Living with Uncertainty in the Age of Runtime Models

Holger Giese[1], Nelly Bencomo[2], Liliana Pasquale[3], Andres J. Ramirez[4], Paola Inverardi[5], Sebastian Wätzoldt[1], and Siobhán Clarke[6]

[1] Hasso Plattner Institute at the University of Potsdam, Germany
{Holger.Giese,Sebastian.Waetzoldt}@hpi.uni-potsdam.de
[2] Aston University, UK
nelly@acm.org
[3] Lero - Irish Software Engineering Research Centre, Ireland
liliana.pasquale@lero.ie
[4] Michigan State University, USA
ramir105@cse.msu.edu
[5] University of L'Aquila, Italy
paola.inverardi@di.univaq.it
[6] Trinity College Dublin, Ireland
siobhan.clarke@scss.tcd.ie

Abstract. Uncertainty can be defined as the difference between information that is represented in an executing system and the information that is both measurable and available about the system at a certain point in its life-time. A software system can be exposed to multiple sources of uncertainty produced by, for example, ambiguous requirements and unpredictable execution environments. A runtime model is a dynamic knowledge base that abstracts useful information about the system, its operational context and the extent to which the system meets its stakeholders' needs. A software system can successfully operate in multiple dynamic contexts by using runtime models that augment information available at design-time with information monitored at runtime. This chapter explores the role of runtime models as a means to cope with uncertainty. To this end, we introduce a well-suited terminology about models, runtime models and uncertainty and present a state-of-the-art summary on model-based techniques for addressing uncertainty both at development- and runtime. Using a case study about robot systems we discuss how current techniques and the MAPE-K loop can be used together to tackle uncertainty. Furthermore, we propose possible extensions of the MAPE-K loop architecture with runtime models to further handle uncertainty at runtime. The chapter concludes by identifying key challenges, and enabling technologies for using runtime models to address uncertainty, and also identifies closely related research communities that can foster ideas for resolving the challenges raised.

N. Bencomo et al. (Eds.): Models@run.time, LNCS 8378, pp. 47–100, 2014.

1 Introduction

Uncertainty can be defined as the difference between information that exists in an executing system and the information that is both measurable and available at a certain point in time [1]. Within the context of software systems, uncertainty can arise from ambiguous stakeholders' needs, the system itself, or its operational or execution environment. For example, a stakeholder can introduce uncertainty by formulating an ambiguous specification [2], the system itself can introduce uncertainty by gathering monitoring information that may be inaccurate and/or imprecise [3], and the surrounding environment can introduce uncertainty by generating inputs that the system cannot interpret [4]. Unfortunately, these sources of uncertainty rarely occur independently of each other. Instead, the effects produced by these sources of uncertainty tend to compound and thereby inhibit the system from fully satisfying its requirements.

Traditional software design approaches tend to address these forms of uncertainty by identifying *robust* solutions at development-time that can continuously exhibit good performance at runtime. Nevertheless, identifying these robust solutions is a challenging task that requires identifying the set of operational contexts that the system might have to support. For systems that will be deployed in highly dynamic environments, identifying a robust solution might be infeasible since it is often impractical to anticipate or enumerate all environmental conditions a software system will encounter throughout its lifetime. Increasingly, software systems, such as context-aware systems and self-adaptive systems (SAS), use runtime models to cope with uncertainty [5,6] by either partially or, if possible, fully resolving sources of uncertainty at runtime. These models, when fed with data monitored at runtime, allow for the dynamic computation of a predictable behaviour of the system.

A runtime model is a knowledge base that abstracts useful information about the executing system, its environment, and the stakeholders' needs and that can be updated during the system's life time. It can be used by either the system itself, humans or other systems. Different types of development-time models and abstractions can be used to this end, including requirements, architectural, and behavioural models. Furthermore, we envision that other kinds of models (e.g., [7]), linking different development models and abstractions, can be used as well. Runtime models can support resolution of some forms of uncertainty, which can manifest, for example, when information that was previously unavailable becomes available during execution. In some cases, however, uncertainty remains present in one form or another, such as when physical limitations in monitoring devices result in monitoring data that is insufficiently accurate and precise for assessing the task at hand. In both cases, model-based techniques can be applied at either development- and runtime to address uncertainty. While development-time techniques focus on the explicit representation of sources of uncertainty that can affect a software system, runtime techniques focus on refining and

augmenting runtime models with monitoring information collected as the system executes.

Research communities are increasingly exploring the concept of uncertainty and how it impacts their respective fields. Some of these research communities include economics [8], artificial intelligence [9], robotics [10], and software engineering [11]. While uncertainty has been studied in parallel by different research communities, a definitive solution for engineering systems that are able to handle uncertainty has not been provided yet. Within software engineering in particular, a key step forward in this direction is to first explicitly represent sources of uncertainty in models of the software system. Current modeling techniques such as variation point modeling [12,13], however, cannot be directly applied as uncertainty cannot be represented by enumerating all possible behavioural alternatives [4]. Thus, new abstractions must be provided to declaratively specify sources of uncertainty in a model at development-time, and then partially resolve such uncertainty at runtime as more information is gathered by the system.

This chapter first elaborates the relation between uncertainty and models - runtime models in particular. Furthermore, it reviews the fundamentals of handling uncertainties such as the relevant forms of uncertainty, the specific relation between time and uncertainty and the current approaches for development-time and runtime. In particular, it explores the role of feedback loops and the typical types of systems with runtime models. It also explores how runtime models can be leveraged to handle and reduce the level of uncertainty in an executing system. Ideally, if uncertainty is viewed as a function over the life-time of the system, then the level of uncertainty should monotonically decrease as design decisions are made and new information becomes available. In practice, however, the level of uncertainty in a software system might increase as new events emerge that were not anticipated previously. Thus, the vision of this roadmap chapter is to use development-time models to identify and represent sources and impacts of uncertainty. If possible, uncertainty should be resolved at development time. Furthermore, once monitoring information provides new insights about the system's behaviour and its surrounding execution environment, the running system should resolve and manage the remaining sources of uncertainty during execution by using runtime models.

This chapter is organized as follows. Section 2 presents a robotic system that will be used throughout the remainder of the chapter as an example of how to address uncertainty through runtime models. Section 3 discusses the general relationships between models and uncertainty. Model-based techniques for addressing uncertainty at development- and runtime are presented and discussed in Section 4. Emerging techniques to handle uncertainty in the context of epistemic, linguistic and randomized uncertainty are discussed in Section 5. Finally, Section 6 summarizes findings, presents research challenges, and future directions.

2 Case Study

In this section, we describe a simplified version of a real robot system. This case study simulates a distributed factory automation scenario[1]. It is used as a running example in this paper to discuss how development-time models and runtime models can be employed to cope with uncertainty. An extended description of our toolchain and development environment can be found in [10]. In the next Section we explain the possible types of uncertainty present in the case study. We also provide a goal, an environmental and an initial behavioural model to illustrate the requirements and the behaviour of the case study.

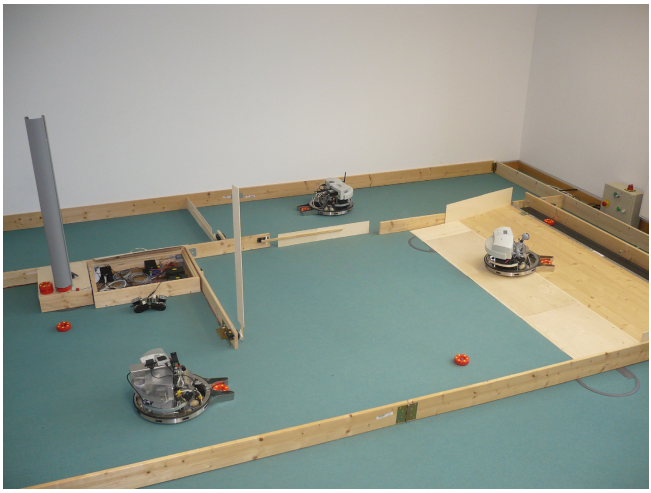

Fig. 1. Case Study: Factory automation robot scenario

In the factory automation scenario depicted in Figure 1, three autonomous robots have to fulfill different tasks to reach overall goals in an uncertain environment. The regular behaviour of one robot is to move around, transport pucks, or charge the batteries. Sensors and actuators are used to monitor the current situation and to navigate through different rooms along changing paths. Each robot is able to communicate with other robots inside the scenario and computational nodes (e.g., servers, mobile phones) outside the scenario. Strict behavioural constraints ensure safety as well as reliability requirements, such as avoiding battery exhaustion and collisions. Beside the transportation of pucks, which is a functional goal, the robots should also satisfy other goals related to the quality of how the functional goals are met (called softgoals). Examples of softgoals are minimization of energy consumption and maximization of throughput. Note that in this scenario throughput is estimated in terms of number of

[1] For more information about our related lab see: `http://www.cpslab.de`

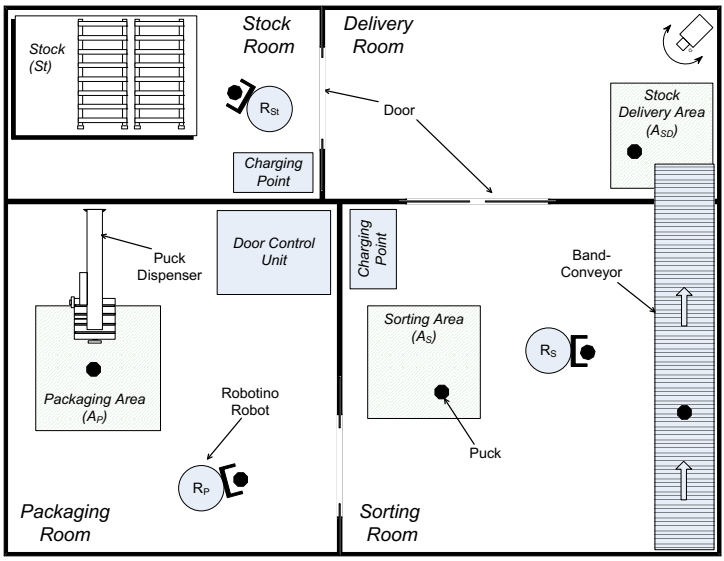

Fig. 2. Structural sketch of the robotic system including three autonomous robots. R_P is a robot that transports pucks from the packaging to the sorting room. Robot R_S (sorting) decides whether the puck is for a customer or the stock. Robot R_{St} transports the puck to the stock.

pucks transported to their final destination within a specific time unit (e.g., a day).

Figure 2 depicts a structural overview of the robot system. The whole simulation scenario is separated into four different rooms (Stock Room S_t, Delivery Room A_{SD}, Door Control Unit A_P and the Sorting Area R_S). In the room lower left of Figure 2, the pucks are packed and dropped for transportation in area A_P by a puck dispenser. The robot R_P transports the pucks from the packaging room to a second room (lower right) and drops it within the sorting area A_S. To maximize the throughput of puck transportation, another robot R_S picks up the puck from the sorting area and drops it on a band-conveyor. Depending on the current goals, the conveyor transports the puck to a customer delivery area outside the scenario (not shown in the picture) or to the stock delivery area A_{SD} (upper right room in the figure). A third robot R_{St} transfers the puck to the stock St. Each robot acts as an autonomous unit. Therefore, the tasks transportation, sorting and stocking are independent from each other. Within this scenario the execution conditions can be changed over time to handle certain loads, optimizing the puck transportation routes or cope with robot failures. As a result, the doors can be opened or closed dynamically. For example, robots can recalculate routes or take over additional transportation tasks. Furthermore, a robot can charge its battery at one of the two charging points if necessary. In

the following sections we will use this scenario to illustrate possible types of uncertainty.

3 Models and Uncertainty

To set the stage for our chapter in this section we first introduce the meaning of fundamental terms such as models, runtime models and uncertainty by using an exemplary goal, context and behavioural model for one robot of the factory automation example. Furthermore, we identify which kinds of runtime models are employed and outline the most common types of systems using such runtime models. Furthermore, we discuss the role that runtime models can play for the different types of systems.

3.1 Models

The definition of a model can vary depending on its purpose of usage. In this chapter, we define a model as follows [14].

Definition 1. *A* model *is characterized by the following three elements: an (factual or envisioned)* original *the model refers to, a* purpose *that defines what the model should be used for, and an* abstraction function *that maps only purposeful and relevant characteristics of the original to the model.*

It is important to note that a model always refers to an *original*. This original can be factual, and either it may exist already or, it may be an envisioned system which does not exist yet. In both cases, the model is used as a representation of the original to ease development or runtime activities.

Models may differ in their *purpose*. For instance, the purpose of a goal-based model [15] is to capture the requirements of a system, while the purpose of a finite-state machine (FSM) is to capture the possible behaviours of the system.

In this paper, we use three different kinds of models for the robotic example. First, a goal model represents the requirements of our scenario. Second, a representation of the structure of the physical space (map) including the location of agents and objects in the physical space is adopted as an exemplary model to specify context information about the system operational environmnet, which may change at runtime. Third, we use a model based on state machines to describe the behaviour of the system with respect to the current goals and the context.

To start, we cover different requirements and constraints in the goal model of our robotic scenario shown in Figure 3. We use the KAOS notation [16] to represent the goal model. This model has a hierarchical structure, since goals can be refined into conjoined subgoals (AND-refinement) or into alternative combinations of subgoals (OR-refinement). When a goal cannot be decomposed anymore, (i.e. it is a leaf goal), it corresponds to a functional or non-functional requirement of the system [16] . In Figure 3, the main goal of the robot system is to

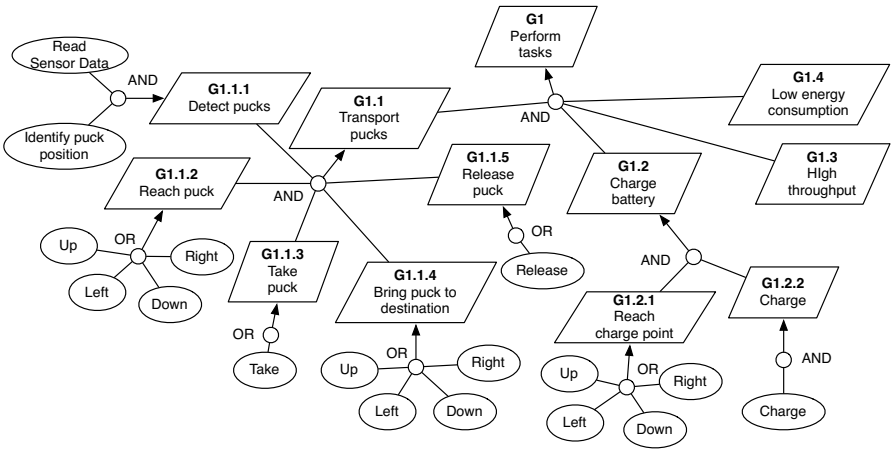

Fig. 3. The KAOS goal model of the Robot System

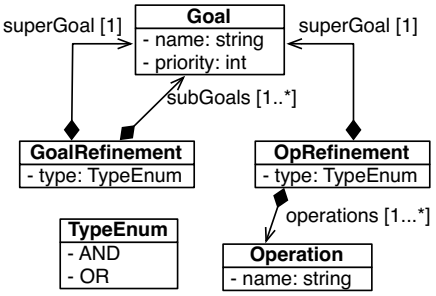

Fig. 4. An excerpt of the KAOS goal metamodel

perform its standard tasks, such as transport pucks ($G1.1$) and charge its battery ($G1.2$). To achieve the goal $G1.1$, the robot has to achieve the following functional requirements: detect a puck ($G1.1.1$), reach the puck ($G1.1.2$), take the puck ($G1.1.3$), bring the puck to its destination ($G1.1.4$) and finally release the puck ($G1.1.5$). A leaf goal that corresponds to a functional requirement can be "operationalized", that is, it can be decomposed into a set of conjoined or disjoined operations that should be executed to meet it [16]. For example, goal $G1.1.2$ is operationalized by operations Up, $Left$, $Down$, and $Right$, since the robot can move up, left, down, and right to reach the puck location. Note that the robot must also perform its standard tasks while satisfying the following non-functional requirements: high throughput ($G1.3$) and low energy consumption ($G1.4$).

In the robot system, we represent these goals according to a simplified KAOS metamodel as depicted in Figure 4. Note that each goal is associated with a

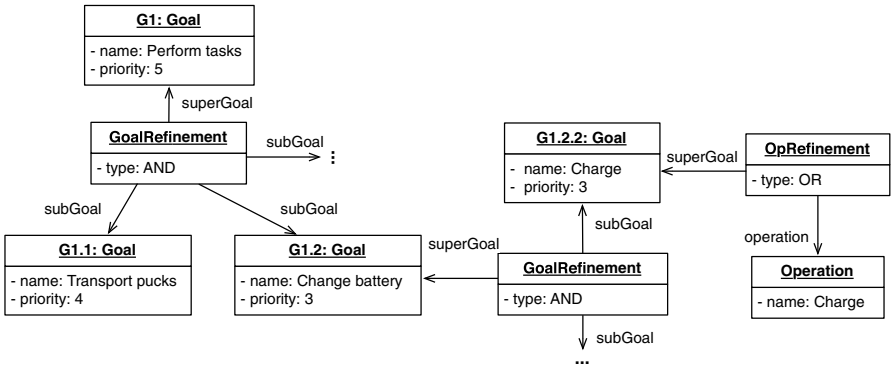

Fig. 5. An instance situation of the goal model shown in Fig. 3 of our robot system

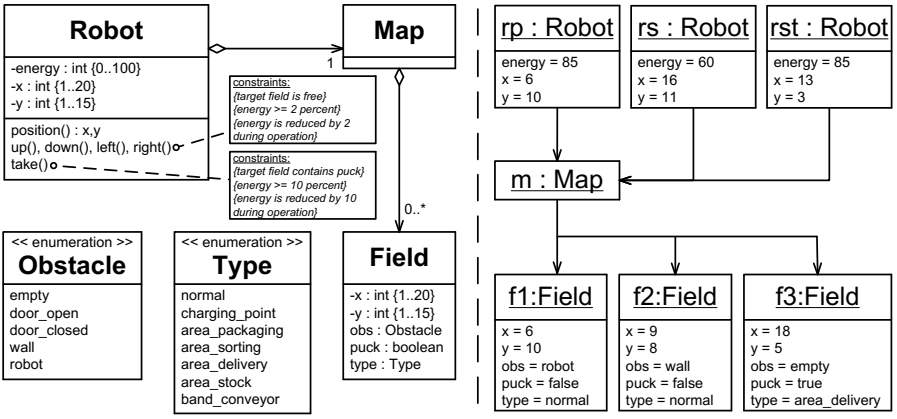

Fig. 6. On the left, the context metamodel of the robot as well as the internal map representation. On the right, an example snapshot of possible instance objects.

name and a priority. Context changes can affect goal priorities and therefore the tasks execution by the robots at runtime. Depending on the priority and the relation among the goals, the robot system weights the requirements during task execution and selects a set of tasks to be performed. Figure 5 presents an excerpt of an instance situation of the goal model shown in Figure 3. Because of the higher priority of the transportation task we have supposed, the robot will prefer this goal until it is necessary to charge the battery. Further constraints can also restrict the robot behaviour charging the battery only in adequate given situations (e.g. the power supply reaches a critical low level).

The second type of model is a structural context model of the robots. With this model we can represent the internal state of the robot, its internal representation of the environment (*Map*) as well as the possible relations between them according to the metamodel on the left in Figure 6. Each robot has an overall

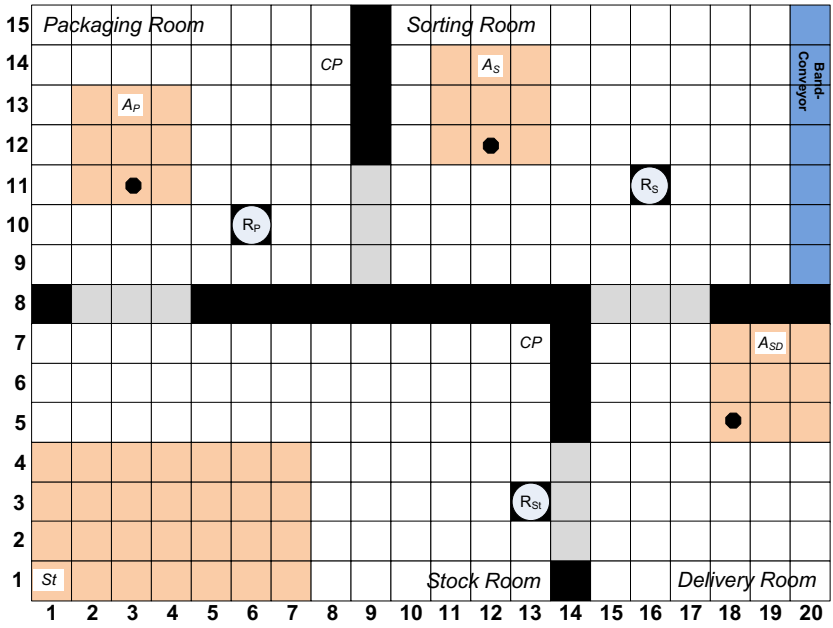

Fig. 7. Discrete grid map representation of the robot laboratory (cf. Fig. 2)

battery power supply with an energy level comprised between 0 and 100 percent. Additionally, its position is given by a x and y coordinate of the map associated with the laboratory. Furthermore, it can perform operations *up(), down(), left(),* and *right()* moving through the laboratory. Executing an operation consumes power as robots have to read sensor data necessary to detect pucks, and move inside the building. Therefore, each operation reduces the overall energy level of the battery. If the robot detects a puck, it can take it for transportation. Each robot maintains an environmental representation for navigation and localization issues.

The environment is represented as a discrete grid (*Map*) as shown in Figure 7. The smallest area in this map grid is a field (cell) with an unique position on the grid. It can contain obstacles as closed doors, walls, or other robots. Furthermore, a puck may lie on a field. Each cell has a type information that indicates special positions in the laboratory as charging points for the robot or the band conveyor (cf. Figure 2).

We assume that a robot is positioned on exactly one field at any time during its movement through the laboratory. Therefore, the operations up, down, left and right can be seen as atomic behaviours. There are no intermediate positions of objects (pucks) and a robot must be on the same field as a puck to grip it.

On the right in Figure 6, three robot instances with different positions and three field objects are shown. The battery of robot R_P contains 85 percent of the overall possible energy and the robot is on the position $x = 6$ and $y = 10$ in

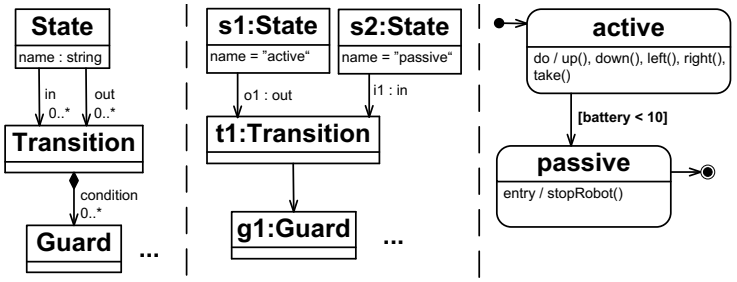

Fig. 8. On the left side: excerpt of the FSM metamodel; in the middle: initial example abstract syntax; on the right side: concrete syntax

the laboratory. The field f_2 on position $(9, 8)$ is a wall and the field f_3 on $(18, 5)$ contains a puck and is part of the delivery area.

The third and last model type for our robot example is a behavioural model in the form of a finite-state machine. Figure 8 shows an excerpt of the metamodel on the left as well as a very abstract initial example of the robot behaviour that is extended later (abstract syntax in the middle, concrete syntax on the right). In this version, the robot has two states. After an activation signal, the robot starts processing its tasks in the *active* state according to the goal model (cf. Fig. 3) until the battery is lower than a threshold of 10 percent. In this case, it will enter the *passive* state and stops all sensors, processing steps and actuators. Consequently, charging the battery automatically is not considered in this first version of our example.

Each model presented focuses on a specific concern of the robot scenario. The definition of a model explicitly states that the *abstraction function* is to eliminate characteristics of the original that are irrelevant as far as the purpose of the model is concerned. At the same time, the model abstraction should preserve the intent and semantics of those details that are relevant to the task at hand.

The same type of models can be used to represent different levels of abstraction. Several types of models and abstractions are often combined in a Hierarchical State Machine (HSM) [17] to provide a unique and comprehensive view of the same software system. In our robotic example, while one FSM can capture only the most abstract states of the robot, such as "the robot is stopped", another FSM can map those abstract states to finer-grained behaviours and state transitions.

If we consider the KAOS goal model (cf. Fig. 3) of our robot example, the original would be the real requirements that are in the stakeholders' minds and the purpose would be to represent the requirements of the system. Note that, for reasons of time and costs, some requirements may not be included in the system implementation, but only those that are relevant to the stakeholders. For this reason, the goal model abstracts the irrelevant goals and requirements and only focuses on the ones that will be implemented in the robotic system.

Moreover, if we consider the map representation, the difference between the real simulation area in the real world and the simplified field representation maintained by the robot is obvious. However, the map model abstraction covers all important aspects that are needed for the robot system offering special functionality, e.g., path planning as well as obstacle detection and avoidance.

In the case of the FSM (see Fig. 8), the original would be the robot and the purpose would be to capture the possible behaviours of the robot. Another case of abstraction is that only the relevant abstract states (modes) of the robot, like moving (*active*) or stopped (*passive*) as well as possible transitions between relevant abstract states, are captured.

A fundamental property of a model is its *validity*. This means that the model must correctly reflect the characteristics (i.e., goals, requirements, behaviours, and components) observed on the original. For example, in a goal model this property would imply that a set of goal operationalizations lead to the satisfaction of the requirements they are associated with. Likewise, given an appropriate input sequence for the FSM presented in Figure 8, a certain sequence of abstract states must be traversed. These states should correspond to the observable behaviour of the robot.

Even if a model is valid, it might not exactly predict the same behaviour of the original as this correlation depends on the model abstraction. Furthermore, it may be difficult to capture and interpret the non-determinism associated with unpredictable events and conditions in a model. In general, it is expected for a model to provide an acceptable degree of accuracy and precision. *Accuracy* measures how correct the model is at representing observable characteristics of the original. That is, accuracy measures the error between a predicted value and the value observed in the original. In contrast, *precision* measures how small the variation is in the prediction made by the model, as compared to the original.

With *over-approximation*, a model is guaranteed to include all possible behaviour of the original, but may also include behaviour that cannot be observed in the original. Therefore, over-approximation can result in false negatives, when a behaviour present in the model results in a failure that is never observed for the original. In contrast, with *under-approximation*, all behaviour captured in the model must also be possible for the original, but not necessarily vice versa. As such, under-approximation can prevent false negatives by ensuring that the characteristics represented in the model are also observable in the original. Note that under-approximation does not preclude the possibility of introducing false positives that take place when the model does not represent all behaviour that the original can exhibit. See for example [18] for a discussion about over- and under-approximation in the specific context of model checking.

3.2 Uncertainty and Uncertainty in Models

The definition of uncertainty depends on its context, origin, and effects upon the system. For the purposes of this chapter, we adopt and modify the definition proposed in [1,19] as follows:

Definition 2. *Uncertainty can be defined as the difference between the amount of information required to perform a task and the amount of information already possessed.*

This concept of uncertainty can be better understood by distinguishing between three main non-mutually exclusive forms of uncertainty: *epistemic*, *randomized*, and *linguistic*.

Epistemic uncertainty is a result of incomplete knowledge. For instance, the operationalization of requirements might be incomplete during system specification, and it can happen as the system designer might not know in advance what operations the system will provide.

Randomized uncertainty can occur due to system and environmental conditions that are either inherently random or cannot be predicted reliably. For instance, a sensor might introduce noise unpredictably into gathered values, thereby preventing a software system from accurately measuring the value of a property.

Lastly, *linguistic* uncertainty can result from a lack of precision or formality in linguistic concepts. For example, the satisfaction of requirements G1.3 and G1.4 in Figure 3 is vague because there is no precise way to express the notion of "high" throughput and "low" energy consumption.

The concept of uncertainty within a model can then be defined by extending Definition 2 as follows:

Definition 3. *Uncertainty is the difference between the information that a model represents about the original - that is relevant to its purpose - and the information that the model could, in theory, represent about the original that would be relevant for its purpose at a certain instant in the system lifetime.*

Uncertainty within a model can affect both the accuracy and precision of a model. Accuracy of a model refers to its the degree of closeness to the original, while precision refers to the degree to which the model is consistent (e.g., lead the system to the same behaviour under the same conditions). Although uncertainty may uniformly affect the entire model, its effects might be irrelevant if they are constrained within attributes that are never queried or evaluated. Thus, the relevance of uncertainty depends upon the criticality of the element that is affected with respect to the purpose of the model.

Dynamic models tend to increase the level of uncertainty over time because of the (possible) continuous updates which are performed to reflect changes in the original. A good example is our context model of the environment. It would be very hard for a robot to maintain a highly accurate model if humans or other moving obstacles frequently appear and disappear in the scenario. These observations can affect the behaviour of a single robot (e.g., path planning and route recalculation) as well as the overall scenario (e.g., new task distribution). As a consequence, this can also render the model imprecise, since the robots' behaviour might not be consistent with respect to previous and equivalent environmental situations. However, this phenomenon is not observable in static models.

In general, uncertainty in a model can be addressed by using internal or external techniques. Internal techniques address uncertainty by increasing the accuracy of the model at the expense of decreasing its precision. Thus, although the outcome of a model prediction might be inconsistent, it is closer to the possible outcomes of the original. External techniques, on the other hand, tend to under-approximate the original by increasing the precision of the model at the expense of decreasing its accuracy. For this reason, post-processing is usually required to ensure the outcome of a model prediction does not exceed certain bounds or thresholds of what the original can exhibit.

For the remainder of this chapter we focus on addressing uncertainty with internal techniques. We consider the predictions of a model, whether it is a requirements, structural or behavioural model, and its post-processing analysis as a *combined* prediction of an extended model. Furthermore, we also acknowledge that a model prediction might be correct while not being fully accurate, and that upper bounds on the prediction error might not necessarily be accounted for in such predictions.

3.3 Runtime Models

Definition 4. *A* runtime model *is a model that complies with Definition 1 and, in addition, is characterized as follows: part of its* purpose *is to be employed at runtime in a system and its* encoding *enables its processing at runtime. The runtime model is* causally-connected *to the original (running system), meaning that a change in the runtime model triggers a corresponding change in the running system and/or vice versa (extended from [20]).*

As outlined in the definition, runtime models differ from other types of models in both their purpose and encoding. Specifically, while development-time models, such as state machines, primarily support the specification, design, implementation, and testing, of a software system, a runtime model captures relevant information of the running system for different purposes, which either are part of the system functional features or are subject to assurance and analysis (non-functional features). Due to the encoding that enables its processing at runtime, other running systems, stakeholders (e.g., final users) and the system itself can alter these models at runtime.

A runtime model can span different types of models (e.g., structural or behavioural models), and can have different degrees of accuracy and precision. Independently on whether changes are automatically included or are externally applied on the model, a runtime model can be used for different purposes. It can be used as a knowledge repository about the system, its requirements, or its execution environment. It can also support adaptation of the system and/or its execution environment, as new information about the original becomes available. An overview about different runtime model categories and the relations between runtime models are described in [21].

In the following sections, we discuss a feedback-loop-based approach that enables handling of runtime models during system operation. Afterwards, we discuss some most common types of system that typically employ runtime models.

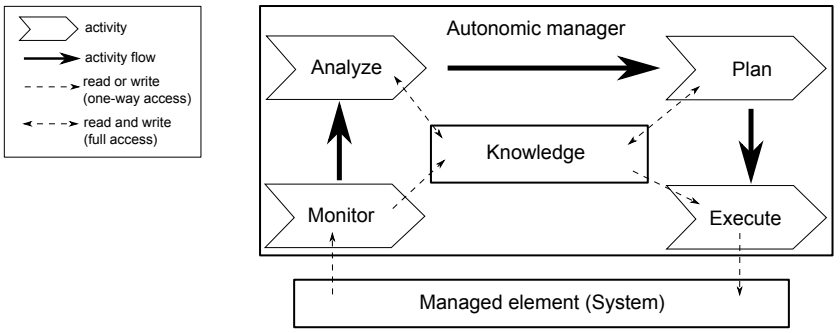

Fig. 9. MAPE-K feedback loop architecture according to [22]

3.3.1 The Feedback Loop in Systems with Runtime Models

Different types of software systems already adopt runtime models to control several aspects of their execution. In some cases, runtime models can be simply used to reconfigure system properties. In other cases, these models can be dynamically updated to reflect changes in the system and its context (the observable part of the surrounding environment).

The MAPE-K feedback loop [22], shown in Figure 9, emphasizes the role of feedback for autonomic computing. At first, it splits the system into a managed element (core system) and an autonomic manager (adaptation engine). It then defines four key activities that operate on the basis of a common knowledge base: *Monitoring*, *Analysis*, *Planning*, and *Execution*. *Monitoring* is primarily responsible for gathering raw data, such as measurements and events, about the state of the managed system. Analysis is used to interpret data collected by the monitoring activity and detects changes in the managed system that might warrant adaptation. Both monitored and analyzed data are used to update the knowledge base of the MAPE loop. Planning reasons over the knowledge base to identify how the managed system should adapt in response to their mutual changes. Execution applies the adaptations selected by the planning activity on the system.

The explicit consideration of runtime models leads to an extended MAPE-K architecture as depicted in Figure 10. A first major refinement is that now the adaptation engine also takes into account - in addition to the core system - its context and requirements as a knowledge base.

In this more refined view Monitoring is gathering raw data, such as measurements and events, about the state of the system and its context. Additionally, monitoring may recognize updates of the requirements. In any case, the accumulated knowledge is stored in the runtime models (M@RT). The Analysis interprets the collected data and detects changes to the system, context and/or requirements that might warrant adaptation. Then it updates the runtime models accordingly. The Planning employs the runtime models to reason about how the running system should adapt in response to changes. The Execution uses the runtime models as basis to realize planned adaptations.

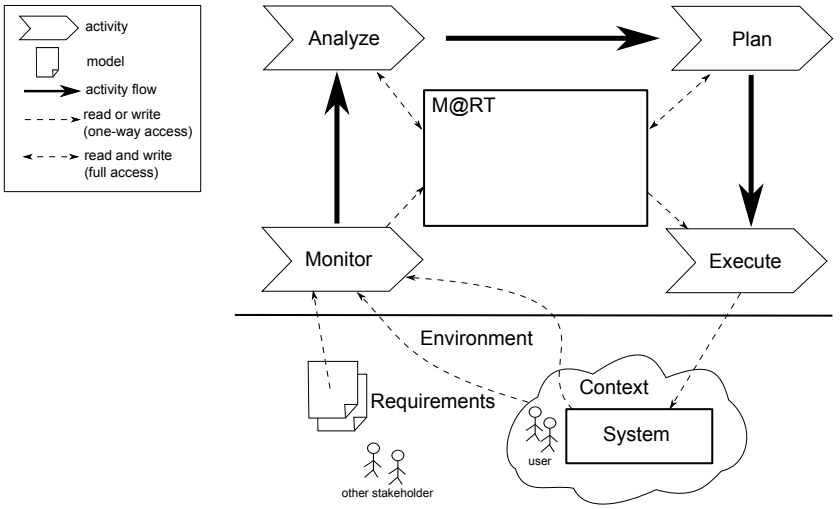

Fig. 10. Runtime Models in an extended MAPE-K architecture

3.3.2 Kinds of Runtime Models

As depicted in Figure 10, we can distinguish different kinds of runtime models depending on their possible original (i.e. subject):

System Models. The first and most common subject of a runtime model is the *system* itself. On the one hand, a runtime model provides an abstract view on the running system. Consequently, to maintain the causal relation, the runtime model has to be adjusted when a represented characteristics of its original changes [23,24]. On the other hand, the runtime model can be used to describe possible future configurations of the running system. Then, to realize the causal relation a related update of the running system has to be triggered. What can be controlled and observed via the runtime model is the system behaviour, as required sensors and actuators can be incorporated into the system if they minimally impact the system non-functional requirements.

Context Models. The *context* of the system – the part of the environment that can be observed by the system – can be a subject of a runtime model. Then, the runtime model represents some characteristics of the context observable via some sensors and the causal connection implies that the runtime model is accordingly updated when the context changes as indicated by changing measurements of the sensors. The case in which the runtime model is used to describe how the context should be changed is more subtle. Here only those changes that are under the indirect control of related actuators can be triggered via a runtime model and its causal connection. Often, only a small fraction of the context can be indirectly controlled via the actuators and sensors. Therefore, while only a few

characteristics are controllable, more characteristics are usually observable via sensors. However, sometimes relevant characteristics cannot be observed directly and then a dedicated analysis is required to derive them indirectly from other observations.

Requirement Models. Last but not least the *requirements* of the system may be subject of a runtime model [25,26]. In this case, either some form of online representation of the requirements exists that is linked to the runtime model by a causal connection or changes of the requirements have to be manually reflected on to the runtime model. In both cases the runtime model carries information about the currently relevant requirements within the system and therefore the system can, for example, check whether the current requirements are fulfilled or try to adjust its behaviour such that the fulfillment of the requirements increases. However, a bidirectional causal relation between the requirements and the runtime model has not been usually considered. This relation would trigger modification of the system requirements from changes in the runtime model. However, if the requirements define a whole set of possible goals for the system, the runtime model can be used to capture which goals are currently selected.

Besides these typical kinds of runtime models, in practice it is also possible to find cases where a single runtime model has multiple subjects. For example, a single model may reflect knowledge about a fraction of the system and the context at once in order to allow analyzing their interplay.

3.3.3 Types of Systems with Runtime Models

Different kinds of systems leverage activities of the MAPE loop to control some aspects of their execution. The rest of this section provides a non exhaustive list of the most common types of systems leveraging runtime models. Note also that these categories may overlap.

Configurable Systems. *Configurable systems* [27] are perhaps the simplest type of software systems that leverage runtime models. Such systems often use runtime models in form of configuration files to determine the concrete configuration and the values of operational parameters that control the behaviour of the overall system. For this reason, no monitoring and analysis process is performed to automatically update the runtime model. Instead, planning and execution processes respectively read the configuration and parameters stored in the runtime model and reconfigure the system accordingly.

Context-Awareness w.r.t. Pervasive Systems. *Context-awareness* [28] describes that a system is able to monitor its context. Context-awareness is regarded as an enabling feature for *pervasive systems* [29,30] that offers "anytime, anywhere, anyone" computing by integrating devices and appliances in the everyday lives of its users. Pervasive systems select and apply suitable adaptations depending on their context. As the user's activity and location are crucial for

many applications, context-awareness has been focused on location awareness and activity recognition. Pervasive systems can leverage runtime models to represent the context and cover all processes of the MAPE loop to foster adaptation. Monitoring acquires the necessary information about the context (e.g. using sensors to perceive a situation). Analysis abstracts and understands the context (e.g. matching a perceived sensory stimulus to a context) and updates the runtime models accordingly. Planning identifies the actions that the system should perform based on the recognized context and execution applies these actions at runtime.

Requirements-Aware Systems. *Requirement-awareness* is the capability of a system to identify changes to its own requirements. *Requirements-aware adaptive systems* [26,31] use runtime models to represent their requirements [32,33], track their changes [25,34] and trigger adaptation in the system behaviour in order to increase requirements satisfaction [35]. Other work [36] proposes to explicitly collect users' feedback during the lifetime of a system to assess the validity and the quality of a system behaviour as a means to meet the requirements. In these systems, requirements are conceived as first-class runtime entities that can be revised and reappraised over short periods of time. Modifications of requirements can be triggered due to different reasons, for example, by their varying satisfaction, or new/changing market needs and final users preferences.

These systems also leverage the activities of the MAPE loop to support requirements-awareness and adaptation. Monitoring collects the necessary data from the system and the context. In addition, if the system is *requirements-aware*, changes in the requirements are taken into consideration. Analysis uses the data about the system and context to update the requirements model or recompute the requirements satisfaction. Planning computes the adaptations to be performed by taking into account the current requirements and assumptions as captured by the runtime models. As a special case, this includes that a requirements changes may result in changes to the system itself (e.g., architectural or behavioural changes). Execution applies selected adaptations on the system.

It has to be emphasized that the use of requirements models at runtime during analysis and planning is conceptually independent of their monitoring (requirements-awareness).

Self-adaptive Systems. *Self-awareness* [37] is the capability of a system to monitor itself. The system can thus detect and reason on its internal changes (e.g., system behaviour, components, failures). *Self-adaptive systems* can in addition to self-awareness also react to observed changes by applying proper adaptations to themselves.

Nowadays, the term self-adaptive systems is used in a very broad sense and it can include self-awareness, context-awareness as well as requirements-awareness. Such systems manage different runtime models that represent the system itself, its context, and its requirements, respectively.

The next section explains how runtime models can be used to handle uncertainty. In particular, Section 4.3 provides further discussion of self-adaptive systems and the use of runtime models in the context of the case study and the MAPE-K loop.

4 Handling of Uncertainty

Nowadays, we can observe the trend to handle uncertainty later at runtime and not already at development-time, as discussed in Section 4.1. To better understand the benefits and drawbacks of using runtime models to handle uncertainty, we first discuss the classical approach to handle uncertainty using development-time models in Section 4.2. In Section 4.3 we explain more advanced solutions to handle uncertainty at runtime and outline how these solutions can benefit from runtime models. The case study has been used to provide specific examples.

4.1 Trend of Handling Uncertainty Later

In classical engineering, uncertainty in the available information about the system and its environment is a major problem. In particular, for models that capture characteristics of the environment it is frequently the case that the exact characteristics are not known at development-time. External techniques for uncertainty for a model such as safety-margins and robustness with respect to these known margins are then often employed to ensure that the developed solutions satisfy the system goals for all expected circumstances (cf. [38]).

Consequently, also in software engineering the classical approach is to build systems that work under all expected circumstances. This is achieved by using models at development-time, which capture the uncertainty internally. Alternatively external techniques can be employed to handle the uncertainty, such that the developed systems work under all circumstances predicted by these development-time models.

However, nowadays it has been recognized that we can achieve smarter and more efficient solutions, when we build systems that are context-aware [28] and/or self-aware [37]. Due to the self-awareness, context-awareness and requirement-awareness, self-adaptive systems become capable of adjusting their structure and behaviour to the specific needs of the current system state, context, and/or requirements. This results in a number of benefits: (1) achievement of better performance and less resource consumption at the same time, (2) minimization of manual adjustments required to the administrators or users, and (3) provisioning of functionality that would be infeasible without the information about the context.

In contrast to the classical software engineering approach, in self-adaptive systems uncertainty concerning the system or context can be handled - to some extent - at runtime and not just at development-time. The classical software engineering approach can only cope with uncertainty that can be handled based on reasonably complex development-time models. A self-adaptive system can in

contrast employ runtime measurements to reduce the uncertainty and adjust its behaviour accordingly. Consequently, self-adaptive systems can handle more situations than classical solutions and their ability to address uncertainty more actively is one of their major advantages. In [39] it is therefore argued that uncertainty should be considered as a first class element when designing self-adaptive systems.

4.2 Handling Uncertainty at Development-Time

This section discusses how models can be used to address uncertainty during development-time. We describe the classical approach, through our case study and then explain the various forms of uncertainty that can arise.

The classical approach tries to exclude uncertainty at the level of requirements in order to have a solid basis for the later development activities. However, it has been observed that stable requirements rarely exist on the long run (cf. [40,41,42]). Watts Humphrey observed that one of the principles of software engineering is that requirements are inherently uncertain [40]: "This creative design process is complicated by the generally poor status of most requirements descriptions. This is not because the users or the system's designers are incompetent but because of what I call the requirements uncertainty principle: For a new software system, the requirements will not be completely known until after the users have used it." Also Lehman's Software Uncertainty Principle [41] states that for a variable-type program, despite many past satisfactory executions, a new execution may yield unsatisfactory results. This is based on the observation that software inevitably reflects assumptions about the real world [43].

During design and implementation, the uncertainty in environment models is usually handled by building robust solutions that simply work for all possible cases. Therefore, the development-time model employed for the environment has to capture all relevant and possible future environments the system will face. In this way, a system designed according to a development-time model should guarantee that in any relevant and possible future environment the required goals and constraints can be satisfied.

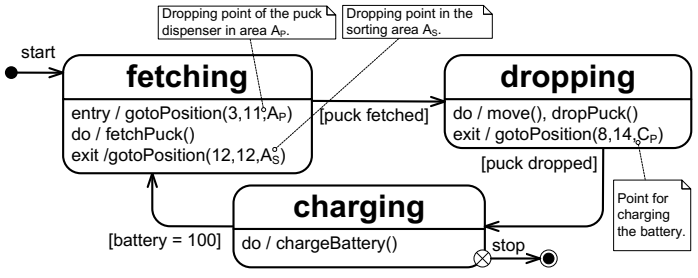

Fig. 11. Behaviour development model of the robot system

Example 1 (Robot Design with Development-Time Models Only). For our case study, the developer must make several design decisions. If we assume that each of our three robots has a clear task and that their overall behaviour fulfills the given goals, a possible solution is the fixed encoding of the different tasks.

According to Figure 2 on page 51, robot R_P must transport pucks from the Packaging Room *to the* Sorting Room *in the specific Areas A_P and A_S respectively. We can model at development-time a state machine that representes the states and transitions necessary for the robot to solve this task. The behaviour model depicted in Figure 11 has three states. First, the robot fetches the puck in the* Packaging Room *(cf. Fig. 2) and transports it according to the sorting position. Due to our discrete grid map model of the laboratory, we can pinpoint the target locations for the robot navigation. The developer knows the maximal distance and whether the robot has enough power for puck transportation. Afterwards, the robot always loads its battery at a fixed charging point avoiding to exhaust its power supply, which is one of the constraints of the system. Furthermore, one can stop the puck transportation while the robot loads the battery or it will fetch and transport the next puck.*

There are many restrictions to the environment, e.g., pucks must always be at the same position, the environment must be very static without relevant disturbances, and goals should not change at runtime ensuring that such a fixed transportation scenario works. This is the case for many systems, such as an assembly line with static working steps in a fixed area. In practice the robot will increasingly diverge over time from the planned trajectory as the move commands are inherently imprecise. In fact only in more restricted cases for embedded systems, such as automation systems with a fixed track layout where the errors of the vehicle movements do not accumulate over time, a solution that is not context-aware really works.

In our specific system design, the handling of the battery loading is one example of resolving uncertainty during development-time. It does not matter how much the power level of the battery is decreased during the task as long as the robot reaches the charging point. Furthermore, the amount can be estimated or measured upfront and results in a simple, not context-aware system solution. Additionally, the fixed encoding of the task further reduces uncertainty due to the fact that no communication between robots and runtime task distribution capabilities are needed.

All the development models are only used for code generation or system implementation. The running system does not reflect or use these models but simply complies with them.

4.2.1 Forms of Uncertainty

The problems of uncertain requirements according to [40,41,42] relate to *epistemic uncertainty* where the requirements captured at development-time may not precisely and accurately reflect the real needs when the system is operating. In the case of the Example 1, the designed behaviour is not able to handle a shift in the priorities of the goals by the operating organization that may occur

over time. Thus, in this case the performance will not be rated as good as in the beginning when the shift in the prioritization of the goals by the operating organization has occurred, as expectations have evolved while the system behaviour stays the same. As a result, a new state machine model must be developed and deployed to the robot. Another related reason for *epistemic uncertainty* is that stakeholders may formulate ambiguous requirements specifications [2], or they may have conflicting interests or uncertain expectations on the resulting software quality. The changes that will occur for the system and environment in between the development-time and when the system is executing may also result in *epistemic uncertainty*. The development-time model of the system or context cannot precisely and accurately reflect the real system and its environment as it is characterized later when the system is under operation.

Also, practical limitations in development and measurement tools can, in principle, cause *epistemic uncertainty* where a development-time model of the system or context cannot precisely and accurately reflect the real system and its execution environment as it is known at development-time. For instance, in the Example 1, it is only possible to measure the initial characteristics of the floor plan with a certain precision and accuracy and at certain points in time. As a result, a development-time model of the floor plan may perhaps never truly reflect the real environment unless, due to abstraction, neither the measurement precision or changes matter after the measurement are relevant.

Furthermore, *randomization* plays a role that may be covered appropriately by probabilistic models such as probabilistic automatas [44]. In our Example 1 the known likelihood of failed operations can be described by probabilistic transitions and still we can determine upper bounds for related unsafe behaviour using probabilistic reachability analysis. If no exact probabilities but rather only probability intervals are known due to epistemic uncertainty, interval probabilistic automata and related analysis techniques (cf. [45]) could still be used.

4.2.2 Time and Uncertainty

For a development-time model, we can observe that the uncertainty may change over time. It may stay stable, increase or even, in rare cases, decrease over time. If the energy level maintained by the robot in our Example 1 changes over time and a design-time model is used, the three cases mentioned above can result in the following situations:

The uncertainty is *increasing over time* as outlined before, which may happen if the set of possible reached states grows over time. If for all activities of the robot only rough upper and lower bounds for the energy consumption are known, after a number of operations, the uncertainty concerning the consumed energy will be quite large. In this case, the model still provides enough information about the system at development-time to build a sufficiently robust design for the system by simply calculating the worst-case and therefore act accordingly. However, the resulting behaviour will be rather sub-optimal as the robot will recharge the battery very early.

The uncertainty *remains constant over time* if the set of possible reached states remains the same size over time. In the case of the robot and the energy consumption this would require that the energy consumption of each operation is exactly known such that the initial uncertainty concerning the state of the battery is neither increasing nor reduced. In this unrealistic case this knowledge can be exploited to build a robust design where the initial worst-case plus the exactly known consumption is employed to determine when to recharge the battery.

The uncertainty is *reduced over time* when after a certain time the state is exactly known as the set of possible reached set of states has collapsed into a smaller set or even a single state. In our example, if loading the battery is blocked we could be certain that after a while the battery will be empty. However, it is rarely the case that such a decrease of the uncertainty in a model can be guaranteed.

While all three cases are possible, we can conclude that unless actively tackled, the uncertainty will increase over time. Consequently, identifying means to counteract this growth in uncertainty is crucial for handling uncertainty properly.

4.2.3 Maintenance and Reengineering for Handling Uncertainty

The standard approach to tackle the aging problem for software is *maintenance*, where required adjustments to changes in the context or requirements are taken into account by a dedicated additional development step. Since the development of the original system has been stopped, in this step the changes in the context and requirements should be identified such that the related uncertainties are reduced. However, often the time for a maintenance step is rather limited and therefore the related analysis is superficial and potentially incomplete. Also, maintenance teams might differ from design and implementation teams, possibly leading to more uncertainty in the form of incomplete understanding.

If the internal quality of the software deteriorates considerably, maintenance is no longer feasible and, instead, dedicated reengineering activities with a reverse-engineering part that recovers lost information about the software and a forward engineering phase are required. Here, reengineering addresses the uncertainty that results from the loss of information concerning the system. Usually, reengineering also has to address changes in the context and requirements since the development of the original system has been stopped to minimize related uncertainties.

4.3 Handling Uncertainty at Runtime

Using runtime models during system operation, as for example it is done for self-adaptive systems, brings up different forms of uncertainty that must be handled. In contrast to the classical software engineering approach, these forms of uncertainty are tackled at runtime and not at development-time.

Before discussing the different forms of uncertainty at runtime, we illustrate in the following example the use of runtime models for the planning step of the MAPE-K feedback loop for our robot case study.

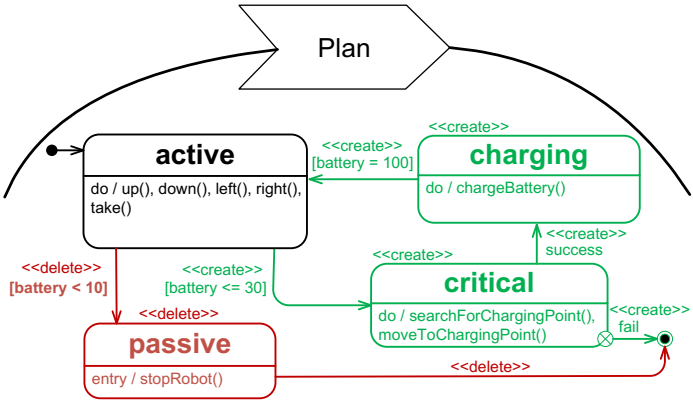

Fig. 12. The planning step of the MAPE-K loop creates a new version of the state machine runtime model

Example 2 (Robot Scenario with Runtime Models). We use the following runtime models in our case study scenario: First, we have an environment model that captures the current position in the floor plan represented by discrete fields in an overall map with current puck positions, obstacles and charging points (cf. Fig. 6 in Section 3) and available puck transportation requests. Second, a goal runtime model exists that includes priorities and constraints of our system and reflects the requirements of the overall system behaviour. Furthermore, the behaviour of the robot should fulfill those constraints and performs according to the given goals and priorities. The abstract syntax as well as a concrete instance situation are depicted in Fig. 3 and 4. The current state information of a robot during operation is represented by a finite state machine runtime model (cf. Fig. 8).

These runtime models can be used and changed within the different activities of the MAPE-K loop. In the following, we assume that the robot system has at least self-awareness capabilities concerning its battery functionality including the current battery level and is context-aware for its current goals. We consider the following exemplary execution steps inside one robot: the monitor activity retrieves information about the current map situation, the possible available pool of operations, the current behaviour specification in the form of a state machine as well as a goal model. It updates this information in the runtime model representation of the robot.

In a next step, the analyze activity is aware of the goals and constraints and it is able to conclude that the battery should not be exhausted. Furthermore, it detects in the possible operation pool of the robot the battery charging capability. As

a result of applying strategies to fulfill the goals, it decides to include a charging mechanism into the robot behaviour. The following third activity of the MAPE approach is the planning step according to the decision of the analyze step before. For example, the planner can adapt the retrieved simple state machine (representing the robot behaviour) depicted in Figure 8. It comes up with a detailed plan to adapt the behaviour from the current state to the envisioned one. An excerpt of the new adapted state machine is shown in Fig. 12. The passive *state as well as ingoing and outgoing transitions are marked as to be deleted and two states* charging *and* critical *(with underlying behaviour) must be created. New transitions reflect the constraints from the goal model. In this case, the robot processes incoming puck transportation tasks in the* active *state. If the battery power level drops below 30 percent, the robot behaviour changes by entering the* critical *state. There, it searches for a charging point nearby and tries to reach it. If the robot successfully reaches one, it charges the battery until it is full in the* charging *state and returns to normal execution behaviour. Otherwise, the battery is exhausted and the robot stops the execution of the task.*

Afterwards, the last activity in the MAPE-K loop takes the plan and applies the planned changes to the real system (it synchronizes the updated runtime model with the real system). After a successful update, the behaviour of the real robot is adapted according to the current goals and constraints.

4.3.1 Forms of Uncertainty

For the case of runtime models *epistemic uncertainty* can arise from multiple sources that include sensing - when building or updating the models - and the passing of time. If the original of a runtime model is monitored at runtime, the resulting update of the corresponding runtime model should, in principle, lead to less uncertainty. This effect may be limited because the measurements at runtime usually include randomized errors and they are limited concerning the *when* and *what*. Looking at *epistemic uncertainty* in our Example 2, retrieval of environment information and subsequent update of the runtime model with information about which fields are blocked as well as the positions of the charging points based on measurements by the robot, would help reduce the level of uncertainty. However, there are details we should take into account as, for example, if the blocked fields change frequently, the effect of reducing the uncertainty would only be temporary. In contrast, if we continuously update the information concerning the location of permanent obstacles (e.g., walls) in the robot of Example 2, the robot will - on the long run, after it has explored the whole area - derive a sufficient floor plan of those permanent obstacles. This will contain only the unavoidable uncertainty due to measurement errors.

If a runtime model is also employed to store the planned changes, its state is somehow permanently evaluated against the original. This should, in principle, lead to a high consistency between the two and therefore a lower level of uncertainty. However, this effect may be limited as also changing the original may include a randomized error due to actuator errors.

When we are able to learn what can happen in the environment, we may be even able to improve the prediction of behaviour for the case of uncertainty associated with *randomization*. For example, in the case of the robot scenario, we may learn how likely temporal blocks occur for certain fields. If a certain transition relates to an activity such as a measurement that fails with a given probability, we may be able to learn the probability for a longer sequence of measurements, as it will be very likely that the number of failed attempts to take the transition divided by the total number of tries converges towards the failure probability. However, there may be problems while using this kind of assumptions. For example, if the assumption that the observed phenomena is related to a probability is not correct, we will likely see no convergence and thus the learning will fail.

The case of *linguistic ambiguities* is slightly different since it covers the cases when the concepts in the model are not known in a precise way. Approaches such as fuzzy automata can be used to deal with this issue. They include a fuzzification and de-fuzzification of the linguistic concepts which allows them to handle this form of uncertainty, while complicating the analysis considerably [46]. If we consider the soft goals in Figure 3 that describe a *low energy consumption* and *high throughput*, these constraints could be good candidates to be specified in a fuzzy automata. Note that the translation of those goals from a linguistic concept to a manageable runtime model may introduce additional uncertainty.

Unfortunately, the sources of uncertainty described above rarely occur independently from each other. Instead, the effects produced by these sources of uncertainty can compound and thereby inhibit the system from clearly assessing the extent to which it satisfies its requirements. Therefore, solutions to tackle composed sources of uncertainty are required. For example, having temporary blocks of certain fields arising in the robot scenario can be related to *randomization* as well as *epistemic uncertainty* and therefore, tackling this issue by only learning probability values would not be enough. Instead, intervals for the probabilities as provided by interval probabilistic automata would be required.

4.3.2 Time and Uncertainty

Furthermore, as in the case of development-time models, for runtime models we can also observe that uncertainty may stay stable, decrease or even increase over time. In case of the energy level maintained by the robot that changes over time, the three cases can result in the following situations:

Even if certain parameters are measured, the uncertainty *increases over time* as the parameters that are not updated over time may also be uncertain. In this case the runtime model represents a partial view that may not provide enough information of the system at runtime to be able to support a solution to cope with the situation. As an example, currently we have not considered in a runtime model that hardware parts of the robot can be worn out. Therefore, the uncertainty about the status of those parts is not handled in any activity of the MAPE-K loop. We can only assume that the quality of the hardware parts

decreases over time but as long as those parts are not broken or fail, we cannot reason about the impact on the robot's behaviour.

The uncertainty remains *constant over time* if the measurements are sufficient to keep the uncertainty within certain bounds. In this case the runtime model can be exploited to chose a proper behaviour that works with the captured circumstances. In the Example 2, the battery level is measured periodically to decide when it is time to load it. We can cope with two constant uncertainty issues in this example. First, if we know the period of the measurement, we can estimate lower and upper bounds of energy decreasing for that specific time slot. Secondly, the used hardware sensor and the runtime model representation has a certain precision that is known upfront and stay in a bound too (assuming that the sensor works correctly).

The uncertainty is *reduced over time* if measurements collect information about the system status and step-by-step increase the accuracy of the corresponding model representation of the system. As an effect, this will reduce the uncertainty over time. Usually, there is saturation of this effect after a while and a certain level of the uncertainty remains (see former case). Otherwise, after a certain time the uncertainty would have been completely eliminated and the characteristics of the original as far as covered by the model are exactly known. A very simple example is the exhaustion of the battery. In that specific case, there is no uncertainty and we exactly know that the battery is empty (even if this is not very helpful).

4.3.3 Feedback Loops and Uncertainty

In classical engineering feedback loops are a well known solution to address uncertainty concerning the environment [47]. Consequently, feedback loops have also been identified as the core element for engineering self-adaptive systems [48]. See also [49] for a discussion of uncertainty in autonomic systems with feedback loops. Therefore, the role of uncertainty in systems with runtime models and related concepts such as self-awareness, context-awareness, and requirement-awareness is best discussed referring to the extended architecture of a feedback loop as outlined in Figure 9.

As explained earlier, the more detailed view of the architecture comprises four key activities which are Monitoring, Analysis, Planning, and Execution (see Figure 13). Each of these activities can be seen as relying on the use of runtime models that serve as a knowledge-base. The runtime models of the system, context, and requirements plus additional strategic knowledge can then be seen as driving the feedback loop.

The basic MAPE-K architecture can be extended to leverage models that can evolve, thereby enabling a software system to cope with uncertainty by learning new properties about itself and its execution environment based on monitoring information that can only be collected at runtime. Specifically, to gradually reduce the level of uncertainty in the system, the four key processes in the MAPE-K architecture can analyze system and environmental data in order

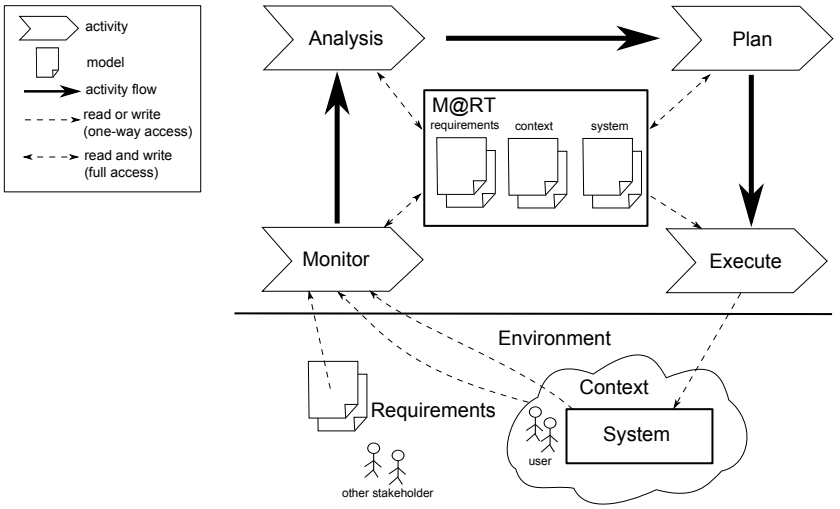

Fig. 13. Resolving uncertainty at run time with an extended MAPE-K architecture

to refine, augment, and revise the information stored in the runtime models, and then leverage that information to guide the adaptation process as necessary.

Next, we will review the objective of each of the four key activities and explore how uncertainty affects each of them. We will also review identified research questions associated with each phase of the MAPE-K loop that can potentially be tackled with the use of runtime models.

Monitor. In the detailed view on the architecture proposed in Figure 14, the monitoring process is primarily responsible for *measuring* raw data, about the current state and/or occurring events of the system, the context, and the requirements, and to *update* the runtime models representing the knowledge about the state, context and requirements.

The monitoring helps the software system to cope with uncertainty by continuously updating the information contained in a runtime model. Nevertheless, as explained above, the sensors used to obtain this monitoring information are limited by the precision and accuracy of their measurements. Moreover, sensors may fail at runtime or report values that the software system may simply be unable to interpret. As a result, even if monitoring reduces the level of uncertainty in a software system, it will depend on its accuracy, precision and frequency. The information it provides ultimately reflects an approximation that may contain some uncertainty.

Research questions associated with the *Monitoring* phase of the MAPE-K loop that can potentially be tackled with the use of runtime models and related to the fact that sensing and monitoring can be imprecise and partial are: *How can we determine the imprecision caused by temporal constraints / delays? Does the monitor engine also need to adapt (i.e., monitoring periods)? How can runtime*

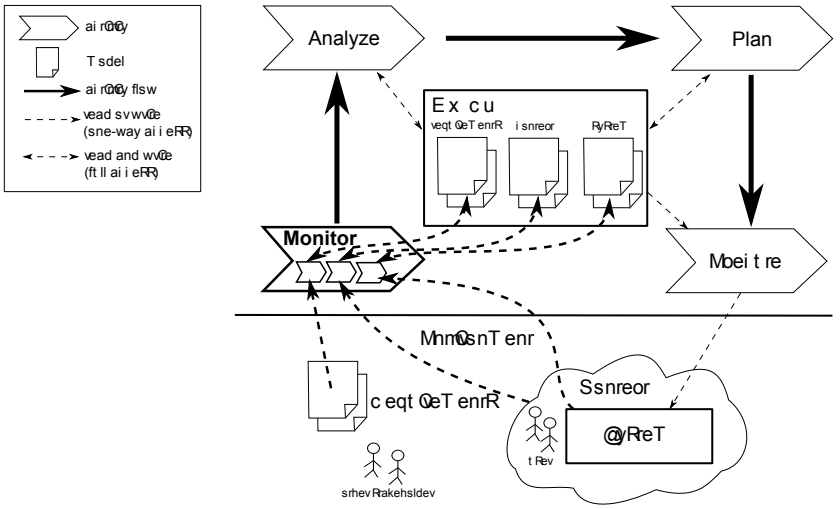

Fig. 14. Monitoring and resolving uncertainty with runtime models

models incorporate or learn new information using machine learning techniques and preserving at the same time the system under a reasonable behaviour? How does the runtime model represent what to monitor and how to do it?

Analyze. The architecture uses the analysis process to interpret data collected by the monitoring process and detect system and environmental changes that might warrant adaptation (see Figure 15).

In case more subtle updates are required, the analysis may, in addition to the monitoring, take as input the most recent data available as well as older data to obtain more accurate initial analysis models of the system and the environment. For example, techniques such as smoothening may allow better capturing of what is known about the system or environment than simply using the last measurement. In this context the complex update can be seen as a learning step that uses the observations made to provide a better runtime model. Accordingly, the employed learning/update strategy can have high impact on how successful the uncertainty is reduced. While more specific strategies may provide highly accurate runtime models, unless severe changes in the system or environment occur, more generic strategies may provide more robustness but solutions that performe worse.

In addition, the analysis activity verifies in a second step that monitoring information satisfies the requirements given, for example, in the form of relevant system and environmental goals and constraints. If necessary, this process also has to trigger an adaptation by the subsequent planning process if it detects that a requirement is, or could potentially become, unsatisfied. Moreover this analysis step concerning the goals and constraints is highly affected by the uncertainty in the runtime models. The analysis step can here only result in the

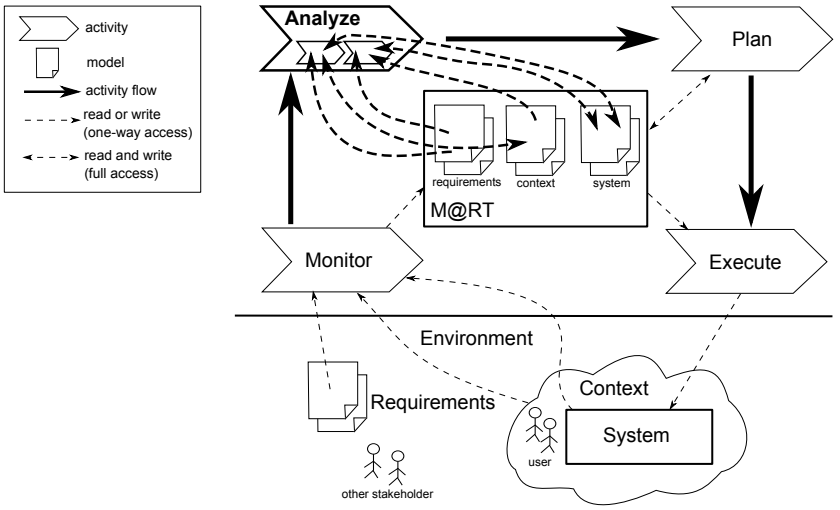

Fig. 15. Analysis and resolving uncertainty with runtime models

required precision and accuracy when the uncertainty present in the runtime model can be successfully handled and therefore, does not make an analysis infeasible. Depending on how the analysis results are presented as models, these runtime models can, due to a higher level of abstraction, contain considerable less uncertainty than what could have been observed for the monitoring.

As diagnosis and analysis can also be ambiguous and imprecise, the following research questions associated with the *Analysis* phase of the MAPE-K loop arise: *What is the effect of analysis techniques to resolve unknown uncertainty? Does the perspective on what is "relevant" for the runtime model need to adapt at run time? Should the criteria for decision-making adapt itself? How can we retain the ability of analysis even if we have incorporated newly learned information? How do the objectives of the analysis are represented and how can they be mapped onto a specific analysis technique?*

Plan. The Planning, if triggered, reads the runtime models enriched by the analysis and performs some reasoning to identify how the running system should be best adapted to changes of the system, context, and/or requirements. It may, for example, identify a plan to change the running system to also take into account a novel system goal.

The planning activity therefore reads the system, context and requirement runtime models and records the planned changes also in the form of a runtime model (see Figure 16). The identified changes can, for example, be captured by modifications of a runtime model of the system that we call $system'$ (see the entity $system'$ in Figure 16). Here, uncertainty only plays a role when identified changes cannot be safely planned as the related current state of the system is not available in the current runtime model of the system. The precision and

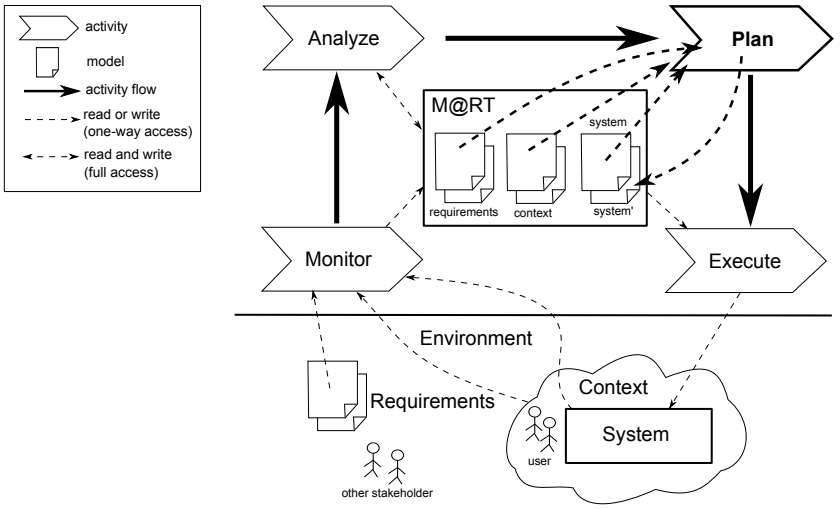

Fig. 16. Planning and resolving uncertainty with runtime models

accuracy of the planning is again determined by the uncertainty of the employed runtime models. For the case of the resulting prescriptive runtime model of the system, the runtime model usually captures only what should be changed and thus will not include any uncertainty at all.

For the *Planning* phase of the MAPE-K loop holds that the final outcome of an applied strategy cannot be accurately predicted and thus the following research questions result: *Should the planner take uncertainty into account? If so, how does it handles strategies when uncertainty exist or what kind of runtime models are more suitable? Furthermore, how can the planning activity be instrumented to achieve a specific objective (e.g., minimise the number of changes to be applied onto the system).*

Execute. The Execution activity directly applies a set of changes for the running system stored in some runtime models (see *system′* in Figure 17) by the planner. Even if these changes can have the direct consequences limited to the system itself, the changes may also indirectly affect the context in the longer run. The execute activity can be seen as the mechanism that support the causal connection which influences the running system according to the updated runtime models.

Assuming that applying the changes always works, the execute activity would guarantee that the employed runtime model of the system is now perfectly in sync with the system. However, in practice the execution activity cannot give such guarantees as its actuators provide only limited accuracy or may even completely fail. Furthermore, the system may have evolved in parallel to the feedback loop such that the planned updates become impossible or the changes do not result exactly in the planned outcome. Overcoming this problem, the

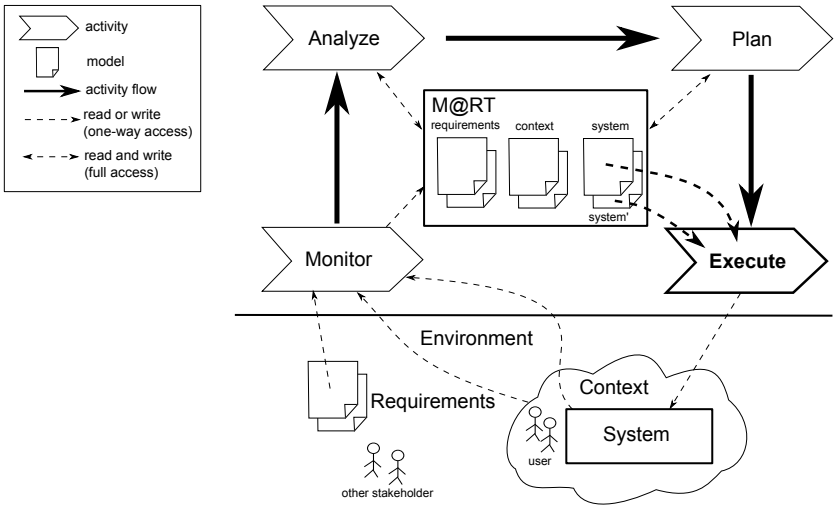

Fig. 17. Executing and resolving uncertainty with runtime models

next loop iteration should detect it and try to solve the inconsistencies again, or otherwise one can try to exclude it in such a way that changes occurring in the managed system can never result in inconsistencies with planned changes.

The effects of the *Execution* phase of the MAPE-K loop on the running system may not be as expected as also external influences as for example, disturbances from the environment or user interactions may exist which are not under control of the MAPE-K loop. Therefore, the following research questions have to be addressed: *Do temporal delays create an inconsistent view of the runtime model? How are the runtime models affected by external influences outside the MAPE-K loop?*

Summary. Uncertainty in self-adaptive systems can arise from multiple sources that include, but are not limited to, the system itself, its environment, and its stakeholders. For instance, the system itself uses its monitoring infrastructure, which may be inaccurate and imprecise, to measure properties about itself (self-awareness). Similarly, the surrounding context can introduce uncertainty because it is dynamic, unpredictable and ever changing, perhaps even leading to the violation of domain assumptions (context-awareness). Lastly, stakeholders can also introduce uncertainty by either modifying the current set of requirements that the system must satisfy, or by the emergence of new business needs or regulations that the system must comply with (requirement-awareness).

The uncertainty in the runtime models usually increases when time passes as the monitored information becomes outdated after a while as discussed in Section 4.3.2. A more frequent execution of the feedback loop can counteract this tendency and also guarantee faster adaptation reactions. However, the chosen frequency has to be cost-effective since it must balance a trade-off between the

quality of the feedback loop and the overhead added to the execution of the system.

Also, a suitable trade-off decision has to be made concerning the accuracy of the runtime models. It is usually not cost-effective to monitor frequently as this not only increases the monitoring costs but also the subsequent analysis and planning activities. Therefore, enough monitoring to enable a suitable analysis and planning is necessary. Again, as in case of the updating/learning strategies, we have here also a trade-off between well-performing solutions within an envelope of expected likely changes and a robust solution. While for the former the required effort can be optimized, for the latter more overhead has to be accepted. Adjusting different activities such as monitoring, analysis and planning that are covered by the runtime models during system operation of higher-level adaptation loops (see Section 4.3.4) enables solutions where the overhead for robustness can be reduced by adjusting and intensifying the specific activities when necessary rather than always run them with a maximal overhead.

4.3.4 Types of Systems with Runtime Models

In the following, we will discuss the implications of using runtime models for different classes of systems ranging from configurable systems to those with full self-managing capabilities.

Configurable Systems. The simplest case of an adaptation loop (not a complete MAPE-K loop) is a runtime model, in which an externally initiated update triggers an adaptation of the system.

This kind of systems is neither self-aware nor context-aware and does not monitor the system itself or the environment. A requirement of self-awareness is that the system must consider itself at a higher level of abstraction. Instead, configurable systems do not actively change themselves and thus the required adaptation triggered by the external update can simply be enforced from outside the running system.

As a typical case, the user configures or changes requirements at runtime. A very simple configurable system takes the potentially updated runtime requirements model and checks it during operation. If constraints are not fulfilled an exception is thrown. Otherwise, the system will perform according to the given configuration parameters. A more elaborated version is that the updated runtime requirement model is used to derive the required behaviour. For example, the system selects a strategy with a good/optimal expected revenue by evaluating possible alternative behaviours and chooses the best performing one according to the actual requirements.

Example 3 (Configurable Robot Scenario with Runtime Models).

In this system scenario, the robot is neither self-aware nor context-aware. For this reason it cannot sense the environment. In this case, the runtime model is a valid map configuration that informs the robot about the actual position of pucks and obstacles. The robot gets an initial model of the environment according to

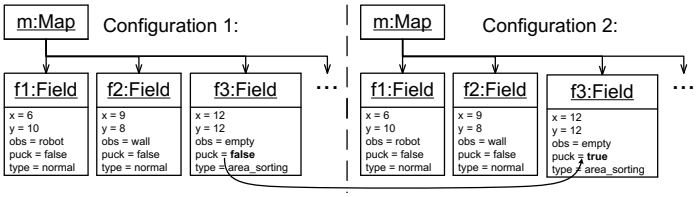

Fig. 18. Static map and instance situation with two possible configuration for the Example 3

the metamodel in Fig. 6 in Sec. 3, maintains all fields in this map and uses the given instance situation for navigation as well as fulfilling its goals. A snapshot of two (partial) environment configurations is depicted in Fig. 18. In this case, a puck is placed on field f_3.

The robot behaves according to the simple state machine in Fig. 19. Each time the robot enters the active *state, it checks the current map configuration data (*loadCurrentMap()*), searches for pucks to transport and calculates routes accordingly. We assume that the parameters in the configuration are valid and triggered from outside. In the* critical *and* charging *states, no adaptation is possible in this example.*

Each change in the map influences the behaviour of the robot because each one must reschedule the transportation tasks or recalculate routes. We have three robots in our scenario (cf. Fig. 2). The R_P *robot transports pucks from the packaging room to the sorting room and this causes a change in the map of the example. The second robot* R_S *is affected by the map change because of a new incoming transportation task. The third robot, which transports pucks to the stock, can simply ignore the change of the map or the overall system does not update the local map of this robot.*

The configuration of the system can easily be extended to other runtime models. However, if the system has to deal with very frequent changes, which causes, for example, a map update, the robots have to read the configuration file and possibly change their behaviour too often. Consequently, this solution is only applicable to rather static environments such as assembly lines with fixed mounted

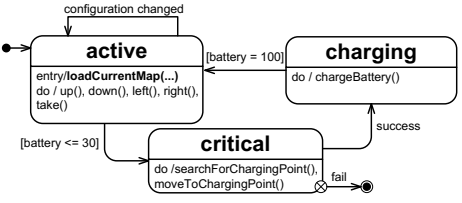

Fig. 19. State machine of a configurable robotic system

robot arms or runtime data that does not change often over time (e.g., the softgoals throughput and low energy consumption).

Context-Awareness and/or Pervasive Systems. According to our definition of context-awareness provided in Section 3.3.3, a context-aware system that is not self-aware does not monitor the system itself but only the environment. Therefore, it usually requires that the system characteristics of interest do not change and thus the required adaptation according to the observed changes in the environment can be simply enforced without taking any changes of the system itself into account.

Example 4 (Context-Aware Robot Scenario with Runtime Models). A context-aware version of the Example 3 has additional sensing capabilities for monitoring the environment. As a result, it continuously corrects the internal environment model according to the measurements and needs no external trigger for updating the map. Additionally to the error correction of the map, the robot corrects its position over time to reduce the error introduced by the wheel actuators.

At deployment time, the robot system gets the same static map instance situation as in Example 3 (cf. Fig. 18). But now, it searches for pucks and reschedule the transportation tasks by itself. The context-aware state machine is shown in Fig. 20. The new sensing, updateMap, searchPuck() *and* correctPositionInMap() *functions are the context-aware parts of the robot and influence the behaviour, e.g., by updating the internal map and a better path planning with less uncertainty over time due to the better environment sensing capabilities.*

Requirement-Aware Systems. These kind of systems conceive requirements as first class entities in the runtime models (c.f. Section 3.3.3). They take care of changes of their own requirements as well as track them over time. According to current constraints or varying needs, the systems adapt the behaviour to fulfill current requirements.

Example 5 (Requirement-Aware Robot Scenario with Runtime Models). In the previous Examples 3 and 4, we have the implicit assumption that the behaviour of the robot system always conforms to the given goals. In a requirement-aware adaptive system, these goals can be considered explicitly. In this example, the runtime model is a valid goal configuration that influences the behaviour of the

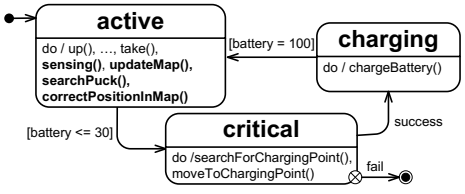

Fig. 20. State machine of a context-aware robotic system

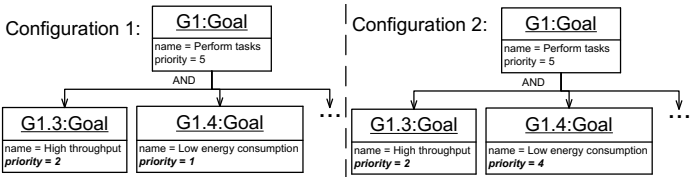

Fig. 21. Two goal configurations for the requirement-aware robotic system

robot. A snapshot of the partial goal configurations is depicted in Fig. 21. The difference between the left and right configuration in the picture is the priority of saving energy during task execution. The robot behaves according to the state machine in Fig. 22. Each time the robot enters the active *state, it checks the current configuration data and calculates the normal drive speed accordingly (*calculateSpeed(config)*). We assume that the parameters in the configuration are valid and triggered from outside. In the* critical *and* charging *states, no adaptation is possible in this example.*

For the two configurations in Fig. 21, the speed of the robot might be much higher for configuration 1 *than for the second one, because of the different priorities of the goals. Saving energy has a higher priority in the second goal configuration, which implies (among other changes) a reduction of the movement speed to an optimal power saving level.*

Self-adaptive Systems. Self-adaptive systems as introduced in Sec. 3.3.3 use the feedback loop to identify and compensate several changes in the system or environment. In essence, it provides the capability to live with the uncertainty related to the changes in the system or environment. They adjust to specific current needs of the different situations that can be identified at runtime.

Example 6 (Self-Adaptive Robot Scenario with Runtime Models).

A self-adaptive version of our robotic scenario extends the monitor activity from the context-aware system in Example 4. The analysis and planning steps are also extended. More precisely, the system runtime models are now an environment model (the first version is initially loaded), a goal model with constraints,

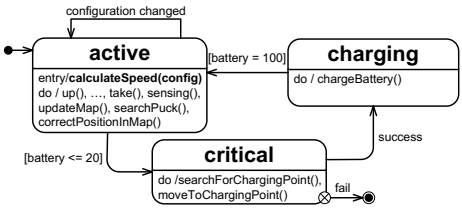

Fig. 22. State machine of a requirement-aware robotic system

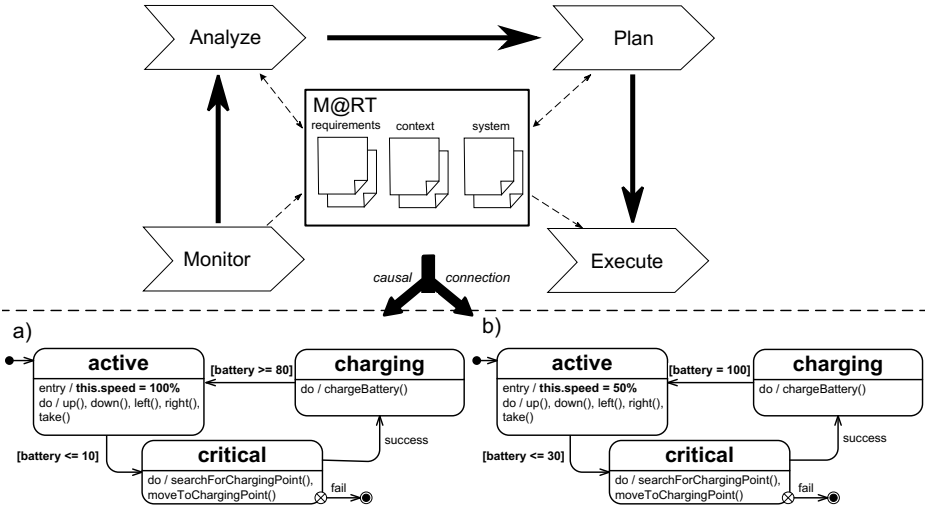

Fig. 23. Self-Adaptive robotic system scenario. The MAPE-K loop has a casual connection to the runtime model in form of a state machine. This state machine is adapted over time from variant a) to b).

and an initial behaviour model in form of a finite state machine. The runtime models are not static in this scenario, which is different from Examples 3 and 4.

Due to the sensing capabilities, the robot system can update its position in the map during runtime observations. Furthermore, it is aware of its requirements and goals. During the analysis and planning step, the self-adaptive system generates a state machine according to the current instance situation and constraints as depicted in Fig. 23.

Let us assume, the analysis step is aware of the possible configuration space of the behaviour parameter in the robot. The first goal parameters can look like the right instance situation in Fig. 21. The subgoal High throughput *has a higher priority than the subgoal* Low energy consumption*. Therefore the planning step can generate a new behaviour model or adapt the existing one as depicted on the lower left in Fig 23 (state machine (a)). Here, the driving speed is set to a maximum and the robot moves are risky because of the reduced battery safety margins (only if battery is lower than 10 percent, the critical behaviour state is entered). Additionally, loading takes a lot of time (especially the last 20 percent) so that this timing behaviour is optimized in a second step. The overall behaviour of the robot must still guarantee that other constraints, e.g., exhausting the battery, are fulfilled during the execution.*

The robot can perform its task according to this specialized state machine until the goals change. Goals can be changed by the user or the system itself. For example, the system can adapt its strategy according to the monitored environment information.

Let us assume, the goals change to the situation as depicted in the second configuration of Fig. 21. Now, the energy saving subgoal has a much higher priority. This is sensed and updated in the runtime model by the monitor step. The analyze activity decides that a behaviour adaptation is necessary and the planning step tries to fulfill the new constraints. In this case, the MAPE-K loop will generate/ adapt the existing state machine as depicted on the right in Fig. 23 (state machine (b)). The new behaviour model (state machine) uses a fix drive speed of 50 percent, which is much more energy efficient than before. Additionally, the safety margin of the battery is much higher (30 percent) and the battery is loaded to the maximum.

Therefore, our adaptive robot system is able to change its behaviour model according to the runtime requirements, context as well as system models. The analysis step must decide whether the adaptation to a new behaviour model is necessary and convenient. Indeed, the planning step must find an acceptable solution in the configuration space and the execute step changes or generates a new behaviour model that is directly used by the robot and therefore forces an adaptation of the system behaviour.

Self-adaptive Systems with Multiple Layers. As advocated in [50] more sophisticated self-management capabilities do not result from a single adaptation feedback loop but from the combination of two loops in two layers. Similarly to adaptive control schemes and robot control architectures, multiple layers - where multiple adaptation loops operate on top of a regular feedback loop - have to be employed. It is outlined in [50] that adaptation related to context-awareness and self-awareness can be handled by a lower level change management layer if the core system stays within certain bounds. For changes of the requirements a higher level goal management layer that adjusts the change management layer is proposed.

Example 7 (2 Layered Self-Adaptive Robot Scenario with Runtime Models). An extended version of our self-adaptive Example 6 includes also adaptation behaviour that happens at the 2nd layer. There, we will determine error handling capabilities if necessary. Furthermore, we assume the same adaptation loop as before in Example 6 with the same change in the requirements.

Fig. 24 shows the influence of each loop to the outcome of the system behaviour (state machine) on two layers. On top, the error handling loop monitors upcoming failures of the system (e.g., the robot does not find a charging station and therefore fails during operation) and the adaptation rate of the underlying MAPE loop. In our example, the analyze step decides to add more robust robot behaviour to guarantee better task execution performance. The key idea for hierarchical loops is that the upper loop only changes the runtime models of the loop below. In our case, the planning activity of the error handling loop manipulates the knowledge base of our introduced MAPE loop in Example 6 by adding additional robot operations (functions) and better analyzing as well as planning capabilities for that loop.

Therefore, if the loop at the bottom is executed, it will detect those new capabilities and can come up with a more sophisticated state machine, which includes now the error handling extensions (or a subset according to the current needs). The bold parts in the state machine depicted in Fig. 24 (state machine (a)) as well as the new error state are the outcome of the indirect influence of the error handling MAPE loop. In another case, the robot can correct its position at runtime in the active *state and/or has more possibilities finding the charging station using the* advancedLaserScan() *operation. Additionally, the robot can now inform other robots about failures and tries to recover its own state in case of failure.*

Another scenario is that the error handling loop detects the decreasing capacity of the battery over time. An additional repairBattery() *function (state machine (b)) can solve this problem and can be removed afterwards in the next adaptation cycle if the full capacity is restored.*

At this point, it is important to mention that the different adaptation loops can influence or work against each other. For example, if the upper loop wants to compensate losses by recharging the battery but the lower loop must consider the High throughput *subgoal (cf. the first configuration in Fig. 18) it may decide to exclude the* repairBattery() *function as shown in the left state machine in Fig. 24 to reach this goal (because repairing the battery will take a lot of additional time). This is one example that the influences of several adaptation loops can be rather complex and has to be designed with care.*

Again, the required subset of all these changes is handled by the upper loop, which can influence the lower loop in each cycle by manipulating the corresponding runtime models accordingly. As an overall effect, the lower loop will generate an adapted state machine that integrates all these changes but still ensures the system goals.

5 Runtime Models for Handling Uncertainty

In this section, we focus on the state-of-the-art in the use of models to mitigate uncertainty. As discussed in earlier sections of this chapter, epistemic, randomized, and linguistic forms of uncertainty can affect the design and operation of a software system. Both epistemic and linguistic forms of uncertainty prevail during the requirements analysis and design, while randomized forms of uncertainty - for the most part - directly affect a software system during runtime.

Development-time uncertainty can compound the different forms of uncertainty explained above and can prevent a software system from delivering its functionality. Therefore, these consequent effects need to be treated during the system execution. During execution, uncertainty may appear in the form of environmental conditions that might have not been foreseen during development-time, because they may be unpredictable by nature. Other sources of uncertainty may be due to unreliable monitoring infrastructure.

We focus on the state-of-the-art of approaches that tackle uncertainty, due to the causes explained above, and which are or can be supported by runtime

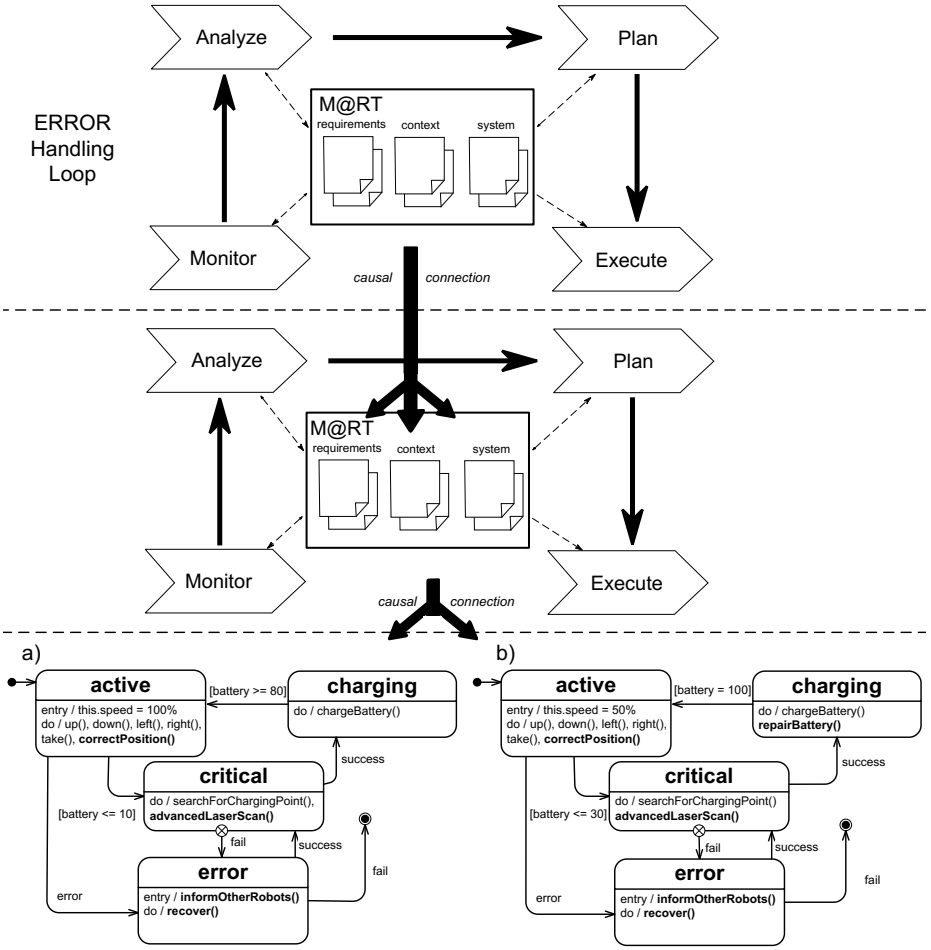

Fig. 24. Self-Adaptive robotic system scenario with multiple MAPE-K loops. The causal connection of the error handling loop influences the knowledge base of the MAPE-K loop below and therefore the outcome of the adapted state machines a) and b) indirectly.

models. The approaches described in the rest of this section are relevant in the context of the use of runtime models, although they focus on system's abstractions that characterize different stages and different qualities of the system life time.

5.1 Forms of Uncertainty

Next we provide a literature review of approaches that use runtime techniques to tackle epistemic, linguistic uncertainty as well as randomized uncertainty.

5.1.1 Epistemic Uncertainty

In [43] authors propose a technique to explicitly document the existence of uncertainty about how architectural decisions contribute towards satisfying non-functional properties (in the form of softgoals). The technique allows developers to deal with uncertainty during both development time and runtime [51]. A Claim can also be monitored at runtime to prove or disprove its validity [51]. *Claims* are particularly useful for ensuring that developers can revisit sources of uncertainty further along the development life cycle, including runtime, when new information may become available [52] tackling directly epistemic uncertainty.

In [53], probabilistic automatas are used to represent uncertainty for dynamically discovered and/or learned behavioural models and, as far as functional property validation is concerned, the uncertainty can be potentially tamed by using appropriate architectural models. Hidden Markov Models (HMM) [54] are typically used to model systems that have markovian characteristics in their behaviour, but they also have some states (and transitions) for which only limited knowledge is available. An example of an approach based on HMM that aims at evaluating the reliability of a software component with partial knowledge of its internal behaviour has been provided in [55]. Feature-based abstract models have been used to represent system's variability and configurations like in [56] to allow efficient symbolic model checking of product-line systems. Variability in the code is provided in context-oriented programming approach [57] as well as in the Chamaleon framework that supports a java extension to allow programming variability explicitly in the code [58]. Such variability is then solved by means of a resource-based analysis at deployment time when information about the execution context becomes available.

5.1.2 Linguistic Uncertainty

Fuzzy sets theory, which represents elements as partial members of a set, has been used in linguistics to deal with vagueness and ambiguity of the statements. In terms of self-adaptive systems, several techniques have been developed to deal with linguistic uncertainty.

An example of the use of Fuzzy theory RELAX [4], a specification language to express requirements that can be affected by uncertainty due to unanticipated

system and environmental conditions. RELAX has been applied [59] to identify sources of uncertainty in the environment and monitor those conditions that pose uncertainty. Using RELAX and KAOS, Ramirez et al. [6] have tackled the fact that design assumptions can also be subject to uncertainty with potential negative consequences on the behaviour of the adaptive system. FLAGS [35] is an approach that distinguishes between crisp and fuzzy goals where goals are specified using linguistics constructors. FLAGS also defines the concept of an adaptive goal to express countermeasures that can be executed when goals are not satisfied.

Torres et al. [46] use Fuzzy sets to underpin an approach that encourages architects to specify the set of requirements of a system as an abstract specification model by using linguistic variables instead of numerical variables, as the latter are more prone to give allow the obsolescence of requirements. By doing so, the authors mitigate the obsolescence of the specification model of the system. The approach allows analysts to create specifications at design-time, while preserving the flexibility afforded by dynamic changes in the "meaning" of non-functional requirements as specific values, thus allowing to effectively assess runtime requirements compliance in non-stationary environments.

5.1.3 Randonmized Uncertainty

Randomized uncertainty is caused both by system and environmental conditions that are either inherently random or cannot be predicted reliably. Therefore, runtime techniques that help provide reliable reasoning and prediction have been developed. This kind of uncertainty has been traditionally expressed in approaches that consider system non functional properties, like performance and reliability. The uncertainty can therefore be accounted for by the use of stochastic models as non functional models. In [60] several techniques have been proposed to take into account non-functional attributes of software under uncertainty. In [61], parametric queuing network models of the performance of different system's configurations are managed at runtime in order to support dynamic reconfigurations of the system in response to unpredictable context variations. Probabilistic automatas [53] discussed earlier also consider dealing with uncertainty about non-functional properties as reliability on the behaviour of components to meet a goal and its costs during runtime.

Feature-based systems representation has been used to support predictive and non predictive system evolution [62], where the feature model can be dynamically evolved to support consistent configuration building. Bayesian models (such as Bayesian Networks [63]) provide a way to express in the non functional setting approaches a la assume-guarantee, typically adopted in the functional world and that we will briefly recall in the following. Bayesian probabilities enable stochastic models to be 'conditioned' to specific events that, in turn, have their own probability distributions. Other sophisticated stochastic models can be used to take into account uncertainty in non-functional validation processes. Discrete-Time Markov Chains (DTMCs) and Continuous-Time Markov Chains (CTMCs) have been used both at development and runtime, in [64] to reason about the

reliability and performance of adaptable service-based application. In particular, the authors have started to study probability-based approaches to tackle the impact of the changes in the environment on the compositions of services and therefore the quality properties or QoS of the the service-based applications [64]. Their focus is on verification and dependability and in particular, on reliability and performance properties. In [7], the authors also focus on non-functional properties that can be specified quantitatively in a probabilistic way and target the challenge of making adaptation decisions under uncertainty. Given a decision that requires a certain configuration, the satisficement of a non-functional property can be modeled using probability distributions. However, differently from [64] they use Dynamic Decision Networks and focus on any non-functional property. Both [7] and [64] use Bayesian machine learning techniques to obtain information and support decision-making for self-adaptation during runtime.

5.2 Kinds of Runtime Models

The techniques described in this section recognize the need to produce, manage and maintain software models all along the softwares life time to support the realization and validation of systems adaptations while the system is already executing. In this section we describe more in depth the different techniques illustrated in the previous section and crucially focus on the runtime models they may involve. Furthermore, examples of the application of some of these techniques have already been introduced in Section 4.3. Additional information about model operations and a categorization of runtime models are further described in [21].

Furthermore, the purpose of this section is to show how to deal with uncertainty by focusing on system's abstractions (i.e. models) that characterize different stages and different relevant qualities of the system's life time. We consider two dimensions: the abstractions' dimension - and its corresponding software artifacts - that are used to explicitly represent uncertainty and system properties, and the properties' dimension. In particular, a system's abstractions have been considered that concern the following: systems models (e.g. architectural and behavioural system models, and coarse grain and fine grain system models), context models, and requirements models. On the properties' dimension, both the functional and non functional properties have been considered. The models described here represent uncertainty explicitly. More precisely, all these models are "loose" representations of the final system, i.e. the system that is actually running. All the approaches reviewed propose techniques to asses either functional or non functional properties on the system's artifact of reference. These models "contain" uncertainty but nevertheless are informative enough to allow assessment of some kinds of properties on the final system. The assessment allows the resolution of uncertainty at runtime.

5.2.1 Systems Models

In [65], runtime models of a system are used to reduce the number of configuration and reconfigurations that should be considered when planning the

adaptations. In [66] variability models are reused during runtime to support self-reconfiguration of systems when triggered by changes monitored in the environment. In [67] architectural models (i.e. configuration graphs) are studied as a means for monitoring, visualizing and recording information about the system adaptations.

In [68] the authors tackle a key issue to support runtime software architectures. First, in their approach it is important to maintain a causal connection between the architecture and the running system to therefore ensure that (i) the architecture model represents the current system, and (ii) the modifications on the architecture model cause the corresponding system changes.

In [69] the authors present a model-driven approach to maintain and update several architectural runtime models using model-driven engineering techniques. The causal connection to the running system is realized by triple graph grammar transformation rules. The approach is implemented and evaluated for the Enterprise Java Beans component standard.

So far, researchers have focused on the use of runtime models for the representation of the architecture of the system with no much advance in the area of the use of runtime models to control and generate system behaviour. In [70] the authors focus on the novel use of runtime models to support the dynamic synthesis of software, and specifically the synthesis of mediators to translate actions of one system to the actions of another system developed with no prior knowledge of the former in order to achieve interoperability. Using discovery and learning methods, the required knowledge of the context and environment. is captured and refined. The knowledge is explicitly formulated and made available to computational manipulation in the form of a runtime model. This runtime model is based on labelled transition systems (LTSs) which offer the behavioural semantics needed to model the interaction protocols to enable the interoperability between the systems. A similar solution to enable components interoperability is presented in [71] . Specifically, the authors present a model-driven approach that integrates an automated technique for runtime identification of message mismatches and the generation of behavioural mediators and their deployment supported by runtime models. However, further research efforts are needed in the area.

5.2.2 Context Models

Beside modeling the system, in order to carry out V&V activities it is also necessary to use a model of the context or the environment. In [72] a probabilistic model of the context evolution is provided in order to allow the dynamic adaptation of a system's configuration by achieving an optimal trade-off between user benefits and reconfiguration cost. Other approaches provide either explicit or implicit representation of the context and of its possible evolutions [73] via context assumptions. Notably in this class we can recall the whole approach to validation that goes under the name of assume-guarantee techniques. Although the original motivation for this approach was to provide compositional means to

validate large systems, this approach can also be characterized as what can be proved in terms of the inner knowledge of a component (the known) and what needs to be provided by the environment in which the component is executed (the unknown). Many approaches exist in the literature that range from the automatic synthesis of assumptions [74] for traditional behavioural models to the extension to probabilistic models [75].

Many other research efforts are devoted to support consistent adaptation of specific type of systems, notably in the service research arena [76]. These attempts, with reference to the MAPE cycle invest the planning activities, and provide solutions that can allow the evolution of the system in response to dynamic unplanned events. Other work handling uncertainty with the system itself is based on monitoring the values of properties over time and using statistical modeling techniques to predict likely future values [77]. For example, estimating the execution time reliably and precisely provides assurances about the suitability of the dynamically-adaptable software within its current operating environment, and may result in a requirement to trigger re-adaptation. Using a dynamically generated predictive model, forecasts are made about the values of any properties that may be analyzed from a series of values monitored over time. Such predictions can be used in the decision-making process of the MAPE feedback loops of self-adaptive systems described earlier.

5.2.3 Requirements Models

As previously discussed, design-time uncertainty can arise due to an imperfect requirements specification where requirements are missing or ambiguous [2,78,79]. Such uncertainty can often lead to a misalignment between the system's design and its original intent. Several techniques have been proposed for dealing with uncertainty at the requirements level, usually focusing either on documenting the existence of uncertainty or facilitating the analysis of how that uncertainty can affect the behaviour of the software system. In [52], the authors argue that requirements for self-adaptive systems need to be runtime entities (i.e. runtime models) that can be reasoned over at runtime.

Welsh et al. [43] have proposed REAssuRE that allows developers to deal with uncertainty during both development-time and runtime. Specifically, the authors used a *Claim* as a marker of uncertainty that explicitly documents the existence of uncertainty about how a system's goal operationalizations contribute towards the satisficement of soft goals. Techniques such as *Claims* are particularly useful for allowing developers to revisit sources of uncertainty further along the development life cycle when new information becomes available [52]. In that context, a Claim can also be monitored at runtime to prove or disprove its validity [51], thereby triggering an adaptation to reconfigure the system if necessary. Furthermore, in [51], the authors have demonstrated how goal-based runtime models can be held in memory in a form that allows the running system itself to evaluate goal satisfaction during execution and to propagate the effects of falsified Claims.

Fuzzy set theory, has been applied to represent and evaluate the satisfaction of functional [80] and non-functional requirements [81]. Ramirez et al. [6], recognize how Claims are also subject to uncertainty, in the form of unanticipated environmental conditions and unreliable monitoring information, that can adversely affect the behaviour of the adaptive system if it mistakenly falsifies a Claim. Therefore, the authors of [6] integrate Claims and RELAX, explained earlier, in order to assess the validity of Claims at runtime while tolerating minor and unanticipated environmental conditions that can trigger unnecessary adaptations and overhead.

Sutcliffe et al. [82] with their PC-RE method allow requirements to change over time in the face of contextual uncertainty. Epifani et al. [83] proposed to use a feedback control loop between models of non-functional properties and their implementations. During runtime, the system makes available information as feedback that is used to update the model to increase its correspondence with reality (hopefully decreasing uncertainty). Analysis of the updated model at runtime makes it possible to detect if a desired property (e.g. reliability or performance) is violated, causing automatic reconfigurations or self-healing actions to therefore meet the desired goals.

6 Research Challenges and Concluding Remarks

In this paper we have studied definitions and different types of uncertainty in the context of model-driven engineering putting emphasis on the use of models@run.time. We have revisited the concept of runtime models and have studied their impact and potential benefits in the management of uncertainty during execution. We have used a simple but illustrative example to discuss how development-time techniques together with runtime models can be used to cope with uncertainty. Also, we have discussed how runtime models can be used to extend the architecture of the MAPE-K loop to better manage uncertainty making use of abstractions (in the form of runtime models) to treat uncertainty as a first class entity during the system life cycle.

Based on the above, we summarize what we consider the most important research challenges, which are mainly explained in the context of the MAPE-K loop. We also argue the need for formal models and tools to support runtime models. Finally, we present some concluding remarks.

6.1 Runtime Models and the Feedback Adaptation Loop

The following are research challenges that have been identified and presented in the context of the MAPE-K loop.

6.1.1 Monitor

Sensing and monitoring can be imprecise and can provide just partial information. Runtime models should be able to make explicit this incompleteness

of information during monitoring through the use of the right abstractions; to therefore make it amenable to subsequent phases and specially the Analysis phase. Finding the right runtime abstractions to use to make available and measurable the uncertainty related to imprecision and partial information during monitoring is challenge that deserves research efforts.

Furthermore, better ways to explore how the system and the environment can interact are needed. We think runtime models can extend their application to represent not only concerns related to the running system but also the surrounding environment. Specifically, testing techniques need to be developed to explore how the software system interacts with its execution environment. These tests should measure whether the software system is capable of satisfying its requirements while facing uncertain conditions. Runtime models can be used to represent uncertainty through a shared boundary between the software system and its execution environment while more information is captured by the system while it is running.

6.1.2 Analyze

Currently, a *"marker of uncertainty"* [43] provides an estimate of a "known-unknown" [84] that identifies and describes parts of a model that are partially known. While markers of uncertainty narrow the scope of uncertainty and make it more manageable at run time, they should be specified in a proper way. Ideally, a marker of uncertainty should identify parts of a model that are partially known and, if possible, describe how they can vary. Regardless of whether a marker of uncertainty is explicit or implicit, techniques applicable at design-time and also runtime are required to facilitate the analysis of how different sources of uncertainty, and their severity, can affect the behaviour of a software system.

Moreover, little attention has been directed to techniques for the synthesis or generation of software using runtime models during execution. In order to design software systems that are able to tackle uncertainty, inferring the knowledge necessary to reason about system behaviour looks like an essential task. Such knowledge can be used to build runtime models during execution. An example is the work presented by the authors of [70] who present early results on how to conceive runtime models during execution, based on information about the running system and inferred using machine learning techniques during the execution of the system.

As new information is acquired, models should be refined. We argue the need of further research on how to include machine learning techniques to be able to incorporate new information while the system is running. Of course, the new acquired knowledge could solve uncertainty but also could incorporate more. In either case, what are the techniques to guarantee that some given properties of the system are preserved to maintain the desired system behaviour remains an open challenge. For example, while the model is fed with new information, the related notion of what is "relevant" to the runtime model may change.The ability of analysis based on the runtime model should be retained in any case.

6.1.3 Plan

In the planning step one has to deal with uncertainty and incompleteness of events in the decision-making process. To evaluate the decision-making process, uncertainty but also dynamicity should be taken into account. We believe that due to uncertainty, probabilistic reasoning and decision planning techniques are required in decision making. Few researchers have already worked using those techniques to tackle uncertainty. For example, Markov Decision Process (MDP) and Bayesian networks have been applied for diagnosis and self-recovery in [85,86]. The authors of [87] use a stochastic Petri net for decision-making in fault-tolerance. In [88] a stochastic Petri net is used as a model to compute the optimal monitoring frequency for crashing failures of a service-oriented system. Bayesian Dynamic Decision Networks have been used to enhance decision-making in self-adaptive systems [7].

The research initiatives named above are novel and represent research progress. However, the runtime models they would require to be applied at runtime would demand considerable amounts of resources (e.g. memory, and CPU) to be done during runtime. Therefore, the application of those techniques still remain a big challenge.

6.1.4 Execute

The use of runtime models imply a causal connection with the running system. Temporal delays in the MAPE-K loop can create an inconsistent view of the runtime model with respect to the running systems. The latter remains a big research challenge.

6.2 The Need for New Forms of Abstractions and Tools

Suitable mathematical abstractions should be applied to formally describe and analyze uncertainty. We believe probability theory, fuzzy set theory, and machine learning techniques should be further investigated for this purpose. Probability theory can be used to describe situations where previous historical data is available and can provide insights about the current design of a software system. For instance, developers can analyze execution data gathered from a previous version of a system to identify which goals and requirements are less likely to be satisfied at runtime. Similarly, fuzzy set theory can be applied to describe types of uncertainty where it is not possible to categorically prove or disprove the validity of a statement. In this manner, fuzzy set theory can be applied to initially produce a more flexible system design that can be progressively tightened as more information about the system and its environment becomes known during the design phase. Fuzzy probability theory extends probability theory with the possibility of expressing uncertainty in the parameters of the probability density function. Lastly, further work is required to develop machine learning techniques to be able to manipulate values of probabilities or parameters of utility functions that change over time and therefore, to be able to quantify the impact of

these values on the evaluation of alternative choices during the decision making process.

6.3 Concluding Remarks

Uncertainty about the running environments of software systems poses issues that software engineers need to face. Therefore, it is becoming increasingly important to come up with new methods and techniques to develop software systems able to deal with uncertainty at runtime. In this chapter we have discussed how runtime models are relevant in a reconceptualization of the development of software systems, which we assert is required to deal with uncertainty at runtime.

To establish a common ground for further discussions, we first introduced fundamental terms such as models, runtime models and uncertainty by using an exemplary goal, context and behavioural model for one robot of the factory automation example introduced earlier in the chapter. We also identified which kinds of runtime models are employed and outlined the most common types of systems using such runtime models. Furthermore, we discussed the role that runtime models can play for the different types of systems described.

Nowadays, we can observe the trend to delay decisions to handle uncertainty at runtime instead of doing it during development-time. To better understand the benefits and drawbacks of handling uncertainty at runtime by using runtime models, first we discussed classical approaches to handle uncertainty using development-time models and followed on considering how more advanced solutions to handle uncertainty at runtime can be used and how they can benefit from runtime models.

Specifically, we have discussed how the concepts of the MAPE-K loop can rely on runtime model techniques updating the knowledge data of the loop to tackle uncertainty during both development and runtime. We have argued how the above allows the management of uncertainty as a first class entity during the system life cycle. The envisioned framework includes a perpetual phase in which the runtime models can evolve, thereby allow the software system to cope with uncertainty by learning new information about itself and its execution environment based on monitored information that can only be collected during execution. We believe that in order to be able to support the extension of the MAPE-K loop proposed in this paper, several key challenges and enabling technologies need to be addressed. Crucially, synthesis of software during execution using runtime models has been identified as key challenge. Furthermore, such a capability requires inference of new knowledge during runtime. Therefore, we believe that machine learning techniques should be further studied to enable the incorporation of new information during the execution of the system while guaranteeing that the behaviour of the system is kept in the required behavioural envelop. Finally, to make this vision feasible new suitable and more efficient mathematical formalisms are also needed.

References

1. Galbraith, J.: Designing Complex Organizations. Organization development. Addison-Wesley (1973)
2. Noppen, J.: Imperfect Information in Software Design Processes. PhD thesis, University of Twente (2007)
3. Ramirez, A., Jensen, A., Cheng, B.H.C., Knoester, D.: Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 568–571 (2011)
4. Whittle, J., Sawyer, P., Bencomo, N., Chen, B.H.C., Bruel, J.M.: RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In: The Proceedings of the 17th International Requirements Engineering Conference (RE 2009), Atlanta, Georgia, USA, pp. 79–88. IEEE Computer Society (September 2009)
5. Welsh, K., Sawyer, P., Bencomo, N.: Towards Requirements Aware Systems: Run-time Resolution of Design-time Assumptions. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Kansas, USA, November 6-10. ACM (2011) (to appear)
6. Ramirez, A.J., Cheng, B.H.C., Bencomo, N., Sawyer, P.: Relaxing claims: Coping with uncertainty while evaluating assumptions at run time. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 53–69. Springer, Heidelberg (2012)
7. Bencomo, N., Belaggoun, A., Issarny, V.: Dynamic decision networks for decision-making in self-adaptive systems: A case study. In: Software Engineering for Adaptive and Self-Managing Systems, SEAMS (2013)
8. Laffont, J.J.: The Economics of Uncertainty and Information. The MIT Press (1989)
9. Uncertainty in Artificial Intelligence, `http://www.auai.org/`
10. Wätzoldt, S., Neumann, S., Benke, F., Giese, H.: Integrated Software Development for Embedded Robotic Systems. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) SIMPAR 2012. LNCS, vol. 7628, pp. 335–348. Springer, Heidelberg (2012)
11. Cheng, S.W., Garlan, D.: Handling Uncertainty in Autonomic Systems. In: Proceedings of the International Workshop on Living with Uncertainties (IWLU 2007), Co-located with the 22nd International Conference on Automated Software Engineering (ASE 2007), Atlanta, GA, USA, November 5 (2007)
12. Ali, R., Dalpiaz, F., Giorgini, P.: A Goal Modeling Framework for Self-Contextualizable Software. In: Proceedings of the Fourteenth International Conference on Exploring Modeling Methods in Systems Analysis and Design, pp. 326–338. Springer-Verlag (2009)
13. Lapouchnian, A., Mylopoulos, J.: Modeling Domain Variability in Requirements Engineering with Contexts. In: Laender, A.H.F., Castano, S., Dayal, U., Casati, F., de Oliveira, J.P.M. (eds.) ER 2009. LNCS, vol. 5829, pp. 115–130. Springer, Heidelberg (2009)
14. Stachowiak, H.: Allgemeine Modelltheorie. Springer-Verlag (1973)
15. Chung, L., Cesar, J., Leite, S.P.: Non-functional requirements in software engineering (1999)
16. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley (2009)
17. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Sci. Comput. Program. 8(3), 231–274 (1987)

18. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-Approximations to Over-Approximations and Back. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 157–172. Springer, Heidelberg (2012)
19. Mula, J., Poler, R., Garciasabater, J., Lario, F.: Models for production planning under uncertainty: A review. International Journal of Production Economics 103(1), 271–285 (2006)
20. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer 42(10), 22–27 (2009)
21. Vogel, T., Seibel, A., Giese, H.: The Role of Models and Megamodels at Runtime. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 224–238. Springer, Heidelberg (2011)
22. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
23. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.-M., Solberg, A., Dehlen, V., Blair, G.S.: An aspect-oriented and model-driven approach for managing dynamic variability. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 782–796. Springer, Heidelberg (2008)
24. Morin, B., Barais, O., Nain, G., Jézéquel, J.M.: Taming dynamically adaptive systems using models and aspects. In: ICSE, pp. 122–132 (2009)
25. Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A., Letier, E.: Requirements reflection: Requirements as runtime entities. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, pp. 199–202. ACM (May 2010)
26. Sawyer, P., Bencomo, N., Letier, E., Finkelstein, A.: Requirements-aware systems: A research agenda for re self-adaptive systems. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, Sydney, Australia, pp. 95–103 (September 2010)
27. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2) (2009)
28. Weiser, M.: The computer for the 21st century. SIGMOBILE Mobile Computing and Communications Review 3(3), 3–11 (1999)
29. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing 2(4), 263–277 (2007)
30. Bellavista, P., Corradi, A., Fanelli, M., Foschini, L.: A Survey on Context Data Distribution for Mobile Ubiquitous Systems. ACM Computing Surveys (2013) (to appear)
31. Souza, V.S., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceedings of the Sixth International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Waikiki, Honolulu, HI, USA, pp. 60–69. ACM (2011)
32. Feather, M.S., Fickas, S., van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behavior. In: Proceedings of the 8th International Workshop on Software Specification and Design, pp. 50–59. IEEE Computer Society (1998)
33. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, pp. 140–147. IEEE Computer Society (1995)
34. Silva Souza, V., Lapouchnian, A., Mylopoulos, J.: (Requirement) Evolution Requirements for Adaptive Systems. In: Proceedings of the 7th International Symposyum of Software Engineering for Adaptive and Self-Managing Systems. IEEE Computer Society (2012) (to appear)

35. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, RE, Sydney, Australia, pp. 125–134. IEEE (2010)
36. Ali, R., Solís, C., Omoronyia, I., Salehie, M., Nuseibeh, B.: Social Adaptation - When Software Gives Users a Voice. In: Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 75–84. SciTePress (2012)
37. Maes, P.: Concepts and Experiments in Computational Reflection. In: Proceedings of the 2nd International Conference on Object-oriented Programming Systems, Languages and Applications, OOPSLA 1987, pp. 147–155. ACM, New York (1987)
38. McManus, H., Hastings, D.: A Framework for Understanding Uncertainty and its Mitigation and Exploitation in Complex Systems. In: Proceedings of the Fifteenth Annual International Symposium of the International Council on Systems Engineering, INCOSE 2005, Rochester, NY (2005)
39. Garlan, D.: Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010, pp. 125–128. ACM, New York (2010)
40. Humphrey, W.: A Discipline for Software Engineering. SEI Series in Software Engineering Series. Addison Wesley Professional (1995)
41. Lehman, M.M., Belady, L.A.: Program evolution: Processes of software change. Academic Press Professional, Inc., San Diego (1985)
42. Parnas, D.L.: Software aging. In: Proceedings of the 16th International Conference on Software Engineering, ICSE 1994, Los Alamitos, CA, USA, pp. 279–287. IEEE Computer Society Press (1994)
43. Welsh, K., Sawyer, P.: Understanding the scope of uncertainty in dynamically adaptive systems. In: Wieringa, R., Persson, A. (eds.) REFSQ 2010. LNCS, vol. 6182, pp. 2–16. Springer, Heidelberg (2010)
44. Rabin, M.O.: Probabilistic automata. Information and Control 6(3), 230–245 (1963)
45. Piech, H., Siedlecka-Lamch, O.: Interval probabilities of state transitions in probabilistic automata. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2012, Part II. LNCS, vol. 7268, pp. 688–696. Springer, Heidelberg (2012)
46. Torres, R., Bencomo, N., Astudillo, H.: Mitigating the obsolescence of quality specifications models in service-based systems. In: MoDRE, pp. 68–76 (2012)
47. Xie, L.L., Guo, L.: How much uncertainty can be dealt with by feedback? IEEE Transactions on Automatic Control 45(12), 2203–2217 (2000)
48. Brun, Y., et al.: Engineering Self-Adaptive Systems through Feedback Loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
49. Cheng, S.W., Garlan, D.: Handling Uncertainty in Autonomic Systems. In: Proceedings of the International Workshop on Living with Uncertainties (IWLU 2007), Co-located with the 22nd International Conference on Automated Software Engineering (ASE 2007), Atlanta, GA, USA, November 5 (2007), http://godzilla.cs.toronto.edu/IWLU/program.html
50. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington, DC (2007)

51. Welsh, K., Sawyer, P., Bencomo, N.: Run-time Resolution of Uncertainty. In: Proceedings of the 19th IEEE International Requirements Engineering Conference, Trento, Italy, August 29-September 2, pp. 355–356 (2011)
52. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, Sydney, New South Wales, Australia, September 27-October 1, pp. 95–103 (2010)
53. Autili, M., Cortellessa, V., Di Ruscio, D., Inverardi, P., Pelliccione, P., Tivoli, M.: Integration architecture synthesis for taming uncertainty in the digital space. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 118–131. Springer, Heidelberg (2012)
54. Ephraim, Y., Merhav, N.: Hidden markov processes. IEEE Transactions on Information Theory 48(6), 1518–1569 (2002)
55. Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L.: Early prediction of software component reliability. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 111–120. ACM, New York (2008)
56. Cordy, M., Classen, A., Perrouin, G., Schobbens, P.Y., Heymans, P., Legay, A.: Simulation-based abstractions for software product-line model checking. In: ICSE, pp. 672–682. IEEE (2012)
57. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology 7(3), 125–151 (2008)
58. Autili, M., Benedetto, P.D., Inverardi, P.: Hybrid approach for resource-based comparison of adaptable java applications. Science of Computer Programming (2012) (to appear)
59. Cheng, B.H.C., Sawyer, P., Bencomo, N., Whittle, J.: A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 468–483. Springer, Heidelberg (2009)
60. Mishra, K., Trivedi, K.S.: Uncertainty propagation through software dependability models. In: Dohi, T., Cukic, B. (eds.) ISSRE, pp. 80–89. IEEE (2011)
61. Caporuscio, M., Marco, A.D., Inverardi, P.: Model-based System Reconfiguration for Dynamic Performance Management. Journal of Systems and Software 80(4), 455–473 (2007)
62. Inverardi, P., Mori, M.: A Software Lifecycle Process to Support Consistent Evolutions. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Self-Adaptive Systems. LNCS, vol. 7475, pp. 239–264. Springer, Heidelberg (2013)
63. Neil, M., Fenton, N., Tailor, M.: Using bayesian networks to model expected and unexpected operational losses. International Journal on Risk Analysis 25(4), 963–972 (2005)
64. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: Continuous assurance of non-functional requirements. Formal Asp. Comput. 24(2), 163–186 (2012)
65. Morin, B., Barais, O., Jézéquel, J.M., Fleurey, F., Solberg, A.: Models@ Run.time to Support Dynamic Adaptation. IEEE Computer 42(10), 44–51 (2009)
66. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic computing through reuse of variability models at runtime: The case of smart homes. Computer 42(10), 37–43 (2009)
67. Georgas, J., van der Hoek, A., Taylor, R.: Using architectural models to manage and visualize runtime adaptation. Computer 42(10), 52–60 (2009)

68. Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., Mei, H.: Supporting runtime software architecture: A bidirectional-transformation-based approach. Journal of Systems and Software 84(5), 711–723 (2011)
69. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proceedings of the 5th Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010) at the 32nd IEEE/ACM International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa, pp. 39–48. ACM (May 2010)
70. Bencomo, N., Bennaceur, A., Grace, P., Blair, G., Issarny, V.: The role of models@run.time in supporting on-the-fly interoperability. Springer Computing (2013); special issue Models@runt.time
71. Hao, R., Morin, B., Berre, A.J.: A semi-automatic behavioral mediation approach based on models@runtime. In: Models@run.time, pp. 67–71 (2012)
72. Mori, M., Li, F., Dorn, C., Inverardi, P., Dustdar, S.: Leveraging State-Based User Preferences in Context-Aware Reconfigurations for Self-Adaptive Systems. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 286–301. Springer, Heidelberg (2011)
73. Hong, J., Suh, E., Kim, S.J.: Context-aware systems: A literature review and classification. Expert Syst. Appl. 36(4), 8509–8522 (2009)
74. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. Autom. Softw. Eng. 12(3), 297–320 (2005)
75. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010)
76. Bucchiarone, A., Marconi, A., Pistore, M., Raik, H.: Dynamic Adaptation of Fragment-based and Context-aware Business Processes. In: Proc. of the 19th International Conference on Web Services. IEEE Computer Society (2012) (to appear)
77. Brennan, S., Cahill, V., Clarke, S.: Applying non-constant volatility analysis methods to software timeliness. In: Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS), WIP Track (2009)
78. Temponi, C., Yen, J., Tiao, W.A.: Assessment of customer's and technical requirements through a fuzzy logic-based method. In: Proceedings of the International Conference on Systems, Man and Cybernetics, vol. 2, pp. 1127–1132. IEEE Computer Society (1997)
79. Liu, X.F., Azmoodeh, M., Gerogalas, N.: Specification of non-functional requirements for contract specification in the ngoss framework for quality management and product evaluation. In: Proceedings of the Fifth International Workshop on Software Quality, pp. 36–41 (2007)
80. Liu, X.F.: Fuzzy requirements. IEEE Potentials, 24–26 (1998)
81. Glinz, M.: On non-functional requirements. In: IEEE International Requirements Engineering Conference, pp. 21–26 (2007)
82. Sutcliffe, A., Fickas, S., Sohlberg, M.M.: Pc-re: A method for personal and contextual requirements engineering with some experience. Requir. Eng. 11(3), 157–173 (2006)
83. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime parameter adaptation. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 111–121. IEEE Computer Society, Washington, DC (2009)
84. Cheng, B.H.C., et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)

85. Robertson, P., Laddaga, R.: Model based diagnosis and contexts in self adaptive software. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 112–127. Springer, Heidelberg (2005)
86. Robertson, P., Williams, B.: Automatic recovery from software failure. Commun. ACM 49, 41–47 (2006)
87. Porcarelli, S., Castaldi, M., Di Giandomenico, F., Bondavalli, A., Inverardi, P.: A framework for reconfiguration-based fault-tolerance in distributed systems. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems II. LNCS, vol. 3069, pp. 167–190. Springer, Heidelberg (2004)
88. Tichy, M., Giese, H.: A Self-Optimizing Run-Time Architecture for Configurable Dependability of Services. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems II. LNCS, vol. 3069, pp. 25–50. Springer, Heidelberg (2004)