Integrity Management in Component Based Systems

Johan Muskens and Michel Chaudron
Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, P.O. Box 513,
5600 MB Eindhoven, The Netherlands

E-mail: J.Muskens@tue.nl, M.R.V.Chaudron@tue.nl

Abstract

There is a need for mechanisms for maintaining and restoring software integrity on deployed systems. Dynamic replacement, removal and addition of components in deployed systems is supported by most component models. This is needed to enable the software on a device to evolve in the period that it is owned by a consumer, but endangers the integrity of the software on these devices.

For high volume consumer devices the challenge is to keep the devices operating reliable and robust in the period that it is owned, used and possibly reconfigured by a consumer. To this end, we propose mechanisms and tools, for maintaining system integrity on deployed systems.

1. Introduction

1.1. Background

Our research was carried out in the context of the Robocop and Space4U projects. The goal of these projects is the definition of a component based software architecture for the middleware layer of high volume embedded appliances. High 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 on this 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 consumer 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. E.g. after adding or replacing components into the component framework the terminal must remain working according to its specification.

In order for the architecture to support the requirements discussed before, we need an architecture that supports:

- maintaining a model of a terminal to reason about itself and to verify it against a preferred state;
- dynamically evaluating non-functional properties.

1.2. Motivation and Research Question

There is a need for integrity management mechanisms. Software systems must be robust and reliable. Now software systems are continuously evolving; both during their development as well as during deployment. Component models provides means for the dynamic replacement, removal and addition of components in a deployed system, to cater for an increasing demand for flexibility. This facility endangers the software integrity of the system. Our research question is: How to ensure robust and reliable operation of dynamically changing software systems? The integrity management mechanisms described in this paper provide means for maintaining and restoring integrity in order to achieve robust and reliable operation of a device.

---

1 These projects are funded in part by the European ITEA program. These are a joint projects of various European companies, together with private and public research institutes.
1.3. Overview

In this paper we describe our approach for integrity management. We divided integrity management into monitoring, diagnosis and repairing. A ‘self-model’ externalized by a terminal is the basis for integrity management in our approach. We verify the current state against the preferred state and evaluate non-functional properties based on this ‘self-model’. In this paper we will focus on the model used to describe the current configuration of the terminal and how this can be used for monitoring and diagnosis.

The remainder of this paper is structured as follows. In section 2 we discuss our view of a terminal. In section 3 we discuss the Robocop component model. In section 4 we discuss the models we use to describe the individual layers of a terminal. In section 5 we illustrate how these models can be used for remote run-time diagnosis of a terminal. Section 6 discusses related work and draws some conclusions.

2. Terminals

The terminals targeted by Robocop and Space4U are high volume consumer devices. They are assumed to have limited resources (CPU, memory, power, etc). The terminal may, but does not need to be, connected to the outside world through some sort of link and may be mobile or stationary. Obviously a connection to the outside world is required to do remote monitoring and diagnosis.

Within a terminal we identify three layers (see figure 1), a platform layer, a middleware layer and an application layer. The platform layer is considered to be relatively static, whereas the middleware- and application layer are more dynamic. This implies that maintaining system integrity will be done through (re)configuring the application and middleware layer. The challenge is to maintain a consistent configuration of these two layers which is suitable for the specific platform. In the remainder of this section we will briefly discuss the individual layers.

2.1. Platform layer

The platform layer consist of an operating system, sometimes an operating system abstraction layer, and some device drivers. The platform layer not likely to change at run-time, but the platform influences which middleware- and application layer configurations are suitable. This means that even though this layer is not likely to change, we do need to take it into account in the monitoring and diagnosis process of the terminal.

2.2. Middleware layer

The middleware layer consists of a run-time environment and a set of executable components. The run-time environment provides the core middleware functionality (registering components, loading of components and unloading of components). The middleware functionality specific for a terminal is provided by a set of components. The set of components can change at run-time. The component model is discussed in section 3.

2.3. Application layer

The application layer consists of a set of applications. This set can also change at runtime, for example the user can download a new application. The type of applications is left unspecified. These applications can be ‘stand-alone’ or they can require one or more robocop components to fulfill their task. This implies that modifications in the application layer can require modifications in the middleware layer.

3. Robocop Component Model

In this section we will discuss the Robocop component model. The Robocop component model is inspired by COM [1], CORBA [5] and Koala [7]. A Robocop component is a set of possibly related models. (see figure 2). 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 \( \text{erm} \in \text{EM} \) and the behavior model \( \text{bmn} \in \text{BM} \). The Robocop component model is open in the sense that new types of models can be added.
3.1. Executable model

A executable component offers functionality through a set of ‘services’ (∈ P(S)). (see figure 2). Services are static entities that are the Robocup 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 Robocup equivalent of an object in OO programming languages.

\[ RC = P(M) \times P(M \times M \times T) \]  \hspace{1cm} (1)
\[ M = EM \cup BM \cup RM \cup \ldots \]  \hspace{1cm} (2)
\[ T = MODELTYPE \times NAME \]  \hspace{1cm} (3)

\[ \varphi : M \mapsto MODELTYPE \]  \hspace{1cm} (4)
\[ \varphi : M \mapsto NAME \]  \hspace{1cm} (5)

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

\[ S = P(P) \times P(P) \]  \hspace{1cm} (6)
\[ P = N \times I \]  \hspace{1cm} (7)
\[ N = STRING \]  \hspace{1cm} (8)
\[ I = P(O) \]  \hspace{1cm} (9)
\[ O = \text{set of operations } o \]  \hspace{1cm} (10)
\[ \text{set of operations } o \]  \hspace{1cm} (11)

3.2. Resource model

Resource Models specify the predicted resource consumption of Executable Components. The Resource Model specifies the predicted resource consumption for all the operations implemented by services of an Executable Component. The resource model contains a number of cost functions. A cost function \( O_{imp} \mapsto \mathbb{N} \) gives the cost \( (c \in \mathbb{N}) \) for an operation \( (o_{imp} \in O_{imp}) \). Per resource \( r \) there can be multiple cost functions \( (r \mapsto P(O_{imp} \mapsto \mathbb{N})) \).

\[ RM = R \mapsto P(O_{imp} \mapsto \mathbb{N}) \]  \hspace{1cm} (12)
\[ O_{imp} = S \times O \]  \hspace{1cm} (13)
\[ R = \text{set of resources } r \]  \hspace{1cm} (14)

3.3. Behavior model

Behavior Models specify the behavior of Executable Components. The Behavior Model specifies the behavior for all operations implemented by services of an Executable Component in terms of operations on interfaces. The behavior \( b \) is modelled as a sequence of operations \( (b \in O^*) \). The behavior is specified for an operation implementation \( (o_{imp}) \).

\[ BM = O_{imp} \mapsto O^* \]  \hspace{1cm} (15)

4. Model of a Terminal

Our approach focuses on integrity management based on a model of a terminal. Terminals are 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 we describe how these models are used for managing integrity.

Next we will describe our model for the context and the terminal. In this paper we 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\( (\theta) \) at which the terminal is used and the environment\( (e) \) in which the terminal is used. We model a specific context \( c \in C \) as follows: \( c =< t, u, \theta, e >. \)

\[ C = T \times U \times \Theta \times E \]  \hspace{1cm} (16)

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: $t = \langle al, ml, pl \rangle$

$$T = AL \times ML \times PL \quad (17)$$

4.1.1. 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:

$$s = \langle \text{provided}, \text{required} >$$

where $\text{provided} \in P(P)$

and $\text{required} \in P(P)$

The structure of an application consists of a set of service instances $(ssi \in P(S))$, services that will be instantiated at run-time and the bindings $(sb \in P(B))$ between these services instances. A binding $(b = \langle si_1, p_1, si_2, p_2 \rangle)$ between two service instances $(si_1$ and $si_2)$ means that two ports are connected, a required port $(p_1)$ of one service instance to a provided port $(p_2)$ of another.

$$AL = P(A) \quad (18)$$

$$A = NAME \times VERSION \times (19)$$

$$D \times STRUCTURE \quad (20)$$

$$NAME = STRING \quad (21)$$

$$VERSION = STRING$$

$$D = P(S) \quad (22)$$

$$STRUCTURE = P(SI) \times P(B) \quad (23)$$

$$SI = S \times NAME \quad (24)$$

$$B = SI \times P \times SI \times P \quad (25)$$

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

$$ML = RUNTIME \times P(EM) \times P(COMPLIES) \quad (26)$$

$$RUNTIME = VERSION \quad (27)$$

$$EM = P(S) \quad (28)$$

$$COMPLIES = S \times S \quad (29)$$

4.1.3. 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 clockspeed, and its cachesize. Storage consists of memory, swap and a set of file systems. Each file system is modeled by its name and size.

$$PL = OS \times CPU \times STORAGE \quad (30)$$

$$OS = NAME \times VERSION \quad (31)$$

$$CPU = VENDOR \times FAMILY \times MODEL \times SPEED \times CACHE \quad (32)$$

$$VENDOR = STRING \quad (33)$$

$$FAMILY = STRING \quad (34)$$

$$MODEL = STRING \quad (35)$$

$$SPEED = N \quad (36)$$

$$CACHE = N \quad (37)$$

$$STORAGE = MEMORY \times SWAP \times P(FS) \quad (38)$$

$$MEMORY = N \quad (39)$$

$$SWAP = N \quad (40)$$

$$FS = NAME \times N \quad (41)$$

5. Remote Diagnosis of a Terminal

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. 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);
- restoring a preferred state (repairing).
Using the model for a terminal as described in section 4 we are able to monitor the configuration of a terminal and verify it. This diagnosis process is illustrated by figure 3. The basis for this process is the model for a terminal. This model is externalized by the terminal and can be fetched by a terminal manager. The terminal manager has access to a database with rules and solutions. The rules indicate desired and undesired configurations. Solutions describe recipes to migrate from an known undesirable state to a desirable state. Using this information a diagnosis can be made and if necessary a repair plan can be generated and executed based on the diagnosis.

In the remainder of this section we will discuss a number of example checks that can be used during diagnosis, and how the detected problems can be repaired. In this paper the focus will be on the diagnosis.

5.1. Blacklist of Services

During the life-cycle of a component bugs in the services that they provide can be discovered. When the bug is discovered after the component has been published then the component can be deployed on an unknown set of terminals. It is impossible, undesirable and probably 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.

\[ D_{BoS}(t) = \{ \text{service blacklist} \} \]

Executable component models that provide services compliant with the services in \( D_{BoS}(t) \) are needed for the generation of the repair script. These compliant services should not be on the blacklist themselves. The database role can provide suggested replacements for the services that are on the blacklist.

5.2. Blacklist of Complies Relations

The set of compliance relations represents the knowledge of the middleware about the compatibility of services. If this knowledge is incorrect this can result into errors. When it is discovered that a complies relation is incorrect after it has been published then the relation can be 'deployed' on a unknown number of terminals. We propose to maintain a blacklist of published complies relations that are incorrect, such that they can be removed during the terminal management activities.

\[ D_{BoC}(t) = \{ \text{complies blacklist} \} \]

The repair script will remove the complies relations that are element of \( D_{BoC}(t) \). After this complies relation is removed the repair script will check whether there are no missing services.

5.3. 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.

\[ D_{MIS}(t) = \{ \text{set of services that are needed by one or more applications on terminal t but that are not provided by the middleware of terminal t} \} \]

\[ D_{MIS}(t) = \{ \text{set of services that is required by the set of applications that is available on terminal t} \} \]

The repair script 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 \( S_b \).

\[ S_b = \{ \text{service blacklist} \} \]

\[ D_{BoS}(t) = \{ \text{service blacklist} \} \]

\[ D_{BoC}(t) = \{ \text{complies blacklist} \} \]

The repair script will remove the complies relations that are element of \( D_{BoC}(t) \). After this complies relation is removed the repair script will check whether there are no missing services.

The repair script 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 \( S_b \).

\[ S_b = \{ \text{service blacklist} \} \]

\[ D_{BoS}(t) = \{ \text{service blacklist} \} \]

\[ D_{BoC}(t) = \{ \text{complies blacklist} \} \]

The repair script will remove the complies relations that are element of \( D_{BoC}(t) \). After this complies relation is removed the repair script will check whether there are no missing services.

The repair script will remove the complies relations that are element of \( D_{BoC}(t) \). After this complies relation is removed the repair script will check whether there are no missing services.
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.

\[
P_c(t) = \left( \cup_{em : ml =< x, y, z >} \wedge (em \in y) \right) \cup \{s_1 \in S \mid (s_2, s_1) \in \{ \cup_{complies : ml =< x, y, z >} \wedge (complies \in z) \} \wedge s_2 \in \{ \cup_{em : ml =< x, y, z >} \wedge (em \in y) \} \}
\]

where \( t = \langle al, ml, pl \rangle \)

\[
S_r(t) = \{s \in S \mid s \in d \wedge \langle name, version, d, structure > \in al \}
\]

where \( t = \langle al, ml, pl \rangle \)

\[
D_{MS}(t) = S_r(t) \setminus P_c(t)
\]

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.

5.4. Memory Exhaustion

Using this diagnosis we check that there is no application on the terminal that consumes more resources that are provided by the terminal. The platform layer model provides the number of resources provided by the terminal. We predict the resource consumption of an application using the resource prediction technique described in [4], and [6]. In the remainder of this subsection we will discuss how this resource prediction technique can be combined with our approach for remote run-time diagnosis.

5.4.1. Per Operation Resource Specification

We predict resource usage based on resource models, behavior models, and a scenario specification. The resource consumption of \( o_{imp} \), an operation \( o \) implemented by service \( s \in S \) provided by an executable component model \( em \in EM \) is specified in a resource model \( rm \in RM \).

\[
r_{rc} = \langle sm, smr \rangle
\]

\[
(\langle rm \in sm \rangle \wedge (em \in sm) \wedge (\langle rm, em, \text{"rm of"} \rangle \in smr) \wedge (s \in em) \wedge (rc \in RC)
\]

\[
o_{imp} = \langle o, s \rangle
\]

5.4.2. Per Operation Behavior Specification

The behavior \((o_1; \ldots ; o_n)\) of an operation \( o \) of service \( s \) provided by executable component model \( em \) can be found in behavior model \( bm \), assuming that \( bm \) is the behavior model of \( bm \).
\[w(a, \{s_1, \ldots, s_n\}, r) = \begin{cases} w(a, \{s_i\}, r) \oplus (w(a_1, \{s_1, \ldots, s_n\}\{s_i\}, r) \oplus \ldots \oplus w(a_k, \{s_1, \ldots, s_n\}\{s_i\}, r)) \end{cases}\]

where

\((a \in i)\)
\((x, i) \in sp) \land (s_i = (sp, y))\)
\(< bm, em, "bm of" \rangle > smo\)
\(s_i \in em\)
\((a_1; \ldots; f_k) = bm(< s_i, o >)\)

In different cases we want to combine the resource usage of the operation executions in different ways. In order to be able to combine the resource usage of operation executions appropriately for each of the cases mentioned before, the resource estimation technique supports resource combiners for each case. The resource combinator \(\oplus\) is used to model resource usage of two sequentially executed operations. The combinator \(\otimes\) is used to model resource usage as part of an operation. The definitions of these combiners may be varied, in order to deal with different types of resources or to measure resource usage in different ways. In this example we consider memory consumption and therefore define the resource combiners as follows (see [6] for explanation):

\[(a, b) \oplus (c, d) = (a - b + c, d) \text{ if } c \geq b\]
\[(a, b) \otimes (c, d) = (a + c, b + d)\]

Now we combine the set of services defined by application \(a\) and weight function \(w\), to form the weight function \(W_{op}\) that calculates resource consumption that results from executing an operation. It is defined as:

\[W_{op}(a, o, i, r) = w(a, o, d, r)\]

where \(a = < x, y, d, z >\)

### 5.4.4. Modeling resource consumption for call sequences

Each operation call that is specified in a service behavior specification may be accompanied with an iterator (\(\ast\)) to indicate that the operation may be called multiple times. For the purpose of resource estimation, we represent an iterated operation call as a lambda expression:

\[o \ast = \lambda l \rightarrow l \times a\]

This lambda expression states that operation \(o\) is called \(l\) times, where \(l\) is variable. The weight function for an iterated operator call can now also be defined as a lambda expression:

\[w((\lambda l \rightarrow l \times a), d, r) = \lambda l \rightarrow l \times w(a, d, r)\]

Such lambda expressions are instantiated with concrete numbers, when resource consumption for a complete scenario is predicted.

### 5.4.5. Per Task Resource Estimation

As part of the resource prediction for a scenario we need to be able to predict resource consumption for a sequence of operation calls. We introduce the notion of a task, which is a sequence of operation calls. A task can be continuous or terminating, a continuous task corresponds to a repeating sequence of operation calls and a terminating task corresponds to a finite sequence of operation calls.

Using function \(W_{op}\) we are able to estimate resource consumption for individual operation calls, we need to predict resource consumption of a task. To that end, we need to know what plausible sequences of operation calls are. Such sequences (which we call task definitions), are determined together with implementors of services. If iterated operations are called in a task definition, the task definition becomes a lambda expression. Defining a task therefore consists of the following two steps:

1. Defining a plausible sequence of operation calls;
2. Instantiating concrete numbers for the lambda expressions, if any.

Instantiating a lambda expression fixes the number of operation calls. Thus, instantiating a lambda expression \(\lambda l \rightarrow l \times f\) with, say, 2 yields a sequence of two calls to \(f\).

We can now define the weight function \(W_t\) for a task, consisting of sequential operation calls. This function computes for each task, e.g. a sequence of operations, the required resources.

\[W_t(a, f, r) = W_{op}(a, o, i, r)\]

for operation call \(f = < i, o >\) and \(i \in I \land o \in O\)

\[W_t(a, g; h, r) = W_t(a, g, r) \oplus W_t(a, h, r)\]

for sequences of operation calls \(g\) and \(h\)

### 5.4.6. Scenario-based Estimation

In the Robocop component model, multiple executable components can work on different tasks that are executed in parallel. A scenario is defined as a set of tasks. This set contains the tasks that are executed in parallel, and for which the resource estimation needs to be made.
We can now define the weight function $W$ for a scenario $S$ and resource $r$, consisting of multiple tasks. This function computes for each task the estimated resource consumption:

$$W_s(a, t, r) = W_s(a, t, r)$$ \hspace{1cm} (58)

$$W_s(a, t_1, \ldots, t_n, r) = \otimes \ldots \otimes W_s(a, t_n, r)$$ \hspace{1cm} (59)

$$W(a, s, r) = TS(W_s(a, s, r))$$ \hspace{1cm} (60)

The function $W$ introduces the functions $W_s$ and $TS$ and a new resource combinator $\otimes$. This resource combinator is used to combine resources that are consumed in parallel. The resources can be combined only after we know what type of scheduling is used. Therefore the introduced resource combinator $\otimes$ is abstract. It is the task of the function $TS$ to do the scheduling and transform the expression in an expression which only contains concrete operators.

### 5.4.7. Applications that Exceed available Resources

The motivation for estimating the resource consumption for an application $a$ executing a scenario is that we want to be able to check whether the resource consumption of will exceed a certain limit. $D_{REA}(t)$ will calculate the set of applications $a$ on terminal $t$ for which the predicted memory consumption for a critical scenario ($W(a, scenario, memory)$) exceeds the memory of the terminal.

$$D_{REA}(t) = \{ a \in al \mid W(a, scenario, memory) > memory \}$$ \hspace{1cm} (61)

where

- scenario is a critical execution of application $a \in al$
- storage $= < memory, x, y >$
- $pl = (k, l, storage)$
- $t = < al, ml, pl >$
  - for arbitrary $k, l, x, y$

### 6. Concluding Remarks

#### 6.1. Discussion

Most systems must operate reliable and robust. Now systems are continuously evolving, even during deployment. This endangers robust and reliable operation of the system. In this paper we presented an approach for managing the integrity of the software on a device during deployment. Now our approach supports remote monitoring and remote diagnosis of a device. Remote repairing is future work. These mechanisms can be used to maintain software integrity and therefore to increase the robustness and reliability of a device.

Our approach has two major advantages. First of all the diagnosis and monitoring can be done remote as well as local. Combinations are also possible. For example some simple diagnosis is done on the terminal and the more complex diagnosis is done on a remote server. The mechanisms we use are light weight for the terminal concerning processing, network traffic as well as storage. Especially for the embedded domain this is a strong advantage.

There are also some open issues that require further research. Now a modest number of checks are available that can be used for diagnosis. We plan to extend this checks. Currently we are working on checks that verify whether the structure of applications adhere to certain design guidelines. Furthermore we observe that the current checks are based on relatively static information. State and configuration of specific instances is not addressed in the current checks. Finally, until now we did little work on repairing the diagnosed problems on a terminal.

### 6.2. Related work

Some work has been done on remote management and configuration of a device. SNMP [2] and SyncML DM [8] are existing device / network management technologies that are widely adopted by industry. These technologies focus on the mechanism needed for remote configuration and adaptation. We focus more on what needs to be adapted (diagnosis) and how (repair script).

Some work has been done on architecture based approaches for self-healing systems [9] [3]. These approaches focus on detecting when to make a particular repair based on architectural. 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 limited to use only the architectural description of a system.

### 6.3. Contributions

In this paper we presented an approach for integrity management. This approach allows monitoring, diagnosis and repairing of the software configuration of a device. Our approach is applicable for component based embedded systems and supports diagnosis and repairing based on architectural descriptions as well as on models describing the extra functional properties of system parts.
References


