Complex Intell. Syst. (2016) 2:111–124
DOI 10.1007/s40747-016-0020-x

ORIGINAL ARTICLE

Run-time architectural modeling for future internet applications

Marina Mongiello[1] · Simona Colucci[1] · Elvis Vogli[1] · Luigi Alfredo Grieco[1] · Massimo Sciancalepore[1]

Received: 10 December 2015 / Accepted: 12 June 2016 / Published online: 24 June 2016

Abstract Interoperability, flexibility, and adaptability are key requirements of future internet applications. Convergence of contents, services, things, and networks can be the cornerstone to fulfill these requirements. Such rich and composite sources of data and processing capabilities call for a structured and formal approach that manages and capitalizes heterogeneous information. This paper proposes an approach to the run-time composition of software system architectures, aimed at addressing goals revealed at runtime. The approach is grounded on a graph model characterized by two control levels: a metamodeling and an instantiation level. At metamodeling level, the graph describes facts that may occur in a scenario of interest, processes triggered by facts, and technologies available to execute processes. The actual occurrence of facts, together with the deriving processes and technologies, is managed at instantiation level, with reference to an application-specific model. In particular, the paper proposes an algorithm that determines an optimal way to manage a change in the run-time environment, by finding a minimum cost path in the model. The usefulness of the proposed approach and its applicability to actual scenarios have been validated in an example smart home environment.

Keywords Internet of things · Self-adaptive systems · Architectural modeling · Process composition

Introduction and motivation

Future internet applications should be able to handle dynamic changes in user experience and interoperability between different technologies, data, and processes. Convergence of contents, services, things, and networks may be a relevant direction to follow towards such objectives.

Complex and composite sources of information become in fact available as data coming from different resources and devices, signals produced from different entities, and events detected from signals processing. Moreover, many kinds of basic events can be combined to detect the occurrence of relevant conditions in complex situations [17,25]. Processing and composition of services may really benefit from all such heterogeneous information.

To address these challenges, adaptive mechanisms for the development and the interoperation of services and applications are emerging [9,24].

The complexity of data sources—ranging from signals, raw data, and simple and complex events—asks for a unique, flexible, and formal way to describe processes and elements related to them.

The main contribution of this article is the proposal of a formal approach to the composition of a software architecture, which is driven by the specification of goals determined at runtime. In general, a goal may be defined as an objective that the system is intended to achieve in the envisioned software and its environment [31].

The approach is based on a model and an algorithm for the automated composition of processes. Moreover, it enables the automated translation of goals into Operational Requirements, i.e., requirements that capture the conditions under which a system component performs an operation to achieve a goal.

The model is characterized by two control levels: a metamodeling and an instantiation level. The metamodel is general and technology independent, i.e., it models different configurations of adaptable software and can be instantiated in different domains. Hence, at the instantiation level, the

✉ Elvis Vogli
elvis.vogli@poliba.it

[1] Politecnico di Bari, Via E. Orabona, 4, 70125 Bari, Italy

model may be implemented in different application contexts and refers to specific run-time environments.

The separation in two levels is useful for deferring at run-time decisions that may be affected by the application context. It also allows for the incremental addition of goals at runtime.

Generally speaking, the design of a software architecture is guided by the questions: "how well does the chosen solutions support the satisfaction of each requirement?" and "which design alternatives were considered?" In a standard design activity, such questions are answered by a design-time decision-making process that compares different problems and solution alternatives. Instead, when a software system has to be composed on-the-fly, answering the second question may not be trivial and may depend on the specific run-time environment.

The model proposed hereby allows for composing, at runtime, a software architecture which is optimal with respect to all the other design alternatives satisfying the same requirements.

The model is a labeled graph, whose vertexes belong to one of the three entities: facts (occurring in the scenario of interest), processes (that may be triggered by facts), and technologies (in which processes may be implemented). Edges in the graph trace possible connections between such entities and are labeled through a function that computes the cost of such connections. The proposed algorithm finds in the model a minimum cost path. This path is an optimal way to orchestrate processes and technologies towards the satisfaction of a goal, which is generated, at runtime, by the occurrence of a fact. The retrieved path also determines the operational requirements in which the goal has to be translated.

The remaining of this paper is organized as follows. "Background" shortly recalls basic principles of the main research solutions adopted in the paper. In "Related work", related literature is analyzed. "Use case scenario" draws a systematic example for a use case scenario. The proposed formal model and composition algorithm are defined in "Adaptive processes composition". The model is instantiated in "Model instantiation" with respect to a fragment of the use case scenario, before discussing conclusion and future work.

Background

In this section, relevant background notions and technologies are shortly recalled, to make the paper self-contained. Specifically, "Self-adaptive systems" introduce self-adaptive systems, "Goals and operational requirements" describe goals and operational requirements, and "REST middleware" reports on REST Middleware.

Self-adaptive systems

A complete and thorough discussion of all issues related to modeling and verification of self-adaptive systems can be found in [14,19]. This section just recalls some definitions and peculiarities of self-adaptive systems which make them useful for the proposed approach.

Among several existing definitions for self-adaptive software, the most significant is probably the one provided in a DARPA Broad Agency Announcement (BAA-98-12) in 1997 [30]. Anyway, all definitions share some key points: a self-adaptive system has to be able to react on its own, by dynamically adapting its behavior, to guarantee a set of quality of service (QoS) requirements. The definition in [35] well summarizes such common features: self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.

Different approaches and models have been recently proposed in the field of self-adaptive applications [38]. The purpose of such proposals is modeling the architecture and verifying the properties of a system. They also share a common objective: designed systems need to have an acceptable level of reliability while still preserving dependability and the flexibility typical of adaptable systems.

In modern software systems, it is always difficult to predict the needs of users and, then, designing a configuration which is always optimal may be really hard. The active involvement of users for a clear understanding of their needs and behavior offers a solution, but it is still an open challenge. At runtime, it may be necessary to vary requirements and then design system components, on the basis of changes arising in the external environment. For example, changes in a sensor network might trigger the execution of processes and the implementation of software components not foreseen at design time.

The work in [48] sees a self-adaptive system as placed in an environment made up by physical and software entities and consisting of a two-layer architecture. It includes a first, managed, subsystem layer, that embeds the application logic, and a managing subsystem, on top of the first one, embedding the adaptation logic. The latter subsystem realizes a feedback loop that monitors the environment and the managed subsystem. The managing subsystem also adapts the managed one in the following cases: self-healing, when dealing with particular types of faults, self-optimizing, when operating conditions change, and self-reconfiguring, when a goal changes. Typically, the managing subsystem is conceived as a set of interacting feedback loops, one for each self-adaptation aspect (or concern). Other layers can be hierarchically added to the system, so that higher level managing

subsystems manage directly underlying subsystems, which in turn can work as managing systems.

Goals and operational requirements

In distributed systems, goals are objectives; the system is intended to achieve through the cooperation of agents in the envisioned software and its environment [31]. A requirement is a goal assigned to an agent in the software design [4]. While functional requirements specify the functionalities to be implemented, non-functional requirements suggest decisions on the architectural model. For example, if the system must ensure security, a proxy should be used to access protected data; if the system must integrate existing components, a distributed architecture should be preferred.

Among non-functional requirements, an Operational Requirement captures the conditions under which a system component may or must perform an operation, to achieve a goal. Thus, operational requirements describe the behavior of the system. They can be described by formal or semi-formal languages: such languages can be operational (i.e., finite-state machines or process algebra) or declarative (logic).

In the so-called adaptive systems, the definition of operational requirements is crucial for defining and describing the system in terms of behavior more than of provided functionalities.

It is still an open problem the translation of high-level goals into operational requirements that should hold on components, parts of the system or operations. Related work in this area is discussed in "Modeling systems goals and requirements".

REST middleware

Nowadays, many vertical M2M solutions have been designed independently for different applications, making the current M2M market very fragmented, which inevitably hinders a large-scale M2M deployment. To decrease the market fragmentation, there have been many efforts from different standardization bodies to define horizontal service layers.

The European Telecommunications Standards Institute (ETSI) has defined with the SmartM2M standard a middleware which has a RESTful architecture [44]. On the other side, OneM2M, where are collaborating more than 200 standardization bodies and companies, is defining a RESTful middleware which will have a global validity [41].

The proposed solutions provide RESTful middlewares which separate the applications from communication domain. The middlewares are accessible via open interfaces and enable the development of services and applications independently of the underlying network. In addition, they provide several service capabilities to enable machine registration, synchronous and asynchronous communication, resource discovery, access rights management, group broadcast, etc.

All the resources in the RESTful middlewares are organized in standardized resource trees and can be uniquely addressed by a Uniform Resource Identifier (URI). Their representations can be transferred and manipulated with verbs (i.e., retrieve, update, delete, and execute).

Related work

The proposed framework models the process of composing, at runtime, a software architecture addressing specific goals arising in the environment of interest. One distinguishing feature of the approach is the adoption of a unique model to represent heterogeneous facets of the composition process. In particular, the same model allows for representing: changes detected in the environment, goals to be reached, operational requirements translating goals, software processes to be composed, and, technological capabilities supporting the composition. As a consequence, both the functional and the adaptation logics of the proposed approach are embedded in the above-mentioned model. This causes different modeling issues to be discussed, when comparing to related literature. In the following subsections, the proposed approach is compared to recent literature in the topics mainly characterizing it.

Approaches more similar to ours can be found in [11,16,37].

The work in [37] adopts the SCA-ASM language, which provides modeling primitives to represent service component assemblies, to express internal service computation, interaction, and orchestration, and to perform fault and compensation handling. SCA-ASM combines the OASIS standard SCA (Service Component Architecture)[1] to model the architecture and the assembly of an application, and the ASM (Abstract State Machine) formal method [10] to specify services' behavior. Therefore, the work in [37] represents one of the few attempts to model both structure and behavior of service components in a unique framework integrating architectural and behavioral views. With respect to this work, we propose a data structure—a graph model—able to specify the dynamic behavior of the composed application, to evaluate cost parameters, and to select the best composition at runtime of a software architecture which is optimal with respect to all the other design alternatives satisfying the same requirements. Therefore, with respect to the approach in [37], our focus is to optimize certain quality requirements in a dynamic manner.

[1] Service Component Architecture (SCA). http://www.oasis-opencsa.org/sca.

Also in [11], a graph-based approach based on graph transformation is used to model self-adaptation. With respect to the types attribute graph grammar used to represent the application and adaptation logic proposed in [11], we use graph model to represent and to manage the adaptation logic with purposes of quality requirements optimization. In fact, a cost function is the base of the path extraction algorithm to find the best application composition.

Third, the approach in [16] relies on a service-oriented paradigm. A rigorous and lightweight theoretical foundation for representing the behavior of heterogeneous things is proposed. The work considers DPWS, a new emergent OASIS standard based on Web Service architectures to support interoperability among heterogeneous things. They propose to extend DPWS to specify the behavior of things. They also propose verification techniques to check if a composition of things fulfills or violates the behavior of those things. In fact, there is still a need to represent explicitly the behavior of things to develop applications in a more rigorous way. With respect to this approach that also refers to the domain of Internet of things adaptive composition, we model a metamodel of the adaptive composition of services and/or application based on behavioral changes in the user's requirements and in external context.

Anyway, with respect to all the existing approaches that we will further describe in the following state of the art, our model is abstract level. In fact, we propose a metalevel for architectural design based on requirements optimization. The metalevel is hence independent from specifications or properties for verification or design time-modeling issues. It is mainly concerned with having a high-level framework to be used for metamodeling of self-adptiveness and integration of applications deriving from several interoperable application domains.

Modeling context-aware and self-adaptive systems

Models for context-aware and self-adaptive systems have been widely studied in the last years, and several surveys have been proposed (see the work in [14,19], just to name the most well known). The work in [39] provides an exhaustive and structured state of the art and compares three approaches to support the implementation of adaptive systems.

A thorough analysis of formal approaches to self-adaptive systems may be found in [46]. According to this study, research proposals seem to fall into two not-overlapping sets: methods providing guarantees about the design of a self-adaptive systems and methods performing run-time analysis to support adaptations with particular guarantees. Only a few studies transfer formalization results over different phases of the software life cycle.

To bridge such a gap, Weyns [45] extends the work in [47] that defines formally founded design models for decentralized self-adaptive systems that cover structural aspects of self-adaptation. The reference model, called FORMS (FOrmal Reference Model for Self-adaptation), offers a vocabulary that consists of a small number of primitives and a set of relationships among them that delineates the rules of composition. In [45], such a model has been integrated in an approach to validate behavioral properties of decentralized self-adaptive systems to guarantee the required qualities. This approach has also been successfully developed in [26], where the authors present a case study of a decentralized traffic monitoring system and use model checking to guarantee a number of self-adaptation properties for flexibility and robustness. The main system processes are modeled with timed automata, and the required properties are specified using timed computation tree logic (TCTL).

Both in [37] and in [45], a decentralized approach to control is taken and adaptation is realized with a MAPE-K (Monitor-Analyse-Plan-Execute components over a shared Knowledge) feedback loop [28]. In particular, the work in [37] realizes the feedback loop described in [48]—and recalled in "Self-adaptive systems"—via MAPE-K.

A conceptual and methodological framework for formal modeling, validating, and verifying distributed self-adaptive systems is presented in [5] by some of the authors of [37]. The authors show how to specify MAPE-K loops for self-adaptation Abstract State Machines. In particular, the concept of multi-agent Abstract State Machines is used to specify decentralized adaptation control using MAPE computations.

The work in [2] proposes the goal-oriented framework SimSOTA for modeling, simulating, and validating MAPE-K feedback loop models of self-adaptive systems. SimSOTA also adopts a decentralized control strategy and a semi-formal notation— UML activity models— to model feedback loops.

In [11], graph transformation is used to model self-adaptation. Modeling of evolving systems based on components is instead adopted in [36]. Models for service choreography and composition are employed in [6] and by the same authors in [7], with specific reference to Future internet applications. Semantic approaches have been introduced in [33] to design self-adaptive architectural models in IoT. The work in [34] adopts the same techniques for run-time verification.

In [12], the authors overview emerging techniques for the engineering of high-integrity self-adaptive software; in the same paper, a service-based architecture aimed at integrating these techniques is introduced. The approach proposed in [42] integrates run-time verification enablers in the feedback adaptation loop of the ASSET adaptive security framework. The scope of this integration is guaranteeing self-adaptive security and privacy properties in the eHealth settings. In [23], the authors present an approach dealing with the run-time verification of behavior-aware composition of things. They propose to check whether a mashup

of things respects the specified behavior of the composed things. The approach is based on mediation techniques and complex event processing and is able to detect and inhibit invalid invocations. As a consequence, things only receive requests compatible with their behavior.

Modeling systems goals and requirements

The modeling of requirements for development and verification has also been widely studied. The work in [32] summarizes most relevant results. In this paper, behavioral adaptation is considered for customizing software, to perfectly meet user's needs in different contexts. The work in [18] models requirement changes at runtime. In this work, requirements are modeled at runtime and derived from a model conceived at design time on the basis of goals.

Moreover, requirements have been studied and formalized with verification purposes in [13]. The formal technique presented in this paper is called *continual verification* and is proposed to ensure reliability and performance requirements of safety-critical systems, even when they evolve. The run-time quantitative verification (RQV) technique has been proposed in [21] to make systems self-adaptive to changing workloads, environments, and goals. The approach is targeted to self-adaptive systems used in safety-critical and business critical applications, characterized by the need to comply with strict non-functional requirements.

Most of the approaches that use specifications, such as formal methods, assume operational requirements to be given. However, deriving *correct* operational requirements from high-level goals is challenging and is often delegated to error-prone processes. Letier and Lamwsveerde [31] propose an iterative approach that allows for the derivation of operational requirements from high-level goals. In this work, goals are expressed in real-time linear temporal logic (RT-LTL). The approach is based on operationalisation patterns. Operationalization is a process that maps declarative property specifications to operational specifications satisfying them. The approach produces operational requirements in the form of pre-, post-, and trigger conditions. The approach is guaranteed to be correct, i.e., the conjunction of the operational requirements entails the goal specification in RT-LTL. The approach is limited to a collection of goals and requirement templates provided by the authors. Moreover, it needs a fully refined goal model that requires specific expertise and is a labour-intensive and error-prone process.

The tool-supported framework proposed in [3,4] combines model checking and Inductive Logic Programming (ILP) to elaborate and refine operational requirements in the form of pre- and trigger conditions. Such conditions are correct and complete with respect to a set of system goals. System goals are in the form of LTL formulas. The approach works incrementally by refining an existing partial specification of operational requirements, which is verified with respect to the system goal. The verification is performed using model checking, which returns a counter example if the considered property is not valid on the model. The counter example is exploited to learn and refine the operational requirements. However, the approach does not support the learning of the operational requirements for a single system component. The approach presented in [15] automatically generates, via a learning algorithm, the assumptions that an environment needs to satisfy for some property to hold. These assumptions are initially approximated, but become gradually more precise by means of counter examples obtained by model checking the system components and the environment. In [22], the authors observe that, in real cases, a component is required to satisfy properties only in specific environments. Moved by these motivations, they directly generate assumptions tailored for a tern (component, property, and environment).

Use case scenario

The basic idea of the proposed approach is illustrated by means of a use case scenario in the following.

It is a cold winter evening, the temperature in the house is low, the heating system is activated to reach soon a temperature that will ensure comfort and well-being to Bob and Mary that are going to come back home after a busy working day. The blinds close to avoid the dispersion of heat. As soon as they get into the house, the lights turn on. Mary goes into the kitchen and set about making dinner; she turns on the oven that will soon bake tasty pork shank. In the laundry, the washer and dryer are temporarily suspended to avoid overload. Bob comes into the living room, where the lights turn on. He is very tired and decides to sprawl on the sofa and enjoy some videos. Therefore, he prepares the projector for watching the video taken by of his GoPRO while skying the previous Sunday on mountain holiday. The video projection begins, and the lights turn down to create soft lights.

Later, Mary goes—as every evening—to the basement to train on sports equipment while waiting for dinner to be ready. The daily news flow on the monitor of the tapis roulant on which Mary is training. Through headset she listens directives of the exercises to be carried out, according to the training program resulting from the control of the calories consumed in the day and of the physical activity already performed. Mary wears her heart rate and distance walked monitors for physical activity. When the goal of daily training is going to be reached, in the bathroom, the heating is switched on, and the whirlpool is switched on to enable Mary to practice proper relaxation after physical activity. Mary goes into the bathroom and the lights turn on, while the basement lights and sports equipment are turned off.

Meanwhile, in the garden, video surveillance cameras found two suspicious individuals climbing on the first floor and forcing a window to enter the house, despite the presence of people. The images sent to the nearby police station trigger the alarm that promptly activates forces to stop the thieves intrusion.

A spark caused by a failure of the electrical systems in the garage makes burst fire and soon the garage is filled with dense smoke. The high level of smoke triggers the fire alarm that immediately reaches the nearest fire department to activate the necessary reliefs.

Adaptive processes composition

In this section, the proposed approach to process composition is detailed. In particular, "The software architecture" presents the principles of the software architecture to be composed. "The model" introduces the graph-based model for run-time composition, while "Fact process technology algorithm" and "Cost function" propose, respectively, a solving algorithm and a description of the cost function labeling the graph.

The software architecture

The hardware infrastructure of the network is composed by motes, with limited memory and computation capabilities. Physical motes are mapped onto logical ones, and have a virtual image at middleware level. The features of the middleware are those of a REST middleware, whose functionalities can be extended through the implementation of adhoc plugins.

Figure 1 shows the software architecture executed at the sensor gateway, which is made up of:

1. a REST middleware with physical motes;
2. a variable number of application or protocol plugins encoding functionalities that can be run-time loaded, depending on the specific goal to be addressed;
3. a master (application) plugin in charge for checking and managing variations in the context, translating them in terms of goals and finding the best composition of plugins addressing the deriving goals [by implementing an algorithm, *Fact Process Technology* (FPT) and proposed "Fact process technology algorithm"].

Sensors' detection is managed at middleware level, where subscribers have to be registered and where updated data can be sent. The master plugin performs an adaptive composition of processes triggered by occurring facts (either retrieved by sensors or caused by the execution of other processes); the composition needs to address a goal by satisfying high level, mainly operational, requirements.

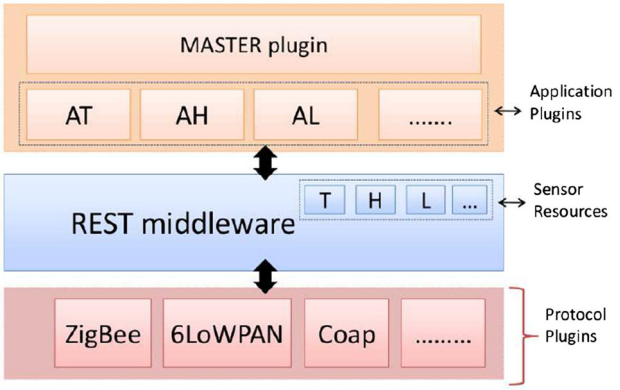

Fig. 1 Software architecture executed at the sensor gateway

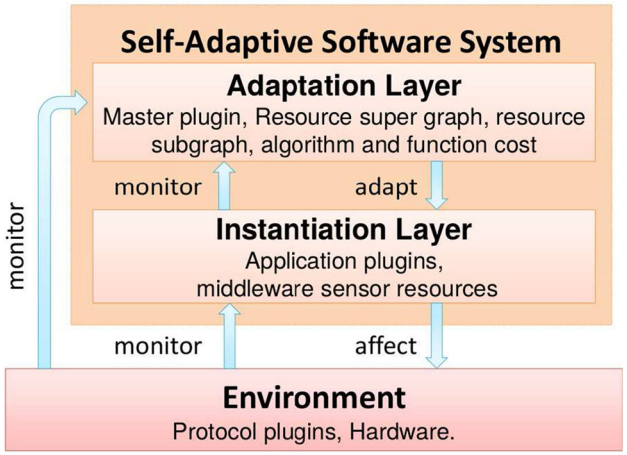

Fig. 2 Schema of the self-adaptive composition

The resulting software architecture is compliant with the schema of self-adaptive system originally proposed in [48] (and recalled in "Self-adaptive systems") and adapted to the proposed model as in Fig. 2. The reader may notice that the adaptation layer works as managing subsystem and embeds all components involved in the composition of a software architecture: the master plugin, the graph model, the cost function labeling the graph, and the composition algorithm. The instantiation layer works instead as managed subsystem and embeds the application plugins and the sensor resources belonging to the middleware which are used to monitor the environment. The adaptation layer may, in fact, monitor the environment either directly (by the master plugin) or through the instantiation level. Moreover, the adaptation layer may adapt components in the instantiation layer, thus affecting the environment.

The model

As introduced in "Introduction and motivation", the model hereby proposed has two control levels: one aimed at meta-modeling and one related to instantiation.

The model is a graph, whose nodes describe retrieved *facts*, *processes* triggered by facts and *technologies* that can be adopted to implement processes. The main advantage of using a graph structure is the possibility to use algorithms well known in graph theory to extract subgraphs satisfying a given goal. In particular, the graph can be visited with the objective to determine the best (according to any preference relation) sequence of processes to be executed and the best related technologies to adopt when retrieved facts require intervention. The retrieved sequence of processes represents the best available design alternative and defines the operational requirements corresponding to the original goal.

Metamodel At metamodeling level, the model is defined as *Resource Super Graph* in the following:

Definition 1 [*Resource Super Graph* (*RSG*)] Let **F**, **P**, **T** be three sets describing facts, processes, and technologies, respectively.

A Resource Super Graph is a weighted directed graph $G = \{\mathbf{V}, \mathbf{E}\}$, such that:

- $\mathbf{V} = \mathbf{F} \cup \mathbf{P} \cup \mathbf{T}$, i.e., a vertex can model a fact, a process or a technology;
- $\mathbf{E} = (\mathbf{F} \times \mathbf{P}) \cup (\mathbf{P} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{F})$, i.e., an edge connects either a fact to a process or a process to a technology or a technology to a fact;
- a function $c : E \rightarrow \Re_+ \cup \{0\}$ labels edges in E as follows:
  1. $c(v, w)$ is the cost of performing a process w triggered by a fact v, for $v \in F$ and $w \in P$.
  2. $c(v, w)$ is the cost of implementing a process v in a technology w, for $v \in P$ and $w \in T$.
  3. $c(v, w) = 0$, for $v \in T$ and $w \in F$.

As stated in Definition 1, nodes in the graph can be distinguished in: *Fact* nodes, *Process* nodes, and *Technology* nodes:

- *Facts* nodes model statuses occurring in the observed scenario; they can be directly perceived by the sensor network or be the result of some *process* implemented in some *technology*.
- *Process* nodes model operations which may be triggered by the occurrence of *facts*; they can distinguished in preprocessing operations—aimed at inferring new *facts* from raw sensor data—and processing operations—aimed at managing facts requiring intervention.
- *Technology* nodes model technological features of computational nodes available to perform *processes*; for example, a node modeling the middleware describes the network type, among other features, while a node modeling a mobile device describes its storage capacity, RAM, screen size, and so on.

Again, according to Definition 1, three kinds of edges are possible in the graph model:

- edges $(v, w) \in F \times P$ imply that a fact v may be managed by a process w with a cost $c(v, w)$.
- edges $(v, w) \in P \times T$ imply that a process v may be implemented by technology w with a cost $c(v, w)$.
- edges $(v, w) \in T \times F$ imply that the implementation of a process in a technology v causes a fact w to occur; the cost associated to such an implication is zero, because it maps an unavoidable effect of the implementation.

Software architectures to be composed are aimed at the satisfaction of goals. In the following, a *Goal* is defined according to the proposed metamodel.

Definition 2 (*Goal*) Given an RSG $R = (V, E)$, defined according to Definition 1, a *Goal* G in R is an ordered pair (s, d) with $s \in F \cap V$ and $d \in F \cap V$.

Intuitively, a goal is defined as a transition from an original status (modeled by the fact node s) to a destination status (modeled by the fact node d).

The reader may note that, given the graph structure of an RSG R, if a goal (s, d) is identified in R, all paths connecting s to d need to match patterns made up by sequences of ordered quadruples $(Fact, Process, Technology, Fact)$. Of course, many of such paths exist, which correspond to different design alternatives addressing the same goal. Nevertheless, this paper only focuses on finding a minimum cost path and to determine *the Resource SubGraph* (*RSubG*) of RSG, defined by such a path. The cost $cost(p)$ of a path p is defined as the sum of costs $c(v, w)$ of all edges (v, w) belonging to p.

Definition 3 [*Resource SubGraph* (*RSubG*)] Given an RSG $R = (V, E)$, defined according to Definition 1, and a goal (s, d) in R, defined according to Definition 2, a Resource SubGraph (RSubG) of R is a direct graph $S = (V', E')$, such that:

- $S \subseteq R$;
- $s \in V'$ and $d \in V'$;
- E' includes a path p connecting s to d, such that $cost(p) \leq cost(r)$ for every path r in R connecting s to d, with $r \neq p$.

A path defining an RSubG identifies also a sequence of operational requirements translating the input goal in terms of pairs $(Process, Fact)$ included in the path.

Instantiation The introduced metamodel is general enough to describe, independently on the application context, all elements involved in the achievement of different goals, through the composition of heterogeneous software processes, that

may be implemented in several available technological solutions.

Nevertheless, at instantiation level, the metamodel needs to be referred to an application scenario, for which specific facts, processes, and technologies have to be identified, together with the relations among them. In fact, observable facts, performable processes, and available technologies depend on the specific scenario, and so do nodes and edges in an RSG.

This activity leads to the definition of an application-specific RSG. Then, when—at middleware level—a variation in the context is revealed by the master plugin, the actual scenario settings have to be processed to determine the best process composition managing such a variation (unless the variation does not require intervention).

In other words, when an occurring fact is revealed, the composition process goes through the following steps:

1. Construction of the contextual RSG: (i) some nodes in the application-specific RSG may become unavailable at runtime and (ii) the relationship between nodes and the related cost may depend on run-time settings.
2. Identification of the Goal: depending on run-time settings, an occurring fact s can be source of one or more goals (s, d) to be addressed through a composition process leading to d.
3. Computation of an RSubG determined by the RSG and the Goal.

Of course, Step 1 does not build from scratch one RSG for each execution, but it is meant to customize the application-specific model on the basis of run-time settings, through a set of rules to be formalized. As an example, at runtime, it may happen that none of the available devices satisfies the requirements of a process or that a process may be adopted to manage an occurring fact only depending on run-time settings.

Moreover, the execution of Step 2 asks for the definition of a set of rules translating occurring facts into goals, even in this case by taking run-time information into account. As an example, a reduction of the temperature in a room may or may not be translated in the goal of increasing the temperature, depending on the preferences set for that room or on the external temperature.

In "Model instantiation", it is provided an example of model instance referred to a fragment of the use case introduced in "Use case scenario".

Fact process technology algorithm

Algorithm *Fact Process Technology* (FPT) is now proposed for finding an RSubG S of an RSG $R = (V, E)$, given R and a goal (s, d) in R.

The algorithm is implemented by the master plugin, which schedules, manages, and monitors facts, technologies and processes execution on the devices. Communication among plugins occurs through the middleware that forwards requests, data, and responses between plugins and sensors, according to low-level protocols; the interaction is instead scheduled and managed by the high-level master plugin.

A running plugin is identified by a quadruple $(Fact, Process, Technology, Fact)$, whose cost is determined through the cost function defined in "Cost function". Algorithm 1 extracts the minimum cost solution for running plugins required by occurring facts.

Data: A Resource SuperGraph $R = (V, E)$, a goal (s, d) in R,
Result: A Resource SubGraph $S = (V', E')$ of R
1 let c be the function labeling R;
2 solve $DIJKSTRA(R, c, s)$
3 read a minimum cost path p_{min} from s to d
4 add all nodes in p_{min} to V'
5 add all edges in p_{min} to E'

Algorithm 1: FPT

In Row 2, the Dijkstra algorithm [20], well-known form graph theory, is adopted to associate to each node $v_i \in V$, the cost of the (minimum) path from s to v_i, and the node v_{i-1} preceding v_i in the minimum cost path. Then (Row 3), the minimum cost path from s to d is read from the results of Dijkstra algorithm and S is consequently built in Rows 4 and 5.

The extracted path can also be seen as a sequence of operational requirements translating the input goal.

Cost function

To define a cost function, we first need to refer separately to the cost of performing a process triggered by a fact and the cost of implementing a process in a given technology. The former cost, $c(v, w)$ for $v \in F$ and $w \in P$ includes: (i) a fixed cost (*plugin fixed cost—pfc*) depending on some computational features known at compile time, such as time and size complexity of the process and (ii) a variable cost (*plugin variable cost—pvc*) depending on run-time factors, such as the values and the size of input data. As a consequence, $c(v,w) = pfc + pvc$ for $v \in F$ and $w \in P$.

The latter cost, $c(v, w)$ for $v \in P$ and $w \in T$, is variable and depends on: (i) the network supporting the implementation and (ii) the status of the device w. As a consequence, $c(v,w) = nc + dc$ for $v \in F$ and $w \in P$, where *nc* stands for *network cost* and *dc* stands for *device cost*.

The cost of the network (*nc*) includes information about the status of the network at the time of plugin execution request, such as connection delay, network bit rate, packet

size, and so on. This cost is zero for the device, where the middleware is installed and, therefore, is crucial for choosing between remote and local executions of a plugin, depending on the network conditions.

The *device cost* is variable and depends on the type of device, which may be either the local one, where the middleware is installed, which is main powered, or a remote device, i.e., a portable device (such as a smartphone or a tablet) usually supplied by batteries.

In the latter case, plugins implemented in a remote device, and *dc* depends only on the availability of computational, storage and energy resources, such as CPU, RAM, storage capacity, and battery.

In the former case, plugins managed by the middleware, *dc* has the following components:

- a *middleware cost (mc)*, which depends on the RAM and/or CPU available in the device, where the middleware is installed; this cost is logically composed by two items: a *middleware preprocessing cost (mpc)*, which is the cost of evaluating whether executing the plugin locally or forwarding it to another device and a *middleware execution cost (mec)*, which applies only to a local execution
- *forwarding cost (fc)* which is the cost of forwarding the plugin to another device.

The factors affecting both *mc* and *fc* are contextual and depend on the number of active connections or services that are being served at the time a request is made.

As a consequence, $dc = mc$ for plugins executed in the local device and $dc = mpc + fc$ for plugins forwarded by the middleware to other devices.

The resulting cost function may be defined as follows:

$$c(v, w) = \begin{cases} pfc + pvc & v \in F \text{ and } w \in P \\ mc & v \in P \text{ and } w \in T, \\ \text{middleware installed in } w, & \\ v \text{ executed in } w & \\ nc + mpc + fc & v \in P \text{ and } w \in T, \\ \text{middleware installed in } w, & \\ v \text{ forwarded to other devices} & \\ nc + dc & v \in P \text{ and } w \in T, \\ w \text{ remote device} & \\ 0 & v \in T \text{ and } w \in F \end{cases}$$

Model instantiation

Before instantiating the model in the smart home example scenario described in "Use case scenario", the main arguments supporting the choice of such a scenario are briefly recalled.

The example is aimed at demonstrating the peculiarities of the proposed approach, which is general purpose and, consequently, could be implemented in any application scenario.[2]

The main peculiarities to be demonstrated are summarized in the following list:

(a) high-level modeling of occurring facts;
(b) flexibility of proposed solutions with respect to environmental settings;
(c) support to the automated extraction of operational requirements from algorithm results.

Although apparently simple, the smart home example scenario in "Use case scenario" allows for showing all the peculiarities above.

For the sake of example, three portions of the described scenario are considered:

1. As soon as they get into the house, the lights turn on.
2. Mary goes into the bathroom, and the lights turn on, while the basement lights and sports equipment are turned off.
3. two suspicious individuals climbing on the first floor and forcing a window to enter the house, despite the presence of people in the house. The images sent to the nearby police station trigger the alarm that promptly activates forces to stop the thieves intrusion.

Figure 3 shows the application-specific Resource Super Graph sufficient to model this fragment of the use case.

The reader may verify that occurring facts, even though quite heterogeneous, are modeled in a small number of nodes that describe, at high level, changes in the scenario of interest. Such changes may be either straightly perceived by sensors (such as a temperature, light, position, and air variation) or revealed by the master plugin by combining information in the environment (such as a safety change). In other words, the model presented hereby allows for representing according to the same format, i.e., as fact nodes in an RSG, pieces of information very dissimilar from each other in terms of granularity.

The same consideration applies to the process nodes: some processes in the example RSG refer to mechanical actions to be performed (such as turn on/off heater, open/close blinds, and turn on/off lights), while some others describe actions involving the exchange of information (such as information forwarding and emergency call). All such processes, although completely different form each other, are formalized as high-level process nodes in the model.

[2] A first attempt of instantiation for the adaptation of cloud-based applications has been addressed in [1].

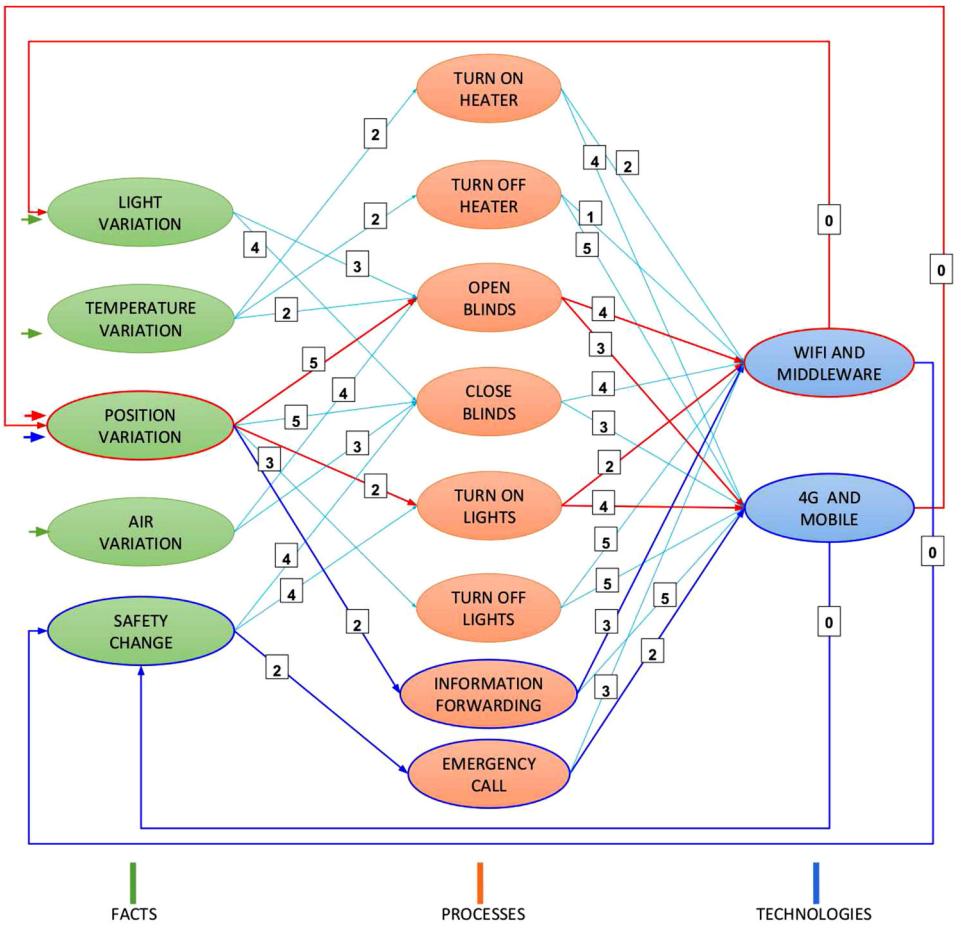

Fig. 3 RSG for the use case scenario

Notably, the only way to represent heterogeneous facts and processes in a unique model is keeping at high level their formalization, as the proposed approach does.

In the following, the three fragments of the example scenario are detailed and used for the demonstrative objectives elicited above.

First fragment The occurring fact initiating the composition is a $Position\_Variation$ (*"As soon as they get into the house..."*). Coherently with the use case description (*... the lights turn on"*), the goal is a pair ($Position\_Variation$, $Light\_Variation$). In fact, if the source fact is the "position variation" (i.e., there are people in the house), then it must be increased the luminosity (i.e, a light variation is produced).

In the RSG, there are different paths that can be followed to address the goal. For example, the $Position\_Variation$ fact is connected to several processes. If the goal is defined as ($Position\_Variation$, $Light\_Variation$), then the solution can be via either $Turn\_on\_Lights$ or $Open\_Blinds$ processes. Moreover, both processes are connected to the two different technology nodes.

The FPT algorithm selects the minimum cost path and determines the RSubG corresponding to the introduced RSG and goal, described in Fig. 4. The RSubG shows that the best solution addressing the goal is through the process $Turn\_on\_Lights$ and the technology $Wifi\_and\_Middleware$. The only operational requirement deriving from the goal is the pair ($Turn\_on\_Lights$, $Wifi\_and\_Middleware$).

We notice that the path suggested by FPT represents one of the process composition alternatives that can be designed to satisfy the goal. In particular, choosing this path implicates discarding all the other design alternatives, i.e., namely, ($Turn\_on\_Lights$, $4G\_and\_Mobile$), ($Open\_Blinds$, $Wifi\_and\_Middleware$), ($Open\_Blinds$, $4G\_and\_Mobile$). In the description of the other two fragments, there is not explicit reference to the discarded design alternatives, for a matter of brevity. Nevertheless, the reader may easily derive them according to the considerations discussed for this fragment.

Second fragment The same line of reasoning as in the first fragment applies to the second one. In fact, if the two situations share the run-time environment (time of the day, luminosity, and so on), also the fragment *"Mary goes into the bathroom and the lights turn on"* is translated into the

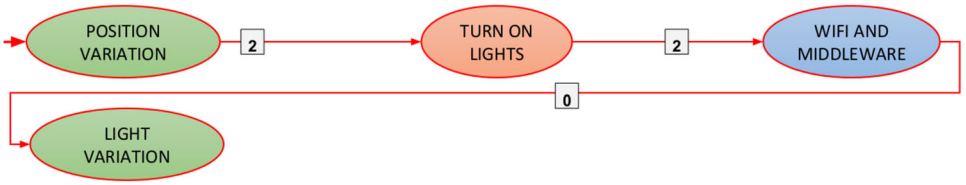

Fig. 4 RSubG extracted for the first fragment of the use case

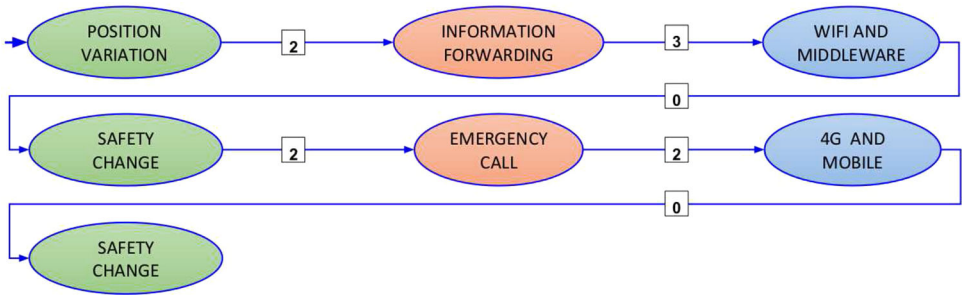

Fig. 5 RSubG extracted for the third fragment of the use case

goal ($Position\_Variation$, $Light\_Variation$) and generates the same RSubG and, thus, the same operational requirement. Analogously, the fragment *"...the basement lights and sports equipment are turned off"* is also translated into a goal ($Position\_Variation$, $Light\_Vari\ ation$), but this time, the RSubG includes the process $Turn\_off\_Lights$ and one of the two available technologies (they share the cost of executing $Turn\_off\_Lights$). If, for example, $4G\_and\_Mobile$ is chosen, this new goal is translated in the operational requirement ($Turn\_off\_Lights$, $4G\_and\_Mobile$).

Third fragment In this case, a $Position\_Variation$ occurs together with run-time settings denoting an intrusion. As a consequence, this fact is translated in a different goal, ($Position\_Variation$, $Safety\_Change$), which means that the processes composition aims at reaching safety in the house. The detected intrusion causes the originating fact to trigger the process $Information\_Forwarding$ (*The images sent to the nearby police station...*), which, independently on the technology implementing it, produces a $Safety\_Change$: the house has become unsafe. This $Safety\_Change$ does not satisfy the goal and needs to be managed by one or more further processes. In particular, the use case selects the $Emergency\_Call$ process, which, again independently on the technology, makes the house safe (*...trigger the alarm that promptly active forces to stop the thieves intrusion.*). The RSubG selected by FPT algorithm is shown in Fig. 5 and defines the following set of operational requirements: {($Information\_Forwarding$, $Wifi\_and\_Middleware$), ($Emergency\_Call$, $4G\ \_and\_Mobile$)}.

The reader may notice that the same fact node, $Position\_Variation$, may be instantiated in the case of "safe" entrance in a room in darkness conditions (Fragments 1 and 2), in the case of "safe" exit from a room in darkness conditions (Fragment 2), and in the case of "unsafe" entrance in the house (Fragment 3). Analogously, the fragments show that: the fact node $Light\_Variation$ has been instantiated in the case of luminosity increment (Fragments 1 and 2) and decrement (Fragment 2); the fact node $Safety\_Change$ is instantiated in the case of change from unsafe to safe status and viceversa (Fragment 3). As suggested in the discussion of the fragments, the master plugin distinguishes the cases of $Position\_Variation$, $Light\_Variation$, and $Safety\_Change$ by sensing the environment. Such a working mode is coherent with the peculiarity *a* in the above list.

When a fact occurs, the master plugin is also in charge for processing information sensed in the environment, to translate the fact in a goal and build the contextual RSG. The analyzed fragments show how the same source fact, $Position\_Variation$, may be managed according to three different contextual RSGs, to satisfy three different goals. This demonstrates peculiarity (b) in the above list. In particular, in the first fragment, the goal is an increment of luminosity and, consequently, the only processes leading to such a $Light\_Variation$ are considered for the composition. In the second fragment, the variation of Mary's position originates two different goals: an increment of luminosity in the room Mary is entering and a decrement of luminosity in the room Mary is leaving. Two different $Light\_Variation$ goals are generated, and different processes are taken into account to reach such goals. As a consequence, two different compositions of processes are retrieved and one design alternative for each of them is discarded. In the third fragment, again, a $Position\_Variation$ generates a completely different goal, because it comes together with sensed information denoting an intrusion. As a consequence, only processes

leading to safety changes are taken into account, and the composition detailed above is proposed to reach safety conditions.

Finally, it is noteworthy that the description of each fragment explicitly reports the operational requirements in which the proposed approach translates the goal. This demonstrates peculiarity (c) in the above list.

Conclusion and future work

This paper introduced an approach to goal-driven architectural composition of adaptable systems. The approach allows for designing, at runtime, the architecture of a software system. The system is conceived to address goals originated by occurring facts revealed at runtime in a scenario of interest.

The approach proposes a formal model, composed by a metamodeling and an instantiation level. The metamodel is general and technology independent, since it describes different configurations of adaptable software and can be instantiated in different, platform-dependent, domains.

The model is based on a graph structure representing scenarios of interest in terms of facts, processes, and technologies. Moreover, it adopts an algorithm specifically proposed to find a minimum cost composition of processes. The resulting composition builds a software architecture satisfying the input goal and, at the same time, defines the operational requirements translating the goal.

The model has been instantiated in a sensor network environment, and the algorithm has been validated in a smart home example scenario. Currently, the model is object of a thorough evaluation aimed at full validation and testing. In the near future, the proposed approach will be extended to the composition of software architectures addressing multiple goals and to different application domains.

References

1. Scandurra P, Mongiello M, Colucci S, Grieco LA (2016) Towards a goal-oriented approach to adaptable re-deployment of cloud-based applications. In: Proceedings of the 6th international conference on cloud computing and services science, pp 253–260. doi:10.5220/0005861602530260, ISBN 978-989-758-182-3
2. Abeywickrama DB, Hoch N, Zambonelli F (2013) Simsota: engineering and simulating feedback loops for self-adaptive systems. In: Proceedings of the International C* Conference on Computer Science and Software Engineering, C3S2E '13, ACM, New York, NY, USA, pp 67–76
3. Alrajeh D, Ray O, Russo A, Uchitel S (2009) Using abduction and induction for operational requirements elaboration. J Appl Logic 7(3):275–288 **(Special issue: abduction and induction in artificial intelligence)**
4. Alrajeh D, Kramer J, Russo A, Uchitel S (2009) Learning operational requirements from goal models. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, pp 265–275
5. Arcaini P, Riccobene E, Scandurra P (2015) Modeling and analyzing MAPE-K feedback loops for self-adaptation. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '15, IEEE Press, Piscataway, NJ, USA, pp 13–23
6. Autili M, Di Benedetto P, Inverardi P (2009) Context-aware adaptive services: the PLASTIC approach. In: Chechik M, Wirsing M (eds) Fundamental approaches to software engineering, 12th International Conference, FASE 2009, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, 22–29 March 2009. Proceedings, vol 5503, Lecture notes in computer science. Springer, pp 124–139
7. Autili M, Inverardi P, Tivoli M (2014) CHOREOS: large scale choreographies for the future internet. In: Demeyer S, Binkley D, Ricca F (eds) 2014 Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, IEEE Computer Society, , Antwerp, Belgium, 3–6 February 2014, pp 391–394
8. Baldauf M, Dustdar S, Rosenberg F (2007) A survey on context-aware systems. Int J Ad Hoc Ubiquitous Comput 2(4):263–277
9. Ben Hamida A, Kon F, Ansaldi Oliva G, Moreira Dos Santos CE, Lorré J-P, Autili M, De Angelis G, Zarras AV, Georgantas N, Issarny V, Bertolino A (2012) An integrated development and runtime environment for the Future Internet. In: Alvarez F, Cleary F, Daras P, Domingue J, Galis A, Garcia A, Gavras A, Karnouskos S, Krco S, Li M-S, Lotz V, Müller H, Salvadori E, Sassen A-M, Schaffers H, Stiller B, Tselentis G, Turkama P, Zahariadis TB (eds) The Future Internet—future internet assembly 2012: from promises to reality, vol 7281, Lecture notes in computer science. Springer, Berlin, Heidelberg, pp 81–92
10. Borger E, Stark RF (2003) Abstract state machines: a method for high-level system design and analysis. Springer-Verlag New York, Inc., Secaucus
11. Bucchiarone A, Ehrig H, Ermel C, Pelliccione P, Runge O (2015) Rule-based modeling and static analysis of self-adaptive systems by graph transformation. In: De Nicola R, Hennicker R (eds) Software, services, and systems—essays dedicated to Martin Wirsing on the occasion of his retirement from the Chair of Programming and Software Engineering, vol 8950, Lecture notes in computer science. Springer, Switzerland, pp 582–601
12. Calinescu R (2013) Emerging techniques for the engineering of self-adaptive high-integrity software. In: Cámara J, de Lemos R, Ghezzi C, Lopes A (eds) Assurances for self-adaptive systems—principles, models, and techniques, vol 7740, Lecture notes in computer science. Springer, Berlin, Heidelberg, pp 297–310
13. Calinescu R, Rafiq Y, Johnson K, Bakir ME (2014) Adaptive model learning for continual verification of non-functional properties. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14, ACM, New York, pp 87–98

14. Cheng BH, Lemos R, Giese H, Inverardi P, Magee J, Andersson J, Becker B, Bencomo N, Brun Y, Cukic B, Di Marzo Serugendo G, Dustdar S, Finkelstein A, Gacek C, Geihs K, Grassi V, Karsai G, Kienle HM, Kramer J, Litoiu M, Malek S, Mirandola R, Müller HA, Park S, Shaw M, Tichy M, Tivoli M, Weyns D, Whittle J (2009) Software engineering for self-adaptive systems: a research roadmap. In: Cheng BH, Lemos R, Giese H, Inverardi P, Magee J (eds) Software engineering for self-adaptive systems. Springer-Verlag, Berlin, pp 1–26
15. Cobleigh JM, Giannakopoulou D, Păsăreanu CS (2003) Learning assumptions for compositional verification. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'03, Springer-Verlag, Berlin, Heidelberg, pp 331–346
16. Cubo J, Brogi A, Pimentel E. Behaviour-aware compositions of things. In: 2012 IEEE International Conference on Green Computing and Communications (GreenCom), IEEE, Nov 2012, pp 1–8
17. Cubo J, Ortiz G, Boubeta-Puig J, Foster H, Lamersdorf W (2014) Adaptive services for the future internet. J Univers Comput Sci 20(8):1046–1048
18. Dalpiaz F, Borgida A, Horkoff J, Mylopoulos J (2013) Runtime goal models. In: Proceedings of the 7th IEEE International Conference on Research Challenges in Information Science (RCIS 2013). Invited paper
19. de Lemos R, Giese H, Müller HA, Shaw M, Andersson J, Litoiu M, Schmerl BR, Tamura G, Villegas NM, Vogel T, Weyns D, Baresi L, Becker B, Bencomo N, Brun Y, Cukic B, Desmarais RJ, Dustdar S, Engels G, Geihs K, Göschka KM, Gorla A, Grassi V, Inverardi P, Karsai G, Kramer J, Lopes A, Magee J, Malek S, Mankovski S, Mirandola R, Mylopoulos J, Nierstrasz O, Pezzè M, Prehofer C, Schäfer W, Schlichting RD, Smith DB, Sousa JP, Tahvildari L, Wong K, Wuttke J (2013) Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos R, Giese H, Müller HA, Shaw M (eds) Software engineering for self-adaptive systems II—International Seminar, Dagstuhl Castle, Germany, 24–29 October 2010 Revised Selected and Invited Papers, vol 7475, Lecture notes in computer science. Springer, Berlin, Heidelberg, pp 1–32
20. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1(1):269–271
21. Gerasimou S, Calinescu R, Banks A (2014) Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, ACM, New York, NY, USA, pp 115–124
22. Giannakopoulou D, Păsăreanu CS, Barringer H (2005) Component verification with automatically generated assumptions. Autom Softw Eng 12(3):297–320
23. González L, Cubo J, Brogi A, Pimentel E, Ruggia R (2013) Runtime verification of behaviour-aware mashups in the internet of things. In: Canal C, Villari M (eds) Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2013, Málaga, Spain, 11–13 September 2013, Revised Selected Papers, vol 393, Communications in computer and information science. Springer, pp 318–330
24. Guinard D, Ion I, Mayer S (2011) In search of an internet of things service architecture: REST or WS-*? A developers' perspective. In: Puiatti A, Gu T (eds) Mobile and ubiquitous systems: computing, networking, and services—8th International ICST Conference, MobiQuitous 2011, Copenhagen, Denmark, 6–9 December 2011, Revised Selected Papers, vol 104, Lecture notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, pp 326–337
25. Guinard D, Trifa V, Wilde E (2010) A resource oriented architecture for the Web of Things. In: Internet of Things (IOT), 2010. IEEE, pp 1–8
26. Iftikhar MU, Weyns D (2012) A case study on formal verification of self-adaptive behaviors in a decentralized system. In: Kokash N, Ravara A (eds) Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, 8 September 2012, Newcastle, UK, vol 91, EPTCS, pp 45–62
27. Iftikhar MU, Weyns D (2014) ActivFORMS: active formal models for self-adaptation. In: Engels G, Bencomo N (eds) 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings, 2–3 June 2014, ACM, Hyderabad, India, pp 125–134
28. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50
29. Kramer J, Magee J (2007) Self-managed systems: an architectural challenge. In: Future of software engineering, 2007. FOSE '07, May 2007, pp 259–268
30. Laddaga R (1997) Self-adaptive software. Technical Report 98–12. DARPA Broad Agency Announcement (BAA)
31. Letier E, Van Lamsweerde A (2002) Deriving operational software specifications from system goals. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT '02/FSE-10, ACM, New York, NY, USA, pp 119–128
32. Liaskos S, Khan SM, Litoiu M, Jungblut MD, Rogozhkin V, Mylopoulos J (2012) Behavioral adaptation of information systems through goal models. Inf Syst 37(8):767–783
33. Mongiello M, Grieco LA, Vogli E, Sciancalepore M (2015) Adaptive architectural model for Future Internet applications. In: Communications in computer and information science, advances in service-oriented and cloud computing. Springer, Switzerland
34. Mongiello M, Pelliccione P, Sciancalepore M (2015) AC-Contract: run-time verification of context-aware applications. In: Inverardi P, Schmerl BR (eds) 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, 18–19 May 2015, IEEE, pp 24–34
35. Oreizy P, Gorlick MM, Taylor RN, Heimhigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL (1999) An architecture-based approach to self-adaptive software. IEEE Intell Syst Appl 14(3):54–62
36. Pelliccione P, Tivoli M, Bucchiarone A, Polini A (2008) An architectural approach to the correct and automatic assembly of evolving component-based systems. J Syst Softw 81(12):2237–2251
37. Riccobene E, Scandurra P (2015) Formal modeling self-adaptive service-oriented applications. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, ACM, New York, NY, USA, pp 1704–1710
38. Salehie M, Tahvildari L (2009) Self-adaptive software: landscape and research challenges. ACM Trans Auton Adapt Syst 4(2):14:1–14:42
39. Salvaneschi G, Ghezzi C, Pradella M (2013) An analysis of language-level support for self-adaptive software. ACM Trans Auton Adapt Syst 8(2):7:1–7:29
40. Strang T, Linnhoff-Popien C (2004) A context modeling survey. In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004—The Sixth International Conference on Ubiquitous Computing, Nottingham/England
41. Swetina J, Guang L, Jacobs P, Ennesser F, Seung SJ (2014) Toward a standardized common M2M service layer platform: introduction to oneM2M. IEEE Wirel Commun 21(3):20–26
42. Torjusen AB, Abie H, Paintsil E, Trcek D, Skomedal Å (2014) Towards run-time verification of adaptive security for IoT in eHealth. In: Weyns D (eds) Proceedings of the ECSA 2014 Workshops and Tool Demos Track, European Conference on Software Architecture, 2014, ACM, Vienna, Austria, pp 4:1–4:8
43. Vögler M, Li F, Claeßens M, Schleicher JM, Sehic S, Nastic S, Dustdar S (2015) COLT collaborative delivery of lightweight IoT applications. In: Giaffreda R, Vieriu R-L, Pásher E, Bendersky

G, Jara AJ, Rodrigues JJPC, Dekel E, Mandler B (eds) Internet of Things. User-Centric IoT—First International Summit, IoT360 2014, Rome, Italy, 27–28 October 2014, Revised Selected Papers, Part I, vol 150, Lecture notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, pp 265–272

44. Vogli E, Ben Alaya M, Monteil T, Grieco LA, Drira K (2015) An efficient resource naming for enabling constrained devices in SmartM2M architecture. IEEE Int Conf Ind Technol (ICIT) 2015:1832–1837
45. Weyns D (2012) Towards an integrated approach for validating qualities of self-adaptive systems. In: Proceedings of the Ninth International Workshop on Dynamic Analysis, WODA 2012. ACM, New York, NY, USA, pp 24–29
46. Weyns D, Iftikhar MU, de la Iglesia DG, Ahmad T (2012) A survey of formal methods in self-adaptive systems. In: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, C3S2E '12, 2012. ACM, New York, NY, USA, pp 67–79
47. Weyns D, Malek S, Andersson J (2012) FORMS: unifying reference model for formal specification of distributed self-adaptive systems. ACM Trans Auton Adapt Syst 7(1):8
48. Weyns D, Schmerl BR, Grassi V, Malek S, Mirandola R, Prehofer C, Wuttke J, Andersson J, Giese H, Göschka KM (2010) On patterns for decentralized control in self-adaptive systems. In: de Lemos R, Giese H, Müller HA, Shaw M (eds) Software Engineering for Self-Adaptive Systems II—International Seminar, Dagstuhl Castle, Germany, 24–29 October, 2010 Revised Selected and Invited Papers, vol 7475, Lecture notes in computer science. Springer, pp 76–107