A domain model for self-adaptive software systems

Alizadeh Moghaddam, F.; Deckers, R.; Procaccianti, G.; Grosso, P.; Lago, P.

DOI
10.1145/3129790.3129824

Publication date
2017

Document Version
Final published version

Published in
ECSA 2017

License
Article 25fa Dutch Copyright Act (https://www.openaccess.nl/en/in-the-netherlands/you-share-we-take-care)

Link to publication

Citation for published version (APA):
A Domain Model for Self-Adaptive Software Systems

Fahimeh Alizadeh Moghaddam
aPS2 and SNE research groups
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
f.alizadehmoghaddam@vu.nl

Robert Deckers
aSoftware and Services (S2) group
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
robert.deckers@atomfreeit.com

Giuseppe Procaccianti
aSoftware and Services (S2) group
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
g.procaccianti@vu.nl

Paola Grosso
bSNE research group
University of Amsterdam
Amsterdam, The Netherlands
p.grosso@uva.nl

Patricia Lago
bSNE research group
University of Amsterdam
Amsterdam, The Netherlands
p.lago@vu.nl

ABSTRACT
Modern software systems operate in a continuously changing environment. To keep up with these changes, they can benefit from self-adaptation mechanisms. A number of modeling frameworks have been introduced to realize self-adaptive software systems. However, they usually focus either on runtime adaptation or on designing self-adaptability. As a consequence, they do not provide a clear link between architecture-level and system-level concepts. Without this link, we cannot ensure that the realized system will deliver the designed-for self-adaptability.

In this paper, we address this problem by introducing a domain model for self-adaptive software systems that encompasses both levels. Our model can be used to facilitate both architecture design (e.g. making better-informed design decisions) and system engineering (e.g. guiding self-adaptation at runtime). We show the application of our model in a case example from the literature where self-adaptation aims at energy efficiency.

CCS CONCEPTS
-Software and its engineering → Software architectures; Software system models; Software evolution;

KEYWORDS
self-adaptive software, domain model, seamless adaptation, software architecture

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ECSA '17, Canterbury, United Kingdom
© 2017 ACM. 978-1-4503-5217-8/17/09 ...$15.00
DOI 10.1145/3129790.3129824

1 INTRODUCTION
During execution, modern software systems co-exist with all sorts of uncertainties in both the environment and the systems themselves. This demands for them to be self-adaptive in order to operate correctly when runtime changes happen. Intuitively, self-adaptive software systems can first detect a (possible) runtime change and then adjust their own behavior to accommodate it. Enabling self-adaptability, however, is not a trivial task. Software architects must resolve a number of challenges:

- The simultaneous achievement of quality requirements might be difficult because of conflicts among them. For example, improving energy efficiency is not always aligned with improvements on performance [1]. The challenge is to make the optimum trade-off between those quality requirements.
- The adaptation process itself is resource-consuming. Raibulet et al. investigate the adaptation overhead from different perspectives, such as cost and adaptation time [19]. Given the fact that most software systems self-adapt for resource optimization, it is required to minimize the adaptation overhead for a seamless adaptation (i.e. transparently autonomous adaptation to runtime changes without external interference [6]).

Existing conceptual models aim to describe the behavior and the characteristics of self-adaptive software systems [2, 11, 16, 17]. However, the association between (design-time) architecture-level concepts and (runtime) system-level concepts remains implicit. If so, software architects can not ensure the realization of self-adaptability. In this work, we address this problem by introducing a domain model for self-adaptive software systems. The term domain model refers to a model that specifies a domain (that of self-adapting software) with related concepts and relationships. Our domain model gives software architects better understanding of the relation between the software design and the corresponding system realization. In the model, they can navigate how the influence of their design decisions propagates to the runtime actions. To show the application of our model, we choose a case example from the literature and explain how the model can describe its self-adaptability. The example focuses on self-adaptation for energy efficiency, which is establishing itself as a critical objective for software systems [4].
This paper is organized as follows: In Section 2 we introduce our domain model for both runtime and design-time concepts. We indicate the mapping between the concepts and the MAPE functionalities. Section 3 describes an example case study from cyber-foraging that instantiates our model. In Section 4 we discuss our arguments regarding the effectiveness of our model. Existing related work is presented in Section 5. We close the paper with Section 6, which includes our conclusions and future directions.

2 THE DOMAIN MODEL

A domain model is a collection of concepts and relationships for a specific domain. We use a domain model to describe self-adaptive software systems. It works as a visual dictionary for software architects to select the necessary mechanisms for self-adaptation. We have created our domain model as largely based on the KISS method [14] applied to the input from domain experts, which has been gathered incrementally for the past two years. For readability, we organize the concepts of the self-adaptation domain into two models: runtime and design-time.

The MAPE model identified by Brun et al. suggests that self-adaptive systems should include four main functionalities: Monitor, Analyze, Plan and Execute [5]. We identify runtime and design-time concepts that are essential for activating each of these functionalities. In the following we present each model by giving a short introduction followed by a more detailed description of the elements corresponding to each MAPE functionality. When necessary, we show the link to the elements by using the italic format.

2.1 The Runtime Model

In our description, the term “system” refers to the mix of the execution environment and its running software. Figure 1 shows the domain model for self-adaptive systems at runtime. It indicates the mapping of the runtime concepts on the MAPE functionalities:

The complete model can be found at: https://tinyurl.com/y85yy6q.
• **Monitor**: The model allows to **Monitor** the system metrics, which are quantifiable indicators of the behavior of software systems at runtime. Monitoring results in the System Metric Data by using two techniques, corresponding to runtime activities: to Measure with Meters, and to Estimate with Estimation Models.

• **Analyze**: From the System Metric Data, the system can detect Actual Runtime Changes, which are (emerging) degradations or improvements in the achievement of quality requirements. For example, a high peak in energy consumption can be a potential threat to the system, while the high availability of more efficient energy sources can be considered as a new opportunity to improve system qualities. From the perspective of the software architecture, runtime changes can be either anticipated or unanticipated. Software systems are typically well equipped for “anticipated” runtime changes at design time. Therefore, the Trend Recognition Algorithm of the system can use the Trend Specification of the metrics to detect anticipated runtime changes.

• **Plan**: For a detected runtime change, the system must find available Adaptation Architectural Tactics. Architectural tactics are design decisions that impact the response of the system to quality attributes [3]. Adaptation tactics, in particular, are mechanisms that enable the system to react when runtime changes arise. As the FORMS reference architecture suggests to separate the self-adaptation concerns and the system functionality concerns, adaptation tactics are realized as system-specific mechanisms [23]. We identify two types of adaptation tactics, namely “software architecture reconfiguration” and “infrastructural reconfiguration”.

An Adaptation Optimization Algorithm selects the optimum realization of the tactics, which results in an Adaptation Plan. The adaptation process is resource-consuming (e.g., time and energy) itself that must be considered in the adaptation plan. Therefore, the optimization algorithm selects a tactic with the minimum adaptation overhead and the maximum improvement on system qualities.

• **Execute**: The execution environment of the system, which consists of a number of devices, adapts based on the Adaptation Plan. If needed the Trend Specification is updated according to the System Metric Data. As mentioned before, trend specifications specify the expected behavior of the system in the form of the system metric data. However, in the presence of runtime changes the expectations from the system can change. For instance, if during the execution, the system is configured to supply its energy from a more efficient energy source, its expected energy consumption can become lower, which should be reflected in the trend specifications.

### 2.2 The Design-Time Model

Figure 2 shows the domain model for enabling self-adaptation at design-time. To realize the functionalities of the MAPE model at runtime, a number of activities are performed at design time. In the following we show the relevant concepts for each relevant MAPE functionality:

• **Monitor**: To define what to monitor in the software system, it is necessary for software architects first to **Identify** the Potential Runtime Changes based on the Quality Requirements. Assessing the potential runtime changes and the quality requirements will lead to the definition of System Metrics. To collect the system metric data, the execution environment can either be instrumented with Meters or implement mechanisms like Estimation Models that are algorithms to estimate varying/constant values. The metric data reported in device specifications are examples of estimation models with a constant value. Meters can be either physical devices or software components to measure the metrics. For instance, physical energy meters are used to measure energy consumption, while the execution time of a computation task can be measured in a software-based manner.

• **Analyze**: Output of this functionality is the detection of runtime changes. To detect any abnormality in the behavior of the software, one first needs to **Specify** Trends for system metrics based on the available capacity in the execution environment. This results in Trend Specifications, which (at runtime) will be used as input by an implemented Trend Recognition Algorithm. Trend recognition algorithms range from simple comparison calculations to complex prediction algorithms. In simple cases, upper/lower threshold values are defined for the target system metrics and if the system metric data exceed these values, a runtime change is detected. The cyber-foraging mobile application introduced in [8] can offload its computation tasks to a nearby surrogate for efficiency purpose. However, if the network delay to the surrogate and the mobile battery level exceed the assigned limits, the offloading plan is re-generated. In more complex cases, the response of the system to quality requirements is predicted using pattern prediction algorithms. The objective is to reduce the chance of occurrence for runtime changes. For instance, Hawarah et al. use Bayesian networks to predict the user behavior in the energy consumption of smart buildings [13].

• **Plan**: According to the identified potential runtime changes, system-specific Adaptation Architectural Tactics must be designed. For instance, in cyber-foraging mobile applications, the design decisions regarding “computation offloading to a nearby surrogate” can be realized as an adaptation tactic in reaction to low battery level in mobile devices. They will be realized as reconfigurations of the software architecture and/or the infrastructure of the system. An Adaptation Optimization Algorithm will select the adaptation tactics at the time of the runtime changes.

### 3 CASE EXAMPLE: ENERGY EFFICIENT OFFLOADING IN MOBILE CYBER-FORAGING

We illustrate the application of our domain model with the help of one example application from cyber-foraging, which is a technique
The cyber-foraging mobile application can offload its computation tasks to external execution environments in order to save energy in the mobile device. Figure 3 shows the high-level architecture of the Maui cyber-foraging system proposed by Cuervo et al. [8]. **Maui Runtime App** is the cyber-foraging mobile application that receives the users computation requests. The app consists of two main components, the **Profiler** and the **Solver**. The Profiler monitors the status of the mobile resources and the network connection to the **Maui Server**. The Solver makes use of the collected data to select the best execution environment, which in this example is either the **Mobile Device** itself or the **Maui Server**. The objective is to extend the battery life of the mobile device by delegating the computation to a nearby surrogate, in this case the Maui server.

Figure 4 shows the instantiated runtime model for the Maui system. Each activity is marked with a number to show the order of realization steps:
(1) Relevant system metrics, i.e. execution time of the computation tasks (the difference between the time the app receives the computation request and the time the computation is completed), the battery level and the network connection delay, are continuously monitored. The mobile battery level and the network delay are measured with the instrumented physical meters. Estimation models are implemented to predict the execution time of the computation tasks when executing locally or remotely.

(2) Based on the energy consumption models, a trend recognition algorithm detects if a (possible) change occurs, which in this example is the faster mobile battery discharge, i.e. a battery discharge that is faster than the predicted trend.

(3) Implemented partitioning strategies will be assessed for this change.

(4) The solver will optimize the strategy by performing a cost-benefit analysis, which is a comparison between the predicted energy consumption of the mobile device when offloading the task and when locally executing. The result of the algorithm is an offloading plan that will be configured accordingly in either the mobile device or the Maui server.

(5) The mobile device and the Maui server will adapt based on the adaptation plan. If the plan is to offload the computation, a number of architectural components must be connected, such as Maui Runtime Controller of the Maui server and additional software components in the mobile device, which will transfer the offload-able computation partition to the remote server.

(6) If needed, the energy consumption model will be updated according to new system metric data.

4 DISCUSSION

Self-adaptability is a property that naturally manifests itself at runtime. Therefore, and not surprisingly, our example case focuses on how the system reacts to runtime change and adapts by re-architecting. It is more challenging to "show" how the domain model supports architects in designing self-adaptive software in a
more effective manner. To this aim, we discuss our arguments in the
following.

We classify the relevant concepts for self-adaptive systems ac-
cording to their contribution to the software lifecycle: for example
to Identify Potential Runtime Changes is relevant at design time
(drawn in light green in Figure 2) and to Monitor System Metrics is
an action occurring at runtime (drawn in black in Figure 1). This
classification helps software architects recognize the link between
“what shall happen” and “what can happen”. Further, with the help
of the concepts from the model that are relevant for both design
time and runtime (drawn in white), the link between software de-
sign and system engineering is both explicit and integrated in the
fact that some concepts exist throughout the full lifecycle.

In addition, to enable seamless adaptation, one first needs to
identify potential runtime changes. Identifying all runtime changes
at design time is challenging (if possible at all) as the runtime con-
tdition of the software systems is not completely known before
execution. We argue that our domain model helps better-informed
reasoning for the identification of potential runtime changes by
offering the potentially-relevant contextual elements. In doing so,
it can increase the number of identified changes. This resembles
the work of Tang & Lau [22] showing that the reasoning behind
the decisions can raise new design issues – in our case new po-
tential runtime changes. As the relation between design decisions
is emphasized in our domain model, the improvement in the rea-
soning can eventually be extended to other decisions, such as se-
lecting suitable adaptation mechanisms. For example, to achieve
self-adaptation, software systems must employ the intelligence for
an effective adoption of adaptation tactics. This means the available
adaptation mechanisms must be selected according to the runtime
changes. An implemented optimization algorithm makes differ-
ent trade-offs between quality requirements to minimize the gap
between the current behavior and the expected behavior of the
system. In our domain model, the activities realizing the “Plan
MAPE functionality both at runtime and design-time, deliver this
trade-off analysis.

In particular, the improvement in reasoning can eventually im-
prove the classification of anticipated and unanticipated changes.
Software architects choose which identified potential changes should
be anticipated for the system and which potential changes should
remain unanticipated. Tools like “probability and impact matrix
are used to perform a risk assessment of each identified change [12].
For instance, one could leave changes with a low impact and with
a small probability of occurrence unanticipated. Better-informed
design decisions can link better “Potential Runtime Change” (a
design-time concept) and “Actual Runtime Change” (a runtime
concept) that are presented in our model.

5 RELATED WORK

A number of survey studies introduce characteristics of existing
self-adaptive solutions [7, 9, 21]. Our focus, however, is on self-
adaptability at the architectural level. The essential role of software
architecture in self-adaptability is investigated by Oreizy et al. [16],
which outline an architecture-based approach that contains high-
level processes to enable self-adaptation. Their approach describes
the two aspects of a self-adaptive system at runtime: evolution
management and adaptation management. In a more recent study,
they discuss a number of architectural styles that can enable self-
adaptation [17]. However, their focus is on the runtime adaptation
by re-configuring software architecture. Their model does not
include design decisions for enabling such reconfigurations in the
form of the MAPE model functionalities.

Andersson et al. propose a classification of modeling dimensions
for self-adaptive systems [2]. The idea is to gather a common vo-
cabulary for engineers to specify self-● properties of self-adaptive
systems. The Rainbow framework [11] adopts an architecture-
based approach with the focus on re-usability. At the same time, it
recognizes a number of mechanisms for specializing the infrastruc-
ture for the sake of applicability. Their focus is on self-adaptation
at runtime with proposing an external architectural layer to mon-
tor and reconfigure the system. They do not address the design
time activities such as change identification and the selection of
adaptation mechanisms.

Weyns et al. present the FORMS (FOrmal Reference Model for
Self-adaptation) reference model that includes specifications of
modeling elements and relationships among them in the design of
self-adaptive software systems [23]. They focus on separation of
self-adaptation concerns (meta-level subsystem) and system func-
tionality concerns (base-level subsystem) at the architectural level.
However, FORMS does not support the trace-ability link between
the design specification and the system’s implementation.

A number of self-adaptation approaches targeting specific do-
main of wireless sensor networks (WSNs) exist. For instance, Ruiz
et al. propose MANNA, which is a decentralized policy-based man-
agement architecture [20]. Based on policies, MANNA achieves
the desired behavior of the system, while it minimizes the gener-
ated traffic by the management layer. Agilla is a middleware for
WSNs that implements a mobile multi-agent approach to support
self-adaptation [10]. Portocarrero et al. introduce a reference archi-
tecture for self-adaptive service-oriented WSNs [18]. These studies
place their focus on specific runtime changes (e.g. undesirable en-
ergy level in sensors) and specific adaptation tactics (e.g. replacing
a low energy node). In comparison, in our model, we emphasize the
design-time activities for “change identification”, “system metric
identification”, and “adaptation tactic selection”, which potentially
help to make better-informed design decisions that realize self-
adaptability at runtime. We provide the fine-grained explicit link
between the design decisions and the runtime behavior of the sys-
tem. We present the design-time concepts and the runtime concepts
in the form of MAPE model functionalities that enable seamless
adaptation.

6 CONCLUSION

This paper presents a domain model for self-adaptive software
systems. The design of self-adaptive software systems is supported
by introducing the relevant concepts at design time and runtime.
Software architects can use our model to enable seamless adaptation
that corresponds to the MAPE model functionalities.

To qualitatively illustrate our model, we used an example from
the literature that enables self-adaptation for the purpose of energy
efficiency. We described how this example system can benefit from
our model to ensure its energy efficiency. In future work we plan
to quantitatively evaluate self-adaptive systems designed as based on our domain model.

Although seamless adaptation is only evident at runtime, we argue that software architects can ensure self-adaptability with the help of our model by employing the runtime concepts of the model as elements of their architecture design. Future work will focus on evaluating in practice the power of our domain model for effective design decision making.

REFERENCES


