Situation-Aware Updates for Cyber-Physical Systems

Kálmán Képes, Frank Leymann, and Michael Zimmermann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
{kepes, leymann, zimmermann}@iaas.uni-stuttgart.de

Abstract. The growing trend of integrating software processes and the physical world created new paradigms, such as, the Internet of Things and Cyber-Physical Systems. However, as applications of these paradigms are rolled out, the management of their software and hardware components becomes a challenge, e.g., an update of a mobile system must not only consider the software but also the surrounding context, such as, internet connection, current position and speed, or users activities, and therefore must be aware of their current context and situations when executed. The timing of executing management tasks is crucial as the surrounding state of the system and its context must hold long enough to enable robust execution of management tasks. We propose an approach to model software updates in a Situation-Aware manner, enabling to observe context, such as, time, environment, application state and people, hence, enabling to execute update tasks at the right time and in the right time-frame. We implemented our approach based on a train scenario within the OpenTOSCA Ecosystem.

Keywords: Cyber-Physical Systems · Context-Aware Systems · TOSCA.

1 Introduction

The trend of integrating the physical world with the so-called cyber world builds upon integration of software and hardware components creating new paradigms, such as the Internet of Things (IoT) [2] and the more general paradigm of Cyber-Physical Systems (CPS) [17]. The wide range of applicable domains, such as health care [19], mobility [18] and energy [1], enable the seamless integration of digital and physical processes with our everyday life. These systems must combine different types of components, starting from commodity hardware to embedded devices, from off-the-shelve software components to specialized real-time functions, e.g., embedded devices that control motors sending data to local databases hosted on edge cloud resources, which are used to aggregate the data before sending it further to the cloud. Maintenance of these heterogeneous components is a complex task, e.g., the exchange of complete components at runtime to achieve a system update. As every system can have its bugs it is crucial to enable updates in CPS, as errors that lead to faults within such a system can have severe safety issues in the real world possibly harming people or resources.

However, management processes that use or configure components must also consider the surrounding environment, i.e., its overall context must be regarded. For example, applying runtime updates of components effect their functional availability and may create severe safety issues in case of errors that can be caused by external environmental changes, such as, changing network connections or actions by people affecting the update process. Additionally, if a CPS should be as autonomous as possible the execution of updates should be started without the explicit affirmation by users every time an update is available. This autonomy can only be achieved if the management system of a CPS is aware of its context, and further, to enable a robust execution of updates, it is important to observe the current state in the context and apply these at the right time. Hence, such systems must be Context-Aware in nature [32]. To enable such an execution it is not only important to know how long a situation will be active in the application context, e.g., are all users absent from the CPS and don't use it, but also how long it takes to execute an update, e.g., how long applying a new firmware takes or how long it takes to restart a database. In summary, it is important to combine both views, how much time does an update have and how long does it take to enable the robust execution of an update.

Therefore, we propose an approach to enable the modeling and execution of management processes based on awareness of the context and worst-case execution time of the needed management tasks, enabling to determine when a system has a proper state and enough time to execute management processes. The main idea of our approach is to model the desired update with addition of specifying which components must be updated only when certain situations are active for enough time to update. Our approach builds on previous work within the DiStOpt project (successor of SitOPT) [23] that enabled to model Situation-Aware workflows, but did not consider the timing of Situation-Aware tasks, and therefore, was only reactive in nature, hence may turn applications not to be available at the right time again. One of the project goals of DiStOpt is decoupling high level knowledge from low level data and queries on top of the received data, and therefore, enable easier modeling of Situation-Aware workflows. The concept of Situation-Aware workflows is used in this paper and applied to updating CPS in a situation-aware manner to enable desired properties of safe and autonomous execution of management processes. Another work we build upon is to enable the generation of workflows from so-called declarative deployment models that only specify what should be deployed and not how. The generated processes from these models are able to install, configure and start software components in an fully automated manner [7], but not under given time constraints such as those given by the current context of a CPS. We extended and combined concepts of both previous works enabling the observation of time in context and execution, therefore, enabling proactive management of CPS.

This paper is structured as follows: In Section 2 we describe a motivating scenario and background information, in Section 3 we describe our approach, in Section 4 we describe our prototypical implementation and related work in Section 5, followed by a conclusion and future work in Section 6.

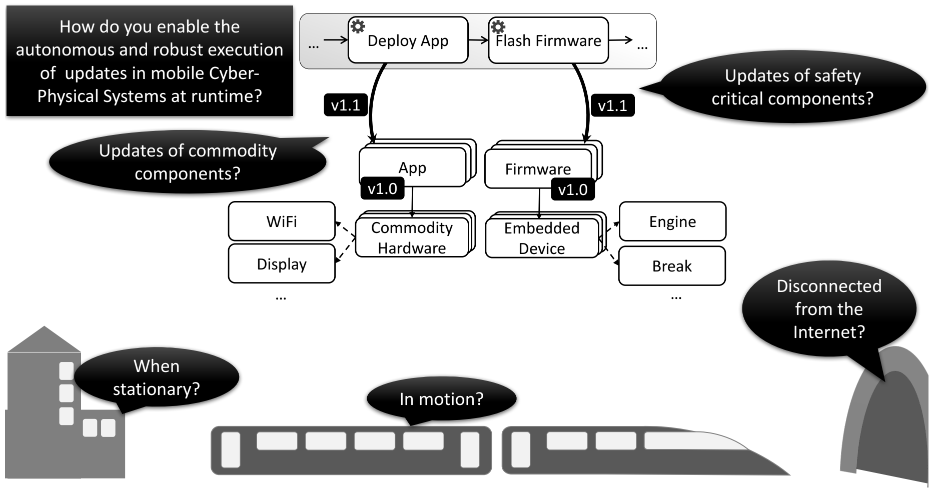
How do you enable the autonomous and robust execution of updates in mobile Cyber-Physical Systems at runtime?
Deploy App
Flash Firmware
v1.1
v1.1
Updates of safety critical components?
Updates of commodity components?
App
v1.0
Firmware
v1.0
WiFi
Display
Commodity Hardware
Embedded Device
Engine
Break
Disconnected from the Internet?
When stationary?
In motion?

Fig. 1. Motivating example for Timing and Situation-Aware execution of updates.

2 Motivation & Background

In this section we describe a motivating scenario based on autonomous trains (see Subsection 2.1) and give brief background information on CPS (see Subsection 2.2), worst-case execution time (see Subsection 2.3) and timed situations based on the DiStOPT project, successor of the SitOPT project (see Subsection 2.4).

2.1 Motivating Scenario

Think about a mobile CPS such as trains consisting of heterogeneous components (see middle in Figure 1), e.g., embedded devices that read sensor data and control actuators to enable safety-critical functions (see firmware hosted on embedded device in the middle of Figure 1). On the other side, software for passengers, such as, multimedia services are running on commodity hardware (see app hosted on commodity hardware in the middle of Figure 1). Each of the component types have different requirements when regarding updates. Updating such commodity components doesn't affect the availability of safety-relevant functions and updating these components could be done at any time when the needed update files are locally available. However, updating all of these components while passengers may want to use them, the availability of those services can be reduced and therefore reduce customer satisfaction. This issue is much more severe when regarding availability of safety-critical components running on embedded devices of the autonomous train. A failure when updating these components, such as a firmware that enables control of brakes and motors, has severe impact on the safety of CPS systems. To enable the automated and robust execution of runtime updates in such a scenario two challenges arise: (i) when do you start and (ii) how much time is left to execute updates before the system has to be available again. For example, in our scenario updating the end-user components should be achieved when those are not needed, i.e., are not actively used by passengers. In case of safety-critical components, the train should be in state where it is safe to execute the update, such as when the autonomous train has stopped at a station, i.e., the train is not in motion. Therefore the update system must know when components are not used and know how much time is left before the components are in use again. Depending on the components to be updated it is not sufficient to only determine the state of the software and hardware components to find a suitable time for update execution. As it is done in research for dynamic software updating [26] it is observing the so-called quiescence of components and executing updates only when the necessary components are not in use. In this field of research the quiescence of components was only observed in the context of applications itself without regarding the physical environment. To determine the perfect time to start an update the overall environment of the system must be observed as well, e.g., whether the train is stopped at a train station or somewhere outside on the tracks, or whether passengers are connected to the wireless network of a train wagon (see lower part of Figure 1). In our train scenario the execution of software updates would be most sufficient when the train is at halt at a station and passengers are disconnected from the network components. Enabling a robust execution of an update in our scenario of autonomous trains means that updates should not disrupt the availability of the system when it is actually needed and be started without the need of human interaction. An update should be execute in a way that it is run at a time slot where the update can take place completely and successfully, e.g., when an update takes five minutes and the train is at a train station for at least the same time, it is safe to update.

2.2 CPS

In research and industry the trend of integrating the physical world and software emerged to paradigms such as the Internet of Things [2] or the more general paradigm of CPS[17]. The wide-range of application scenarios for CPS, such as, energy [34], health care [19] and mobility [18] implies the growing integration within our everyday lives. These applications build on heterogeneous hardware and software components such as embedded devices to commodity hardware or from off-the-shelve components to specialized software. As these components may be deployed and run within physical environments and interacting with users in a physical manner, CPS in cases can be seen as real-time systems executing (near-)real-time tasks that have deadlines on the execution of their functions to enable properties such as safety. These properties are crucial for CPS as missing deadlines may put people into danger, e.g., when an airbag of a car reacts to late. Therefore it is crucial that CPS are highly resilient to errors and do not create severe faults, hence, it is crucial to immediately apply (critical) updates of the heterogeneous CPS components to make the system more resilient over their lifetime. This also must be regarded when executing management tasks such as updates, as the execution of as such may have severe side effects as well.

2.3 Tasks and their worst-case execution time

The timing of the update operations does not only depend on the duration a situation is active, but, the execution time of the operations as well, especially their worst-case execution time (WCET) [40]. This enables our approach to proactively decide whether to execute operations based on the currently active situations. However, determining the WCET of an operation is not trivial and in general not possible to calculate an exact value in modern systems, i.e., it is only possible to get approximations of the WCET [40]. But especially, in the case of CPS calculating the WCET is crucial, e.g., airbag control in cars or emergency braking in trains. The precision of a so-called WCET upper bound that specifies how long an operation will take at most depends on the underlying hardware. The closer the software is to the hardware the more precise a WCET bound can be calculated, which can be done by so-called static analysis or measurement-based approaches. While the static approaches calculate a WCET bound based on the source code and target hardware, the measurement-based approaches calculates upper bounds on previous executions of operations [40]. This problem of calculating upper WCET bounds gets more complicated in modern Cloud or Edge Computing scenarios, as the underlying hardware is visualized and heterogeneous, further complicating the problem getting proper WCET values. In our example, the most sensitive part to update is the firmware on the embedded device, as flashing the system disables safety-critical functionality and must not be aborted while executing. However, as firmware is the first software layer on top of embedded devices the WCET of flashing can be practically determined. Hence, the WCET of tasks that are sensitive and close to the hardware is easier to determine, and therefore, usable within our approach.

2.4 Timed Situations

The overall goal of our approach is to execute activities of a process at the right time, e.g., in our motivating scenario updates on safety-critical components should only be executed when the train is at a train station, while other software components may only be updated when passengers don't use them. Another important aspect is that the execution should be aborted as less as possible, in case of hardware, such as ECUs in trains or cars, retries of the flashing process reduces lifetime and may even make the hardware unusable when the flashing process fails [20]. Therefore, we extends our previous approaches on modeling situations within the SitOPT project [39][8], where a situation was defined as a state in the context of an object. To observe situations, context data from objects, such as, machines, software components or users are processed to calculate a particular situations state. We extended situations to give a guarantee on how long they are active, enabling to use the timing-aspect of situations directly while executing situation-aware tasks. These specify at runtime for how long a situation will be either active or inactive, hence, enable to proactively manage the execution of tasks. For example, if we know that the necessary situations are available for enough time, e.g., the train is stopped at a station and passengers don't use specific components for at least a minute, we can execute an update.

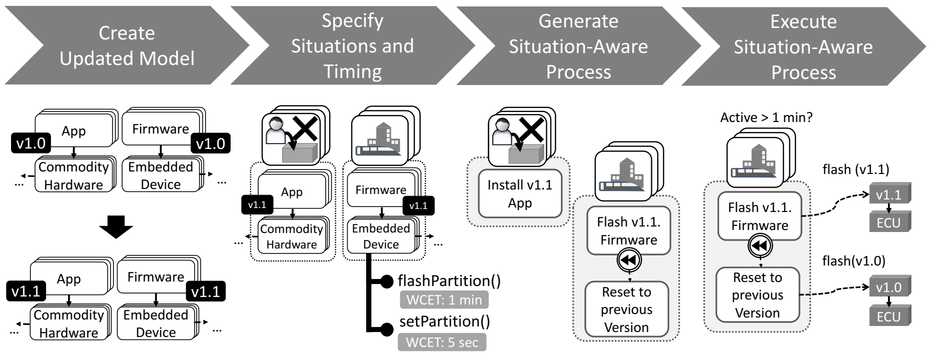

Fig. 2. Overview of the UPTIME method.

3 UPTIME - Update Management for Things in Motion and Environments

In this section we describe our concept to enable the situation-aware modeling and execution of software updates, based on our method called UPTIME (see Figure 2) which is based on a so-called *Situation-Aware Deployment Metamodel* and *Time-Aware Situational Scopes* (see Subsection 3.2). The first is used to model an update while the second is used to for situation-aware execution.

3.1 Overview

The main idea of the UPTIME method is to specify components and their exposed operations with the situations that must be active long enough in the current context of the application to execute such operations properly. The first step in the UPTIME method is to model the desired update from a source to a target deployment model, i.e., model the components and relations which shall be installed and configured in the running systems via deployment models (see first step in Figure 2). In the second step the target deployment model is annotated with situations (see Subsection 2.4) and the worst-case execution time (WCET, see Subsection 2.3) of the available operations and their compensation operations. For example, in our scenario an operation to flash embedded devices, and also to compensate previously executed flash operations, is annotated with a WCET of one minute and the situation that a train shall be at a station, indicating that the update of a firmware should only be executed when the train is at a station for at least a minute at runtime (see on the right side in Figure 2). In case of components used by passengers, we only specify the situation that no passenger is using these components. Both steps are based on the Situation-Aware Deployment Metamodel described in Subsection 3.2 in detail. After specifying situations and execution times, in step three we generate an imperative deployment model, i.e., a process from the given source and target deployment models that contains activities using the operations of modeled components in an order

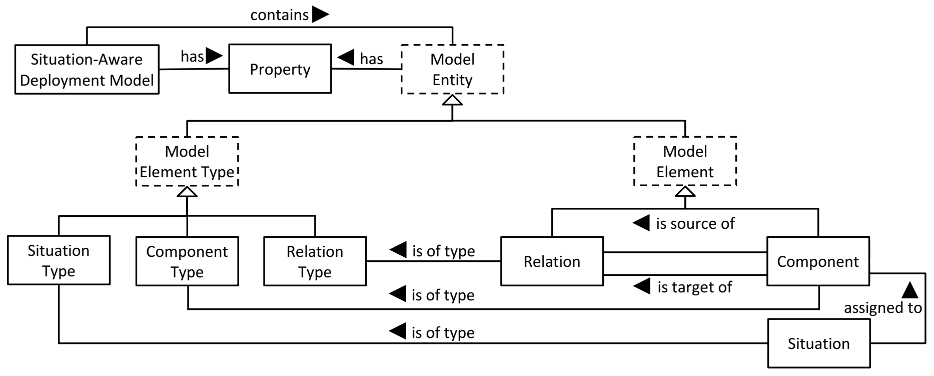

Fig. 3. Metamodel of Situation-Aware Deployment Models (SADM).

to update the system (see Subsection 3.3). The order of generated processes are derived by determining which components must be removed, updated or newly installed, and using the available management operations to achieve this. In addition, each annotated operation from the original declarative models which are needed within the generated process is added to a so-called Time-Aware Situational Scope. These scopes control the execution of these operations by observing the specified situations and therefore can calculate whether there is enough time to execute an operation according to its annotated WCET. The execution of operations in such scopes is only started when the proper situations are active for a duration that is enough to execute management tasks and possibly their compensation, i.e., revert already executed operations. This strategy enables to keep a system in a consistent state even if the execution of operations fail as a scope will start to compensate already executed update operations in time.

3.2 Situation-Aware Deployment and Process Metamodels

The Situation-Aware Deployment Metamodel (SADMM) enables to model the deployment of an application by specifying components, relations and situations . In our method the current state and updated state of an applications' configuration is modeled within two models of a SADMM. Figure 3 gives an overview of the metamodel with our extension for situations, which will be presented in the following. An instance of a SADMM, a SADM, is a directed and weighted multigraph that represents structural configuration of an application under a set of situations, i.e., it represents components and relations that can be started and configured at runtime, however, only when the specified situations are active their exposed operations can be used. Let $\mathcal{M}$ be the set of all SADMs, then $d \in \mathcal{M}$ is defined as a tuple:

$d = (C, R, S, CT, RT, ST, type, operation, sits, active)$

The elements of the tuple d are defined as follows:

- C: The set of components in d. Each $c_i \in C$ represents an application component that should be configured and started.
- $R \subseteq C \times C$: The set of relations, whereby each $r_i = (c_i, c_j) \in R$ represents a relationship between two components: c_i is the source and c_j the target.
- S: The set of situations in d, whereby each $s_i \in S$ specifies a certain situation in the context of an application.
- CT: The set of component types whereby each $ct_i \in CT$ describes the semantics of components, e.g., it specifies that a component is firmware.
- RT: The set of relation types whereby each $rt_i \in RT$ describes the semantics of relations, e.g., it specifies whether a relation is used to model that a component is hosted on or connected to another.
- ST: The set of situation types whereby each $st_i \in ST$ describes the semantics of situations, e.g., it specifies that a situation is used to determine whether a train is moving.
- *type*: The mapping function, which assigns the component type, relation type and situation type to each component, relation and situation.
- *operation*: The mapping function, which assign operations to components via their type, therefore, the mapping *operation* maps each $ct \in CT$ to a set of tuples $(id, wcet)$ where $id \in \Sigma^+$, with Σ being an alphabet and Σ^+ denoting all strings over Σ, and $wcet \in \mathbb{R}_{>0}$, specifying the available operations and their WCET.
- *sits*: The mapping function, which assigns each component in d its set of situations under which it is valid to be deployed. Therefore, *sits* maps to each $c_i \in C$ a set $s_i \subseteq S \cup \epsilon$ where ϵ denotes that any situation is feasible, allowing that a component is always valid to be deployed.
- $active : S \times \mathbb{R}_{>0} \rightarrow \{true, false\}$, where $active(s,t) = true$ if and only if the situation s is active for the duration t.

To achieve the adaptation needed within our method, i.e., update from the current configuration to target configuration, different methods can be applied to determine an abstract process of needed activities to be used in executable languages. In the following we will define a meta model for such processes and so-called Time-Aware Situational Scopes (TASS).

Let a process G with a set of TASS be a directed and labeled graph represented by the tuple $G = (A, L, c, scopes)$.

- A is the set of activities. We separate A into $A_B \cup A_C = A$ where A_B is the set of activities implementing the desired deployment logic and A_C is the set of compensation activities.
- L defines the order of the control flow of process G and a control connector $l \in L$ is defined as a pair (s,t) where $s, t \in A_B$ specifies to execute the activity s before t.
- c is the mapping of activities to their compensation activities via $c : A_B \rightarrow A_C \cup \{NOP\}$, allowing activities to not have a compensation activity by using NOP.
- *scopes* is the set of TASS in G.

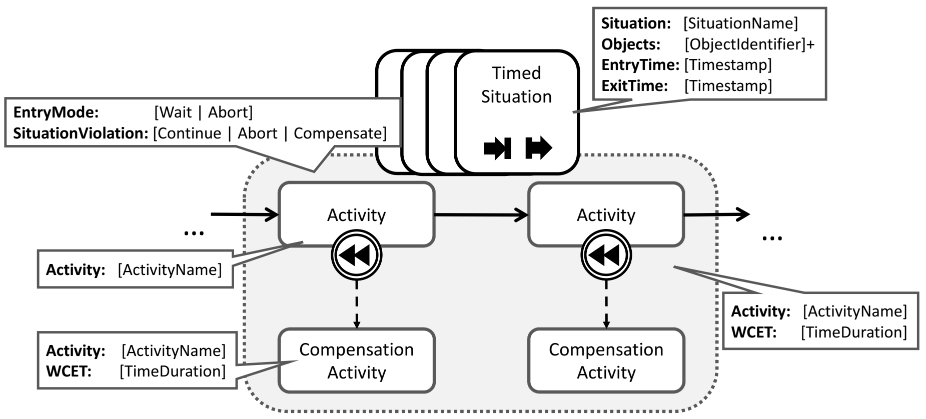

Fig. 4. Time-Aware Situational Scope.

We define a TASS $scope \in scopes$ to be an acyclic subgraph of G defined as $scope = (A_{scope}, L_{scope}, S, situations, active, guarantee, wcet_{duration})$ with the elements defined as the following:

- A_{scope}, L_{scope} is the subgraph of G spanned by A_{scope}
- S is the set of situations, whereby each $s_i \in S$ specifies a certain situation in the context of an application.
- $situations : A \rightarrow \mathcal{P}(S)$ is the mapping from A to S to indicate that activities may only be executed when all annotated situations are active.
- *active* is the mapping $active : S \times \mathbb{R}_{>0} \rightarrow \{true, false\}$, where $active(s, d) = true$ if and only if the situation s is active for the duration d, analogous to those in a SADM.
- *guarantee* is the function to determine the minimal duration of a scope to be safely executed. We define it as $guarantee(A_{scope}) \quad := \quad min(\{d | a \in A_{scope} \wedge s \in situations(a) \wedge active(s, d) = true\})$, i.e., we search for the minimal duration all relevant situations are active.
- $wcet_{duration}$ is the mapping of activities to their WCET with $wcet_{duration} : A \rightarrow \mathbb{R}_{\geq 0} \cup \{\perp\}$ denoting the duration it takes for an activity to finish with its execution, or $wcet_{duration}(a) = \perp$ means that it has no duration defined.

In addition, a TASS must hold that $\forall a \in A_{scope} : wcet_{duration}(c(a)) \neq \perp$, i.e., all activities in a scope have a compensation activity with a WCET defined and $\exists a \in A_{scope} : situations(a) \neq \emptyset$ meaning that at least one activity of *scope* defines which situations of S must be regarded at runtime. From these definitions the WCET time of a *scope* can be calculated by finding a path $p = a_1 \rightsquigarrow .. \rightsquigarrow a_i \rightsquigarrow .. \rightsquigarrow a_n, a_i \in scope$ for which $\sum_1^n wcet_{duration}(a_i)$ is maximal, as the graph in *scope* is acyclic by definition. We define the set p^{max} to be the set of paths with the maximal WCET for a TASS. However, to enable the robust execution the most important aspect is that, even in case of errors, compensating already executed tasks must be achieved before the currently active situations are changing their state. Therefore it is important to know how much time the overall compensation can take, we need to calculate how much time the longest path of compensation activities takes to finish. In our model the compensation of activities is in reverse order of the actual tasks and therefore we must find a path $p_c = b_1 \rightsquigarrow .. \rightsquigarrow b_i \rightsquigarrow .. \rightsquigarrow b_m, b_i = c(a_i)$ for which $\sum_1^m wcet_{duration}(b_i)$ is maximal, analogous to the WCET for a set of activities in a *scope*. We define the set p_c^{max} to be the set of paths with the maximal WCET for the compensation activities of *scope*. Calculating how long a *scope* has time to execute the tasks or compensation is based on the time situations are active. After, finding the longest paths p^{max} and p_c^{max} inside a *scope* we take all situations $s_i \in situations(a), a \in A_{scope}$ and take the minimum guaranteed duration that the situations s_i are active, via $guarantee(A_{scope})$. The $guarantee(A_{scope})$ becomes the deadline for the execution of *scope*, which means to properly enter such a scope at runtime the compensation path $p_c \in p_c^{max}$ of scope must take less time to compensate as the situations have time to change their state, therefore, $\sum_{b \in p_c} wcet_{duration}(b) < guarantee(A_{scope})$. However, this just guarantees, that when the tasks themselves take to long they can be compensated in time, in addition, to give a better assurance that the task will actually be executed, we can check that $\sum_{a \in p} wcet_{duration}(a) + \sum_{b \in p_c} wcet_{duration}(b) < guarantee(A_{scope})$, stating that we have enough time to execute the provisioning tasks and are still able to compensate if any error may occur.

3.3 Generating Situation-Aware Processes

In Algorithm 1 we describe a simple method to generate such processes as directed and labeled graphs. It starts by first calculating the maximum common and deployable subgraph *mcs* between the current configuration $currentConf$ and the target configuration $targetConf$ (see line 2 in Algorithm 1). The set *mcs* contains all components between the two configurations which hold the following: if a component in *mcs* needs another to be used properly it must also be in *mcs*, e.g, a software component must be hosted on hardware component. To know which components we have to terminate to get to the target configuration, we calculate the component set *toRemove* by removing *mcs* from the current configuration $currentConf$, because *mcs* contains components we can reuse (see line 4 in Algorithm 1). On the other hand, to know which components need to be started we create the set *toStart*, by removing all components in *mcs* from the set $targetConf$ as we only need to start components which are not in the set of reusable components *mcs* (see line 5 in Algorithm 1).

Afterwards *terminate* and *start* tasks for each of the sets $toTerminate$ and *toStart* are created, respectively (see lines 9-16 in Algorithm 1), as well as for their compensation tasks. The ordering of *terminate* activities are to stop first source nodes (see lines 18-20 in Algorithm 1) and the *start* activities start with sink nodes of the graph (see lines 21-23 in Algorithm 1). Note that π_i selects the i-th component of a tuple. In other words, we update components from the 'bottom' of the graph, on the other hand termination tasks are executed

Algorithm 1 createProcess($currentConf, targetConf \in \mathcal{M}$)

```
1: // Get maximum set of common components and ensure that it is deployable
2: $mcs := maxCommonAndDeployableSubgraph(currentConf, targetConf)$
3: // Find with $mcs$ the components that have to be started or terminated
4: $toTerminate := currentConf \setminus mcs$
5: $toStart := targetConf \setminus mcs$
6: $A_B := \{\}, A_C := \{\}, L := \{\}, scopes := \{\}$
7: // For each component that will be terminated or started add a $terminate$ or $start$
8: // task that will use its operations to terminate or start an instance
9: for all $c \in toTerminate$ do
10:     $A_B \leftarrow A_B \cup (c, terminate)$
11:     $A_C \leftarrow A_C \cup (c, start)$
12: end for
13: for all $c \in toStart$ do
14:     $A_B \leftarrow A_B \cup (c, start)$
15:     $A_C \leftarrow A_C \cup (c, terminate)$
16: end for
17: // Add control link based on relations between components
18: for all $r \in \{e | e \in \pi_2(currentConf)\} : \pi_1(r), \pi_2(r) \in toTerminate$ do
19:     $L \leftarrow L \cup ((\pi_1(r), terminate), (\pi_2(r), terminate))$
20: end for
21: for all $r \in \{e | e \in \pi_2(targetConf)\} : \pi_1(r), \pi_2(r) \in toStart$ do
22:     $L \leftarrow L \cup ((\pi_2(r), start), (\pi_1(r), start))$
23: end for
24: $termSinks := \{t \mid t \in A_B : \pi_2(t) = terminate \wedge \nexists e \in L : \pi_1(e) = t\}$
25: $startSources := \{t \mid t \in A_B : \pi_2(t) = start \wedge \nexists e \in L : \pi_2(e) = t\}$
26: // Connect the sinks and sources, termination tasks before start tasks
27: $L \leftarrow L \cup (e, r) : \forall t_1 \in termSinks, \forall t_2 \in startSources$
28: // For paths of activities with annotated compensation activities create a TASS
29: for all $p = a_1 \rightsquigarrow .. \rightsquigarrow a_i \rightsquigarrow .. \rightsquigarrow a_n, a_i \in A_B, c(a_i) \neq NOP, sits(c(a_i)) \neq \epsilon$ do
30:     $scopes \leftarrow scopes \cup (\{a_1, .., a_n\}, \{(a_1, a_2), .., (a_{n-1}, a_n)\},$
31:     $sits(a_1) \cup sits(c(a_1)) \cup sits(a_2) \cup sits(c(a_2)).., ..)$
32: end for
33: return graph $(A_B \cup A_C, L, c, scopes)$
```

in reversed order. We connect the termination and start activities so they are ordered with the goal that everything is terminated first and started after (see lines 24-27 in Algorithm 1) by connecting the sinks of termination tasks with the sources of start tasks. The last step of the algorithm is to add all TASS of the process by finding all paths for which there are compensation activities defined. In addition, all of the compensation activities of such a path must have annotated situations which must hold inside a TASS.

In summary, the main idea of our approach is to model and execute updates tasks based on combining the context of an application with the worst-case execution time of the used management tasks. This combination allows to execute tasks at runtime without the need to reduce the availability, as they can be executed when the overall system and its context are in a suitable state.

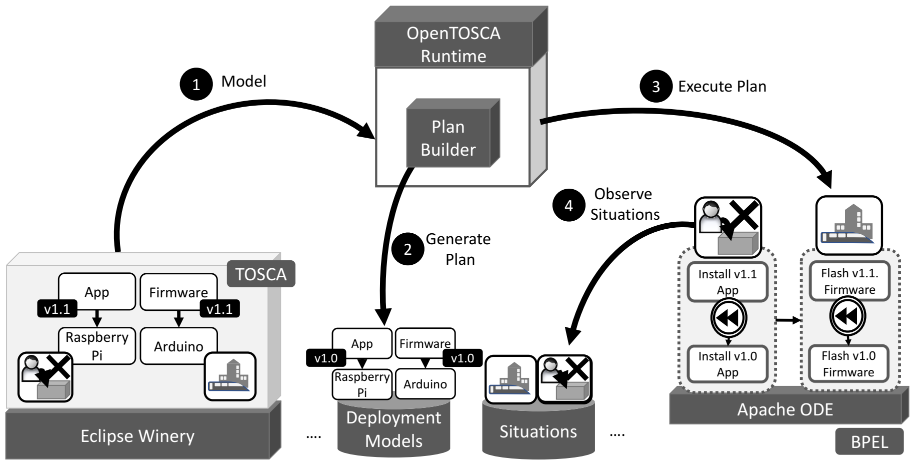

Fig. 5. Prototypical implementation within the OpenTOSCA Ecosystem.

4 Prototypical Implementation and Discussion

In this section we describe our prototypical implementation. To validate the practical feasibility of our concepts, we used the deployment modeling language TOSCA [30] for the following reasons: (i) it provides a vendor- and technology-agnostic modeling language and (ii) it is ontologically extensible [4]. Further, for our prototype we extended the OpenTOSCA ecosystem [9][1] providing an open source TOSCA modeling and orchestration implementation. While the declarative modeling within the ecosystem is based on TOSCA, the imperative deployment models are generated and implemented in the workflow language BPEL [29] which use the specified operations of components.

To implement the first two steps of our proposed method we used the TOSCA modeling tool Eclipse Winery [25][2] and modeled a prototypical deployment model based on a Makeblock mBot [3] programmable robot consisting of an embedded device (i.e. an Arduino based mCore Control Board) and different sensors and actuators, such as, ultrasonic sensor or a motor. The robot was also connected to an attached Raspberry Pi 2 [4] to enable to control the robot via WiFi by a software adapter that receives control messages via the MQTT protocol. To enable updates while the robot is in use we annotated to each component of the deployment model a situation. The Raspberry Pi represents the commodity hardware hosting software for users, therefore, the *Not in Use* situation is annotated, while the embedded device of the robot was annotated with a *At*

[1] https://github.com/OpenTOSCA/container
[2] https://github.com/eclipse/winery
[3] https://www.makeblock.com/mbot
[4] https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

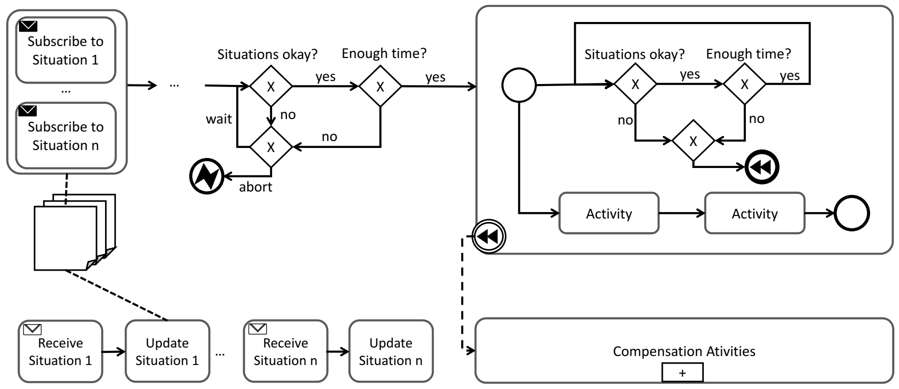

Fig. 6. Mapping TASS to BPMN to illustrate the mapping to BPEL.

Station situation, which where calculated by scripts implemented in Python. We measured the flashing times of the embedded device by the Raspberry Pi and annotated an WCET upper bound of 1 minute for the flashing operation. Additionally, we used data by the Deutsche Bahn[5], the german railway, about the local metro, S-Bahn, from the year 2017. A simple analysis of the data delivered that the mean time a train is at a stop is 65.31 and median at 60.0 with a standard deviation of 41.87 seconds [6]. The minimal time a train was at a station was 60 seconds, while the longest was to be 101 minutes, therefore the worst-case time of 1 minute of updating the robot is feasible. The WCET, situations and TASS specification annotations where mapped to so-called TOSCA Policies that can be used to specify non-functional requirements, e.g., as in our case, the situation-aware execution of operations. The generation of a situation-aware process in step 3 of our method is based on our previous work on generating so-called TOSCA Plans [7] to enable the automated deployment and configuration of application components and their relations. We added the generation of TOSCA Plans which can update running application instances from a source deployment model to a target deployment model. To implement our Time-Aware Situational Scope concept we extended the code generation according to subsection 4.1 to add the subscription to situations, evaluating the current state of situations and the allowed execution time. The generated process, i.e. TOSCA Plans, are implemented in the workflow language BPEL and executed on the compatible workflow engine Apache ODE, while the necessary data, such as TOSCA models, instance data, situations and operations execution time are stored within the OpenTOSCA TOSCA runtime.

[5] https://data.deutschebahn.com/dataset/data-s-bahn-stuttgart-ris-archiv-daten
[6] https://github.com/nyuuyn/sbahnstuttgart

4.1 Mapping to standard-compliant process languages

We implemented our concept of TASS within the OpenTOSCA Plan Builder that can generate imperative processes in the workflow language BPEL [29][7]. The Plan Builder is based on previous work [7] and was extended to enable generation processes able to adapt a running application instance to a new model it belongs to, hence, enable to model and execute updates. However, in the following we will describe a mapping of TASS to BPMN[31][8] which is depicted in Figure 6, allowing a better understanding of the mapping as BPEL has no standardized graphical notation like BPMN. The mapping assumes that situations are available via a service, such as middlewares sharing context [24]. We used the Situation Recognition Layer from the SitOPT project which we integrated in previous work within OpenTOSCA, it was however extended to enable software components to write the entry- and exit times of situations, i.e., enabling to manipulate the data of how long situations are either active or not.

The general idea of mapping TASS to BPMN (see Figure 6) is first to subscribe to all situations referenced in the process and, additionally, enable to receive notifications when they change and update internal situation variables. A concrete Time-Aware Situational Scope is added to a BPMN process as follows: At the beginning of the activities to execute, three gateways are used to check whether the situations are active at all and whether the time the situations are active is enough to execute the operations and compensation activities of the scope. If one of the conditions doesn't hold, the control flow either aborts the overall execution of the activities by throwing a fault, or starts a loop for the condition evaluation, depending whether the TASS has the *EntryMode* defined as *abort* or *wait*, respectively. When all of the conditions hold the first time they are evaluated, the activities specified within the TASS are executed according to the control flow. While these are executed, the conditions of the situations and the available time is continuously monitored in parallel to the actual business logic, and if the conditions aren't met anymore or the business logic takes longer then the available time, the currently running tasks are aborted and compensated by the surrounding compensation handlers.

The implemented mapping to BPEL is almost analogous to BPMN and differs based on wrapping the business functions into their own BPEL *scope* (used as an equivalent to a BPMN in-line subprocess) activity which is added to a *flow* activity (enabling the graph-based ordering of activities) containing two *scope* activities which are used to check the situations state, just as in the BPMN mapping. However, instead of looping via control flow edges we wrap the situation observation activities into a *while* loop activity, as it is not allowed to model cycles in a *flow* activity. When the situations states are checked within the while loop and are found to be not valid anymore a fault will be raised that starts the compensation activities and therefore compensates already completed activities which were added in the generation phase of our approach.

[7] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[8] https://www.omg.org/spec/BPMN/2.0/

5 Related Work

In the following we describe the related work of our approach, which are from dynamic software updating [26], Context-Aware Systems (CAS) [3][24], real-time systems/transactions [21] and runtime adaptation [38]. While the field of dynamic software updating tries to solve the problem of runtime updates, CAS enable applications to observe their environment to enable reactively or proactively change application behavior and structure, and real-time systems focus on the execution of tasks under timing constraints.

Kramer and Magee [26] present a model to enable Dynamic Software Changes, i.e., the execution of software updates at runtime. One part of the model is to detect a so-called *quiescent* state of software components, i.e., detect software components that are not used or do not affect the running application and its running transactions when updating. In a CPS observing only these types of components is not sufficient as CPS are integrated within physical environments consisting of software, hardware, people and other physical resources, which can create different kinds of errors that can have severe impact on the surrounding. Therefore, runtime update methods must include context information to determine a proper update state, and therefore be Context-Aware.

In the field of Context-Aware Systems the research goal is to enable applications to work contextually, i.e., adapt the behavior or structure according to their context. Context according to Dey [14] is every information that can describe the situation of an entity. To model context different modeling techniques can be used, such as, ontologies, markup or simple key-value models [5][36][6]. On top of these models middlewares can be used to distribute the current state of the context to the applications[3][24]. In research, context-aware systems were applied to multiple different fields, e.g. the Internet of Things [32]. However, most approaches lack the timing aspect in regards to context-aware deployment and management of applications [35]. One application on Context-Awareness to processes is presented by Bucchiarone et al. [10] to enable the runtime adaptation of business processes. The main parts of their method is a main business process containing abstract activities that are implemented at runtime by business process fragments that are added to the main process based on goals. These goals describe what a process or fragments can achieve at runtime and is therefore used to adapt the process to the desired goal of the overall business process. While the presented approach enables context-aware adaptation of processes, it doesn't handle aspects such as timing and therefore robust execution. This is the case in the work of Avenoğlu and Eren [**?**] as well. They combined a workflow engine with a Complex-Event Processing system to enable observing context. Another Context-Aware Workflow approach is presented by Choi et al. [12] that integrates context data into the workflows on top of the presented middleware, however, it misses the timing as well. Although the presented works integrate context into their approaches they only enable reactive adaptation while to properly manage software updates in CPS the adaptation, i.e., or execution of tasks must be managed proactively. For example, Grabis demonstrates in [15] how the proactive adaptation of workflows can benefit the performance of the adaptation, however, focuses only aspects of the workflow itself not the context as within our approach. Vansyckel et al. [37] present an approach which uses proactive adaptation of application configurations based on context and it's cost to adapt the application in the present and future context. Although they regard timing of the adaptation and context they regard it as a part of a cost function. The proactive context-aware execution of workflows builds on the timing, i.e., how long a particular context is valid in the environment and therefore introduces timing constraints. Specifying time constraints, such as, deadlines on execution of a certain task (e.g. triggering of airbags in the automotive field), are crucial for the development of applications regarding QoS, such as, safety, performance and proper handling of context. In research works integrating time into process models is not new, previous work focused on applying different timing constraints on tasks of a process [33][11], however these do not consider the dynamic change of deadlines, which is natural in context.

In contrast, the field of real-time transactions or scheduling tasks are executed before a certain soft- or hard deadline [13]. A soft deadline is a point in time where a task should be finished, while a hard deadline is a point in time where the task must be finished. The work by Zeller et al. [41] optimized automotive systems in regards to real-time components. In contrast, our approach focuses on the execution of such (re-)configurations at the right time and before a deadline. Most real-time transaction scheduling algorithms work with fixed deadlines, however, in the real world hard deadlines are not always easy to calculate, especially when regarding context and situations. To cope with such problems Litoiu and Tadei [27] presented work which regards deadlines as well as the execution time of tasks as fuzzy. This enables the execution of tasks in uncertain context, whereas our approach assumes hard deadlines for situations. However, it does not regard the compensation of running tasks and assume that behavior of deadlines can be mapped to fixed functions.

In the field of runtime adaptation a related study was done by Grua et al. [16] on self-adaptation in mobile applications, which found that methods for adaptation mostly regarded timeliness in a reactive manner. However, a study by Keller and Mann [22] found contributions that tackle the issue of timing adaptation of an application. Their findings show that there is limited work which handles all parts of an adaptation from the timeliness view, in the study they found only a single paper which handles all phases (time to detect the need for adaptation, executing the adaptation,..) when adapting an application. A detailed view on the issue of timing adaptation is given by Moreno [28]. Moreno tackled the issue of timing adaptation within a MAPE-K loop with different methods such as using a Markov Decision Process or Probabilistic Model Checking.

In general, applying updates in CPS can be viewed as adapting an application under timing constraints to not disrupt its availability. However, which timing constraints are used to manage the adaptation is depending on the use case itself. Some works only regard system properties such as message delays of the approach or the application itself, however, we argue it is equally important to regard the system context to cope with timing issues when adapting.

6 Conclusion and Future Work

The execution of software updates in Cyber-Physical Systems combines challenges from different fields. First they integrate a plethora of heterogeneous software and hardware components that have to be managed differently, but in addition to software and hardware, such systems have to be able to regard the current state in the environment, i.e., Cyber-Physical Systems must be Context-Aware and especially when executing management tasks such as software updates. As updates disrupt the functionality of systems and therefore availability they must regard context at runtime. When the execution is started at the wrong time or is disrupted by the environment itself, they can create serious safety issues and at least reduce the availability of components. In our scenario of trains, components either used by passengers or to control safety-critical functions wouldn't be available if the update would be executed at runtime without regarding context, such as, the current usage of components or the position of a train. Therefore it is beneficial to regard Cyber-Physical Systems as Context-Aware Systems.

In this paper we presented a method to enable the Situation-Aware execution of software updates in Cyber-Physical Systems. The method is based on modeling the update in a declarative model and specifying the situations which have to be active and timing constraints on the operations of the updated components to determine their worst-case execution time. Based on this model our method generates a Situation-Aware process with so-called Time-Aware Situational Scopes which can update the system according to the initial model by calling the component operations, however, these are only executed when the modeled situations are active and their worst case execution time doesn't exceed the available time situations are active. Therefore, operations will only be executed when the context is suitable for updates. We implemented our approach within the OpenTOSCA systems which was already able to start update processes when situations occured, however, missed the proper timing and therefore could lead to issues as mentioned.

In the future we plan to extend the presented concept to enable Situation-Aware execution of operations with deadlines and execution times that are not crisp, enabling to apply the Situation-Aware execution of operations on cloud resources as well as their execution times are significantly harder to determine, because of multiple virtualization layers on top of the actual hardware. This could be achieved by using measurement-based methods to determine a worst-case execution time of operations, which take the times of previous executions. Another track of future work is the evaluation of the proposed method on available simulation data, such as, a whole countrys' train network.

Acknowledgements

This work was funded by the DFG project DiStOPT (252975529).

References

1. Andr, C., Quijano, N., Mojica-nava, E.: A Survey on Cyber Physical Energy Systems and their Applications on Smart Grids. 2011 IEEE PES Conference on Innovative Smart Grid Technologies (ISGT Latin America) (2011). https://doi.org/10.1109/ISGT-LA.2011.6083194
2. Atzori, L., Iera, A., Morabito, G.: The Internet of Things: A survey. Computer Networks (2010). https://doi.org/10.1016/j.comnet.2010.05.010
3. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing (2007). https://doi.org/10.1504/IJAHUC.2007.014070
4. Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., Leymann, F.: A systematic review of cloud modeling languages. ACM Comput. Surv. **51**(1), 22:1–22:38 (Feb 2018). https://doi.org/10.1145/3150227
5. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive and Mobile Computing **6**(2), 161 – 180 (2010). https://doi.org/https://doi.org/10.1016/j.pmcj.2009.06.002
6. Bolchini, C., Curino, C.A., Quintarelli, E., Schreiber, F.A., Tanca, L.: A Data-oriented Survey of Context Models. SIGMOD Rec. **36**(4), 19–26 (Dec 2007). https://doi.org/10.1145/1361348.1361353
7. Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., Wettinger, J.: Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In: International Conference on Cloud Engineering (IC2E 2014). pp. 87–96. IEEE (Mar 2014)
8. Breitenbücher, U., Hirmer, P., Képes, K., Kopp, O., Leymann, F., Wieland, M.: A situation-aware workflow modelling extension. In: Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services. pp. 64:1–64:7. iiWAS '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2837185.2837248
9. Breitenbücher, U., Endres, C., Képes, K., Kopp, O., Leymann, F., Wagner, S., Wettinger, J., Zimmermann, M.: The OpenTOSCA Ecosystem - Concepts & Tools. In: European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016,. pp. 112–130. INSTICC, SciTePress (2016). https://doi.org/10.5220/0007903201120130
10. Bucchiarone, A., Marconi, A., Pistore, M., Raik, H.: Dynamic adaptation of fragment-based and context-aware business processes. Proceedings - 2012 IEEE 19th International Conference on Web Services, ICWS 2012 pp. 33–41 (2012). https://doi.org/10.1109/ICWS.2012.56
11. Chama, I.E., Belala, N., Saidouni, D.E.: Formalization and analysis of timed bpel. In: Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014). pp. 483–491 (Aug 2014). https://doi.org/10.1109/IRI.2014.7051928
12. Choi, J., Cho, Y., Choi, J., Choi, J.: A Layered Middleware Architecture for Automated Robot Services. International Journal of Distributed Sensor Networks **10**(5), 201063 (2014). https://doi.org/10.1155/2014/201063
13. Davis, R.I., Burns, A.: A Survey of Hard Real-time Scheduling for Multiprocessor Systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (Oct 2011). https://doi.org/10.1145/1978802.1978814

14. Dey, A.: Understanding and using context. Personal and ubiquitous computing **5**(1), 4–7 (2001). https://doi.org/10.1007/s007790170019
15. Grabis, J.: Application of predictive simulation in development of adaptive workflows. In: Proceedings of the Winter Simulation Conference 2014. pp. 996–1004 (Dec 2014). https://doi.org/10.1109/WSC.2014.7019959
16. Grua, E.M., Malavolta, I., Lago, P.: Self-Adaptation in Mobile Apps: a Systematic Literature Study. In: 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 51–62 (May 2019). https://doi.org/10.1109/SEAMS.2019.00016
17. Gunes, V., Peter, S., Givargis, T., Vahid, F.: A survey on concepts, applications, and challenges in cyber-physical systems. KSII Transactions on Internet and Information Systems (2014). https://doi.org/10.3837/tiis.2014.12.001
18. Guo, Y., Hu, X., Hu, B., Cheng, J., Zhou, M., Kwok, R.Y.K.: Mobile Cyber Physical Systems: Current Challenges and Future Networking Applications. IEEE Access **XX**(c), 1–1 (2017). https://doi.org/10.1109/ACCESS.2017.2782881
19. Haque, S.A., Aziz, S.M., Rahman, M.: Review of Cyber-Physical System in Healthcare. International Journal of Distributed Sensor Networks **10**(4), 217415 (2014). https://doi.org/10.1155/2014/217415
20. Hassan, R., Markantonakis, K., Akram, R.N.: Can You Call the Software in Your Device be Firmware? Proceedings - 13th IEEE International Conference on E-Business Engineering, ICEBE 2016 - Including 12th Workshop on Service-Oriented Applications, Integration and Collaboration, SOAIC 2016 pp. 188–195 (2017). https://doi.org/10.1109/ICEBE.2016.040
21. Kejariwal, A., Orsini, F.: On the definition of real-time: Applications and systems. In: 2016 IEEE Trustcom/BigDataSE/ISPA. pp. 2213–2220 (Aug 2016). https://doi.org/10.1109/TrustCom.2016.0341
22. Keller, C., Mann, Z.Á.: Towards understanding adaptation latency in self-adaptive systems. In: 15th InternationalWorkshop on Engineering Service-Oriented Applications and Cloud Services (WESOACS). Springer (2019)
23. Képes, K., Breitenbücher, U., Leymann, F.: Situation-aware management of cyber-physical systems. In: Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019). pp. 551–560. SciTePress (May 2019). https://doi.org/10.5220/0007799505510560
24. Knappmeyer, M., Kiani, S.L., Reetz, E.S., Baker, N., Tonjes, R.: Survey of context provisioning middleware. IEEE Communications Surveys and Tutorials **15**(3), 1492–1519 (2013). https://doi.org/10.1109/SURV.2013.010413.00207
25. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – A Modeling Tool for TOSCA-based Cloud Applications. In: Proceedings of the 11^{th} International Conference on Service-Oriented Computing (ICSOC 2013). pp. 700–704. Springer (Dec 2013)
26. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering **16**(11), 1293–1306 (Nov 1990). https://doi.org/10.1109/32.60317
27. Litoiu, M., Tadei, R.: Real-time task scheduling with fuzzy deadlines and processing times. Fuzzy Sets and Systems **117**(1), 35 – 45 (2001). https://doi.org/https://doi.org/10.1016/S0165-0114(98)00283-8
28. Moreno, G.A.: Adaptation Timing in Self-Adaptive Systems. Ph.D. thesis, Carnegie Mellon University (4 2017)
29. OASIS: Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Organization for the Advancement of Structured Information Standards (OASIS) (2007)

30. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2013)
31. OMG: Business Process Model and Notation (BPMN) Version 2.0. Object Management Group (OMG) (2011)
32. Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context Aware Computing for The Internet of Things: A Survey. Communications Surveys Tutorials, IEEE **16**(1), 414–454 (2014). https://doi.org/10.1109/SURV.2013.042313.00197
33. Pichler, H., Eder, J., Ciglic, M.: Modelling processes with time-dependent control structures. In: Mayr, H.C., Guizzardi, G., Ma, H., Pastor, O. (eds.) Conceptual Modeling. pp. 50–58. Springer International Publishing, Cham (2017)
34. Shrouf, F., Miragliotta, G.: Energy management based on Internet of Things: Practices and framework for adoption in production management. Journal of Cleaner Production (2015). https://doi.org/10.1016/j.jclepro.2015.03.055
35. Smanchat, S., Ling, S., Indrawan, M.: A Survey on Context-aware Workflow Adaptations. In: Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia. pp. 414–417. MoMM '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1497185.1497274
36. Strang, T., Linnhoff-Popien, C.: A Context Modeling Survey. In: First International Workshop on Advanced Context Modelling, Reasoning And Management at UbiComp 2004, Nottingham, England, September 7, 2004 (September 2004)
37. Vansyckel, S., Schäfer, D., Schiele, G., Becker, C.: Configuration Management for Proactive Adaptation in Pervasive Environments. In: 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems. pp. 131–140 (Sep 2013). https://doi.org/10.1109/SASO.2013.28
38. Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T.: A survey of formal methods in self-adaptive systems. In: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering. p. 67–79. C3S2E '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2347583.2347592, https://doi.org/10.1145/2347583.2347592
39. Wieland, M., Schwarz, H., Breitenbücher, U., Leymann, F.: Towards situation-aware adaptive workflows: SitOPT - A general purpose situation-aware workflow management system. In: 2015 IEEE International Conference on Pervasive Computing and Communication Workshops, PerCom Workshops 2015 (2015). https://doi.org/10.1109/PERCOMW.2015.7133989
40. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem&mdash;overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (May 2008). https://doi.org/10.1145/1347375.1347389
41. Zeller, M., Prehofer, C., Weiss, G., Eilers, D., Knorr, R.: Towards Self-Adaptation in Real-Time, Networked Systems: Efficient Solving of System Constraints for Automotive Embedded Systems. In: 2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems. pp. 79–88 (Oct 2011). https://doi.org/10.1109/SASO.2011.19