Information and Software Technology 111 (2019) 159–176

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

GoalD: A Goal-Driven deployment framework for dynamic and heterogeneous computing environments

Gabriel S. Rodrigues[a], Felipe P. Guimarães[b], Genaína N. Rodrigues[a,*], Alessia Knauss[c], João Paulo C. de Araújo[a], Hugo Andrade[c], Raian Ali[d]

[a] University of Brasilia, Brasilia, Brazil
[b] Brazilian Space Agency, Brasilia, Brazil
[c] Chalmers University of Gothenburg, Gothenburg, Sweden
[d] Bournemouth University, Bournemouth, United Kingdom

ARTICLE INFO

Keywords:
Autonomous deployment
Contextual goal modelling
Heterogeneous computational resources
Deployment planning

ABSTRACT

Context: Emerging paradigms like Internet of Things and Smart Cities utilize advanced sensing and communication infrastructures, where heterogeneity is an inherited feature. Applications targeting such environments require adaptability and context-sensitivity to uncertain availability and failures in resources and their ad-hoc networks. Such heterogeneity is often hard to predict, making the deployment process a challenging task.
Objective: This paper proposes GoalD as a goal-driven framework to support autonomous deployment of heterogeneous computational resources to fulfill requirements, seen as goals, and their correlated components on one hand, and the variability space of the hosting computing and sensing environment on the other hand.
Method: GoalD comprises an offline and an online stage to fulfill autonomous deployment by leveraging the use of goals. Deployment configuration strategies arise from the variability structure of the Contextual Goal Model as an underlying structure to guide autonomous planning by selecting available as well as suitable resources at runtime.
Results: We evaluate GoalD on an existing exemplar from the self-adaptive systems community – the Tele Assistance Service provided by Weyns and Calinescu [1]. Furthermore, we evaluate the scalability of GoalD on a repository consisting of 430,500 artifacts. The evaluation results demonstrate the usefulness and scalability of GoalD in planning the deployment of a system with thousands of components in a few milliseconds.
Conclusion: GoalD is a framework to systematically tackle autonomous deployment in highly heterogeneous computing environments, partially unknown at design-time following a goal-oriented approach to achieve the user goals in a target environment. GoalD has demonstrated itself able to scale for deployment planning dealing with thousands of components in a few milliseconds.

1. Introduction

Ubiquitous computing advances in sensing and information infrastructure made it possible to have ad-hoc networks and pervasive computing where solutions are composed on the fly according to the needs and context of a system. Smart phones, watches, TVs, and cars are examples of daily objects empowered by computing and networking capabilities. Such diversity of resources represents a highly heterogeneous computing environment, one formed by different types of devices, with different resources profiles and which are, at best, partially known at design-time. Ubiquitous Computing [2], Internet of Things (IoT) [3], Assisted Living [4] and Opportunistic Computing [5] are examples of computing architectures that have to be typically designed for such highly heterogeneous computing environments. Software deployment is the process of getting a software ready for use in its operational environment [6]. It includes selecting suitable artifacts to deploy, moving them to the target environment, configuring the environment, and starting the execution.

In highly heterogeneous computing environments, *Deployment planning* is especially challenging [7], given the variability and uncertainty of the host environment. Manual configuration would not scale as they require the deployment to be executed by a person with knowledge about the application internals and the environment dynamics [8]. The use of scripts is commonly used in cloud environments for an automated software deployment [9]. The scripts are designed at design-time and strive to cater for different environmental conditions. *Software*

* Corresponding author.
E-mail addresses: gabrielsr@aluno.unb.br (G.S. Rodrigues), felipe.guimaraes@aeb.gov.br (F.P. Guimarães), genaina@unb.br (G.N. Rodrigues), alessia.knauss@chalmers.se (A. Knauss), sica@chalmers.se (H. Andrade), rali@bournemouth.ac.uk (R. Ali).

https://doi.org/10.1016/j.infsof.2019.04.003
Received 2 July 2018; Received in revised form 1 April 2019; Accepted 8 April 2019
Available online 9 April 2019
0950-5849/

store is another alternative approach where developers upload metafiles to the store for software configurations, target devices and their fitness. The deployment would require end-users to access the store interface, searching for the application, and initiating the installation of the application which may require also some further information and permissions from the users about their devices. Both approaches do not cater for highly heterogeneous environments as variability is limitedly handled through the preparation of scripts and metafiles to deal with a set of common environmental and devices profiles, known a priori.

We argue that the challenges related to deployment in highly heterogeneous computing environments are threefold: (i) heterogeneity: the system is meant to run in a broad range of configurations of computing environments;(ii) uncertainty at design-time: the system architect/developer cannot precisely ascertain the configuration of the end user computing environment; (iii) autonomous deployment: the need for autonomy in the system to cater for the ad-hoc environment settings. To provide a systematic engineering approach and autonomous support to assist deployment, such rationale shall be captured and followed.

In fact, solutions for such autonomous deployment have been already proposed in the realm of business process orientation [10,11]. Nevertheless, such approaches may render too heavyweight for being deployed on resource-constrained devices [7]. Moreover, business process based solutions will be able to either orchestrate or choreograph once the components are already deployed, since it will only be able to manage the executable methods, services and events to the corresponding workflow engines for execution [12] once the required and suitable component infrastructure is deployed. There are also proposals to address autonomous deployment in the realm of dynamic software product lines [13–15] where the focus is on devising adaptation strategies based on feature configurations for deployment. Nevertheless, configuration and behavior are related and it is not always possible to change one without changing the other:

> The need for both capabilities of independent yet coordinated adaptation of behavior and configuration requires an extensible architectural framework that makes explicit how different kinds of adaptation occur [16].

More recently, goal-oriented engineering has paved the way for solutions able to leverage seamless configurations, behavior, and strategies in self-adaptive systems [17–20]. As Weyns points out as the fourth wave on engineering self-adaptive systems:"… *(goal-driven adaptation) puts the focus on key elements for the concrete realization of self-adaptive systems"* [20]. In particular, goal modelling captures what stakeholders intend to achieve in terms of goals and alternative strategies they may adopt. Such strategies can be expressed through AND/OR refinements, rendered as alternative sets of executable tasks [21–23]. Context Goal Models (CGMs) extend goal models [24] by adding contextual conditions on the need for goals and suitability of such alternatives. GORE has been also used to guide and architecture design [25–29]. The variability in goal modelling provided a basis for the engineering of self-adaptive systems [18,30,31]. CGM would be a suitable baseline model for our highly heterogeneous environments, both for their accommodation of the variability in the context and strategies and also for their ability to capture experts' rationale as a decision tree.

In this paper, we propose GoalD as a framework that follows a goal-oriented approach for deployment in highly heterogeneous computing environments. Our work is intended to investigate the capability of contextual goal modeling to help manage variability at deployment in a scalable fashion. The GoalD framework comprises an offline and also an online phase. In the offline phase, GoalD relies on an abstract model, which can be seen as a domain model, which presents (i) *what* the system is required to achieve (i.e., the goals), (ii) *how* it can achieve the goals (i.e., its alternative strategies) and the corresponding tasks and their software components and (iii) the *restrictions* to the strategies (i.e., the resources and contexts needed). In the online stage, GoalD uses automated reasoning to encompass (i) the deployment planning to decide on suitable bundles for the target computing environment and (ii) the architecture deployment configuration by fetching and binding artifacts from their repositories. In addition to that, one of the main points of novelty of our approach is the linkage between the configuration process and the strategic interests and requirements of the system. This helps rationalising the selection process and validating it against the system's main mission and strategies to meet it, modelled as pathways to reach goals. The algorithms and the reasoning of GoalD and its knowledge base are built for that level of presentation and this would also make their interpretation easier. GoalD facilitates taking a decision at the early stages of development and eliminating strategies which can become cumbersome or redundant and will also detect situations where additional equipment is needed. We harness these benefits of Goal Oriented Modelling approaches for this problem.

As a potentially relevant application sector for GoalD can be health care and well-being solutions, we evaluate GoalD on a TeleAssistance System (TAS) case study, which was originally introduced in [32] and further implemented as an exemplar by Weyns and Calinescu in [1] to be used for evaluations of self-adaptive solutions. Our results demonstrate the usefulness and scalability of GoalD in planning the deployment of a system with thousands of components in a few milliseconds.

The paper is structured as follows: Section 2 introduces the background and illustrative scenario. Section 3 presents our proposed GoalD approach. Section 4 reports on our evaluation results of GoalD. Section 5 depicts related work and Section 6 concludes the paper and outlines future work.

2. Goal model and contexts

Goal-oriented analysis is a powerful technique to capture and represent the systems and software requirements at the intentional level, the goal level, and the strategies that can be adopted to achieve them. The variability representation in such Goal Oriented Requirements Engineering (GORE) makes it suitable for developing adaptive systems [33]. Tropos [21] is one of the methodologies that utilizes goal modeling for depicting requirements at design time. The goals in Tropos are associated with actors (social or system actors) and may be hierarchically refined via AND/OR decompositions into other goals or tasks/plans which ultimately achieve them by means of executable processes. Contextual Goal Models (CGM), proposed in [24], also capture the relation between the environment surrounding the system, e.g., the computing resources or weather status, and its goal achievement strategies. *Context* is a concrete description of the environment presented in a manageable, decidable, and formalized form. In CGM, contextual conditions are associated with variation points in the goal analysis hierarchy that enable decisions on which alternative strategie(s) to adopt. Similar to goal analysis, context analysis allows to refine a high level environment description into a formula of verifiable pieces of information (facts).

2.1. Tele assistance system

To illustrate our approach throughout the paper, we will use the Tele Assistance System (TAS) [32] based on the reference implementation of the exemplar provided by Weyns and Calinescu [1]. TAS is a technology service that provides care to elderly and chronically sick people, enabling them to continue to live in their own homes. We have chosen this particular example to make it easy to point out variability in environments and resources.

In an assisted-living scenario, with wearable technologies, it is clear that the environment surrounding the user may vary. While designed to be used at home, TAS probably would be usable outside, in a mall or in a friend's house. Additionally, the combination of embedded sensors may vary due to power shortages or mere lack of necessity in some

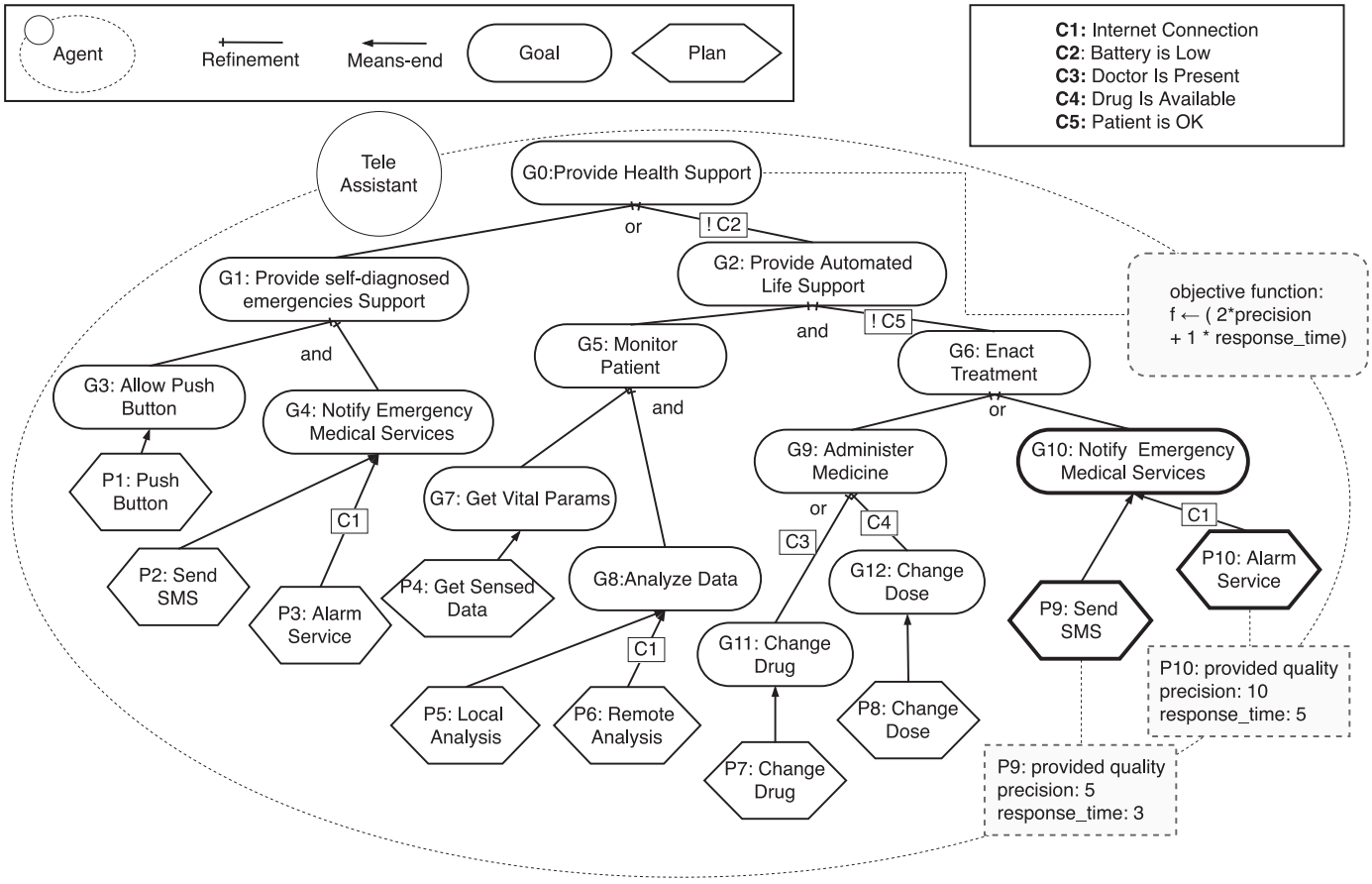


Fig. 1. CGM of the Tele Assistance System (based on [1,32]).

cases or situations. These "situations" are what we understand as the CGM's contexts: "user is asleep", "night time", or "user is at home". A resource availability may similarly be understood as a context (e.g., "user's cardiac rate is measurable").

The CGM presented in Fig. 1 depicts a CGM that may represent TAS behavior and intents. In this example the goals or tasks P1, P3, G7, G8, P7, P8 and P10 emulate the TAS behavior as described in [32]. For the sake of argumentation and to proper exemplify the features introduced by GoalD, we have further decomposed some of those goals into different alternatives. This allows GoalD to adjust TAS's expected behavior under different contexts. It should be noted that, under the context [C1, C3, C4], the original behavior is still valid according to this CGM. In our model, the root goal *G0: Provide Health Support* is AND-decomposed into two sub-goals: *G1: provide self-diagnosed emergencies support* and *G2: Provide automated Life Support*. Each of these is then further refined into more fine-grained goals or tasks. For AND-decompositions like the one from G0 every sub-goal must be achieved whereas in OR-decompositions like G3 any applicable alternative is suitable.

Every OR-refinement in the model introduces variability to the system, allowing it to achieve the root goal in different ways. Each of these different alternatives may provide different quality of service (QoS) levels. Examples of QoS levels can be seen illustrated in tasks P9 and P10 of Fig. 1. The tasks P9 and P10 of our example provide different precision and response times. Additionally, the model designer defines a utility function for the goal model to define what it understands as a better alternative. GoalD then combines the expected QoS levels and the applicable configurations to choose the one which is expected to provide the higher QoS levels for the user under the current context.

The CGM also presents some context annotations on some edges, like the one linking goals `G2: Provide Automated Life Support` and `G6: Enact Treatment`. These annotations represent the context(s) in which an alternative is applicable. For instance, Enact Treatment (goal G6) is only available when the patient is not ok (!C5). The context conditions of the goals propagate to their AND-OR refinements. During the time that the patient is well, Change Drug (goal G11) and Change Dose (goal G12) are not applicable. Similarly, on OR-decompositions, every context restriction also introduces variability to the model although such variations are not within the system's control and may improve or impair the system without prior notice. This is why we believe that handling such changes effectively is paramount to a system in a highly heterogeneous computing environment.

3. The GoalD approach

Our approach for goal-driven automated deployment (GoalD) is divided into two separate stages: *offline* and *online* activities. This differentiation is in line with the established concepts on software development processes for variability configuration and adaptive systems (e.g., [34]), in which variability and self-adaptation capabilities are designed and implemented in the offline stage. It includes the preparation of software bundles which will be used during the online stage activities: dynamic planning and automated deployment. This enables us to tackle heterogeneity and constant changes that are quite intrinsic to modern computing environments, such as IoT, cloud or wearables. Furthermore, it allows us to handle a large spectrum of available alternatives to provide better QoS levels to the users and deal with the context-dependent applicability of such decisions and to foresee the impact of the adaptation on the overall QoS.

The main objective of GoalD is to provide an adaptable deployment framework to harness the decision making complexity to provide a higher levels of quality to the users, even in the face of contextual

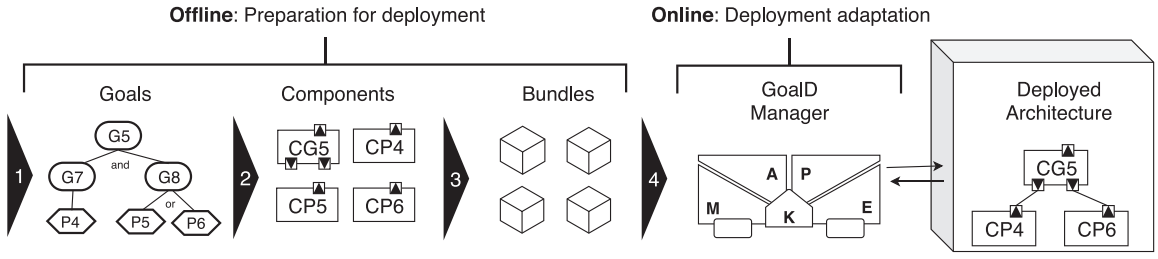


Fig. 2. Overview of the steps in GoalD.

variations and unavailability of resources. Fig. 2 provides an overview of GoalD's two stages (*online* and *offline*).

At design time, the *offline* stage encompasses most of the creative work, usually carried out by software engineers devising the goal model and applying patterns to concretize the conceptual model into components which will be used during the *online* phase. Starting out from the system's contextual goal model, in activity (1) the specialists map the requirements to their contextual dependencies and to the QoS levels they are expected to provide via model annotations (see Section 3.1). Activity (2) comprises the application of the mapping proposed in Section 3.2 effectively extracting composable software components from the contextual goal model. Finally, during activity (3) these components are packaged, described by metadata, and published in a repository (see Section 3.2.1).

Since GoalD does not make any assumptions on the underlying application, the definition of a quality measure is deliberately wide, leaving the semantics to the analyst defining it. These QoS levels may then be expressed in both goals and tasks in order to select the dependencies delivering the highest expected QoS levels through the utility function defined for the model. An example of the application of the utility function could be on the means-end selection for Goal G10. This goal may be implemented through tasks P9 or P10. Assuming that context C1 is active and the defined utility function ti be ($precision * 2 + response\_time * 1$) then task P10 would be selected with an expected QoS level of 25 ($10 * 2 + 1 * 5$) instead of the task P9, which would give a QoS level of 13 ($10 * 2 + 1 * 3$).

In activity (4), following Fig. 2, the *online* stage of GoalD concerns itself with the proper selection of applicable components for the system's environment at runtime. GoalD utilizes automated reasoning and scalable algorithms to support this activity. Fig. 2 represents the autonomous deployment by the GoalD manager, which autonomously builds deployment strategies at runtime following the MAPE-K loop to decide on the suitable bundles to be deployed, e.g., capable of providing the highest QoS level for the root goal under the current computing environment; and handling deployment adaptation reactively and proactively to respond to context changes by finding redeployment alternatives to newly available contexts. Section 3.3 gives further details for the online stage of GoalD.

3.1. Goal modeling in GoalD

GoalD starts from a generic CGM and specializes it with a richer *resource* notion. It maps goal hierarchies to software components and environmental and resource dependencies to contextual restrictions, hampering the applicability of a goal decomposition alternative. This allows us to plan and deploy software onto variable and dynamic computing resources under dynamic external environments. In this sense, from now on we assume that a CGM context may either represent a reification of the system's surroundings or the availability of a given resource in the computing environment. As such, the target computing devices and their corresponding available resources can be associated with a fully qualified context. Therefore, we may consider a set of resources/contextual conditions relating to the available resource set as a `context` in the environment.

We illustrate this based on the TAS example (Fig. 1). The context conditions are either computational resources (`C1`: Internet connection) or environmental restrictions (`C5` - *Patient is OK*). The contextual conditions are represented as labels on the goal decomposition edges and represent that the associated computing capability must be available in the environment to consider this decomposition viable.

Let us consider the sub-tree beneath `G9: Administer Medicine` and a world situation in which internet connection is currently available (`C1`), a doctor has just arrived (`C3`) and no drug is being administered (`!C4`). In this scenario, changing the dosage of a medicine is not viable since there is no previous dosage to be changed but all the remaining alternatives are applicable. Therefore the enactment of a plan under these circumstances would necessarily involve `G9: Administer Medicine` or `G10: Notify emergency services`. Task `P7: Change drug` is currently available and may be used in an enactment plan it depends on a context (`C3 - Doctor is present`) which is currently active. On the other hand, plans involving `P8: Change dose` are automatically deemed unfeasible due to the nonexistence of the necessary context (`C4`).

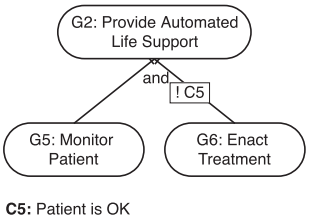
Do note however that since context `C2` is active, goal `G9` is achievable through tasks `P7` and `P8` independently. The availability of multiple decomposition alternatives like this case, any of which could achieve the desired results, enables GoalD to choose the one offering the higher QoS alternative. The GoalD framework includes the capability of estimating these QoS levels and choosing the alternative with the highest quality. This selection method is described in Section 3.3.

To enable this feature the software engineers must: (1) define the provided QoS levels of each available task and (2) include at the goal a utility function for composing all of the QoS levels of each child tasks/-goals into a single and comparable metric. To do so, each task (leaf node) is annotated with a set of tuples $\langle quality_i dentifier, value \rangle$ where $quality_i dentifier$ is a string that uniquely identifies the metric and *value* is a positive numerical representation, where higher values signify higher qualities. Goals on the other hand are annotated with another set of tuples: $\langle quality_i dentifier, weight \rangle$ where $quality_i dentifier$ is the string identifying the metric and *weight* is a numerical value that will be used as a multiplier for such metric when instantiating the goal's utility function.

Given that the available context of a plan can only be known at deployment time, the model encompasses them all and uses the available contexts as a filter on the goal's alternatives upon deployment. A context condition is said to be satisfied if its associated resources are present in the current state of the environment or if its world condition is active. In Fig. 1, goal *G9: Administer Medicine* has two alternatives to be achieved: by executing task `P7: Change Drug` or `P8: Change Dose`. The enactment of a plan involving task `P7` is feasible if, and only if, the context condition `C3` (Doctor is present) is satisfied. In summary, our CGM's context conditions act as restrictions on the applicability of a plan.

3.2. From goals to components

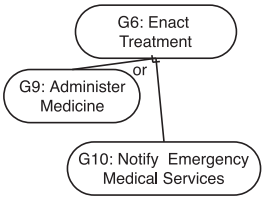
Components are units of composition with contractually specified interfaces and explicit context dependencies [35]. Components and




```
interface ProvideAutomatedLifeSupport
    {}
interface MonitorPatient {}
interface EnactTreatment {}

component
    ProvideAutomatedLifeSupportImpl {
  provides ProvideAutomatedLifeSupport
      ;
  requires [
    MonitorPatient,
    condition(!patientIsOk,
        EnactTreatment)];
}
```

Fig. 3. Model excerpt of an AND-refined goal and corresponding definitions.




```
interface EnactTreatment {}
interface AdministerMedicine {}
interface NotifyEmergencyMedicalServices
    {}

component EnactTreatmentImpl {
     provides EnactTreatment;
  requiresAny [
    AdministerMedicine,
    EnactTreatment
  ];
}
```

Fig. 4. Model excerpt of an OR-refined goal and corresponding definitions.

interfaces can be described using architecture description languages (ADLs). Yu et al. [31] adapt the Darwin ADL [36] with elements borrowed from one of its extensions, namely Koala [37]. They also provide a method for relating goals with components, but without taking into account context conditions.

In GoalD we build on Yu et al. proposal [31] ACL to define patterns to map CGM elements into architectural elements. Since these patterns are able to convert both OR- and AND-decompositions, their application result in a specification for the implementation architecture of the system. In particular, it allows the identification of components in need of implementation from the goal hierarchy pathways of the CGM.

The system architectural elements are `components` and `interfaces`. An `interface` element represents a contract between different components. Therefore a `component` must declare the interface(s) it adheres to and `provides` functionality for. It may also declare the interface(s) it `requires` functionality from. These metadata elements (`interfaces`, `provides` and `requires`) and the *architecture specification* may be algorithmically derived from the original CGM, based on the following derivation patterns for AND- and OR-refinements. Finally, the outputted components and interfaces should be included with concrete implementations by the system's developers.

For each AND-refined goal, the conversion pattern will produce:

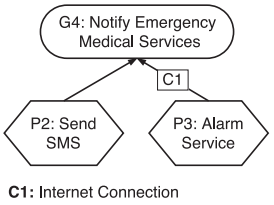
1. An interface specification for the goal;
2. An interface specification for each of its refinements;
3. A component specification that provides the interface defined in 1, requires the interfaces defined in 2 possibly having a conditional statement describing the context in which it is necessary.

To illustrate the application of this pattern, let us consider goal `G2 - Provide Automated Life Support`. This goal is decomposed into `G5 - Monitor Patient` and `G6 - Enact Treatment` through an AND-refinement. This allows the presented pattern to be used. The application of step 1 will result in the definition of an interface for the goal itself `interface ProvideAutomatedLifeSupport{}`, step 2 will result in two other interfaces, one for each refinement (`interface MonitorPatient{}` and `interface EnactTreatment{}`). Finally, step 3 will require a `component ProvideAutomatedLifeSupportImpl`, which provides interface `ProvideAutomatedLifeSupport`. This component will always require interface `MonitorPatient` and, conditioned to context `patientIsOK` being false, also require `EnactTreatment`. The resulting ADL is depicted in Fig. 3.

In a similar fashion, OR-refinements will result in a component, but in this case, there is no need to deploy all of them, therefore replacing the `requires` with the `requiresAny`. Whereas in an AND-refinement, an unachievable sub-goal incurs that the root goal is also unachievable, in an OR-refinements it merely means that it should not be deployed. This goal will only be deemed unachievable if all of its sub-goals are also unachievable. Fig. 4 illustrates an OR-refinement, where `G6 - Enact Treatment` is refined into `G9 - Administer Medicine` and `G10 - Notify Emergency Medical Services`.

This pattern can be repeated indiscriminately since both AND- and OR- refinements will always result in the generation of components. This cycle will only be broken when we consider means-end refinements. A Means-end refinement depicts all the different strategies or manners in which a goal may be implemented. From such refinements will derive one interface specification and multiple components specifications that provide the implementation of such an interface. Thus, the pattern for the means-end refinement of a goal is as follows:

1. An interface specification for the goal itself and;
2. A component specification for each of its refinements which may depend on a context;




```
interface NotifyEmergencyServices {}

component AlarmService {
  provides NotifyEmergencyServices;
  condition InternetConnection;
}

component SendSms {
  provides NotifyEmergencyServices;
}
```

Fig. 5. Model excerpt of a Means-End refinement and corresponding definitions.

For illustrating this pattern we may focus on goal `G4 - Notify Emergency Medical Services`, as depicted in Fig. 5. This goal is a means-end refinements achievable by either `P2 - Send SMS` or `P3 - Alarm Service`. If we apply the conversion pattern to goal `G4`, it would produce `interface NotifyEmergencyService` (step 1) and components `SendSms` and `AlarmService` (step 2).

It is noteworthy that, differently from AND- and OR-refined goals, the components derived from a means-end refinement provide, *i.e.,* implement, their parent node's interface. This means that either component may be used to achieve the decomposed goal. This is also one of the main sources of the deployment variability scope, since AND-refinements do not pose a choice at all and OR-refinements will always attempt to deploy as many as possible. From an architectural level perspective, this variability also characterizes the decoupling of components, in accordance to the goal model.

Whenever a refinement edge has an associated context restriction, such restriction is mapped to a `condition` element on the component, as exemplified by `AlarmService` component's `condition InternetConnection`. In this case, the plans are associated with the context `C1 - Internet Connection`. Particularly, contextual restrictions associated with OR-decompositions may limit the applicability of a given alternative under certain computing environment contexts.

By associating AND/OR-decompositions and context conditions in the component analysis, we may map the CGM's variability and its conditions to the system's architecture itself. For example, components `AlarmService` and `SendSms` both provide the same interface (`NotifyEmergencyServices`) but with different context conditions. This means the same goal may be achieved by deploying either component as long as its required contextual condition is in effect.

By using the proposed patterns, the variability presented in the goal model is catered for usage on the system's architecture. Such architectural variability enables deployment adaptation in response to heterogeneity in the target computing devices and to the system's surrounding environment.

Once the component/interface is mapped by using GoalD patterns, including those interface details should be a simple matter of further specifying the component/interface with its proper signatures.

3.2.1. Packaging components

As previously described in Section 3.2, at the architectural level goals are contractually specified as interfaces which, in turn, are implemented by components. From a deployment point of view, components and interfaces represent deployment units and should be packaged as files for distribution, usually referred to as *bundles*. GoalD bundle packages encompass components, interfaces and their associated metadata in order to allow automated deployment.

Bundles metadata describe the bundle's contents: its goals, dependencies, and context conditions. They are characterized by the following information: (i) `name`, (ii) `conditions`, (iii) `defines`, (iv) `implements` and (v) `depends`. Firstly, `name` metadata uniquely identifies a bundle. Secondly, `conditions` metadata describes the context conditions needed to deploy a bundle onto any given environment. Lastly, `defines`, `implements`, and `depends` metadata specify the bundles inter-dependencies and are used to analyze contractual responsibilities between bundles. The attribute `defines` declares that this bundle *defines* the contract for a set of goals; `implements` declares that this bundle provides a possible *implementation* for a set of tasks/goals; `depends` declares that this bundle has a dependency relationship towards other bundles that *define* and *implement* a goal/task.

All components packaged together in a bundle will be delivered together. In order to favor low coupling in the GoalD approach, components and interfaces should be packaged into separated bundles, so they can be delivered only to environments where they are required and further used. In order to maximize the flexibility of systems following the GoalD deployment approach, we package interfaces and components in two bundle types: *definition* and *implementation*, respectively.

A *definition* bundle packages interfaces – In a *definition* bundle, the metadata `defines` contains the list of goals it provides definition for. For example, the interface definition for goal *G0* of the Tele Assistance System, namely `Provide Health Support`, is packaged into a *definition* bundle along with the following metadata:

```
name: ProvideHealthSupport.def
defines: ProvideHealthSupport
```

Implementation bundles package components – In such bundles, the `implements` metadata contains the list of goals provided by its packaged components; the `depends` metadata contains the list of interfaces that packaged components depend on and; the optional `quality` metadata defines a mapping between arbitrary labels and their related values. As an example, the components for goal *G2* presented in Fig. 3, namely `Provide Automated Emergency Support`, is packaged in an *implementation* bundle along with the following metadata:

```
name: ProvideAutomatedLifeSupport.impl
implements: ProvideAutomatedLifeSupport
depends:MonitorPatient, conditional(!patientIsOk,
    EnactTreatment);
quality: [
    response_time: 18
]
```

Note that `ProvideAutomatedLifeSupport.impl` depends on the interface definition and actual implementation of `MonitorPatient` and occasionally `EnactTreatment`. Therefore, at deployment time, components that `defines` and `implements` interface `MonitorPatient` and `EnactTreatment` should be included in order to successfully deploy `ProvideAutomatedLife Support.impl` bundle.

Finally, the `condition` metadata of *implementation* bundle reflects the context conditions of packaged components. For example, in the Tele Assistance Service, the components `AlarmService` and

SendSms are packaged into separate bundles with the following metadata:

```
name: AlarmService.impl
implements: NotifyEmergencyServices
conditions: InternetConnection
quality: [
    precision: 10
    response_time: 5
]
name: SendSms.impl
implements: NotifyEmergencyServices
quality: [
    precision: 5
    response_time: 3
]
```

AlarmService and SendSems are bundles that implement the same goal NotifyEmergencyServices. However, SendSms is always applicable while bundle AlarmService depends on context InternetConnection being active. This architectural scheme keeps the variability introduced by the CGM as well as the components' decoupling at deployment level. Do note also that both bundles advertise their expected quality level. These bundles do advertise their expected precisions. These quality levels may have any label and any positive value. The semantics for the value is that the higher the value, the better QoS is provided. The lack of precise definition of the labels and the simplistic interpretation of its attributed value are due to the fact that thoroughly defining a QoS level and its comparison is beyond the scope of this paper. It serves mostly as a way to indicate such levels and enable the algorithm to choose between those available options. Such quality levels should be introduced by the bundle's software designers during packaging.

After components and their corresponding metadata information (quality levels, goals and contexts dependencies) are properly packaged into bundles, they are registered to a repository so that they can be distributed to the target devices. During the registration process, a bundle is uploaded to the repository and its metadata is processed and stored in the repository database. This way such information may later be queried for the next step of GoalD: deployment planning.

3.3. The autonomous deployment in GoalD

During the online phase, the focus shifts from the system's definitions to the autonomous deployment and adaptation of the packaged bundles in response to context changes. In this stage, GoalD (1) monitors the currently active contexts, (2) takes such context set alongside the currently available computing devices into account to pinpoint applicable alternatives, (3) estimates the delivered QoS of each alternative, (4) picks the one expected to deliver the highest QoS levels and list the bundles needed by it, (5) deploys those bundles onto the computing devices available, and (6) restart the cycle by monitoring contextual changes. This new monitoring phase enables GoalD to identify and respond to newly available contexts by adapting the current deployment. This adaptation may either reactively handle a bundle that is not applicable anymore or proactively increase the provided QoS levels by choosing alternatives that became applicable, without disrupting the system's proper execution.

The starting point for the deployment process is the explicit definition of the root goals the stakeholders want to achieve/deploy and the target computing environment via a deployment request. From this set of goals and their CGM sub-trees, GoalD creates a DVM (Deployment Variability Model) relating its goals, contexts, available bundles, and provided qualities. A DVM is a tree-like structure composed of Variability Elements (VE) that describe the alternative implementation options for each goal, ordering them by the expected delivered quality level. DVM follows the principles of a model at runtime [38]. Through the analysis of the alternatives at each node of the DVM tree, GoalD is able to reason about the deployment and choose, for the current context, the alternatives set providing the best QoS and devise a deployment plan to enact it. Such deployment plan consists of a set of deployable bundles that allow for the achievement of all selected root goals on the available computing environment upon deployment and execution. The deployment itself is executed by fetching the selected bundles from the repositories and binding them to the available resources. At runtime, GoalD stores the DVM in order to respond to eventual context changes through re-planning the deployed components and adapt the deployment accordingly.

After executing the deployment according to the goal model, the managed system is considered *available*, which means that it is able to fulfill its defined goals as long as the required resources are available. In an event of a context change, the availability of the system could be threatened if the required resources render unavailable. In case a resource becomes unavailable, GoalD can recover from the system availability by adapting the deployment if there is an applicable alternative. Then, the system will be unavailable until an adaptation occurs by deploying bundles of the applicable alternatives. As such, the unavailability time encompasses the time to identify the context change, analyse it, come up with an adaptation plan, and execute that plan.

For example, in TAS, in an event of loss of the internet connection, the system would render unavailable if the Alarm Service was deployed at the moment of the connection loss and no Send SMS task is deployed. TAS would become available again by deploying Send SMS as it is an applicable alternative to fulfill goal G4: Notify Emergency Medical Services, following Fig. 1.

However, in an event of a change that allows for an increase of the provided QoS, there is no unavailable period as the system continues to operate with the current deployment setup while the adaptation cycle is executed simultaneously.

In this section we present the GoalD framework in detail: (1) the runtime framework, (2) its knowledge metamodel and (3) the algorithms to synthesize and update the DVM as well as to create a deployment plan that best suits the system goals.

3.3.1. GoalD manager and runtime framework

Fig. 6 illustrates the perspective of the self-adaptation control loop of our GoalD Manager, realized through a MAPE-K feedback loop [39]. A MAPE-K loop consists of five elements, the Monitor, Analyze, Plan, Execute, and Knowledge Base. The *Knowledge Base* stores all necessary information (also those produced by the MAPE elements). For example, it keeps a model of the system goals (the DVM), the system's current deployment, and also a data structure to correlate each context to the VEs that depend on it (Context-VE Mapping).

The *Monitor* subsystem is responsible for identifying the system's current operating context and notify the Analyze subsystem whenever a *context change* occurs.The *Analyze* subsystem updates the DVM identifying parts rendered inapplicable under the new context and/or evaluate newly available alternatives for the possibility of delivering higher quality levels than those currently being delivered. The *Plan* subsystem is then triggered to (1) parse the updated DVM; (2) envision a new system deployment applicable under the changed environment context and; (3) engineer a deployment plan for adapting the previous deployment into the new one. Finally, the *Execute* subsystem performs the adaptation through the removal of components that have become unnecessary and the installation of the newly required ones.

Whenever the *Monitor* subsystem identifies a context change, the *Analyze* subsystem is immediately triggered to evaluate the need of locally adapting the model. Adapting locally decreases the analyzing cost and, consequently, the recovery time. However, should a context change incur no effect on the deployment, the *Analyze* subsystem will proactively act by attempting to identify newly viable alternatives to adapt into a

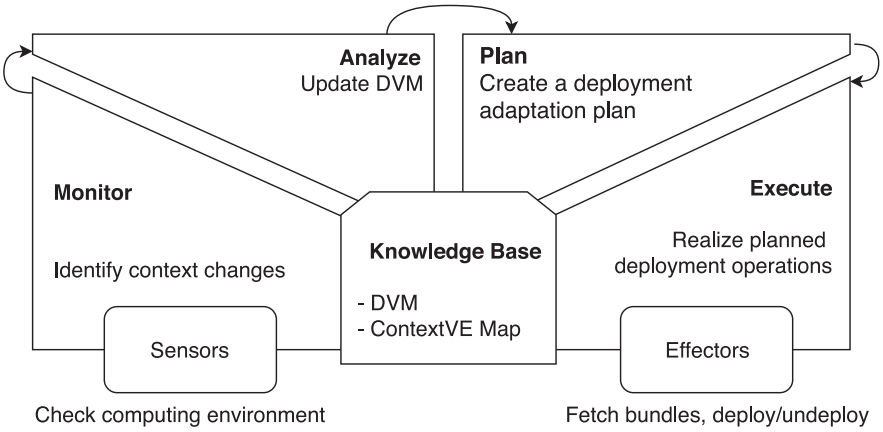


Fig. 6. MAPE-K Perspective of GoalD.

deployment setup to deliver a higher QoS than the current deployment. If such an opportunity is found, the *Plan* subsystem will update the DVM and ultimately construct a *deployment plan*. This plan is forwarded to the *Execute* subsystem for deployment. Upon receipt of a deployment plan, the *Execute* subsystem will carry out the planned deployment operations. These operations involve the *fetching* of the bundles in the deployment plan from the repository, the subsequent *install* or *uninstall* of bundles and the wiring of software components. The *Execute* subsystem is also responsible for updating the *Knowledge Base* with the current deployment state.

3.3.2. Knowledge metamodel

To properly handle context changes, GoalD makes use of a *Knowledge Base*. GoalD's *Knowledge base* consists of the system's DVM and a Context-VE mapping derived from the components' metadata. The DVM is a model depicting the available alternatives and it is constructed from information extracted from the bundles available at the *Repository*. The Context-VE mapping consists of a correlation between each context and the set of VEs requiring it. This mapping is used to assert whether a context change render the current deployment unapplicable.

The deployment metamodel of GoalD is the underlying data structure that defines major conceptual elements of the framework's autonomous deployment. The metamodel of GoalD consists of seven major elements: (1)*Variability Element (VE)*, (2)*Context*, (3)*Bundle*, (4)*KnowledgeBase*, (5)*Repository*, (6)*Dependency* and (7)*Alternative*. These elements are used to reason about the system goals, context and deployment at runtime. Fig. 7 presents the GoalD metamodel.

A *VE* reifies a goal and contains a *Bundle* that `defines` such goal and a set of *alternatives*. In particular, *rootVE* represents the most important, significative and abstract goal to be achieved by the system.

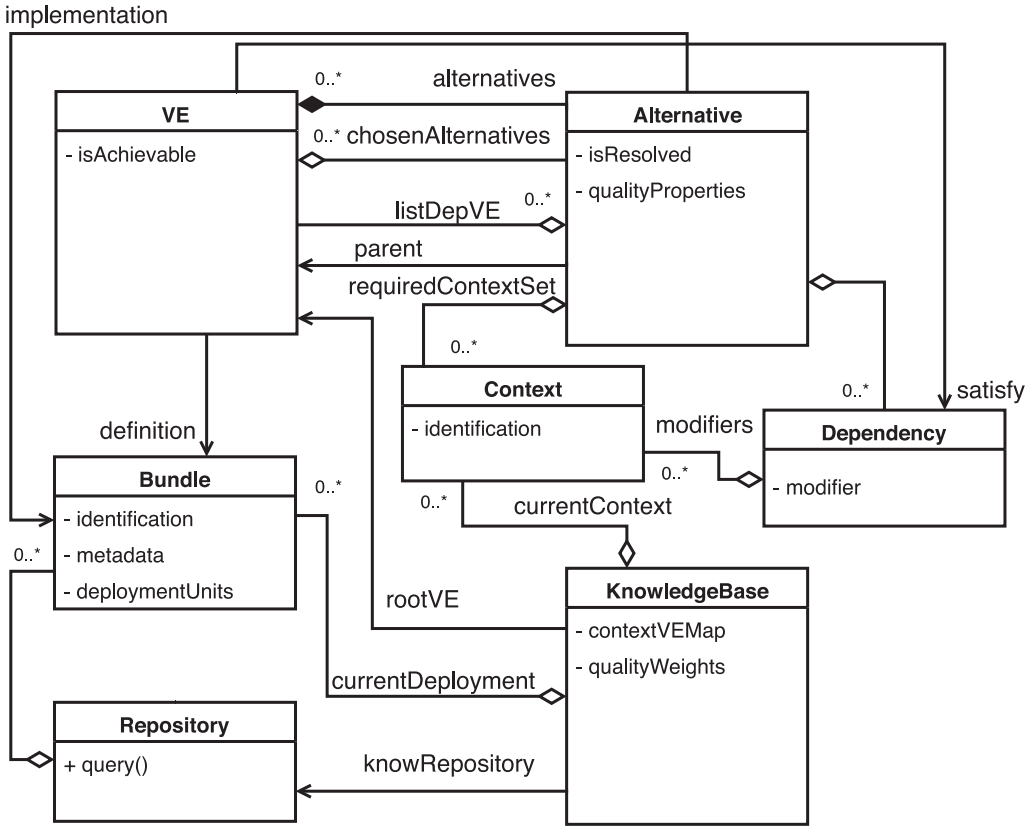


Fig. 7. GoalD-runtime metamodel.

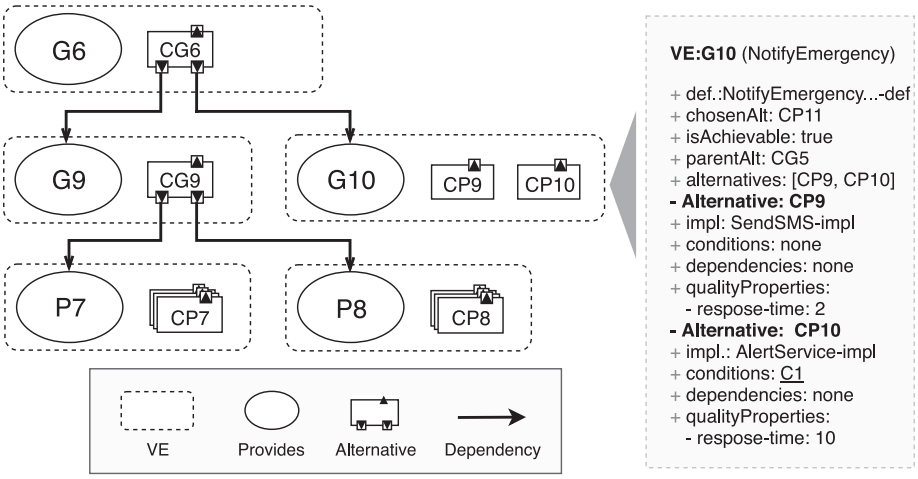


Fig. 8. DVM: Deployment Variability Model.

For TAS, this goal is `G0: Provide Health Support`. Each *Bundle*, as previously described in Section 3.2.1, has a unique identifier, the deployment units and some metadata describing the goals/tasks they *defines, implements* and/or *depends* on. It is through such metadata that the relationships between bundles are represented: a bundle with dependencies relies on other bundles to provides their *definition* and *implementation*. An `Alternative` represents a possible strategy for resolving a *VE*.An *Alternative* aggregates a set of `Dependencies` that, whenever deployed altogether, are able to achieve the *VE*'s goal. It also advertises the expected QoS level it is expected to deliver (`qualityProperties`) and the contexts in which it is will or will not be applicable. A *Context* in GoalD represents either a resource's availability or a world predicate at any given moment. A *Dependency* has a modifier that represent whether it is compulsory (AND-decompositions), optional (OR-Decompositions) or conditional (Context dependencies).

Repository stores the DVM and may be queried via the `query` method. This method receives a set of dependencies as an argument and returns all the *VE* objects that implement such dependencies. It is this intricate dependency relationship between different goals via *depends* and *implements* annotations that defines the complete DVM model. Another important constituent part of the runtime metamodel of GoalD is the *KnowledgeBase*. The *KnowledgeBase* gathers the runtime models for a given computing environment. It stores the DVM, represented by the *rootVE*, the DVM's root node and a Context to VE mapping (*ContextVEMap*). This map is used to pinpoint the goals affected by a given context change. It enables GoalD to efficiently evaluate a change and optimize the adaptation process.

Fig. 8 illustrates the DVM correspondent to TAS CGM's (see Fig. 1) sub-tree rooted at goal `G6`. This Figure depicts the VEs for goals `G6`, `G9` and `G10` as well as tasks `P7` and `P8`. At each VE there is a set of possible component alternatives (`CG9`, `CP10`,...). `G6` has only one alternative given that it is an AND-decomposition that necessarily needs both `G9` and `G10`. A similar logic applies to `G9`. On the other hand, since `G10` has two means-end alternatives, it is left free to choose component `CP9` and/or `CP10`.

An alternative is deemed *resolved* if its context restrictions are satisfiable and it either (1) has no dependencies (like CP7, CP8 CP9 and CP10) or (2) all of its dependencies have already been resolved. In turn, a VE is considered *achievable* if it is possible to find at least one resolvable alternative. Should there be multiple deployable alternatives, the choice of any given alternative is conditioned to its applicability in the current environment and the level of quality it advertises. The chosen alternative (*chosenAlternative*) will be the one providing the highest QoS levels. Each *Alternative* is associated with an implementation bundle. An *Alternative* also has its own set of dependencies, reflecting the dependencies of its associated bundle. Such process is performed by GoalD following the elements defined by the MAPE-K loop.

3.3.3. Monitor

As previously stated, the Monitor subsystem is responsible for identifying the system current operating context. The implementation of such monitoring is very dependent on the context(s) it is supposed to monitor. Therefore, the context monitoring module is out of scope of this paper. However, the role it plays is paramount for the adaptation mechanism, in the sense that the Monitor subsystem is the trigger of the whole adaptation process.

3.3.4. Analyze

The Analysis subsystem is used to update the DVM of the system according to its goals and current context. Analyze is called during the system start up and in the event of a context change. However, handling an initial deployment is slightly different from handling a context change. During the first deployment managed by GoalD has no DVM to guide itself. On the other hand, while handing a context change the system already has an instantiated DVM and can take advance of it in order to optimize the analysis.

In order to create an initial deployment, GoalD queries the repository for the root goal which results in the root VE. Algorithm 1 is then used to create a DVM.

A context change may have (1) a negative effect, (2) a positive effect or (3) no impact in deployment. A context change with a negative effect on the deployed system will render some of its components unusable, *i.e.*, the context conditions upon which the component depended on are not satisfied anymore. A context change with a positive effect will render applicable components that previously were not thus allowing their usage to improve the QoS levels of the system and will not disrupt the current deployment. Finally, the contexts change has no impact in deployment if it do not render some of its components unusable nor render applicable components with better QoS then the current components.

After a context change is reported by the monitoring stage, the analysis stage starts by verifying each and every node that depends on such context in order to define whether for that particular node the current deployment is not viable or may be improved therefore triggering its adaptation planning.

Whenever a negative impact context change occurs, it is possible to recover the failed deployment as long as it is possible to find other components or other alternatives to fulfill the deployment goals. As such, even a context change with negative impacts can, by making new components deployable, offer new opportunities and new alternatives to recover the achievability of the root goal. Moreover, a context change can also provide us with opportunities to improve the delivered system's quality of services by making deployable components with higher quality of services than those currently deployed.

Algorithm 1: resolveVE.

```
1  Function resolveVariabilityElement (ve)
2      ve.chosenAlternative ← NULL;
3      if !isApplicable(ve) then
4          ve.isAchievable ← True;
5          ve.chosenAlternative ← NULL;
6          updateContextMap(ve.satisfy.modifier, ve)return ve;
7      end
8      foreach ve.alternatives as alt do
9          updateContextVEMap(alt.requiredContextSet, ve) ;
10         if not checkCtx(alt) then
11             alt.isResolved ← FALSE ;
12         end
13         else
14             var isResolved ← TRUE ;
15             if notalt.listDepGoals.isEmpty() then
16                 alt.listDepVe ← queryRepo(alt.listDepGoals) ;
17                 foreach alt.listDepVe as depVE do
18                     var result ← resolveVariabilityElement(depVE);
19                     if not dependencyIsSatisfiable(result) then
20                         isResolved ← FALSE ;
21                         break ;
22                     end
23                 end
24             end
25             alt.isResolved ← isResolved AND
               hasAtLeastOneAltForEveryAnyDep(alt);
26             ve.chosenAlternative
               =selectBetterAlternativeSet(ve.chosenAlternative, alt);
27         end
28     end
29     ve.isAchievable ← checkAltValidity(ve.chosenAlternative) ;
30     return ve;
```

3.3.5. Plan

The process of creating and updating a DVM under a given context, loosely based on Guimarães planning algorithm [40], is presented in Algorithm 1. This algorithm works as follows: first, it decides on the applicability of the VE itself (line 3). Whenever the contextual conditions for dependency application are inactive, the alternative is deemed achieved and returns the VE without introducing any new alternatives to the deployment plan. It also updates the ContextVEMap with a mapping from every context described in the dependency modifier to the current VE. On the other hand, should the goal be applicable, the algorithm proceeds by iterating over the VE alternative list (line 8). For each alternative, the *contextVE* map of the *KnowledgeBase* is updated with the alternative context and the associated VE (line 9).

Then the context conditions of the associated implementation bundle are checked against the current context via *checkCtx* function (line 10). If *checkCtx* returns `FALSE`, it means that at least one context condition of the alternative does not hold. In that case, the alternative is marked as not resolved (line 11). Otherwise, the *checkCtx* returns `TRUE` then the alternative dependencies are checked (lines 14–27).

Whenever an alternative has no dependencies, no further check is needed and the algorithms immediately marks such alternative as resolved and returns it after the `selectBetterAlternative` function chooses it over a `NULL` plan. Otherwise, if the list of dependencies is not empty, the planning stage begins by checking each dependency as such: the repository is queried for the `alt` dependencies and returns a list of VEs which are then assigned to the *listDepVE* attribute (line 18). For each VE in *listDepVE*, `resolveVariabilityElement` function is recursively called (line 18). The method *dependencyIsSatisfiable* then evaluates the dependency modifier to decide on its satisfiability. For an AND alternative, where all of the dependencies must be deployed, if any VE of its *listDepVE* cannot be resolved then the alternative as a whole is considered unresolved (line 20) and the resolve process halts. In contrast, for an ANY (OR alternative), the method will return true even if such dependency was not resolved so that the algorithm may keep on trying the remaining options. Should any be deemed resolved, the VE is also considered resolved and the alternative set is returned. If none of the alternatives is applicable, then the VE is considered unresolved. Should *isResolved* retain `TRUE` value after resolving all the dependencies and at least one alternative has been chosen for each dependency then the alternative as a whole is considered resolved (line 25). This alternative is finally compared with the (possibly NULL) VE's currently `chosenAlternative` to decide on the alternative expected to deliver the highest QoS level. Such comparison is performed firstly by selecting any resolvable alternative over an unresolved one and, should both be resolved, secondly by applying the model's utility function, expressed through the *KnowledgeBase*'s qualityWeight, to the *qualityProperties* advertised by each alternative. The applicable alternative able to deliver the highest QoS level is then selected and returned by the *selectBetterAlternativeSet* method. Then the process moves on to resolving the next VE alternative and repeats itself.

The last step, after iterating over all alternatives, is setting the VE's `isAchievable` attribute to `TRUE`, when an alternative has been chosen or `FALSE`, otherwise (line 29) and returning the VE itself (line 30).

The output of the *resolveVariabilityElement* algorithm is a VE tree root element that contains the best available alternative in its `chosenAlternative` attribute. Its *listDepVE* attribute will have another VE for each dependency, each one with its own *chosenAlternative* already assigned, recursively decomposing into smaller sub trees, up to alternatives without dependencies.

Algorithm 2 is used to locally update the DVM in an event of a context change. Firstly, the *KnowledgeBase* is updated with the changes

Algorithm 2: Handle Context Change.

```
   Input: ContextChange contextChange
   Result: Bool isAchievable;
1  Function handleChange(contextChange)
2      updateCtx(contextChange) ;
3      var affectedElements ←
       getAffectedVariabilityElements(contextChange) ;
4      for affectedElements as ve do
5          var tempVE = resolveVariabilityElement(ve) ;
6          while not tempVE.isAchievable AND tempVE.parentAlt != NULL do
7              tempVE = tempVE.parentAlt.parentVE ;
8              tempVE ← resolveVariabilityElement(tempVE) ;
9          end
10         if not tempVE.isAchievable then
11             return FALSE ;
12         end
13     end
14     return TRUE
```

in context (line 2) and the currently deployed VEs which will be affected are listed, with the assistance of the Context-VE Mapping from the `KnowledgeBase` (line 3). At this point, GoalD attempts to locally patch the DVM at the affected VEs in order to minimize the re-deployment effort. For each *affectedVE* Algorithm 1 is called (line 6). If the resulting VE's `isAchievable` attribute is `TRUE`, the DVM has been successfully patched for this VE and no further processing is needed. Otherwise, if the VE cannot be achieved in the new context, than the current alternative is inapplicable and the analysis process is restarted at the VE's parent node so that other alternatives may be considered (lines 6–9). This process is repeated until an achievable plan is

found or until the VE has no parent, *i.e.*, until the planning is performed at the rootVE.

If an achievable plan was found, then the condition at line 10 will not be triggered and the algorithm moves on to the next `affectedElement`. If the planning was attempted at the `rootVE`, then all the possible alternatives have been attempted and no deployment plan is possible. In this case, the condition at line 11 will be triggered and the algorithm will halt returning a value of FALSE.

If all affected VEs are successfully patched and the algorithm has not halted yet, then the adaptation has been successful and the algorithm returns TRUE. This indicates that the DVM has been updated and that the deployment plan can be executed.

As previously mentioned, in order to provide a deployment plan, a set of operations must be defined. Based on the DVM structure, the Plan subsystem engineers a list of commands to deploy the specified bundles at runtime by traversing the DVM collecting bundles associated to chosen alternatives, which constitutes the target deployment bundle set; (3) compare the target deployment with the current deployment, creating a plan consisting of (a) commands to install bundles present onto the target deployment and not currently deployed and (b) commands to uninstall current bundles not needed in the target deployment.

After updating the DVM, the next step is to obtain the target deployment, which can be carried out by traversing the DVM and collecting the bundles associated with the chosen alternatives. Algorithm 3 initially

Algorithm 3: Create Deployment Plan.

```
   Input: VE ve
   Result: List bundles
1  Function createDeploymentPlan(ve, currentDeployment)
2      var targetDeployment ← new Set ;
3      targetDeployment.include(ve.definition) ;
4      targetDeployment.include(ve.chosenAlternative.implementation);
5      for ve.chosenAlternative.depVEList as depVE do
6          var depBundles ← collectChosenBundles(depVE) ;
7          targetDeployment.includeAll(depBundles) ;
8      end
9      var deploymentPlan ← new DeploymentPlan;
10     deploymentPlan.toBeIncluded
       ←currentDeployment.diffSet(targetDeployment);
11     deploymentPlan.toBeRemoved
       ←targetDeployment.diffSet(currentDeployment);
12     return deploymentPlan
```

comes up with the target deployment (lines 2–8), which can be described as follows. Firstly, it initiates variable bundles with an empty set (line 2). Then it includes *ve.definition* and the *ve.chosenAlternative.implementation* bundle to the set (line 3–4). Then for each dependency of the chosen alternatives it recursively includes the bundles set to the target deployment(line 5–7).

3.3.6. Execute

Having the target deployment set, the algorithm moves on to comparing it to the current deployed bundles set. The deploymentPlan object has its two attributes set on lines 10 and 11. On line 10, the `targetDeployment` is compared to the `currentDeployment` to identify the bundles that are in the target deployment and are not yet available. These bundles are then included in the deployment plan `toBeIncluded` attribute and will be deployed upon the plan's enactment. Analogously, the `currentDeployment` is compared to the target to identify bundles that are deployed, not needed anymore and therefore should be removed during the plan's enactment (line 11). At last, the plan itself is returned and the algorithm is complete (line 12). The same concepts of deployment planning are used in a slightly different way in order to create the initial deployment and to handle context changes. As the system initially has an empty deployment, the initial deployment plan will consist of *INSTALL commands* for all bundles in the target deployment and an empty `toBeRemoved` command set.

Once a deployment plan is devised, executing the plan is a matter of executing each command in the plan. The change mechanism for the deployment is a platform agnostic concept of GoalD. Under a Java platform, such as the one used for our implementation and evaluations, the OSGi [41] framework is an alternative that may be used to fetch and bind the components.

4. Evaluation

In the evaluation of GoalD, we use the Goal-Question-Metric (GQM) evaluation methodology [42]. To evaluate the achievement of this goal, we define the questions and metrics presented in Table 1.

Our first evaluation goal, G1, is to investigate the feasibility of the approach by evaluating whether the implementation of the proposed framework is capable to autonomously plan the initial deployment and adapt it as needed and as expected. In order to make an actual evaluation, we implemented the TAS in GoalD following the exemplar, as an actual self-adaptive system implementation, provided by Weyns and Calinescu [1].

Our second goal, G2, aims at evaluating GoalD planning capability to adapt to a changing environment. More concretely, we evaluate three perspectives of the adaptation: (1) its accuracy on identifying a context change that impacts its availability, (2) the time required by GoalD to provide a new deployment planning in face of failure, (3) the effect on the deployment QoS when a context with more reliable resources becomes available for deployment.

Our third goal, G3, aims at providing a more comprehensible scalability evaluation of GoalD. To handle heterogeneity, GoalD enables the provisioning of various artifacts in the repository that can achieve the same goal and fitting different context conditions. We name the number of artifacts present in the repository that provide the same goal as variability level. A high level of variability is beneficial to cater for diverse context but, at the same time, can affect the scalability of the planning because of the exponential nature of the decision tree in goal model, which can be computing intensive.

We implemented GoalD in Java version 1.8 structured as a Maven project[1]. The code for the execution of the evaluation, the data obtained and scripts used to analyze it are available on a public repository[2]. The experiments were conducted using Apple MacBook Pro-(Mid 2015) with a 2.2 GHz Intel Core i7, 16GB DDR3 1600MHz memory, with macOS (10.13.5). JDK(10.0.1 64bits) was used to build and run the project.

4.0.7. Goal 1: Evaluate GoalD capability to deploy and execute a system

In order to answer *Q1.1*, an implementation of the Tele Assistance Service was developed by using the GoalD methodology. This was achieved by making use of the OSGi Technology [41] as a runtime environment for the packaging of the components in bundles and for the execution of the TAS workflow. The goal model specified in Fig. 1 was mapped into 30 diferent bundles (15 of type definition, 15 of type implementation).

For a fair comparison between the two implementations, we also implemented two execution strategies using GoalD: the *Retry* strategy, in which two attempts are made in order to obtain a service upon failure, and the *Reliable* strategy, which selects an equivalent service with the lowest failure rate, or lowest cost if a tie, when facing a service failure. Then we compared our results to theirs, which is presented in Table 2, by running the experiment in ten series of a hundred executions. Although the standard deviation is not reported in the original work of TAS [1], we consider a usual 10% standard deviation for the values reported by

[1] GoalD's source code https://github.com/lesunb/goald/

[2] https://github.com/lesunb/goald-evaluation/

Table 1
GQM devised plan.

Goal 1: Evaluate GoalD capability to deploy and execute a system.	
Question	Metric
1.1 How similar is the system behavior when deployed using GoalD in comparison to a reference implementation of TAS?	Failure rate
Goal 2: Evaluate GoalD capability to adapt to a changing environment.	
Question	Metric
2.1 Does GoalD correctly identify context changes that will impact the system's availability?	Precision of adaptation decision.
2.2 What is the impact of a context change that renders the current deployment infeasible?	Mean time to repair
2.3 What is the impact of a context change that allows the deployment to deliver higher QoS levels?	Mean time to failure
Goal 3: Evaluate GoalD scalability	
Question	Metric
3.1 How does the planning scale over an increasing number of bundles?	Time elapsed to come up with an initial deployment plan.
3.2 In the presence of context change, how does the planning scale over an increasing number of bundles?	Time elapsed to come up with a deployment adaptation plan.

Table 2
Failure rate comparison between TAS implementation in GoalD and Weyns & Calinescu.

Approach	No adaptation	Retry	Select reliable
GoalD	$(21.5 \pm 1.2)\%$	$(0.7 \pm 0.2)\%$	$(0.6 \pm 0.2)\%$
Weyns & Calinescu	18%	0.5%	0.09%

Weyns & Calinescu. As such, we can notice that the values for strategies *No Adaptation* and *Retry* fall very closely to the same range.

Nevertheless, the values obtained for the *Selected Replication* are quite different from each other. We validated our results with the two TAS authors, who confirmed there might have been some typo in their original. They reckon our results are legitimate. We also note that the adaptation policy in GoalD does not consider the possibility of choosing the less costly option (in case of a reliability tie), but only the selection of most reliable in case of a failure. In order clarify whether GoalD is actually performing the adaptation strategy as expected, we further evaluate such feature in the forthcoming goal G2 of our GQM.

The implementation code, along with the log of the executions of this specific goal can be found at https://github.com/jcosta9/OSGi/

4.0.8. Goal 2: evaluate GoalD capability to adapt to a changing environment

To verify that GoalD achieves its second goal - i.e., able to adapt a deployment to respond to changes in the system's environment - we simulated environmental context changes. Every context change is reflected in an entry in the experiments' log file which may be represented as a timeline. An excerpt of the timeline is illustrated in

Fig. 9 illustrates the contexts (lines 1–5), the components (lines 6–19), and the resulting system availability (line 20) as horizontal bars over time (t). Throughout the timeline, the system availability under the ever-changing environment is an outcome of the described context scenarios (as depicted in Fig. 9). As previously mentioned in Section 3.3, the unavailability in the GoalD approach will take place when context change incurs on the required resources unavailable (e.g., due to an outage). The dotted vertical lines represent those moments of sudden unavailability.

The excerpt in Fig. 9 depicts the first 20 hours of execution. Initially, contexts C1, C2 and C5 were active. In this environment, GoalD deployed components *ProvideSelfDiagnosedEmergenciesSupport, PushButton* and *AlarmService*. The option to choose the *Provide Self Diagnosed Emergencies Support* strategy (G1 in Fig. 1) over G2 as well as the *Alarm Service* (P3) over *Send SMS* (P2) was the expected choice as G2 was not applicable (battery was low) and P3 delivers a better quality level. This initial configuration was deployed and the system became available in 11ms. This deployment supported the system up until the moment t=2h when context `battery is low` was inactivated. At this point, GoalD adapts the system deploying goal G2 alongside G1.

At nearly t=3h30m, the internet connection is lost (!C1), therefore rendering the deployed components `RemoteAnalysis` and `AlarmService` unusable as both depend on context C1 to provide their functionality. This disruptive change is identified by GoalD and, in response, components `LocalAnalyisis` and `SendSms` are deployed, successfully restoring the system's availability. Such configuration is kept until 4h, when the connection with the Internet was restored. The availability of an internet connection does not affect any of the deployed components; however the QoS levels provided by `AlarmService` and `RemoteAnalysis` are greater than those offered by their counterparts, so GoalD proactively deploys those to increase the system's overall QoS levels.

At t=6h, an issue with the patient is detected (context C5 is deactivated) and again GoalD responds to this change by including components `EnactTreatment`, `AdministerMedicine`. In time t=7h a doctor arrives (context C3) and allows for drugs to be prescribed, moment in which GoalD deploys component `ChangeDrug`. Finally, in t=9h, the prescribed drug is made available (context C4) and changes in its dose may occur therefore deploying the corresponding component `ChangeDose`.

Analogous analysis follows for time at nearly 13h, 15h, 16h, and 17h, where a dotted vertical red line shows a context change followed by a sudden unavailabilty in the order to 34 to 66ms. Therefore, the unavailable moments are in the order of 270 milliseconds, and the overall availability of our implementation of TAS in OSGi following GoalD is ≈ five9′s.

In the following paragraphs we will address the GQM plan's G2 questions.

`Q2.1: Does GoalD correctly identify context changes that will impact the system's availability?`

GoalD supports context changes that lead to deployed components' unavailability or that lead to the availability of previously unusable components capable of providing higher QoS levels. For example, in the test case depicted in Fig. 9, the context changed a few times. Five of the changes (instants t=3h30m, t=13h, t=15h, t=16h, t=17h) incurred in system unavailability and the deployment had to be repaired. For other context changes, the deployment was not negatively affected but was changed to provide a better QoS (e.g., instants t=2h and t=7h).

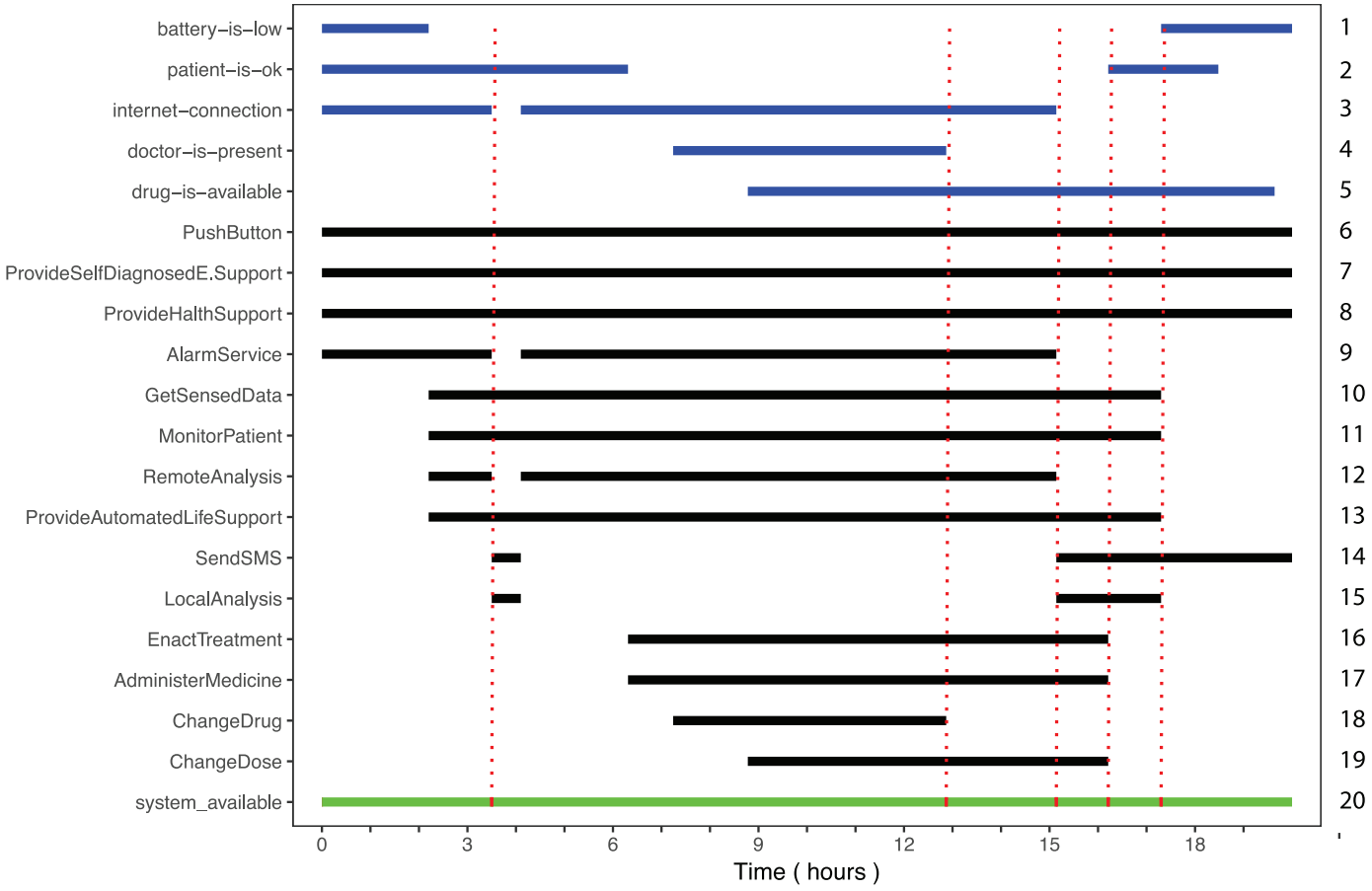


Fig. 9. Component Activation in GoalD.

Table 3
Analysis of the accuracy of context changes detected in GoalD Manager.

Precision	Recall	F-measure
1.0	1.0	1.0

To evaluate the correctness of the GoalD analysis and planning, we created a list of assertions that evaluate whether the proposed changes are valid under varying contexts. The assertions followed the template (*premise -> conclusion*) where the premise was composed of a set of propositions related to contexts while the conclusion was composed of a set of propositions related to a target deployment. An example of such propositions is: (*!internet-connection*) -> (*!AlarmService-impl* and *!RemoteAnalysis-impl*), which means that in a context where the system has no internet connection the AlarmService-impl and RemoteAnalysis-impl should not be deployed. From the TAS goal model 1 we elaborated 10 propositions that describe the expected deployment configuration. Then we evaluated all possible context changes that could occur in the TAS case study: a total of 160 possible context changes. (2^5 possible initial contexts with 5 possible changes for each one).

Evaluating each proposition with each possible adaptation we found 96 true positives, 64 true negatives, 0 false positives and 0 false negatives. The results are summarised in Table 3.

The results show that the precision of GoalD in correctly identifying the need for context changes (precision) as well as identifying the relevant ones (recall) are both quite high as the computed f-measure is 1.0

`Q2.2: What is the impact of a context change that renders the current deployment infeasible?`

We analyse Q2.2 by measuring the *Mean time to repair* in the TAS in those scenarios where the system became unavailable, followed by a deployment repair provided by the GoalD manager.

A context change renders the current deployment infeasible if it renders at least one deployed component unavailable (e.g a lost in the Internet connection renders *Remote Analysis* unavailable). The system is unavailable from the moment of the environment change until the deployment is repaired. In Fig. 9 we have two examples of that: around 2h, when the context c1 is deactivated and around 11s when the context battery-is-low is deactivated. In that case, the needed time to repair the deployment encompass: (i) the time needed to identify a context change in the monitored environment, (ii) the time to analyze the change impact (updating the DVM), (iii) devise a deployment adaptation plan and (iv) conclude the execution of the adaptation.

In order to measure the *Mean time to repair* we used the TAS case study and evaluated in all possible context changes that could render the deployment infeasible. In our experiments the obtained *Mean time to repair* was (21.6 ± 1.1) ms. This means that the time for TAS to devise a deployment strategy in GoalD while facing component unavailability due to context change would be quite negligible. This analysis indicates that GoalD might perform reasonably well for systems that can afford a minimum downtime of 30ms.

`Q2.3: What is the impact of a context change that allows the deployment to deliver higher QoS levels?`

To answer this question we analyzed context changes that present opportunity to provide a deployment with higher QoS levels. In other

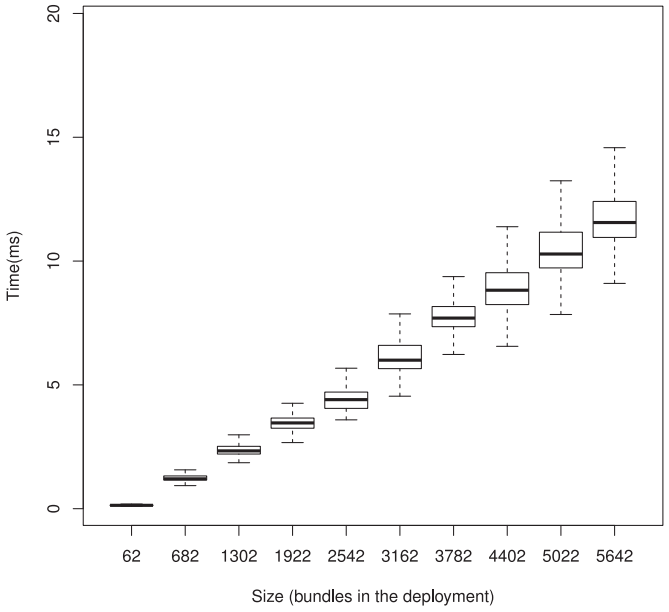


Fig. 10. Scalability over increasing number of bundles in the initial deployment.

words, when the context changes so that two conditions are satisfied: (i) at least one of its components provides a QoS higher than a currently deployed one and (ii) such context triggers an adaptation to improve the deployed availability.

We should note that, in the observed scenarios we had no impact on the system availability while adapting to increase in quality as all the adaptation process is executed while the systems is still in a valid configuration. In summary, we observed that a GoalD managed system will not render unavailable while GoalD manager deploys the (new) components with higher QoS.

4.0.9. Goal 3: evaluate GoalD scalability

To answer the question 3 we implemented one last test which randomly generate large deployment scenarios. Since TAS has a relatively small-medium size, we further cloned it and artificially proliferated the components so that we can evaluate scalability to come up with a deployment plan on a large-scale model.

A repository as big as 430,500 bundles was randomly generated. The bundles generated had different dependency levels. To evaluate the impact of the various hierarchical levels on the deployment planning time, we have performed 1000 batches of 10 deployment requests, where each request contains from 1 to 10 goals. The generated plans encompassed up to 5642 bundles.

In order to answer question 3.1 as for how the algorithm scales over the number of bundles in the deployment plan, is depicted in Fig. 10 and the *observed time vs plan size* is shown as boxplots. Despite the presence of outliers (< 10%), the experiment shows that less than 15 milliseconds were necessary to come up with an initial deployment plan of more than 5000 bundles. Additionally, a polynomial time trend can be inferred from the experiment. While adapting to changes, the experimental results shows that all adaptations plans but one were devised in less than 2 milliseconds.

Finally, to answer question 3.2, adaptations were triggered by inserting random changes in contexts that affect the deployment. Each context change resulted in an adaptation plan comprising from 0 to 276 commands. A command could be INSTALL a new bundle or UNINSTALL a current deployed bundle. Since there is a one-to-one relationship between commands and bundles that were either installed or uninstalled, the number of commands executed also represents how many bundles were affected by the adaptation triggered by the context change. The scalability of the adaptation is depicted in Fig. 11a and b. Fig. 11a shows the time elapsed to come up with an adaptation plan (analysis and planning) as a function of the number of deployed bundles prior the context change. Fig. 11b is a bloxplot of the same dataset plotted as a function of the number of commands present (and therefore the number of affected bundles) in the deployment plan. Results show that GoalD planning in the presence of context change has performed under 6ms in the worst case scenarios, disregarding the outliers, which are analysed in the sequel.

We hypothesised that the outliers depicted in Figs. 10 and 11 were caused by spurious factors that could be related to the computing platform where the experiments are conducted (e.g non-deterministic execution of the Java Garbage Collector, process preemption, frequency of cache misses). We validated that hypothesis by re-executing the outliers 10 times each. Results in Table 4 show that the outliers when re-executed fall indeed within the range of the boxplot quartiles of the original executions.

4.1. Threats to validity

Construct validity – The major threats here are the correctness of the implementation of the TAS and of the proposed approach. To overcome the threat of TAS implementation, it has been thoroughly tested following the specification and implementation of TAS exemplar. At least two authors of this paper reviewed the implementation in OSGi and checked the plausibility of the evaluation results based on the

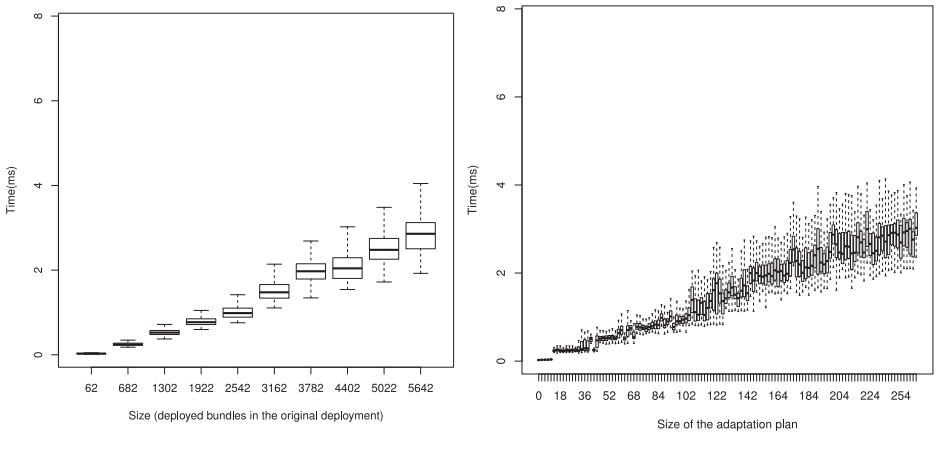


Fig. 11. Adaptation scalability over increasing (a) deployed bundles and (b) number of commands in the adaptation plan.

Table 4
Comparision between original execution and re-execution of the outliers in Figs. 10 (top) and 11 (bottom).

Deployment size	Initial deployment (ms)	Outliers re-execution (ms)
62	(0.0695 ± 0.0215)	(0.0671 ± 0.0126)
682	(1.01 ± 0.189)	(0.963 ± 0.155)
1302	(2.18 ± 0.439)	(2.17 ± 0.426)
1922	(3.31 ± 0.576)	(3.51 ± 0.893)
2542	(4.66 ± 0.744)	(4.76 ± 0.613)
3162	(6.20 ± 0.799)	(6.31 ± 0.703)
3782	(7.88 ± 0.871)	(8.55 ± 1.74)
4402	(8.70 ± 0.808)	(8.83 ± 0.932)
5022	(11.6 ± 2.35)	(11.0 ± 1.52)
5642	(12.2 ± 2.26)	(11.5 ± 0.891)
Deployment Size	Adaptation (ms)	Outliers re-execution (ms)
62	(0.0301 ± 0.0192)	(0.0189 ± 0.0126)
682	(0.257 ± 0.0734)	(0.234 ± 0.155)
1302	(0.540 ± 0.105)	(0.497 ± 0.426)
1922	(0.798 ± 0.133)	(0.813 ± 0.893)
2542	(1.05 ± 0.285)	(1.11 ± 0.613)
3162	(1.52 ± 0.257)	(1.57 ± 0.703)
3782	(2.00 ± 0.288)	(2.01 ± 1.74)
4402	(2.10 ± 0.402)	(2.20 ± 0.932)
5022	(2.54 ± 0.465)	(2.90 ± 1.52)
5642	(2.87 ± 0.504)	(2.74 ± 0.891)

experience they acquired through the TAS specification provided. Results in Table 2 has shown that results for TAS in GoalD are in the same range of the results of original TAS implementation, except for the *Select Reliable* option, which were validated by at least two authors of the original TAS exemplar. Those authors acknowledged the correctness of our results and that there could be a typo in the TAS exemplar paper. In addition, our CGM for TAS could also represent a threat to the construct validity. Despite the fact that we creatively devised scenarios by means of context variations for TAS, the deployment adaptation by GoalD activated the same components that would be expected to be activated in the orginal TAS scenarios, where applicable.

Internal validity – Our approach showed itself quite trustworthy for evaluation purposes of the correct adaptations, as presented in Table 3. GoalD identified the correct context changes followed by the provision of the correct deployment planning, whenever needed. Nevertheless, unveiling all combinations of deployment conditions involved in a system's operation is inherently non-deterministic, which could represent a threat to any adaptation process. Moreover, in this experiment, we dealt with a considerable small number of context conditions. Also, the components configuration and artifacts deployment were manually implemented. All these issues might represent a threat in complex scenarios with a great number of contexts. For future work, we plan to revise our evaluation framework and adopt the most suitable evaluation approaches described in [43] to overcome such a threat. In particular, the SOLAR tool could be a potential alternative as it implements the metrics proposed by [44], which could be used to evaluate the adaptability of the systems implemented using the GoalD approach.

External validity – Although our approach is platform independent, and the required time for deployment planning and adaption has shown itself negligible (*cf.* Fig. 10), we do reckon the limitation of the evaluation as TAS is a prototype only of a health care system in the Ambient Assisted Living domain, while other domains could be considered instead (e.g., robotics, autonomous driving, etc). Further evaluation must be performed to generalize the correctness and scalability of the results. Despite all efforts to implement TAS with new features to enrich the deployment adaptation experimentation in GoalD, further study must be done to verify the applicability in real-world scenarios.

Reliability – The experiments we conducted include certain randomness in the contexts sets of the TAS application setup, which reflects the inherent uncertainties of self-adaptive systems. For this reason, there is a threat that the results may not be the same if the study would be conducted again. To overcome this threat, we ran the experiments over a period of time that represent a large number of adaptation loops. In addition, the ability to reproduce our evaluation results has been facilitated by means of publicly available GitHub projects containing the source code of GoalD framework as well as the conducted and reported experiments. Also, to avoid cumbersome configuration effort, we have used and provided Maven configuration files so that GoalD framework and its experiments can be automatically built and replicated by any interested reader or contributor.

5. Related work

In this section we discuss most related work whether they are goal oriented, handle heterogeneity, and support autonomous deployment. We should note that by goal-oriented we mean those work where the goal model is key to the deployment management strategy. Thus, we compare to those work able to (i) promote the seamless integration between system requirements, deployment strategies and its configurations as well as (ii) make the goal model as key architectural element of offline and online stages to make feasible the autonomous deployment process.

Angelopoulos et al. [30] present an approach to handle variability at three different dimensions: goals, behavior, and architecture. Variability can occur at the goal level as an OR-refinement or context selection; at the behavior level as different plans flows; and at the architecture level addressing the variability of components and their implementations. GoalD can augment this approach by handling variability as a deployment problem and explicitly captures and caters for the different settings of the hosting environment. Pradhan et al. [45] propose a goal-based solution to achieve resilience in mobile cyber-physical systems via self-configuration and autonomous deployment. Although such approach does benefit from a goal-based solution as well, there is no seamless integration between requirements, strategies and configurations/context conditions. Therefore, considering what we mean by goal-oriented approach, Pradhan et al. do not fall into such category. Instead, a colored petri net model is used to capture the behavior and to analyse the system properties at design time. At runtime, queries to a database need to take place in order to statically store deployment actions.

Ali et al. [46] explore the optimization of the deployment for a given context variability space. Contextual Goal Models (CGM) are used to represent aspects of the environment elicited because of their relation to the solutions presented in the goal model. GoalD puts a primary focus on the context related to the computing environment and enriches the notion of resources and the mapping between goal achievement alternatives and software artifacts.

The Dynamic Software Product Line (DSPL) paradigm is motivated by a rapid production of software from a set of reusable assets to fit variability in users requirements and system environments. Bencomo et al. [47] use an SPL approach to adaptation by associating an architecture variability model with an environment variability model.Mizouni et al. [48] use a feature model associated with context requirements. Casquina et al. propose the Cosmapek [15], an adaptive deployment infrastructure that relying on a reflective architecture and implementing MAPE-K main stages as a means to achieve dynamic deployment of DSPL. However, they perform the MAPE-K activities based on static models using runtime binding as pointed out by Eleuterio and Rubira [49]. They represent the variability in a feature model, representing static and dynamic features in a single model with its own notation, that is translated to XML for runtime use. Additionally, they use UML component diagram to statically build the architectural model. Overall, the use of DSPL and its associated approaches is mainly focused on runtime adaptation where software systems switch amongst already implemented and deployed artifacts/features configurations. Although the approaches do contemplate configuration variability as well as behavior control, they do not follow a coordinated and seamless integration as GoalD does by having system state, system goals and

Table 5
Comparison between GoalD and most related approaches.

Work by	Goal oriented	Handle heterogeneity	Autonomous deployment
Ali et al.[46]	Yes	No	No
Angelopoulos et al. [30]	yes	No	No
Casquina et al. [15]	No	yes	No
Bencomo et al. [47]	No	yes	No
Mizouni et al. [48]	No	yes	yes
Pradhan et al. [45]	No	yes	yes
Gunalp et al.[54]	No	yes	yes
GoalD	**Yes**	**Yes**	**Yes**

environment assumptions/constraints integrated into the CGM where offline and online stages fully rely on. In addition, GoalD is able to handle the step preceding that, i.e., the deployment stage and its decision making based on the contextual conditions, as supported by Braberman et al. [16,18].

Leite et al. [14] propose an approach for autonomous deployment on inter-cloud environments, based on DSPL as well. It relies on abstract and concrete features models and constraint satisfaction problem solver to create a computing environment using resources distributed across various clouds. The approach heavily depends on design-time created deployment scripts and requires prior creation of model knowledgeable about the environment. GoalD caters for the uncertainty about the environment and enables a more open approach to its heterogeneity.

Autonomous deployment solutions have been already proposed by means of business process orientation as pointed out the survey of Chang et al. [11]. Approaches such as [10,12] focus mostly on the (deployment of) the execution workflow that the already deployed components must follow or are following based on a BPMN events representation. Therefore, business process based solutions will be able to either orchestrate or choreograph once the components are already deployed, since it will only be able to manage the executable methods, services and events to the corresponding workflow engines for execution [12] once the required and suitable component infrastructure is deployed. In addition, such approaches may render too heavyweight for being deployed on resource-constrained devices [7]. However, we envision that future versions of GoalD could potentially benefit from an integration with such business-oriented solutions where GoalD could follow reconfiguration strategies based on the services that require target constraints.

Rainbow is a framework for architecture based self-adaptation [50]. It keeps a model of the architecture of the system and can be extended with rules to analyse the system behavior at runtime, find adaptation strategies and perform changes. It separates the functional code (internal mechanisms) from adaptation code (external mechanism) in a schema called external control, influenced by control theory [51]. As such, Rainbow focuses on changing component instances and bindings and also tune in behavior through operational parameters. Nevertheless, the framework does not explicitly account for building strategies that control the functional behavior of the system components.

MUSIC project provides a component-based middleware for adaptation and proposes to separate the self-adaptation from business logic and delegates adaptation logic to generic middleware. Aligned with GoalD, it adapts by evaluating at runtime the utility of alternatives, to choose a feasible one (e.g., the one evaluated as with highest utility) [52].

Differently from Rainbow, MUSIC and Rondo, GoalD tackles the problem of autonomous deployment by leveraging the use of goals through the CGM as a means to devise adaptation strategies, while preserving traceability between requirements, behavior and configuration. Consequently, GoalD enables a seamless integration with the perspective of the current foundations of self-adaptative systems [20,53]. The deployment configuration strategies of GoalD arise from the CGM and it autonomously adapts according to the available resources at runtime following our algorithms for handling context change and the variability structure of the CGM. As a result, GoalD is a resourceful framework at runtime, which frees the system designer from manually specifying adaptation policies at design-time, which could be limited for anticipated scenarios and could be cumbersome to configure for a great number of computational resources.

GoalD has been greatly inspired by MORPH, a reference architecture for configuration and behavior of self-adaptation [18], particularly on their reconfiguration strategy manager. MORPH promotes the alignment between configuration and behavior for self-adaptation by means of a GORE approach. The goal model is a key data architectural element in the MORPH repository in order to combine assumptions on the environment and requirements to achieve the software objectives. GoalD does share the same purpose of the MORPH's reconfiguration strategy manager where it not only enacts current strategies, but also "capitalises on opportunities afforded by a change in the environment." Likewise, "should a new component become available, or statistics on its performance improve, this would be reflected in the knowledge repository and a preferred alternative strategy may be deployed" in GoalD. While MORPH is a reference architecture, GoalD paves the way to a concrete means to achieve the principles of MORPH focusing in detail on autonomous deployment for highly heterogeneous and dynamic environments.

Gunalp et al. [54] present Rondo, which is an approach for continuous deployment for dynamic and service-orientd applications. In RONDO, a specialist has to specify deployment a priori in terms of resources and their desired target states. They use an operational model to drive the adaptation: implemented strategies to move monitored resources to the target states. In addition, the evaluation that is closest to ours is Gunalp et al. [54] in Rondo. Their results for an experiment run on 2 different service platforms (Wisdom and iCasa) shows that their deployment method runs within the range of 90-110ms and variability adaptation around 236ms. A comparison to GoalD shows that GoalD has a major advantage given that Gunalp et al. deployed at most 107 bundles, while ours shows an even greater number (from 62 to 5642 bundles) but runs in under 15ms in all situations depicted in our Figs. 10 and 11. Nevertheless, their setup configurations and the experiments conducted were quite different from ours and this would render a misleading impression that GoalD is superior, unless another case study was conducted with Wisdom and iCasa in GoalD approach. For this reason, as we point out in the external threat to validity section, we do recognise the limitations of our experiments and that further evaluation must be performed to generalize the correctness and scalability of the results.

Table 5 summarizes the comparison between GoalD and most related, based on (i) goal-oriented management strategy, (ii) ability to handle heterogeneous computational resources, and (iii) autonomous deployment.

6. Conclusion and future work

In this paper, we presented GoalD, an approach to systematically tackle autonomous deployment in highly heterogeneous and dynamic computing environments. GoalD supports the design of a system with a variability space, providing a foundation to handle heterogeneity. GoalD bridges between the strategic level of requirements modelled as goals,

and the architecture understood as inter-related components and interfaces. Moreover, GoalD supports variability at deployment time by finding the correct set of artifacts that allows the system to achieve the system and stakeholder goals in each hosting computing environment. Also, GoalD supports dynamicity of resources by adapting the deployment in response to changes in the computing environment.

Our evaluation of the approach shows that GoalD is capable of determining a plan for a reasonably large scenario in just a few milliseconds. The results obtained are promising, and we plan to enrich GoalD in the near future by extending the framework to distributed systems so that one instance of GoalD may choose between deploying locally or delegating to another known instance in the network. As for the next steps, we plan to extend the decision making process of GoalD for adaptation policies including more fine-grained quality criteria, like dynamically calculating the goals' QoS levels based on their selected decompositions or contextually-dependent quality levels for tasks. The usage of machine learning is also under consideration for this purpose. Furthermore, we plan to conduct an extensive evaluation of GoalD integrating into devices with heterogeneous hardware using different kinds of SBCs (Single Board Computers), like the Raspberry Pi boards, with different hardware extensions attached. We also plan to evaluate systems implemented with GoalD using self-adaptive evaluation methods, such as the ones mapped in [43].

GoalD extends the literature of self-adaptive systems and service-oriented computing by tackling variability and uncertainty of the hosting computing environment using autonomous deployment. We view deployment as a continuous and adaptive process to cater for resources' dynamic availability and failure in a way that takes the achievement of requirements, seen as goals, as a guiding factor. When new requirements are added or new resources are available, the deployment needs to be adjusted. By doing this, our work provides an additional layer and criteria to the deployment decision making process and contributes to making it more holistic.

Acknowledgements

Gabriel Rodrigues was partially funded by CAPES/CNPq grant number 1438159 (2014–2015) while affiliated with University of Brasilia. Genaína thanks CNPq for partial support under grant number 306017/2018-0.

Conflict of Interest

The authors declare no conflict of interest.

References

[1] D. Weyns, R. Calinescu, Tele assistance: a self-adaptive service-based system exemplar, in: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE Press, 2015, pp. 88–92.
[2] G. Bell, P. Dourish, Yesterday'S tomorrows: notes on ubiquitous computing's dominant vision, Personal Ubiquitous Comput. 11 (2) (2007) 133–143, doi:10.1007/s00779-006-0071-x.
[3] L. Atzori, A. Iera, G. Morabito, The internet of things: a survey, Comput. Netw. 54 (15) (2010) 2787–2805, doi:10.1016/j.comnet.2010.05.010.
[4] T. Kleinberger, M. Becker, E. Ras, A. Holzinger, P. Müller, Ambient intelligence in assisted living: enable elderly people to handle future interfaces, in: Proc. of the 4th International Conference on Universal Access in Human-computer Interaction: Ambient Interaction, Springer, 2007, pp. 103–112. http://dl.acm.org/citation.cfm?id=1763296.1763308.
[5] S.D. Smaldone, Improving the Performance, Availability, and Security of Data Access for Opportunistic Mobile Computing, Rutgers University, New Brunswick, NJ, USA, 2011 Ph.D. thesis. AAI3474990.
[6] A. Carzaniga, A. Fuggetta, R.S. Hall, D. Heimbigner, A.v.d. Hoek, A.L. Wolf, A characterization framework for software deployment technologies, Technical Report, 1998.
[7] D. Miorandi, S. Sicari, F. De Pellegrini, I. Chlamtac, Internet of things: vision, applications and research challenges, Ad Hoc Netw. 10 (7) (2012) 1497–1516, doi:10.1016/j.adhoc.2012.02.016.
[8] Jesper Andersson, A deployment system for pervasive computing, in: International Conference on Software Maintenance, 2000, pp. 262–270, doi:10.1109/ICSM.2000.883058.
[9] D. Spinellis, Don't install software by hand, IEEE Softw. 29 (4) (2012) 86–87, doi:10.1109/MS.2012.85.
[10] K. Dar, A. Taherkordi, H. Baraki, F. Eliassen, K. Geihs, A resource oriented integration architecture for the internet of things, Pervasive Mob. Comput. 20 (C) (2015) 145–159, doi:10.1016/j.pmcj.2014.11.005.
[11] C. Chang, S.N. Srirama, R. Buyya, Mobile cloud business process management system for the internet of things: a survey, ACM Comput. Surv. 49 (4) (2016) 70:1–70:42, doi:10.1145/3012000.
[12] A. Yousfi, C. Bauer, R. Saidi, A.K. Dey, Ubpmn: a bpmn extension for modeling ubiquitous business processes, Inf. Softw. Technol. 74 (2016) 55–68, doi:10.1016/j.infsof.2016.02.002.
[13] Models@run.time, N. Bencomo, R. France, B.H.C. Cheng, U. Aßmann, D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, A. Kobsa, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum (Eds.), LNCS, 8378, Springer International Publishing, Cham, 2014.
[14] A.F. Leite, V. Alves, G.N. Rodrigues, C. Tadonki, C. Eisenbeis, A.C.M.A.d. Melo, Automating resource selection and configuration in inter-clouds through a software product line method, in: 8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015, pp. 726–733.
[15] J.C. Casquina, J.D.A.S. Eleuterio, C.M.F. Rubira, Adaptive deployment infrastructure for android applications, in: 2016 12th European Dependable Computing Conference (EDCC), 2016, pp. 218–228, doi:10.1109/EDCC.2016.25.
[16] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, S. Uchitel, Morph: a reference architecture for configuration and behaviour self-adaptation, Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE 2015, ACM, New York, NY, USA, 2015, pp. 9–16, doi:10.1145/2804337.2804339.
[17] D.F. Mendonça, G.N. Rodrigues, R. Ali, V. Alves, L. Baresi, GODA: a goal-oriented requirements engineering framework for runtime dependability analysis, Inf. Softw. Technol. 80 (2016) 245–264, doi:10.1016/j.infsof.2016.09.005.
[18] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, S. Uchitel, An extended description of MORPH: a reference architecture for configuration and behaviour self-adaptation, in: R. de Lemos, D. Garlan, C. Ghezzi, H. Giese (Eds.), Software Engineering for Self-Adaptive Systems III. Assurances, Springer International Publishing, Cham, 2017, pp. 377–408.
[19] F.P. Guimarães, G.N. Rodrigues, R. Ali, D.M. Batista, Planning runtime software adaptation through pragmatic goal model, Data Knowl. Eng. 109 (2017) 25–40. Special issue on conceptual modeling – 34th International Conference on Conceptual Modeling. doi: 10.1016/j.datak.2017.03.003.
[20] D. Weyns, Software engineering of self-adaptive systems: an organised tour and future challenges, in: K.K.C. Kang, S. Cha (Eds.), Handbook of Software Engineering, Springer, 2017.
[21] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, J. Mylopoulos, Tropos: an agent-oriented software development methodology, Auton. Agent. Multi. Agent Syst. 8 (3) (2004) 203–236, doi:10.1023/B:AGNT.0000018806.20944.ef.
[22] A. Dardenne, A. van Lamsweerde, S. Fickas, Goal-directed requirements acquisition, Sci. Comput. Program. 20 (1–2) (1993) 3–50, doi:10.1016/0167-6423(93)90021-G.
[23] E.S.-K. Yu, Modelling Strategic Relationships for Process Reengineering, University of Toronto, Toronto, Ont., Canada, Canada, 1996 Ph.D. thesis. UMI Order No. GAXNN-02887 (Canadian dissertation).
[24] R. Ali, F. Dalpiaz, P. Giorgini, A goal-based framework for contextual requirements modeling and analysis, RE J. 15 (4) (2010) 439–458, doi:10.1007/s00766-010-0110-z.
[25] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: Future of Software Engineering, 2007. FOSE'07, IEEE, 2007, pp. 259–268. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4221625.
[26] M. Morandini, L. Penserini, A. Perini, Towards goal-oriented development of self-adaptive systems, Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS '08, ACM, New York, NY, USA, 2008, pp. 9–16, doi:10.1145/1370018.1370021.
[27] L. Penserini, A. Perini, A. Susi, M. Morandini, J. Mylopoulos, A design framework for generating BDI-agents from goal models, Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '07, ACM, New York, NY, USA, 2007, pp. 149:1–149:3, doi:10.1145/1329125.1329307.
[28] J. Pimentel, M. Lucena, J. Castro, C. Silva, E. Santos, F. Alencar, Deriving software architectural models from requirements models for adaptive systems: the STREAM-a approach, RE J. 17 (4) (2012) 259–281, doi:10.1007/s00766-011-0126-z.
[29] A. Van Lamsweerde, From system goals to software architecture, in: Formal Methods for Software Architectures, Springer, 2003, pp. 25–43.
[30] K. Angelopoulos, V.E.S. Souza, J. Mylopoulos, Capturing variability in adaptation spaces: athree-peaks approach, ER, LNCS, 9381, Springer, 2015, pp. 384–398.
[31] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, J.C. Leite, From goals to high–variability software design, in: Foundations of Intelligent Systems, Springer, 2008, pp. 1–16.
[32] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, P. Spoletini, Validation of web service compositions, IET Softw. 1 (6) (2007) 219–232.
[33] M. Morandini, F. Migeon, M.P. Gleizes, C. Maurel, L. Penserini, A. Perini, A goal-oriented approach for modelling self-organising MAS, ESAW, LNCS, 5881, Springer, 2009, pp. 33–48.
[34] J. Andersson, L. Baresi, N. Bencomo, R.d. Lemos, A. Gorla, P. Inverardi, T. Vogel, Software engineering processes for self-adaptive systems, in: R.d. Lemos, H. Giese, H.A. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II, LNCS, Springer Berlin Heidelberg, 2013, pp. 51–75.
[35] I. Crnkovic, M. Larsson, Component-based software engineering-new paradigm of software development, Invited talk and report, MIPRO (2001) 523–524.

[36] J. Magee, J. Kramer, Dynamic structure in software architectures, in: ACM SIGSOFT Software Engineering Notes, 21, ACM, 1996, pp. 3–14. http://dl.acm.org/citation.cfm?id=239104.
[37] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, The koala component model for consumer electronics software, Computer 33 (3) (2000) 78–85, doi:10.1109/2.825699.
[38] G. Blair, N. Bencomo, R. France, Models@ run.time, Computer 42 (10) (2009) 22–27, doi:10.1109/MC.2009.326.
[39] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess, J.O. Kephart, An architectural approach to autonomic computing, Auton. Comput. Int. Conf. on 0 (2004) 2–9, doi:10.1109/ICAC.2004.8.
[40] F. Pontes Guimaraes, G. Nunes Rodrigues, D. Macedo Batista, R. Ali, Pragmatic Requirements for Adaptive Systems: A Goal-Driven Modeling and Analysis Approach, Springer International Publishing, Cham, 2015, pp. 50–64.
[41] The OSGi Alliance, OSGi Service Platform Core Specification, Release 4.1, 2007. http://www.osgi.org/Specifications.
[42] V.R. Basili, G. Caldiera, H.D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering, Wiley, 1994.
[43] C. Raibulet, F.A. Fontana, Evaluation of self-adaptive systems: a women perspective, in: ECSA (Companion), ACM, 2017, pp. 23–30.
[44] D. Perez-Palacin, R. Mirandola, J. Merseguer, On the relationships between QoS and software adaptability at the architectural level, J. Syst. Softw. 87 (2014) 1–17, doi:10.1016/j.jss.2013.07.053.
[45] S. Pradhan, A. Dubey, T. Levendovszky, P.S. Kumar, W.A. Emfinger, D. Balasubramanian, W. Otte, G. Karsai, Achieving resilience in distributed software systems via self-reconfiguration, J. Syst. Softw. 122 (C) (2016) 344–363, doi:10.1016/j.jss.2016.05.038.
[46] R. Ali, F. Dalpiaz, P. Giorgini, Requirements-driven deployment, in: Software and Systems Modeling, 13, 2014, pp. 433–456, doi:10.1007/s10270-012-0255-y.
[47] N. Bencomo, P. Sawyer, G.S. Blair, P. Grace, Dynamically adaptive systems are product lines too: using model-driven techniques to capture dynamic variability of adaptive systems., in: SPLC, 2008, pp. 23–32. http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/papers/BencomoDynAdapt.pdf.
[48] R. Mizouni, M.A. Matar, Z.A. Mahmoud, S. Alzahmi, A. Salah, A framework for context-aware self-adaptive mobile applications SPL, Expert Syst. Appl. 41 (16) (2014) 7549–7564, doi:10.1016/j.eswa.2014.05.049.
[49] J.D.A.S. Eleuterio, C.M.F. Rubira, A comparative study of dynamic software product line solutions for building self-adaptive systems, Comput. Inst., Univ. Campinas, 2017 (C-17-05).
[50] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: architecture-based self-adaptation with reusable infrastructure, Computer 37 (10) (2004) 46–54, doi:10.1109/MC.2004.175.
[51] D. Garlan, B. Schmerl, S.-W. Cheng, Software architecture-based self-adaptation, in: Y. Zhang, L.T. Yang, M.K. Denko (Eds.), Autonomic Computing and Networking, Springer US, 2009, pp. 31–55.
[52] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, U. Scholz, MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 164–182.
[53] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A.B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A.V. Papadopoulos, S. Ray, A.M. Sharifloo, S. Shevtsov, M. Ujma, T. Vogel, Software engineering meets control theory, in: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, in: SEAMS '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 71–82.
[54] O. Gunalp, C. Escoffier, P. Lalanda, Rondo atool suite for continuous deployment in dynamic environments, in: International Conference on Services Computing, IEEE, 2015, pp. 720–727, doi:10.1109/SCC.2015.102.