Maintaining Terminal Integrity and Context-Aware Reconfiguration

Johan Muskens[1], Otso Virtanen[2], Michel Chaudron[1], and Ronan Mac Laverty[3]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{J.Muskens,M.R.V.Chaudron}@tue.nl
http://www.win.tue.nl

[2] Helsinki Institute for Information Technology (HIIT)
Basic Research Unit (BRU)
P.O. Box 26, FIN-00014 University of Helsinki, Finland
otso.virtanen@cs.helsinki.fi
http://www.hiit.fi

[3] Nokia Research Center, Helsinki
P.O. Box 407 FIN-00045 NOKIA GROUP, Finland
ronan.maclaverty@nokia.com
http://www.nokia.com

Abstract. There is a need for terminal management mechanisms for high volume embedded devices. Current technology provides mechanisms for adaptation of software on embedded devices during their deployment. These mechanisms give the opportunity to adapt a device to the wishes of the consumer at a specific location or time. While being useful to the user, these mechanisms might also jeopardize the software integrity on these devices. In this paper we present mechanisms that can be used to adapt the software configuration of a device depending on the context in which it is used. Furthermore we present the mechanisms that can be used to maintain software integrity in a dynamically changing system. The challenge is to increase the value of a device by run-time adaptation of the software configuration to the needs of the consumer, and to maintain software integrity during run-time changes to ensure robust and reliable operation of a device.

1 Introduction

1.1 Background

Our research was carried out in the context of the Robocop and Space4U projects[4]. The goal of these projects is the definition of a component based software architecture for the middleware layer of high volume embedded appliances. High

[4] These projects are funded in part by the European ITEA program and they are joint projects of various European companies, together with private and public research institutes.

volume embedded appliances vary from cellular phones and personal digital assistant to Internet and broadcast terminals like set top boxes, network gateways and digital television sets. Where the Robocop project solves a number of critical issues like the enabling of software IP exchange and supporting (distributed) developments based on resource constrained, robust, reliable and manageable components, the Space4U project extends these goals by including the awareness of the framework and its components with respect to fault-, power- and terminal management related aspects. Our contribution to the Space4U project focuses on the terminal management activities.

In the context of Robocop and Space4U, a terminal is a high volume embedded device. Its management focuses on the period that a terminal is owned and used by a consumer. Integrity refers to functional aspects of the software inside the terminal but also to timing issues and resource use. The terminal must work according to its specification. Context-aware adaptation refers to modifying the terminal based on the location where it is used, the time on which it is used or the user that uses it.

1.2 Motivation

The purpose of our research is to investigate how to increase the value of a device by tailoring it to the context in which it is used and how to achieve robust and reliable operation of devices with dynamically changing software configurations.

There is a need for mechanisms that maintain software integrity in embedded devices. Robust and reliable operation of a device require the software on the device to be consistent and suitable for that device. Producers of embedded devices face the challenge of developing a continually increasing amount of software while time-to-market should preferably decrease. Time-to-market can be reduced by deploying a core system first and upgrading it with additional features when the system is already owned and used by a consumer. This results in embedded systems that are continuously evolving, which significantly complicates the task of integrity management. The system integrity management framework developed within Space4U provides mechanisms for maintaining the integrity of the software in embedded devices.

With the increasing capabilities of embedded devices it is possible to tailor the devices to the context in which they are used. A mobile phone can be used as a shopping list in the supermarket and to look up departure times at the airport. Tailoring the device to the context increases the value of the device to the consumer. The context-aware configuration framework developed within Space4U provides the mechanisms for tailoring devices depending on the context in which they are used.

1.3 Overview

The remainder of this paper is structured as follows. Section 3 discusses the requirements on the architecture imposed by the system integrity management and the context aware frameworks. Our approach is based on models, in Section 4

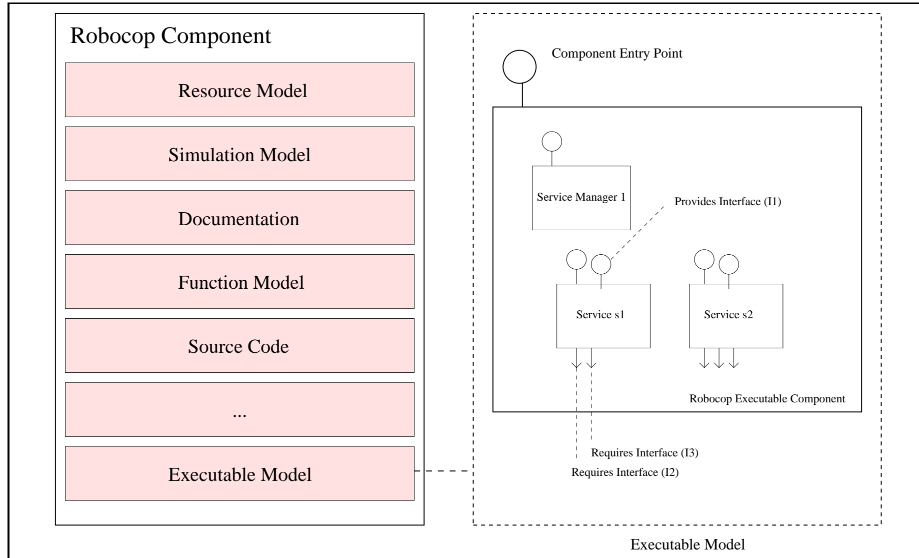

Fig. 1. Robocop component model.

we discuss why and present the models. Section 5 discusses the mechanisms we developed for integrity management. Section 6 discusses the mechanisms we developed for context-aware adaptation. Finally we present some concluding remarks in Section 7.

2 Robocop Framework

In this section we will discuss the Robocop framework. We will focus on the component model and run-time environment since these are strongly related to integrity management and context-aware configuration.

2.1 Component Model

The Robocop component model is inspired by COM [1], CORBA [8] and Koala [9]. A Robocop component is a set of possibly related models. (see Figure 1). Each model provides a particular type of information about the component. Models may be in human-readable form (e.g. as documentation) or in binary form. One of the models is the 'executable model', which contains the executable component. Other examples of models are the resource model and the behavior model. The Robocop component model is open in the sense that new types of models can be added.

An executable component model offers functionality through a set of 'services'. (see Figure 1). Services are static entities that are the Robocop equivalent of public classes in Object-Oriented programming languages. Services are

instantiated at run-time. The resulting entity is called a 'service instance', which is the Robocop equivalent of an object in OO programming languages.

A Robocop service may define several interfaces. We distinguish a set of 'provides' ports and a set of 'requires' ports. The first defines the interfaces that are implemented by a service. The latter defines the interfaces that a service needs from other services. An interface is defined as a set of operations.

2.2 Run Time Environment

The core responsibility of the Robocop run time environment (RRE) is to handle requests for service instances. The RRE maintains a database (registry) that contains the executable component models available, the services implemented by these executable component models, and compatibility information of these services. The Robocop framework allows for run-time adaptation. This means that components can be added and removed from the registry at run-time. Part of the Robocop framework is an optional download framework (RoboCop Download Process, RCDP). The download framework enables loading Robocop components to a terminal. Executable component models can be registered after they are resident on the device.

3 Additional Requirements on Architecture

Our approach in solving the context-aware configuration started by requirements gathering and identification of the needed roles or frameworks in a terminal to enable this reconfiguration. Early on we also decided to first investigate the switching and downloading of components to a terminal based on context and to leave the application level adaptation for future work. Nevertheless, the solution we have designed could be easily modified to offer notifications about context changes to application layer as well.

Dynamically changing components in a terminal based on context clearly calls for an accompanying framework that provides mechanisms for guarding the terminal by making sure that the components downloaded are behaving correctly and that they contain no defects. This is where the *System Integrity Management* (SIM) framework is needed. The system integrity management framework is explained in detail in Section 5.

For the context-aware framework in terminal we identified the following needed sub-systems:

- Interaction with the Robocop run-time environment (RRE) and download framework,
- Context monitoring services and
- A placeholder for context data.

As previously described, the RRE provides an API e.g. for un/registering components and instantiating services. The download framework makes it possible for a terminal to fetch components from an outside repository. An entity

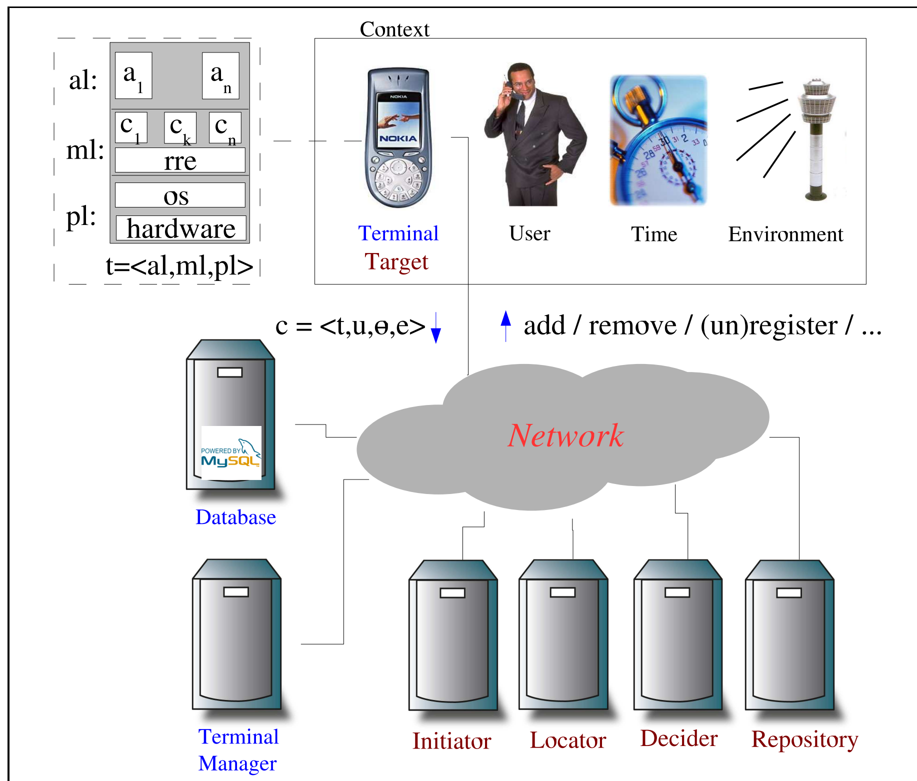

Fig. 2. High level set-up.

interacting with these sub-systems is needed in terminal reconfiguration and is therefore part of the context-aware framework.

Context monitoring means that the terminal should be able to see e.g. which devices or Bluetooth beacons are nearby by periodic scans of the neighborhood. The beacons might emit an XML-formatted string describing the location or an URL which could be further mapped to the needed components in this specific context, for details see [6].

The last item in the list is where the terminal keeps all the context related data that is considered to be useful for reconfiguration purposes. This data structure should be accessible through a simple API e.g. for querying and depositing data. We decided to use a blackboard-based approach which has been used previously in other context-aware frameworks as well [13]. The structure of this data, i.e. models, is further described in Section 4.

4 A model based approach

Our approach focuses on integrity management and adaptation based on a model of a terminal and the context in which it is used. Terminals as well as the context in which they are used can be considered to be very complex things that contain a large number of details. We use a model to abstract from the details and to simplify the complex things we are dealing with. In our approach terminals maintain and externalize a self-model. In Section 5 and Section 6 we describe how these models are used for managing integrity and adaptation.

Next we will describe our model for the context and the terminal which is maintained in the terminal and needed for reconfiguration of components. In the remainder of this paper we will use the following convention: sets are identified in uppercase, elements out of these sets are identified in lowercase. For example, T is the set of all terminals and $t \in T$ identifies one specific terminal.

The context is determined by the configuration of the terminal(t), the user(u) using the terminal, the time(θ) at which the terminal is used and the environment(e) in which the terminal is used. A somewhat similar division has been previously identified e.g. in [3] and [11]. We model the specific context $c \in C$ as follows: $c =< t, u, \theta, e >$.

$$C = T \times U \times \Theta \times E \tag{1}$$

In the remainder of this section we will discuss the models for the terminal, user, time and environment individually.

4.1 Terminal model

Within Robocop and Space4U we distinguish an application layer, middleware layer and platform layer on a terminal. We model these layers individually. In our model a terminal $t \in T$ is modeled as follows: $t =< al, ml, pl >$

$$T = AL \times ML \times PL \tag{2}$$

Application layer model The application layer (al) is modeled as a set of applications. Each application a is described by a *name*, a *version*, its dependencies d on services and the *structure* that it will create when launched. The dependencies of an application are modeled by a set of services ($d \subseteq S$) that are needed by the application in order to execute.

Services provide their functionality through a number of named interfaces called ports. In order to provide this functionality a service might require functionality from other services. This dependency is made explicit through the use of required ports, which are also named interfaces. We model a service $s \in S$ as follows: $s =< provided, required >$, where $provided \in \mathbb{P}(P)$ and $required \in \mathbb{P}(P)$.

The *structure* of an application consists of a set of service instances ($ssi \in \mathbb{P}(S)$), services that will be instantiated at run-time and the bindings ($sb \in \mathbb{P}(B)$) between these services instances. A binding ($b =< p_1, p_2 >$) between two service

instances means that two ports are connected, a required port (p_1) of one service instance to a provided port (p_2) of another.

$$
\begin{aligned}
AL &= \mathbb{P}(A) && (3)\\
A &= NAME \times VERSION \times D \times STRUCTURE && (4)\\
NAME &= STRING && (5)\\
VERSION &= STRING && (6)\\
D &= \mathbb{P}(S) && (7)\\
S &= \mathbb{P}(P) \times \mathbb{P}(P) && (8)\\
P &= \mathcal{N} \times I && (9)\\
\mathcal{N} &= STRING && (10)\\
I &= \mathbb{P}(O) && (11)\\
STRUCTURE &= \mathbb{P}(SI) \times \mathbb{P}(B) && (12)\\
SI &= S \times NAME && (13)\\
B &= P \times P && (14)
\end{aligned}
$$

Middleware model The middleware layer consists of a run-time environment (*runtime*), a set of registered executable components models ($sem \in \mathbb{P}(EM)$), and a set of complies relations between services ($sc \in \mathbb{P}(COMPLIES)$). The run-time environment is modeled by its version number. An executable component model is modeled by the a set of implemented services ($ss \in \mathbb{P}(S)$). A complies relation consist of one service (s_1) that is compliant with another (s_2), therefore we model a complies relation as follows:$complies =< s_1, s_2 >$

$$
\begin{aligned}
ML &= RUNTIME \times \mathbb{P}(EM) \times \mathbb{P}(COMPLIES) && (15)\\
RUNTIME &= VERSION && (16)\\
EM &= \mathbb{P}(S) && (17)\\
COMPLIES &= S \times S && (18)
\end{aligned}
$$

Platform model The platform layer consists of an operating system (*os*), a *cpu*, and *storage*. The operating system is identified by its *name* and *version*. The cpu is identified by its *vendor*, the *family* that its part of, the clock*speed*, and its *cache*size. Storage consists of *memory*, *swap* and a set of filesystems. Each filesystem is modeled by its *name* and size.

$$
\begin{aligned}
PL &= OS \times CPU \times STORAGE && (19)\\
OS &= NAME \times VERSION && (20)\\
CPU &= VENDOR \times FAMILY \times MODEL \times && (21)\\
&\quad SPEED \times CACHE \\
& && (22)
\end{aligned}
$$

$$VENDOR = STRING \tag{23}$$
$$FAMILY = STRING \tag{24}$$
$$MODEL = STRING \tag{25}$$
$$SPEED = \mathbb{N} \tag{26}$$
$$CACHE = \mathbb{N} \tag{27}$$
$$STORAGE = MEMORY \times SWAP \times P(FS) \tag{28}$$
$$MEMORY = \mathbb{N} \tag{29}$$
$$SWAP = \mathbb{N} \tag{30}$$
$$FS = NAME \times \mathbb{N} \tag{31}$$

4.2 User model

The user model consists of information regarding the identity, group, role, location and activity. Relevant standards and existing definitions to describe in more detail this layer have been addressed e.g. in Open Mobile Alliance's[5] LOC (location working group) and IMPS (presence information standard).

$$U = IDENTITY \times LOCATION \times ACTIVITY \tag{32}$$
$$IDENTITY = NAME \times GROUP \times ROLE \tag{33}$$
$$NAME = STRING \tag{34}$$
$$GROUP = STRING \tag{35}$$
$$ROLE = STRING \tag{36}$$
$$LOCATION = STRING \tag{37}$$
$$ACTIVITY = STRING \tag{38}$$

4.3 Time model

The time layer has information of the current time in standard UTC format. To make it more interesting for adaptation purposes some higher-level properties of time might also be included. These higher-level properties can be e.g. $WEEKEND - NOT_WEEKEND, NIGHT - DAY, WORKING_HOURS - NOT_WORKING_HOURS$ and inference of the current property can be in simplest form based on current time.

$$\Theta = CURRENT_TIME \times HIGH_LEVEL \tag{39}$$
$$CURRENT_TIME = STRING \tag{40}$$
$$HIGH_LEVEL = STRING \tag{41}$$

[5] `http://www.openmobilealliance.org/`

4.4 Environment model

The environment model consists of information about nearby devices and the information they possibly emit that can be found with the already mentioned Bluetooth scanning. Other information falling into this model might include sensor readings from the environment (e.g. temperature) which can be further processed to higher level context information. An example of an inference framework enabling this is given in [7].

5 Managing integrity on mobile terminals

Software systems are continuously evolving; both during their development as well as during deployment. To cater for an increasing demand for flexibility, the Robocop architecture provides means for dynamic replacement, addition, and removal of components. In Robocop, as well as in other component models, service interfaces and service implementations are separated to support 'plug-compatibility'. This allows for different services, implementing the same interfaces to be replaced. This facility introduces the task of maintaining a consistent software configuration that is suitable for the terminal. This task involves:

- monitoring of the current software and hardware configuration of the terminal,
- verifying the real configuration against a preferred state (diagnosis) and
- restoring a preferred state (repairing).

Using the model for a terminal and context as described in Section 4 we are able to monitor the configuration of a terminal and verify it. To this end we developed a solution based on three roles:

Terminal role This role has two main responsibilities. First this role externalizes the model of the terminal and context. Second this role provides a mechanism to execute some basic actions (adding, removing, and replacing executable component models) from a remote device.

Terminal manager role This role is responsible for monitoring, diagnosis, and repairing. Monitoring comes down to retrieving the models externalized by terminals. Based on the retrieved models and rules provided by the database role the terminal manager is able to create a diagnosis report. A repair script can be generated based on this diagnosis report and known solutions also provided by the database. This repair script can be executed using the mechanisms provided by the terminal role to execute basic actions from a remote device. For an overview of this typical integrity management scenario see Figure 3.

Database role This role provides integrity rules, that can be used during diagnosis. Furthermore this role provides known solutions for certain problems that can be found during diagnosis.

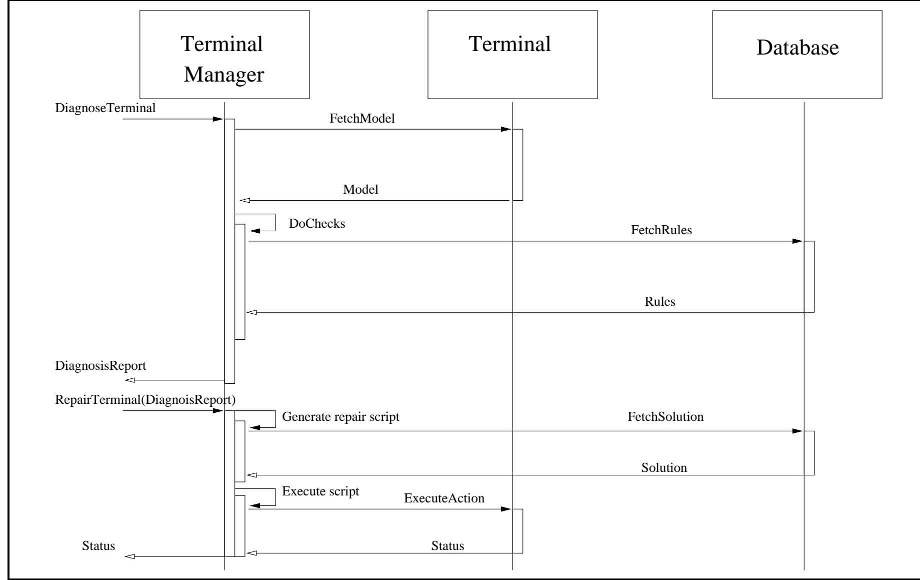

Fig. 3. Integrity management scenario.

These roles can be deployed on different devices or all on one device. The only restriction is that the terminal role must be deployed on the device that is managed. In the remainder of this section we will discuss two possible checks that can be used during the diagnosis phase.

5.1 Blacklist of services

During the life-cycle of a component bugs in the services that they provide can be discovered. If the bug is discovered after the component has been published, the component can be deployed on an unknown set of terminals. It is impossible, undesirable and even illegal to track the deployment of every component. We propose to maintain a database that contains the services in which a bug has been discovered, such that these services can be replaced by the terminal management activities.

D_{BoS} is a (partial) function that takes an element $t \in T$ and gives the set of services provided by the middleware of the terminal t that are on the blacklist S_b.

$$S_b = \text{service blacklist (set of services with known faults)} \tag{42}$$

$$D_{BoS}(t) = (\cup em : ml =< x, y, z > \wedge (em \in y)) \cap S_b \tag{43}$$
$$\text{where } t =< al, ml, pl >$$

For the generation of the repair script we will need executable component models that implement services that are compliant with the services that are

on the blacklist, but are not on the blacklist themselves. The database role can provide suggested replacements for the services that are on the blacklist.

5.2 Missing services

Applications can require a number of services to fulfill their task. In a dynamically changing terminal, to which new applications are downloaded and from which applications are removed, it can be hard to keep the set of services provided by the middleware consistent with the applications. We will now discuss how to verify that all required services are provided by the middleware based on the application layer- and middleware layer model.

$D_{MS}(t)$ gives the set of services that are needed by one or more applications on terminal t but that are not provided by the middleware of terminal t. $D_{MS}(t)$ is the difference between $S_r(t)$ and $P_c(t)$. S_r is the set of services that is required by the set of applications that is available on terminal t, this comes down to the union of the dependencies of these applications. $P_c(t)$ is the set of services that can be provided by the middleware of terminal t, this comes down to union of the set of services provided by the middleware with all the services that are compliant with the services provided by the middleware according to the complies relations that are registered.

$$\begin{aligned}
P_c(t) = {} & (\cup em : ml =< x, y, z > \wedge (em \in y)) \cup \\
& \{s_1 \in S \mid \\
& (s_2, s_1) \in (\cup complies : ml =< x, y, z > \wedge \\
& (complies \in closure(z))) \wedge \\
& s_2 \in (\cup em : ml =< x, y, z > \wedge (em \in y))\} \\
& \text{where } t =< al, ml, pl > \\
& \text{and } closure(z) \text{ gives the transitive closure} \\
& \text{of the complies relations in } z
\end{aligned} \tag{44}$$

$$\begin{aligned}
S_r(t) = {} & \{s \in S \mid s \in d \wedge \\
& < name, version, d, structure > \in al\} \\
& \text{where } t =< al, ml, pl >
\end{aligned} \tag{45}$$

$$D_{MS}(t) = S_r(t) \setminus P_c(t) \tag{46}$$

Generation of the repair script can be straight forward. The repair script can add executable component models that provide the missing services or remove the applications that require these missing services.

6 Adapting mobile terminals based on context

So far we have presented all the models (i.e. terminal, user, time and environment) and the integrity management mechanisms (i.e. the blacklist of components and missing services) but we have not yet depicted their usage in detail

in context-aware reconfiguration of a terminal. A scenario illustrating our approach is therefore outlined. Consider a user equipped with a Robocop terminal arriving to an airport. The context framework in the terminal notices the arrival to the airport (e.g. through periodic scanning of the environment or location data usage) and receives an identifier that can be further mapped to a list of components which are preferred in this specific location. One of the components offers a timetable functionality (in this case the component is also executable) using two existing components in the terminal. In this case one of the existing (i.e. a component registered to the registry) components in the terminal does not offer a specific service required by the application and a missing service script is prepared to repair the situation. Luckily the component repository contacted by the device contains an updated version of the component in question that has one more provided interface which in this case is the one needed by the timetable component. The component is downloaded to the terminal and registered to the RRE.

The user executes the timetable application and easily finds all the needed departure times of flights etc. Later on that day the user notices that the device is not working correctly or more specifically two applications using the same newly downloaded component are not executing properly. The user therefore starts the *check terminal status* -procedure from the management interface of the terminal. After this the SIM framework sends all the model data deposited to the terminal to a server offering integrity checking services. The server contains information about the newly downloaded component which has been blacklisted after the user downloaded it by a developer who has noticed problems with this specific component after it was released. Luckily, a bug-free implementation of the component can be found from the repository and this component is downloaded and registered to the RRE.

7 Conclusions

Current technology allows for run-time adaptation of software on embedded devices. This gives the opportunity to adapt a device to the wishes of consumer at a specific location or time. This also introduces some risks concerning the software integrity on the device.

There is a need for mechanisms that maintain software integrity in embedded devices. Robust and reliable operation of a device require the software on the device to be consistent and suitable for that device. In this paper we present the mechanisms we used to maintain a consistent software configuration on a device. We applied these mechanisms in the context of high volume consumer electronic devices. Our approach for software integrity management has two large advantages:

- Support for local integrity management as well as for remote integrity management. The roles we use in our solution can be deployed in a flexible manner. This allows for a terminal manager to be deployed on a terminal or a remote server.

– Management based on models allows to abstract from the details. Using the models we are able to consider integrity at a configuration level. Simplicity is achieved by abstracting from the details of the specific service instances.

Some work has been done on remote management and configuration of a device. SNMP [2] and SyncML DM [10] are existing network / device management technologies that are widely adopted by industry. These technologies focus on the mechanisms needed for remote configuration and adaptation. We focus more on what needs to be adapted (diagnosis), how (repair script), and when (depending on context).

Some work has also been done on architecture based approaches for self-healing systems [5] [12]. These approaches focus on detecting when to make a particular repair based on architectural styles. In their approach, an architectural style is a set of constraints. Constraint violation is cause for inducing a repair. The difference with our approach is that our approach allows diagnosis and repairing based on different kind of models (resource, behavior, etc). Our diagnosis is not restricted to use only the architectural description of a system.

The value of a device for a consumer can be increased by customizing the device to the wishes of the consumer in a specific context. For example a mobile telephone can present an e-commerce application when visiting a mall or an airplane arrival and departure schedule application in an airport. This paper presented mechanisms for context-aware configuration applicable for component based embedded devices. Related work with similar kind of context-aware mechanisms and entities supporting download and managing context information in component middleware can be found in [4].

8 Acknowledgments

We are grateful to all the partners of the Robocop and Space4U projects for their contributions: Philips Electronics, Nokia, CSEM, Saia Burgess, ESI, Fagor, Ikerlan, University Polytechnic de Madrid, Helsinki Institute for Information Technology and Eindhoven University of Technology.

References

1. D. Box. *Essential COM.* Object Technology Series. Addison-Wesley, 1997.
2. J. Case, R. Mundy, D. Partain, and B. Steward. Introduction to version 3 of the internet-standard network management f ramework. Technical Report RFC 2570, SNMP Research, TIS Labs at Network Associates, Ericsson, Cisco Sy stems, 1999.
3. G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical Report TR-2000-381, Dept. of Computer Science, Dartmouth College, 2000.
4. D. Conan, C. Taconet, D. Ayed, L. Chateigner, N. Kouici, and G. Bernard. A proactive middleware platform for mobile environments. In *IASTED International Conference on Software Engineering*, Feb. 2004.

5. E. Dashofy, A. van der Hoek, and R. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*. ACM, Nov. 2002.
6. T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic. People, places, things: Web presence for the real world. *Mobile Networks and Applications*, 7.
7. P. Korpipää, J. Mäntyjärvi, J. Kela, H. Keränen, and E.-J. Malm. Managing context information in mobile devices. *Pervasive Computing*, 2(3), July-September 2003.
8. T. Mowbray and R. Zahavi. *Essential Corba*. John Wiley and Sons, 1995.
9. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, Mar. 2000.
10. Open Mobile Alliance. SyncML Device Management initiative consolidated into OMA, http://w ww.openmobilealliance.org/tech/affiliates/syncml/ syncmlindex.html, 2002.
11. B. N. Schilit, N. I. Adams, and R. Want:. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, Dec. 1994.
12. B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing sys tems. In *Fourteenth International Conference on Software Engineering and Kno wledge Engineering*, 2002.
13. T. Winograd. Architectures for context. *Human-Computer Interaction*, 16:401–419, 2001.