL meta-abstractions. The model focuses on the notion of *action*, a task executed by one or more sensor nodes. Actions are separated into *local*, whose effect is limited to the node where the action is invoked (e.g., acquiring a reading from the attached temperature sensor), and *distributed*, whose effect spans multiple nodes.

Distributed actions are further divided into *tell*, *report*, and *collective* actions. In essence, the former two export the functionality of protocols in the `COLLECTION` and `DISSEMINATION` classes at the MPL level. For the former, the needed data acquisition occurring on each involved node is specified by a local action given as parameter to the report action. Similarly, collective actions provide the functionality of `P2P`-class protocols within MPL.

Importantly, distributed actions may optionally have *modifiers* associated with them, "customizing" their behavior. makeSense provides two types of modifiers, *target* and *data operator*.

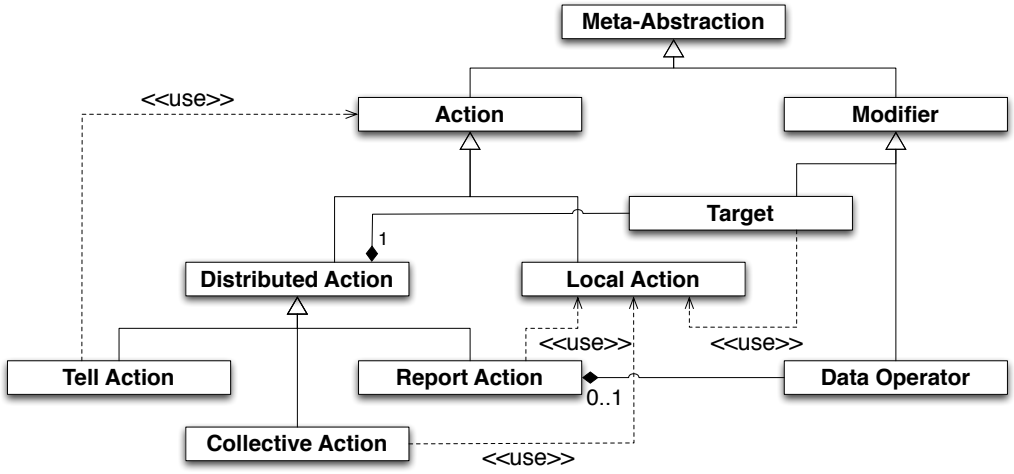

Figure 4.2: A model for the meta-abstractions of makeSense MPL.

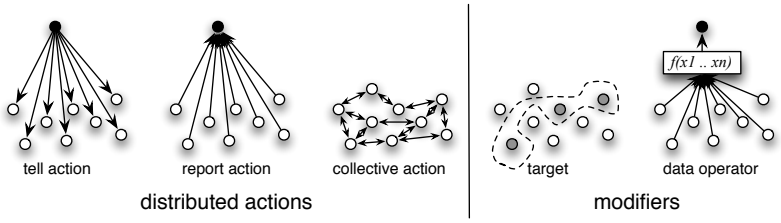

Figure 4.3: Distributed actions and modifiers.

In heterogeneous scenarios, programmers must be able to map actions to the set of nodes of interest. A *target* identifies a set of nodes satisfying application constraints, and gives the ability to apply a distributed action to the nodes in this set. Instead, a report action may have a *data operator*, specifying processing performed on the results after gathering and before they are returned to the caller, e.g., to filter or aggregate the data. Figure 4.3 provides a graphical intuition of the relationship between distributed actions and modifiers.

To create an instance of a meta-abstraction, a class implementing its interface must be defined in the core language. As abstraction implementations typically closely interact with the operating system, methods of abstraction classes are implemented in C using a native code interface provided by the core language.

4.3.2 Modified Meta-model and Interfaces

We embed a notion of dependability requirement in makeSense MPL by providing an additional modifier that customizes the behavior of distributed actions by imposing given dependability

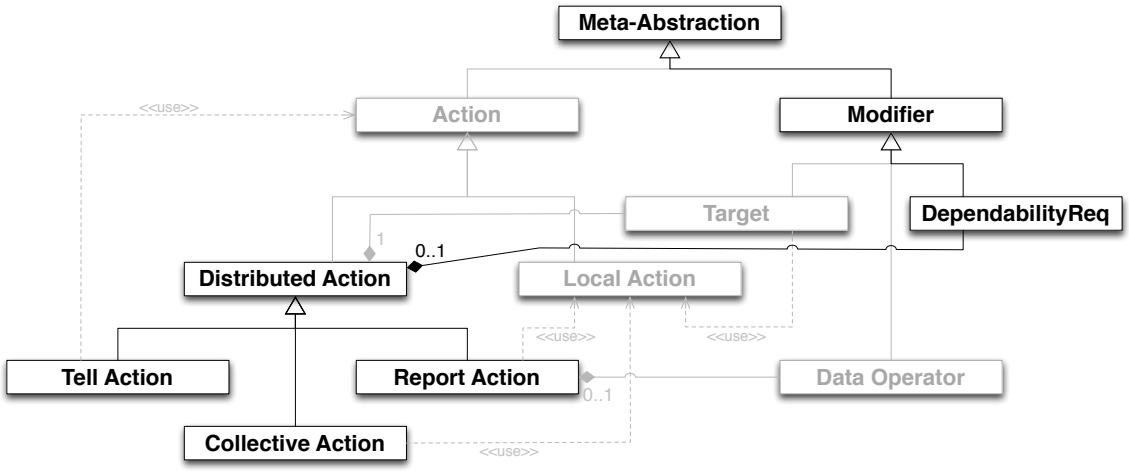

Figure 4.4: A modified meta-model for MPL meta-abstractions: *DependabilityReq* is added to specify dependability requirements for distributed actions. Greyed-out concepts are not affected by the modification.

Listing 4.3: MPL interface for specifying dependability requirements.

```
1 interface DependabilityReq extends Modifier {
2   Objective getObjectiveMetric();
3   List getConstraints();
4 }
```

requirements. Figure 4.4 shows the modified MPL meta-model. A distributed action may have zero or one associated *dependability requirement*(s). As a result of the original structure of the MPL meta-model, such a dependability requirement may concretely apply to *tell*, *report*, or *collective* actions.

Listing 4.3 shows the MPL interface for `DependabilityReq`. Concrete objects implementing that interface are generated by a factory class called `RELYonIT` with a unique method `DependabilityReq createDependabilityReq(String name)` used to extract from the back-end support the encoding of a named dependability requirement, based on the XML specification described in Section 4.2. Such objects are immutable: indeed, methods in Listing 4.3 only allow to gather information from the object without changing it.

`DependabilityReq` objects are passed as parameter to a `setDependabilityReq (DependabilityReq r)` method or returned by a `DependabilityReq getDependabilityReq()` method, both added to the `DistributedAction` interface. The implementation of both methods delegates the processing to the underlying back-end support. An example use is

```
DependabilityReq sample_requirement = RELYonIT.createDependabilityReq("Sample Requirement");
Report report_action = new MyReport();
report_action.setDependabilityReq(sample_requirement);
```

Depending on what type of distributed action the requirement needs to be applied to and what concrete abstraction is currently in use for the same meta-abstraction, the back-end support will carry out the necessary operations to change the underlying protocols' behavior. As already mentioned, this may entail switching the protocol, selecting a different set of operating parameters among those found in the protocol bundle, or running the adaptation logic, if available. The latter may be implemented by extending the existing self-optimization framework in makeSense [6].

4.4 Embedding within Contiki/C

The current practice of programming sensor networks largely relies on low-level languages, mostly based on C [19]. The Contiki operating system is a sensor network programming platform based on C with the addition of a few dedicated macros that provide concurrency and synchronization primitives. In the following we describe an approach to embed dependability requirement specifications in Contiki/C.

Similar to makeSense, the unique name specified in the XML file for a given dependability requirement is used in Contiki/C to obtain a language-specific encoding of the requirement itself, used to set the desired dependability requirement for given protocols classes. An example use is

```
dep_req_t sample_requirement = relyonit_dependability_req("Sample Requirement");
collection_set_dependability(sample_requirement);
```

where the dependability requirement specification for "Sample Requirement" derives from the XML example in Listing 4.2. Specifically, the back-end support provides a C function

```
dep_req_t relyonit_dependability_req(char* name);
```

that maps the named dependability specification in the XML file to an opaque data structure `dep_req_t` that stores a C-specific encoding of the requirement. This is given as input to C functions specific to the three protocol classes we introduce in Section 4.2, such as

```
void protocolclassname_set_dependability(dep_req_t requirement);
```

that are able to parse the information in `dep_req_t` and take care of the necessary operations. Similar to makeSense, these include switching protocols, selecting a different set of operating parameters among those in a protocol's bundle, or running the adaptation logic, if available.

4.5 Implementation Notes

The implementation of the two designs above is actually similar. Before the optimization process takes place, the XML files containing the dependability requirements must be parsed to collect the actual requirement specifications and the associated names. Next, we must generate language-specific encodings of the dependability requirements at hand, and link them to the back-end support implementation. In addition, the protocol bundles must be formed by coupling the implementation of the chosen protocols with the sets of operating parameters and their adaptation logic, if available.

At run-time, when a dependability requirement is set for which optimized parameters are known, the back-end support simply applies the parameters found in the corresponding bundle. Should there be no knowledge on what parameters to use for satisfying a given dependability requirement based on current network conditions, the underlying back-end support will execute the adaptation logic in the same bundle.

5 Conclusion

In this deliverable, we discussed the facets of dependability relevant for RELYonIT, and briefly surveyed existing work related to a specification of dependability requirements. Next, we defined an analytical framework for their specification in RELYonIT and described two designs to embed such specification within existing sensor network programming languages.

In particular, the two designs we outlined have pros and cons. Unlike Contiki/C, the makeSense-based design allows programmers to enjoy a high-level of abstraction in implementing the application-specific processing. On the other hand, the current code base for Contiki/C includes more protocol implementations that the protocol selection and optimization tool may pick from. Also, automatically generating C code for the back-end support in Contiki/C may be assisted by plenty of existing code generation tools that are not available for MPL.

Our immediate plan is to identify specific tools for parsing XML and for automatically generating C code. Based on this, we will assess the expected implementation work for the two designs and settle on a specific choice.

Bibliography

[1] F. Aburub, M. Odeh, and I. Beeson, "Modelling non-functional requirements of business processes," *Information and Software Technology*, vol. 49, no. 11, pp. 1162–1171, 2007.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[3] F. Bai, N. Sadagopan, and A. Helmy, "A framework to systematically analyze the impact of mobility on performance of routing protocols for adhoc networks," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies.* IEEE, pp. 825–835.

[4] L. M. Cysneiros and J. C. S. do Prado Leite, "Nonfunctional requirements: From elicitation to conceptual models," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 328–350, 2004.

[5] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LC)*, 2004.

[6] F. Casati and F. Daniel and G. Dantchev and and J. Eriksson and N. Finne and S. Karnouskos and P. Moreno Montera and L. Mottola and F. Oppermann and G.P. Picco and A. Quartulliz and K. Römer and P. Spiess and S. Tranquilliniz, and T. Voigt, "Towards business processes orchestrating the physical enterprise with wireless sensor networks," in *NIER track, 34th International Conference on Software Engineering (ICSE)*, 2012.

[7] W. Fawaz, B. Daheb, O. Audouin, M. Du-Pond, and G. Pujolle, "Service level agreement and provisioning in optical networks," *Comm. Mag.*, vol. 42, no. 1, pp. 36–43, 2004.

[8] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Languange.* Addison-Wesley Professional, 2004.

[9] A. Gunasekaran, C. Patel, and R. E. McGaughey, "A framework for supply chain performance measurement," *International journal of production economics*, vol. 87, no. 3, 2004.

[10] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, 2002.

[11] M. Jackson, "The world and the machine," in *Proceedings of the 17th international conference on Software engineering (ICSE)*, ser. ICSE, 1995.

[12] ——, *Software requirements and specifications.* Addison-Wesley.

[13] R. Jain, *The art of computer systems performance analysis*. John Wiley & Sons Chichester, 1991, vol. 182.

[14] A. Keller and H. Ludwig, "The wsla framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003.

[15] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., 1998.

[16] Y. Liu, Z. Ma, R. Qiu, H. Chen, and W. Shao, "An approach to integrating non-functional requirements into uml design models based on nfr-specific patterns," in *Quality Software (QSIC), 2012 12th International Conference on*, 2012.

[17] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck, "A service level agreement language for dynamic electronic services," *Electronic Commerce Research*, vol. 3, no. 1-2, pp. 43–59, 2003.

[18] B. Meyer, *Design by contract*. Prentice Hall, 2002.

[19] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, 2011.

[20] J. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 2008.

[21] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. IEEE, 2001, pp. 249–262.

[22] N. Yigitbasi, A. Iosup, D. Epema, and S. Ostermann, "C-meter: A framework for performance analysis of computing clouds," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2009, pp. 472–477.

[23] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele, "ptunes: runtime parameter adaptation for low-power mac protocols," in *Proceedings of the 11th international conference on Information Processing in Sensor Networks (IPSN)*, 2012.

