Rule-based technical management for the dependable operation of networked Building Automation Systems

Malte Burkert, Joe Volmer and Heiko Krumm
Department of Computer Science
Technische Universität Dortmund
(malte.burkert, joe.volmer, heiko.krumm)@tu-dortmund.de

Christoph Fiehe
MATERNA Information & Communications
Dortmund
christoph.fiehe@materna.de

***Abstract*—The ongoing integration of various devices and services into nowadays buildings as part of networked building automation systems (BAS) demands for flexible and dynamically adaptable system architectures as they are supported by service orientation, which, moreover, can provide suitable abstractions when integrating new technologies and systems arising for example from the Internet of Things. The increasing complexity of such service systems, however, is accompanied by increasing dependability requirements. Therefore a comprehensive and automated technical management system including service lifecycle management, dynamic (re-)configuration, distributed (re-)deployment, self-repair etc. has to be accomplished. Furthermore the management tasks and the required abstraction of the managed system should span a defined state space in order to enable formal verification of system characteristics like dependability. This work reports on a technical management system utilizing general and system-specific event-condition-action-rules (ECA rules) to enable the dependable operation of BAS. The management tree does not only serve as homogeneous management interface to the BAS, but spans in conjunction with the ECA rules the desired state space.**

I. Introduction

New technologies and systems arising from the fields of Cyber Physical Systems and the Internet of Things require an abstraction when cooperation with building automation systems (BAS) is desired. E.g. heating, ventilation and air conditioning systems can be supported by local weather forecasts delivered by weather stations over the internet or mobile devices can provide additional information on building visitors. It can obviously be noted, that all participating systems deliver some sort of service and communicate by certain standards or at least provide a common interface and representation of exchanged data. This challenge is often solved by service oriented architectures (SOA) introducing a suited abstraction of all devices and systems in the form of services. The heterogeneity and joint operation of services stemming from different domains in BAS lead to new arising challenges. Building automation hardware which is vendor specific and strongly structured needs to be integrated into the SOA based landscape. The hardware and BAS-specific components partly require context-aware management, have a different demand for dependability and might be state-full, which contradicts to the concept of SOA.

Our approach to dependable BAS relies on the utilization of the management tree [1] as a hierarchical abstraction of all participating components of BAS (hardware and software) and a rule engine executing event-condition-action-rules (ECA rules) as reactions on system changes. The management tree propagates events to the rule engine, which reflect changes of BAS components (e.g. installation of a controlled shutter or outage of a computing resource for automation services). The reconfiguration and adaption of the BAS is realized by modifying nodes and values of the management tree, so that a defined state space is given by this abstraction of BAS infrastructures. On the one hand this management structure provides the functional capabilities for the comprehensive management of BAS and on the other hand it enables by means of the defined state space a formal verification of system characteristics like dependability.

II. Related Work

BAS are also known as Building Management Systems (BMS) [2] or Building Automation and Control Systems (BACS) [3]. The common issue is the separation and diversification of contributing heterogeneous system components within nowadays buildings. Our approach tries to overcome the heterogeniety of different technologies by abstraction and providing an executable unified management interface to the integrated systems.

Zhang et al. report in [4] on an OSGi platform (Open Services Gateway initiative) and agent based control system architecture for smart home. As autonomic computing in smart homes has to cope with similar issues as heterogeneous BAS service landscapes, they propose a three-tier system architecture in order to handle different devices and services. The management agents are system specific and are installed on home gateways as OSGi bundles. The so called gateway operator handles all management capabilities through the management agents. The presented approach lacks a simple solution for verifying the correctness of the management actions. The state space is not defined in terms of any abstraction of the individual systems, which impede formal verification of system characteristics.

The authors in [5] present the policy management for autonomic computing, a generic policy middleware platform which is a representative of policy-based management systems introduced by Sloman in [6]. Our presented management system follows the same paradigm. The ECA rules are basically executable policies defined on the suited runtime abstraction of BAS. Furthermore the rule lifecycle and executions can be

978-3-901882-89-0 @2017 IFIP

observed by the management system and maintenance personal in order to assess the effects and correctness of the desired objectives.

Dohndorf et al. [7] and Dai et al. [8] describe the concept of policy-based management and the refinement of an abstract system description divided into the layers use cases & aspects, services & domains and components & devices. They introduce the concept of different patterns in order to control the refinement process, which is executed as part of the building automation lifecycle in our approach. Refinement, Evaluation and Control Patterns are introduced as mean to relate abstract model elements with elements on the next layer of abstraction, introduction of abstract status variables aggregating concrete values and derivation of lower-level policies from higher-level ones. This concept enables the utilization of runtime models for the semi-automatic generation of runtime configuration without introducing additional artifacts to the system lifecycle. We adopt this approach in order to automatically generate the configurations for our management rules.

III. Self-managed Technical Management System

The technical management (TM) system of a BAS must be able to cope with the speed at which building and room usage constraints change. Therefore, it must be equipped with self-management capabilities in order to adapt its behavior to changing conditions and requirements as well as to react on system intrusions or faults. It must configure and reconfigure itself, continually tune and optimize itself, protect and recover itself and keep its complexity hidden from users and administrators.

Our approach comprises a hierarchical object model implemented by a distributed **Management Tree** that covers protocol-specific parameters for management data acquisition through specific handler implementations realized in the form of self-contained software components. All data that needs to be visible within the management scope must be provided by management variables, in the form of two different kinds of variables:

- **Configuration variables**: these variables are solely written by the management system in order to adapt BAS behavior to changing conditions and requirements.
- **Status variables**: these variables are written by the system and represent the current system state.

The management tree consists of *interior* nodes and *leaf* nodes. Interior nodes can have children whereas leaf nodes have primitive values. Base value types are: Boolean, Short, Integer, Long, Float, Double, String, Binary and Object. Nodes can have an *Access Control List* (ACL), associating operations allowed on those nodes with a particular principal represented as a String value. Furthermore, nodes can have meta data describing the actual nodes and their siblings. Leaf nodes may have default values specified in their meta data. Meta nodes define allowed access operations (Get, Add, Replace, Delete and Execute).

The whole monitoring and control logic is implemented in modular **ECA rules** composed by *events*, *conditions* and *actions*. These rules represent high-level objectives and constraints imposed by the building operator, tenants, users and system administrators. At runtime, rules are present in the form of Java bytecode that can be executed efficiently. They form small reusable code building blocks, which are loosely coupled in a way that allows interaction without the need for direct dependencies. To achieve this behavior, rules' scope is limited by convention to only the information that are available through the Management Tree. Rule interaction takes place on the basis of this shared data structure. Each structural or value assignment change within the management variables triggers an event publication.

A **rule manager** has been implemented that is dynamically equipped with rules that carry out the actual BAS management process. The rule manager is connected to the Management Tree and controls the rule lifecycle. A rule comprises at least one path pattern and the actual rule implementation as a Java class. Optionally, a rule can specify an event mask that acts as a filter expression that allows the rule manager to filter out all those events of a specify type a rule is not interested in. A rule instance is created immediately after deployment and gets registered in the OSGi service registry. This approach complies with OSGi best practices by making use of the recommended whiteboard pattern for service interaction. The rule transitions form the *initial* state to the *resolving* state indicating that it is now under control of a rule manager. Within this state the rule manager tries to find at least one matching path in the management tree to which the rule gets attached. When the rule specifies multiple path patterns, rule resolution requires at least one existing path for each pattern. The rule transitions to the *active* state when path pattern resolution was successful. Rule activation is lazy and can be delayed when at its time of deployment no matching paths exist. Structural changes may lead to its activation and deactivation in the course of time. Whenever an event is published describing a change within a subtree, the rule manager dispatches the event to all active rules that are attached to this subtree or to any other containing subtree.

The Management Tree is equipped with a **rule monitoring handler** providing a subtree that makes rule resolution and rule execution itself monitorable. For each rule instance there is a dedicated subtree containing status variables describing its state, a timestamp when the rule was deployed, a timestamp when the rule was activated and information about its last execution. The execution subtree comprises a timestamp when the execution has started, a timestamp when the execution has finished, whether execution was successful or not and a sequence of all those Management Tree methods with start time, end time, input and output parameters that were called during rule execution. Technically, this is achieved by creating a monitorable dynamic proxy of the Management Tree service interface that gets injected in the rule instance. It is capable of documenting every method call that was made on this service. The execution subtree is built up incrementally. Whenever there is monitoring information available they get provided immediately in the corresponding rule instance subtree. This approach allows us to react with minimum delay when rule execution violates time constraints or when interim results do not fulfill dedicated requirements.

IV. Rule based technical management for distributed service landscapes

ECA rules can be divided into *general* and *system-specific* ones. General ECA rules realize generic management objectives, which do not need an individual configuration. E.g. the dependency resolving rule (DR rule) presented in [9] is a representative of general ECA rules. Its management objective

is to monitor the runtime platform and detect unresolved OSGi bundles in order to resolve them by autonomously installing all required packages. As can be noticed this rule does not need further information as all required data can be explored from the managed system and the participating management components like the artifact repository.

The service redundancy rule (SR rule) (also described in [9]) is a system-specific rule as it requires information on the specific services, which are expected to be distributed redundantly. The SR rule factory (an implementation of the OSGi Service Factory pattern) is leveraged to provide one SR rule instance per SR rule configuration. The SR rule requires for instance information on the service identifier of the service to be installed multiple times, the number of desired instances and eventual service properties.

A. Service Dependency and Binding Rule

The distribution of services across multiple computing nodes requires an appropriate solution for the dynamic remote service binding and resolution of service dependencies. This does not only mean to initially set up the service distribution and bindings, but also the dynamic adaptation to system changes and failures, so that a dependable operation of the service landscape can be achieved. The notion of the service dependency and binding rule (SDB rule) is the autonomous installation of depending services, configuration of endpoints and related proxies, so that distributed services depending on each other can transparently call each other without even caring about the physical service location.

The SDB rule is associated with a path pattern with the management tree path, which represents the bundle and service states in the whole runtime system on all configured computing nodes as well as the state of the computing nodes itself. Due to the service lifecycle being part of the bundle lifecycle the state change of bundles has to be observed in order to start the evaluation of the SDB rule. Furthermore failures of computing nodes have obviously to be handled too.

Figure 1 depicts the activity diagram of the evaluation of the SDB rule. Since designed as ECA rule, the SDB rule is defined trough event(s), condition(s) and action(s). The events triggering the evaluation of the SDB rule are in conformance with figure 1:

- An OSGi **bundle**'s state changes to **resolved**: as soon as a bundle changes its state to resolved the lifecycle of potentially included OSGi service components start. Thus possibly required (remote) service dependencies have to be resolved and respective bindings have to be set up.
- A **computing node** is **gone**: being the fundamental infrastructure for the runtime platform of all services, the failure, outage or removal of a computing node could possibly lead to unfulfilled service dependencies due to remote services bound to remote endpoints hosted on the missing node. Thus services might have to be distributed and deployed differently and the corresponding remote service proxies have to be re-configured accordingly.
- Not as bad as the outage of a computing node is the **disappearance of a service**. The SDB rule's objective is to assure the continuous operation of all eventually depending services. Thus substitutions for depending services have to be found or newly installed and the services have to be re-bound (eventually remotely by remote service proxies) in order to provide an ongoing operation.

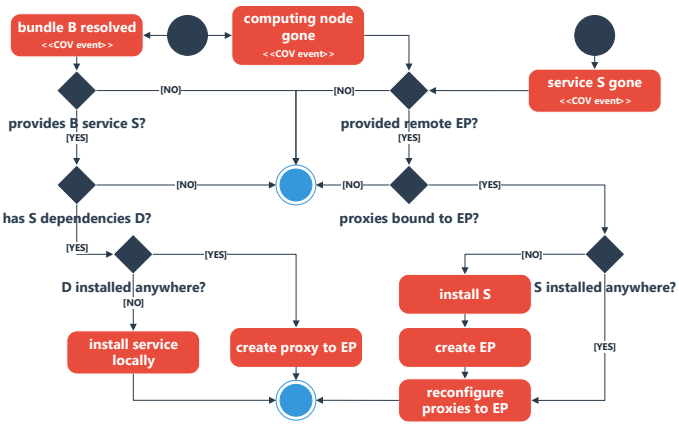

Fig. 1. Activity diagram of the service dependency and binding rule evaluation

The conditions being evaluated before the rule actions are executed depend on the observed system change.

Event: Bundle State Change

If the event comes from the bundle state change, the evaluated conditions are:

- Does the resolved bundle provides a service S?
- Is the service S depending on other services D?

If both conditions evaluate to true, the rule has to distinguish if the depending services D are already installed in the runtime environment or have to be installed in order to satisfy the dependencies of S. If the dependencies are already available missing remote service proxies are created and pointed to the respective endpoints, otherwise missing dependencies are tried to be deployed on the same computing node where the bundle has been installed on in order to prevent the unnecessary introduction of remote service calls. The information of the endpoint description for the proxy creation is taken from the management tree, which provides among others the service properties of all available services. The endpoint properties like port and service interface are for instance part of the service properties.

Event: Computing node or service gone

In the case of a gone computing node or service, the conditions are:

- Has the computing node provided remote service endpoints for hosted services? / Has a remote service endpoint been provided for the gone service?
- Have proxies been bound to the former remote service endpoints?

If both conditions are positively evaluated, the rule either re-configures the proxies, which have been bound to the old remote service endpoint to an equivalent endpoint, or: a new instance/ new instances of the required service(s) is/are installed, remote service endpoint(s) has/have to be created and the existing proxies have to be re-configured according to the newly created endpoints.

This rule enables the location transparent operation of BAS

services, respects the dynamics of BAS and is aware of failures, thus supports a dependable distributed operation.

B. Resource aware Service Re-distribution Rule

The resource aware service re-distribution rule (RSR rule) enables the service specific resource monitoring and automatic re-distribution of the respective service a priori a possible failure or outage of the service may occur due to a lack of computing resources. This categorizes the RSR rule as a system specific ECA management rule requiring external configuration from the runtime model. Thus a rule instance is instantiated per configuration by the usage of the RSR rule factory, which is implemented according to the OSGi service factory pattern. The configuration of the RSR rule is as follows:

- **Service Identifier** [mandatory]: the service identifier of the service type of which the underlying computing nodes should be monitored
- **Service Properties** [optional]: only services with the provided service properties are handled by the rule
- **CPU-threshold** [mandatory]: the relative CPU load which should not be exceeded
- **MEM-threshold** [mandatory]: the relative or absolute free memory value which the current value should not fall below

Figure 2 shows the activity diagram of the RSR rule evaluation. According to the ECA paradigm, the rule consists of events, conditions and actions in order to realize the corresponding management tasks. The triggering events are as follows:

- The CPU load of the computing node the service is operated on exceeds a specific threshold.
- The memory consumption of the service related computing node exceeds a defined threshold.

Unlike the conditions of the SDB rule, the conditions of the RSR rule do not depend on the triggered events. As soon as any resource threshold is exceeded the rule checks whether the value is continuously too high or is only assessed as a peak value. If that is the case the rule explores the resource load of all other available computing nodes. If all computing nodes are too busy a management reporting event is generated asking for manual system adaption of a system maintainer. If a computing node with sufficient resources is available, the service is additionally installed on this computing node, the service is configured according to the original one and the required resource monitoring for the service is set up. The original service is removed after the new service is successfully installed.

V. Conclusion

The applicability of the presented concept has been determined on our physical small scale model of a BAS, which is equipped with real building automation hardware and custom small electronic components for the actors and sensors (partly controlled by MCUs, but automated by DDCs) [10].

The presented work reports on a solution for the required distributed operation of service oriented architectures in the cooperative operation of BAS and the internet of things. The presented technical management system supports the management of heterogeneous service landscapes like BAS and furthermore

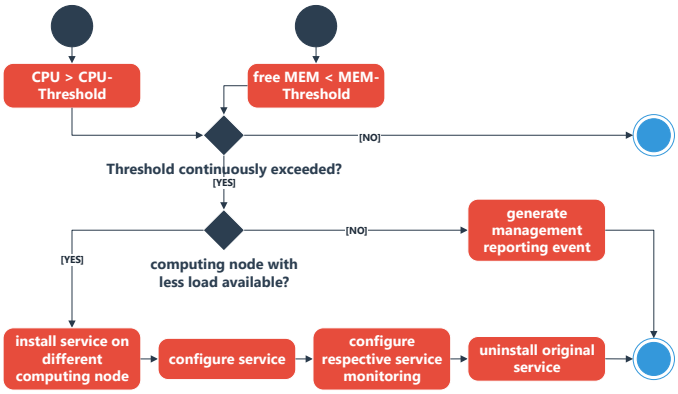

Fig. 2. Activity diagram of the resource aware service re-distribution rule evaluation

enables formal verification of different characteristics. The formal verification of dependability has not been investigated yet, but is part of forthcoming work.

We have exemplarily presented two management rules supporting dependable operation of large and complex BAS: service dependency and binding rule and resource aware service re-distribution rule. The objectives and execution of the rules have been presented as well as the realization through the rules of the automatic service lifecycle management, (remote) service binding and the resource aware monitoring and (re-)distribution of services.

This work has been funded by the German Federal Ministry of Education and Research (BMBF) and is a result of the investigations under reference number 01IS13019 in the European ITEA research project *Building as a Service* (BaaS).

References

[1] M. Burkert, H. Krumm, and C. Fiehe, "Technical management system for dependable Building Automation Systems," in *ETFA, 2015 IEEE 20th Conf. on*, Sept 2015, pp. 1–8.

[2] S. Wang and J. Xie, "Integrating building management system and facilities management on the internet," *Automation in construction*, vol. 11, no. 6, pp. 707–715, 2002.

[3] C. Aghemo, L. Blaso, and A. Pellegrino, "Building automation and control systems: A case study to evaluate the energy and environmental performances of a lighting control system in offices," *Automation in Construction*, vol. 43, pp. 10–22, 2014.

[4] H. Zhang, F.-Y. Wang, and Y. Ai, "An osgi and agent based control system architecture for smart home," in *Proc. ICNSC*, March 2005.

[5] D. Agrawal, K.-W. Lee, and J. Lobo, "Policy-based management of networked computing systems," *IEEE Communications Magazine*, vol. 43, no. 10, pp. 69–75, Oct 2005.

[6] M. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, no. 4, pp. 333–360, 1994. [Online]. Available: http://dx.doi.org/10.1007/BF02283186

[7] O. Dohndorf et al., "Tool-supported refinement of high-level requirements and constraints into low-level policies," in *POLICY, 2011 IEEE International Symposium on*, June 2011, pp. 97–104.

[8] N. Dai et al., "Osami commons - an open dynamic services platform for ambient intelligence," in *ETFA, 2011 IEEE 16th Conf. on*, Sept 2011.

[9] M. Burkert and H. Krumm, "Dependency Management in Component-Based Building Automation Systems," to appear in DASC, 2016 IEEE 14th International Conf. on, 2016.

[10] M. Burkert, J. Esdohr, and H. Krumm, "A Small-Scale Model House Evaluation platform for Building Automation Systems," in *ETFA, 2016 IEEE 21th Conf. on*, Sept 2016, pp. 1–8.