Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems

Bradley Schmerl[1], Jesper Andersson[2], Thomas Vogel[3], Myra B. Cohen[4], Cecilia M. F. Rubira[5], Yuriy Brun[6], Alessandra Gorla[7], Franco Zambonelli[8], and Luciano Baresi[9]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
schmerl@cs.cmu.edu
[2] Linnaeus University, Växjö, Sweden
jesper.andersson@lnu.se
[3] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
thomas.vogel@hpi.de
[4] University of Nebraska, Lincoln, NE, USA
myra@cse.unl.edu
[5] University of Campinas, Campinas, SP, Brazil
cmrubira@ic.unicamp.br
[6] University of Massachusetts, Amherst, MA, USA
brun@cs.umass.edu
[7] IMDEA Software Institute, Madrid, Spain
alessandra.gorla@imdea.org
[8] University of Modena and Reggio Emilia, Modena, Italy
franco.zambonelli@unimore.it
[9] Politecnico di Milano, Milano, Italy
luciano.baresi@polimi.it

Abstract. Self-adaptive software systems adapt to changes in the environment, in the system itself, in their requirements, or in their business objectives. Typically, these systems attempt to maintain system goals at run time and often provide assurance that they will meet their goals under dynamic and uncertain circumstances. While significant research has focused on ways to engineer self-adaptive capabilities into both new and legacy software systems, less work has been conducted on how to *assure* that self-adaptation maintains system goals. For traditional, especially safety-critical software systems, assurance techniques decompose assurances into sub-goals and evidence that can be provided by parts of the system. Existing approaches also exist for composing assurances, in terms of composing multiple goals and composing assurances in systems of systems. While some of these techniques may be applied to self-adaptive systems, we argue that several significant challenges remain in applying them to self-adaptive systems in this chapter. We discuss how existing assurance techniques can be applied to composing and decomposing assurances for self-adaptive systems, highlight the challenges in applying them, summarize existing research to address some of these challenges, and identify gaps and opportunities to be addressed by future research.

1 Introduction

Modern software systems typically have to operate in complex and diverse environments and conditions. For example, business and cloud-based systems must cater for a wide range of load and customer profiles, and systems that manage physical elements must deal with uncertainties in the physical world. Self-adaptive systems form a category of software that changes, reconfigures, or fixes itself as it is running. Much research has been conducted into different methods for constructing self-adaptive systems, for example by integrating control loops to manage systems or by using self-organizing or bio-inspired principles [42]. Self-adaptive systems often attempt to maintain or achieve system goals in the face of uncertainty, and are usually constructed to provide some confidence that a system at run time will continue to operate appropriately, even in changing and uncertain circumstances.

While various methods for constructing self-adaptive systems have proven successful in a number of domains, *assuring* the self-adaptive aspects of these systems remains a challenge. Assuring self-adaptive systems requires run-time validation and verification (V&V) activities [45]. This is mainly because the combination of self-adaptive configurations and the environments that they encounter leads to a state explosion that makes static V&V challenging. One way to address this challenge is to apply techniques for decomposing and composing assurances. For safety-critical systems there is a large body of work on constructing safety cases, or more generally assurance cases, that construct assurance arguments about these kinds of systems. Reasoning about assurances in safety-critical systems may shed some light on how to provide these assurances for self-adaptive systems. Typically, assurances involve decomposing top level goals into argumentation structures that involve sub-goals, strategies for achieving the goals, and defining evidence that can be collected to show that the goals are achieved. Top level goals can also be composed together to provide assurances about a system with multiple goals, to reuse some assurances for goals in similar systems, or to provide assurances in systems of systems.

In this chapter we discuss the challenges related to decomposing and composing assurances in self-adaptive systems. In Section 2 we give some background on assurances, focusing on assurance cases as a framework for guiding decomposition and composition of assurances. We also introduce an example that will be used to illustrate the challenges. In Section 3, we survey existing self-adaptation assurance research that has either discussed how to compose assurances, or could be used to help build an assurance argument. Section 4 identifies a set of challenges associated with composing and decomposing assurances. In Section 5 we discuss some emerging research in assurance cases that should be followed to help with composition and decomposition and outline the challenges that arise when applying assurance cases to the context of self-adaptation. In Section 6 we provide some concluding remarks.

2 Preliminaries

In this section, we briefly introduce self-adaptive systems as we conceive them in the scope of this chapter. Then, we discuss techniques for software assurance in safety-critical systems, and describe an illustrative example that we use throughout the chapter.

2.1 Self-Adaptive Systems

Current and emerging software systems are increasingly complex and distributed, and are called to operate in open-ended and unpredictable operational environments. On one hand, such uncertainty challenges the capabilities of a system to maintain its business goals if the configuration is to remain static. On the other hand, changing environments or the system itself may also modify the goals and requirements for which the system was originally structured and configured.

To tackle the above situation, human intervention has historically been required. However, human intervention is generally impossible due to the inherent decentralized nature of modern systems, or simply infeasible due to economic or temporal reasons. Accordingly software systems have to become *self-adaptive* in their behaviour, that is, capable of dynamically adapting their configuration and/or structure in an autonomous way without human supervision, in order to respond to changing situations without malfunctioning or degrading quality of service unacceptably [16, 42].

In modern software systems, self-adaptation can take place both via mechanisms integrated in individual components as well as in groups/collectives of components (e.g., [8–12]), and that have the goal of modifying something in the behaviour of a component or a collective (e.g., [6, 7]).

The study of both individual and collective adaptation mechanisms has a long history. Individual adaptation is a very important thread of research since the early years of intelligent agents [35] and reflective computing [48], and several architectures and mechanisms to enable adaptation have been proposed so far, including the recent IBM autonomic computing approach [38]. All proposals for self-adaptation at the level of individual components rely on the integration, within each component, of a closed control loop. In the control loop, a specific control component (e.g., the "autonomic manager" in the autonomic computing approach, or the "meta-component" in reflective approaches) monitors and analyses the current operational and environmental condition of the component, and plans and executes appropriate adaptation actions as needed. The predominant pattern for self-adaptation that has emerged for structuring an autonomic manager is MAPE-K [38], where each of the activities that need to occur as part of adaptation are Monitoring – or sensing – the system and the environment, Analysing the system to determine whether the current state of the system requires adaptation, Planning, which determines what adaptations to perform, and Executing to effect changes in the system. All of this is coordinated through Knowledge.

For collective adaptation, the simplest approach is to integrate a single controller in charge of managing a whole collective with a single control loop, but this approach has challenges when scaling to realistic systems. For this reason, a variety of patterns for coordinating multiple controllers and control loops has been investigated [41, 50].

For both individual and collective adaptation uniform models and tools supporting the design and development of self-adaptive systems are still missing. Furthermore, there are few methods for assuring that self-adaptive systems adapt correctly, with respect to performing as designed to achieve their intended goals, doing so in a safe and consistent manner, and ensuring that adaptations result in legal systems respecting their design and business constraints.

2.2 Assurance Cases

Self-adaptation can be decomposed into a number of activities the use of which can span from design to run time. It is therefore not possible to provide a single assurance mechanism to provide guarantees about self-adaptation. In fact, as we shall see, there exists a collection of assurance techniques that can be used during these various activities. These techniques need to be applied and structured in a principled way in order to provide assurances. We need, therefore, to carefully consider how assurances should be decomposed into these assurance activities to ensure that the activities do in fact help assure overall goals, and to ensure that we are only doing assurance activities that help meet our goals. Furthermore, because self-adaptive systems are increasingly being composed, we need methods and approaches for composing the systems' associated assurances to assure global properties about the collective adaptive system.

To do both of these things, we can look at how assurances are handled in safety-critical systems. In this section, we discuss some solutions for software assurance and propose that assurance cases could be a good starting point for decomposing and composing assurances for self-adaptive systems. In the area of safety critical systems, there has been considerable research in software assurances. An assurance can be defined as a justified measure of confidence that a system will function as intended in its environment of use.

Assuring that a system satisfies some quality and functional goals requires the construction and evaluation of a reasoning approach based on claims, arguments, evidence, and expertise. For example, a *safety case* presents a structured demonstration that a system is acceptably safe in a given context. In other words, it is a comprehensive presentation of evidence linked by argument to a claim. For example, if we are trying to assure a claim *Claim1*, then an assurance case might decompose this claim into two subclaims, *Claim2* and *Claim3* that are easier to show, with some argument that says that if *Claim2* and *Claim3* are true, then *Claim1* is true. We could then provide some evidence that shows that each of *Claim2* and *Claim3* are true. Structuring evidence in such a way means that an expert can make a judgement that the argument makes sense and thus, if that evidence is provided, have confidence that the system is acceptably safe. *Assurance cases* are a generalization of safety cases to construct arguments that are about more than just safety.

While this rationale can be presented textually in documentation, it has proven useful to use graphical notations that help define and present the argumentation structure for assurance cases. The structure of an assurance case can be graphically represented using, for instance, the Goal Structuring Notation (GSN) [37] or Claims-Argument-Evidence (CAE) [5]. GSN is a well-accepted graphical notation to show how claims (or goals) can be broken down into sub-claims, and eventually supported by evidence, making clear the argumentation strategies adopted, the rationale for the approach (assumptions, justifications), and the context in which claims are stated. In general, arguments are structured hierarchically: claim, argument, sub-claims, sub-arguments, evidence. It is essential that assurance cases are presented in a clear structure, and GSN can capture the elements most critical for arguing a case (claims, evidence, argument strategy, assumptions, relation of claims to sub-claims and evidence) to build a convincing case.

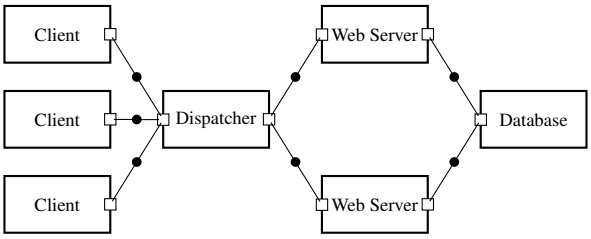

Fig. 1. Architecture of the Znn web system.

2.3 Illustrative Example

Throughout this paper we will use a simple example to illustrate how assurances could be decomposed and composed for self-adaptive systems, and some of the challenges in doing so. The example that we will use is Znn, a typical web system serving news articles and related images, that is implemented as a three-tiered web service using a standard LAMP stack (Linux, Apache, MySQL, PHP). Figure 1 shows the Znn architecture with one dispatcher, which shares the load evenly between two web servers, and one database, which stores images for served articles. While Znn itself is not self-adaptive, it provides APIs that allow a control loop to be added onto it to manage quality of service goals. Examples of a quality goal include keeping the response time below two seconds, which is related to how many servers can be used by the dispatcher and the detail of the content that each server produces (e.g., as reported in [18]). Another related goal might be to keep the operational costs below a certain threshold.

To support control loops on top of it, Znn provides a number of APIs for probing the state of the system and effecting changes. For example, it is possible to affect the configuration of Znn by changing the number of web servers or modifying the detail of the content served. Both of these simple changes can affect response time, which is information that can be retrieved from Znn using probes.

Thus, self-adaptation can be added to Znn by integrating a control loop that takes inputs from Znn probes and effects changes on Znn, via the API. For example, Rainbow [28] takes the probe inputs, abstracts them to values in the architectural model of the system, and then conducts an analysis of this architecture to determine if something is wrong (e.g., the response time is too high). If corrective action should be taken, Rainbow balances various business quality concerns in order to decide the best effects to make in the system [17]. For example, it will trade-off increasing the number of servers (thereby increasing costs), decreasing content detail, with the effect these will have on response time, and choose an option that has the highest overall utility.

Znn is an interesting example for decomposing assurances because we would like to assure that the response time is below the threshold. For example, we want to be able to answer questions such as how does one construct this argument, and what forms of evidence can be provided by a self-adaptive system? We can consider two aspects of composition of assurances in Znn. First, if there is a self-adaptive system that is trying to assure response time goals, and we want to combine this with an assurance that costs do not go above a certain amount, we want to be able to reason about how the assurances

can be defined, whether there are any conflicts and how they are resolved, and the kinds of evidence and strategies that can be used to reason about the assurance. Second, we are interested in similar questions if two self-adaptive systems are being composed (e.g., Znn, for which performance and cost are important, and another system that uses the same infrastructure but has an additional goal of security).

Assurance Cases for Znn Self-Adaptation. As discussed in Section 2.2, assurance cases could be used to organize assurances for self-adaptive systems and to identify evidence that can be provided for those assurances. Among others, such evidence can be based on observations, testing, simulation, and the process used to construct the system. The argument around an assurance case represents a high-level explanation of how evidence combines to show that the goals (or claims) will be met. The evidence and arguments are usually structured as a tree, with high-level goals being decomposed into increasingly fine-grained sub-goals that are eventually supported by evidence.

As mentioned in Section 2.2, one common way to document assurance cases is to use the Goal Structuring Notation (GSN) [30] to structure the assurance case as a tree. GSN has nodes for *claims* (or *goals*) that need to be shown and form part of the argument. These can be decomposed into sub-claims or strategies. *Strategies* describe how the claim is to be shown for the assurance case, and then *evidence* or *solutions* are activities or evidence that is used to support the claim. Associated with each claim or strategy is a *context*, which states the assumptions under which the claim is made or in which the strategy is valid. If it is necessary to state the assumptions that a strategy relies on to be valid, or to justify a strategy, these can be documented via *Assumptions* or *Justifications* in the assurance case. Finally, goals or strategies that have not been decomposed can be denoted by placing a diamond underneath them in the graphical notation. The graphical legend for these items in GSN are denoted in Figure 2.

Consider a high-level goal for Znn that it will be able to reply to all requests within 2 seconds. An example of an assurance case for this goal is shown in Figure 3. An engineer may choose a strategy of designing two versions of Znn, one for normal operation (G2) and one for high-load operation (G3), and a method for adapting the system by switching between these versions when the assumptions change (i.e., the user load changes). The sub-goals G2 and G3 would then show that the response time goals are met in each of these versions under the different load contexts. To obtain evidence for each of these sub-goals, architectural performance analysis (e.g., based on queuing theory) for that version and validation processes throughout development such as component-level testing might be used. In this context, assumptions concerning individual components that are made in the architectural analysis can be supported by evidence from component-level tests. Evidence would then be the results of the analysis and tests, which give one the confidence that each sub-goal is met in its context. In this way, an assurance is decomposed into sub-goals and evidence while the strategies form the arguments for the assurance case.

The assurances cases for G2 and G3 would proceed as normal for the static design and assurance of the various Znn modes/versions. We will not discuss these further here. Instead, we are concerned with composing and decomposing assurances that relate to the *self-adaptive* part of the system. Such an example happens in Znn when the user

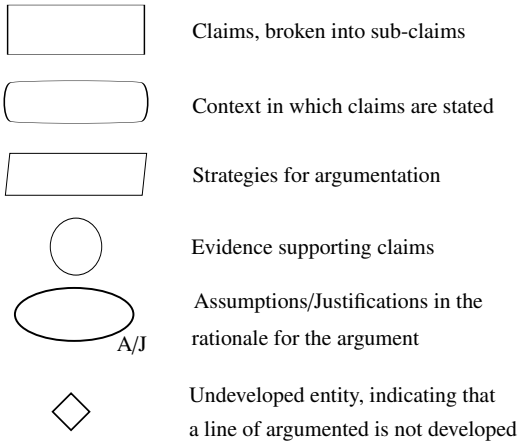

Fig. 2. Goal Structuring Notation legend.

load forces a change in modes (e.g., the load changes from 3000 requests per second to 7000 requests per second). In this case, we want to provide the assurance that the mode switch happens within 1 second (G4). We will elaborate the assurance case of G4 to illustrate both, how we might use assurance cases to guide decomposition and composition, and to highlight some of the challenges associated with each of these.

3 Assurance Decomposition and Composition in Self-Adaptive Systems

The focus of much of the research in self-adaptive systems to date has been to engineer systems that can maintain stated goals, even in the presence of uncertain and changing environments. There is existing research in assurances for self-adaptive systems that either addresses how to compose assurances, or can be used as part of an argument in assurance cases. The purpose of this section is to summarize some of this research, and to illustrate how it might apply to the problem of decomposition and composition specifically.

As argued above, it is not possible to provide a single, monolithic assurance for the goals of a self-adaptive system. Assurance cases can provide a way to organize existing work on assurances into self-adaptive system. We can organize this existing work in the following areas:

Evidence types and sub-goals for use in assurance case decomposition. Each of the classic activities of self-adaptation - monitoring, analysis, planning, and execution - have existing techniques that help to provide evidence for goals that can be used in assurance cases. These approaches could be used in assurance case decomposition.

Assurance composition based on the MAPE-K loop. Once assurances have been decomposed, we need ways to recompose the assurances. Many of these will need to

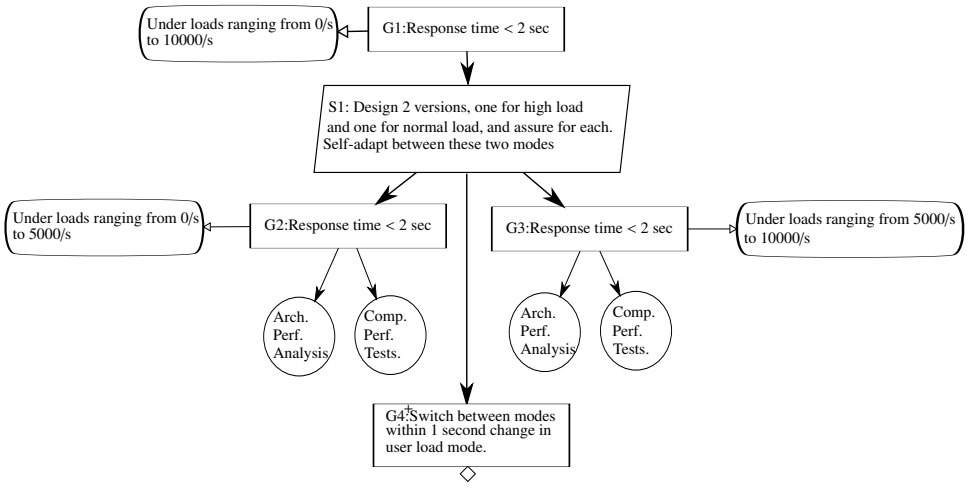

Fig. 3. An example Goal Structuring Notation diagram for Znn.

be managed at run time. There is work on integrating various verification tasks into self-adaptation as a way to provide assurances as an intimate part of self adaptation.

3.1 Evidence Types for Use in Assurance Case Decomposition

When considering decomposition, it is necessary to identify what kinds of evidence and sub-goals could be used in assurance cases. In this section, we summarize some of the on-going research into this, organizing it into a discussion of work that could be used for evidence from each of the MAPE-K activities.

Monitoring. Casanova [13, 14] provides some foundational work that can be used to provide evidence that there is sufficient knowledge to diagnose a problem in a system. In [13], the information theoretic concept of entropy is used to determine how much information is needed by a statistical and model driven diagnostic approach. The amount of information needed can be updated autonomously. In [14], formal criteria are given for establishing the maximum theoretical accuracy bounds for diagnostics given a set of observations, and the minimum bounds for accuracy (in the case of single fault systems). This theory can be used to provide evidence that enough monitoring is in place for specific diagnosis, even when some of the components in the system cannot be observed but might be the cause of problems.

Requirements may also be used to provide evidence of sufficient monitoring. In [1] they use contextual goal models to identify monitoring requirements. Different contexts provide different facts that can be monitored, and can lead to problems in the monitoring requirements (such as redundancy and inconsistency). SAT-solvers are used to produce equivalent and less costly monitoring requirements that can be shown to be optimal for monitoring the requirements.

The above approaches show that it is possible to provide evidence that a self-adaptive system is *monitoring enough information* about the target system. However, assurances related to the granularity, timing, and effect of the monitoring on the target system have yet to be investigated.

Analysis. The analysis activity is related to interpreting information gained from monitoring in the context of whether an adaptation should be done. Analysis can range from checking the correctness of the model to applying mathematical analysis of the model. For example, [39] uses performance and workflow models to analyse traffic coming into a distributed transactional system. These models are used at design time to determine initial resource provisioning given assumptions about the environment, and at run time to determine correct provisioning to new workflows and configurations. A similar approach is used [2] to deal with Denial of Service attacks, where performance models are used to determine whether to divert traffic to a checkpoint, which then issues a challenge to determine if the traffic originates from a bot. In this case the soundness of the mathematical models and their updating and use at run time provide evidence that performance goals will be met. Performance models were also used in [19] to provide evidence that appropriate constraints, monitors, and mitigations are designed into a self-adaptive system to manage performance.

In general, assurances providing sound analytical models or simulations (e.g., [26]) can be used to provide evidence that analysis is sufficient to decide if some adaptation needs to be performed. One thing to note here is that parts of the analysis can be done at design time, and parts at run time. Furthermore, if the part of the analysis that is done at run time needs to be done in a timely manner, and in this case evidence needs to be given that the analysis will, for example, detect problems in enough time to do something about them.

Planning. As mentioned above, self-adaptive systems must produce adaptations that return systems from an undesired state to a normal state. Ideally, we would like to be able to provide evidence that adaptations always achieve this transition, but because of various sources of uncertainty, this is not possible. However, there has been some work that is making strides in providing some evidence. In the case where adaptations are being chosen from a set that is predefined, probabilistic model checking can be used to determine the adaptation that has the higher likelihood of success. In [43], the authors show how this approach may be used to provide evidence about the effect of adaptations on system quality objectives, and how this may be used to provide evidence that all states in some region of the state space of the system will be improved by selected adaptations. This evidence helps to show that strategies have sufficient coverage of some part of the state space. Adaptive CTL (AdaCTL) is defined in [22] to also provide some analytical evidence that there is sufficient coverage of adaptations to achieve desired goals in changing, but enumerated, environments. The summary of formal methods used in self-adaptive systems also concentrates on assurances for planning [49]. They note that many of the formal methods used for assuring planning happen during the design of the self-adaptive system, and not at run time. Filieri [24, 25], on the other hand, describe a number of strategies for using probabilistic model checking at run time for self-adaptive

systems where goals are expressed as temporal logical formulas, including state elimination and algebraic approaches for making it more tractable at run time.

Execution. When an adaptation is triggered, we want to be able to assure several things. For example, we want to assure that the system correctly executes the effects and that the system and model are (eventually) consistent, we want to be able to assure that, if two adaptations can execute simultaneously, that they do not interfere with each other in unpredictable ways, and we require assurances that the execution will not in fact have a deleterious affect on the qualities that we are concerned about. Some work has been done in trying to answer these questions. In Veritas [27] some of these assurances can be evidenced by run-time testing. As the system is adapting, so too should the test cases. Veritas uses a genetic algorithm approach to evolve test cases, and utility functions to choose and prioritize test cases that need to be run to assure that the system is still executing safely and correctly.

3.2 Assurance Composition Based on the MAPE-K Loop

One of the challenges for providing assurances for self-adaptive systems is how to integrate assurances into the process of self-adaptation at run time. In the context of composition, this can be seen as developing techniques to allow evidence to be collected and collated at run time. For example, Tamura et al. [45] discuss the need for validation and verification (V&V) in self-adaptive systems, and argue that run time V&V tasks should be integrated into the activities of self-adaptation. They integrate the V&V tasks into the MAPE-K loop.

While this approach does not provide any specific techniques for providing assurances, it does define a framework for integrating and positioning self-adaptive elements that could provide some evidence for assurance cases, and could be used to structure this evidence in assurance cases. In particular, service level agreements can be thought of as high level goals in an assurance case, and so the Runtime Validator and Verifier, in checking that after execution of an adaptation the goals will be met. It is conceivable that other pieces of evidence (such as evidence that a model accurately reflects the system being managed) could be incorporated into this structure.

Another aspect of V&V discussed in [45] is viability zones, which are the set of possible systems states in which goals can be achieved that evolve with environment and context changes. This is elaborated upon in [49], which identifies adaptation zones as a way to understand the state space of self-adaptive systems, and as a framework for understanding the use of model checking in providing evidence of self-adaptive behaviour. They identify four zones: (1) Normal behaviour, which is the state where the system is running in its designed functionality; (2) Undesired behaviour, which is where a system is not meeting its goals or properties and requires adaptation; (3) Adaptation behaviour, which is when the system is adapting itself to fix the undesired behaviour, and (4) Invalid behaviour which are behaviours that the system should never exhibit (e.g., system deadlock). They then identify model checking work that assures properties in each and between these zones. This work can be use to identify the evidence types that have been used for assuring different parts of self-adaptation, and organizing such evidence around these parts that can aid in composition.

Model-checking evidence for the most part concentrates on assuring the design of the self-adaptive system. In [40], the authors outline an approach to testing implemented self-adaptive systems. They use a Failure Mode and Effects Analysis on the activities in the MAPE-K loop that allows them to categorize different possible problems that need to be assured (in this case, tested). The seven categories that they identify range from providing assurances that sensor information is correctly interpreted to assuring that adaptation effects are correctly effected in the system. All of these categories need to be assured for model based testing to be considered comprehensive. This work provides another way to organize the assurance activities that could be used to compose an assurance case for a particular goal.

4 Decomposing and Composing Assurances to Self-Adaptation

As we have argued in this chapter, we need an approach that organizes assurance techniques and the results they produce into a rational argument where justifications of the goals for the self-adaptive system can be checked and assessed throughout the system's life cycle. In this section we discuss some of the challenges with decomposition and composition of assurances. The challenges for decomposition and composition discussed below constitute the set of requirements any approach must manage.

4.1 Decomposition of Assurances

Assurance cases decompose assurance problems by breaking high level goals into subgoals for which it easier to provide evidence. This evidence can be combined through argumentation and judgement to provide confidence that the goals will be met. Not surprisingly, we can use assurance cases as a guide for thinking about decomposition of assurances for self-adaptive systems. In the previous section, we saw how existing work in assurances for different activities of MAPE-K can be used as evidence. In this section, we describe some of the challenges associated with decomposing the assurances.

Decomposition of Goals. A fundamental tenet of assurances cases is being able to decompose goals. In goal-oriented requirements engineering [46], goals describe the objectives that the software system should achieve. Such goals are used in the requirements engineering process for "eliciting, elaborating, structuring, specifying, analysing, negotiating, documenting, and modifying requirements" [46, p. 249]. An abstract goal (i.e., the root of a goal tree) is systematically and iteratively refined to subgoals until each subgoal (i.e., the leaves of the goal tree) can be satisfied by a set of tasks a single agent can perform. Such an agent can be a human or a software component. Approaches extending the principles of goal-oriented requirements engineering have been proposed to address self-adaptive software systems [15, 44].

For *functional* decomposition of goals, the system is decomposed into multiple components. For each component we have to establish evidence that it correctly realizes its tasks to achieve the subgoal assigned to it. For instance in Znn (c.f. Section 2.3), we may provide evidence for the correct behaviour and processing time of the dispatcher

regardless of the size and configuration of the server pool, that is, whether the dispatching of requests works and how much time it takes. This evidence focuses on an individual component of the system and the related subgoal but contributes to the assurance of the overall response time goal of the system. Hence, functional decomposition can be exploited to decompose assurances and establish evidence for components or subsystems.

To decide whether and how to decompose *extra-functional* goals, on the other hand, is more challenging. Unlike functional goals, which can more easily be decomposed into somewhat independent pieces, extra-functional goals are cross cutting with many interdependencies, for example resource usage, which is split over many functions. Extra-functional goals may be decomposed if they are orthogonal/independent from each other, or if the interdependencies can be managed. The latter requires knowledge about how the goals affect each other and how the goals can be balanced. Utility functions are one means to do that [17, 34]. An example from the Znn case is the response time goal that may be assured without considering the associated costs and using simulation or predictive analysis. However, this may result in over-provisioning, which is not desired. Hence, we have to consider both concerns, response time and costs, together. The question then arises whether each of them can be completely assured independently and whether the resulting assurances can be composed afterwards (cf. Section 4.2). The composition might require further assurances to obtain the required confidence that a composition works. If two or more goals are tightly coupled with each other, it might not be reasonable to decompose and assure each of them individually. In such cases, we may need to keep them together in the decomposition structure. In such situations assurances are not provided for leaf goals in the goal tree but rather for a subtree. The same holds for the self-management of Znn. Considering the requirements of performing an adaptation safely and within a certain time (c.f. Goal 4 in Figure 3), we may establish evidence for both aspects individually. However, it is conceivable that executing a guaranteed safe adaptation takes more effort and time than an ad hoc adaptation. Hence, both requirements must be jointly handled when constructing and assuring them.

Decomposition Strategies Specific to Self-Adaptation. In the previous section, we discussed how we might use goal decomposition to decompose assurances, and in Section 3.2 we discussed how existing self-adaptive techniques might be considered as evidence in a decomposition. Another way to consider self-adaptation in the role of assurances is as a technique itself for achieving some goal in the system, and in such cases we need to provide assurances for the self-adaptation mechanism itself.

The performance goal of Znn is achieved by a controller automatically adapting the Znn architecture shown in Figure 1 by scaling up and down the number of web servers in response to the varying load. As depicted in Figure 3, we simplify the problem and consider only two versions of Znn, one with a smaller pool of web servers for normal load and one with a larger pool for high load. If we install a self-adaptation mechanism on top of Znn, that is, a controller that automatically reconfigures the architecture of Znn by switching between the version, we must provide assurances for the controller and the controller's interface to Znn and the environment. The addition of a self-adaptation strategy requires that we provide evidence for the strategy-specific goals (c.f. Goal 4

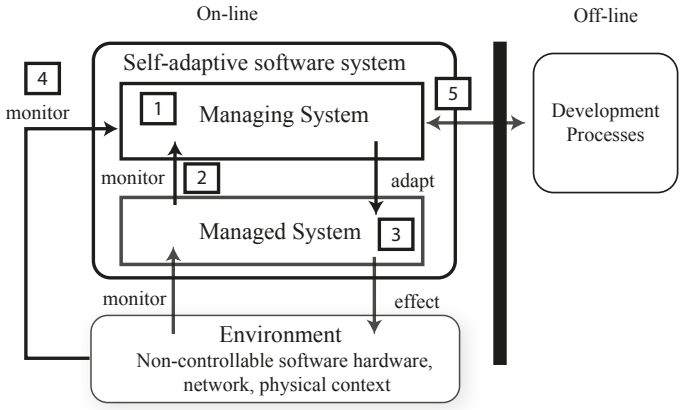

Fig. 4. Architectural Reference Model for Self-Adaptive Software Systems.

in Figure 3) in the argumentation structure. This calls for a further decomposition of assurances concerning the self-adaptation mechanism. To guide this decomposition, we have identified strategies that are specific to self-adaptive software systems.

We propose that the decomposition of goals, either functional or extra-functional ones, and the identification of evidence types and techniques are guided by the reference model for self-adaptive systems depicted in Figure 4. It provides an architectural perspective on self-adaptive systems and is helpful to identify architectural concerns for self-adaptation that require assurances and that should be included in the argumentation structure.

A managing system monitors the managed system and the environment to make a decision about adapting the managed system if the goals are not satisfied or if their satisfaction steadily decreases. For instance, in the Znn example, a controller monitors Znn to observe the current architecture and the response time, and it monitors the network to observe the number of connected clients as an indicator for the current load imposed on Znn. If the monitored response time violates the performance goal, the controller decides about scaling up the web servers and the number of web servers to be added, which is eventually translated to Znn by executing this adaptation. Based on Figure 3, the controller would switch to the Znn version designed for the high load. Finally, this architectural reconfiguration of Znn that switches to the high-load version should bring back the system into a state that fulfils the performance goal.

Consequently, besides providing assurances for the managed system such as Znn, we have to establish assurances for the managing system. We exemplify this with its functional goals, where we must provide convincing evidence that the managing system

- makes a correct decision of when and how to adapt the managed system (cf. **1** in Figure 4),
- correctly monitors the managed system **2** and the environment **4** and that the assurances and assumptions provided for managed system and environment are correct such that the managing system can rely on them,

– correctly adapts the managed system 3 that in turn must change according to this adaptation,
– correctly interacts with the development process 5, for example, an administrator directly adapts the running Znn instance in a situation that the managed system cannot handle, or an engineer tunes the adaptation strategies of the managing system to improve the performance of the self-adaptation.

A similar exercise for the managing system's extra-functional goals can be guided by the same reference architecture. Using this approach, the managing and managed systems do not have to be considered as black boxes and their decompositions can be taken into account. We may repeat the decomposition and refine the managing system to monitor, analyse, plan, execute, and knowledge components as proposed by MAPE-K [38] and consequently, assurances can be provided for the individual components rather than for whole controller.

These aspects (1 to 5) must be covered by assurance cases. For the performance goal of Znn, evidence is required that Znn properly implements the monitor 2 and adapt 3 interface such that the controller can rely on certain timeliness and accuracy of monitored data and on the eventual execution of a reconfiguration. Moreover, evidence is required that the assumption concerning the environment 4 hold, for instance, that the controller can reliably derive the user load on Znn from the network. Finally, evidence is needed that the controller itself works properly 1. For instance, a controller typically should fulfil the properties of stability, accuracy, settling time, and overshoot [32] in addition to maintaining the managed system in a state that fulfils the goals such as the performance goal in the case of Znn.

Challenges. So far, we have outlined how we might use assurance cases and the MAPE-K pattern as a framework for decomposing assurances. We now summarize the challenges.

Time- and Lifecycle-related decomposition: The connection between on-line and off-line assurances (cf. Figure 4) raises a number of challenges. On-line techniques are embedded in the self-adaptive system while off-line techniques work in the development or maintenance environment of the system. Though both kinds of techniques are used while the system is running, the distinction becomes relevant if costly assurance techniques such as model checking cannot be used on-line and thus have to be performed off-line. One challenge here is to understand which evidence and goals are more suitable for run time collection and verification, and which are more suitable earlier. For example, providing evidence through model checking tools is difficult to do at run time because of state explosion and computational complexity, but doing this earlier may not account for uncertainty. In this case, we need to explore ways to parameterise the model checking so that parts of it can be done at run time and parts at design time. But a general challenge is how to use evidence collected before deployment to inform and make efficient any run-time analysis. We also need to develop guidelines of how to decompose the evidence along the time dimension.

Matching evidence with goals: Decisions have to be made about which evidence types and techniques should be used for the assurance of which goals, and how we

know that we have enough evidence to assure a goal. This requires knowledge about the different evidence types, such as the level of confidence that they provide (e.g., simulation results refer to individual traces of the system while model checking results refer to the whole state space of the system) or the costs of using them (e.g., model checking can be infeasible due to the problem of state-space explosion).

Assurance additivity and independence: During decomposition of assurances, assurance cases are mostly assumed to be global to the system, and so at the top level are assumed to be independent and complete — conflicts are resolved with the goal tree and rationale. For self-adaptive systems, this global assumption will not hold if we are composing goals and systems at run time. In this case, we need to be able to reason about assurance additivity, independence, and conflict resolution. The identification of assurances that remain independent and can be added together is important for composition — in this case, composition is relatively straightforward. An important aspect of this challenge is to develop a set of sufficient criteria for assurance independence. Part of this challenge may be alleviated with strong provenance and annotations of global assumptions made during decomposition. At the systems of systems level, the subsumption of goals and conflicts in goals must also be identified. For example, if we have an assurance case in Znn for a goal of the response time being less than 2 seconds, and we are composing with an assurance for a goal of less than 5 seconds, is the former assurance case sufficient?

4.2 Composition of Assurances

An orthogonal approach to decomposition is *composition*. While the aim of decomposition is to make the task of gathering individual assurances simpler via modularization, composing assurances aims to construct an argument by assembling arguments together. To achieve composition, global system information is required. Decomposition may remove the larger context of the system, however it is necessary for each assurance to maintain its provenance and running context, as well as have a clear position within the argumentation structure. As mentioned in Section 4.1, individual decomposed assurance cases have a close analogy with unit tests, while composing assurances is more closely aligned with integration and system testing. Unit tests are run without the global view of the system, and may either over- or under-approximate system behaviour, while integration and system tests consider the environment under which they are run. Bate and Kelly [3] argue that to compose assurances for modular systems the modules should align with the hardware and software components. They also point out the need to consider trade-offs in goals which we discuss below. However, if decomposition should be performed via goals and evidence, then we argue that composition should follow these dimensions as well. Our discussion below assumes that we take this view of composition.

Types of Composition. In composition of assurances (either goal-based or evidence-based), individual facts are aggregated to confirm that a goal holds. Composition can be of three types (1) composing assurances from a single system with a single set of goals, (2) composing two or more individual systems each with their own goals (and assurance

cases), or (3) composing systems of systems, with multiple systems, multiple goals and multiple assurance units for each. In the first type, if our argumentation structure from the original decomposition exists, then composition can simply gather the individual assurances using the provided argumentation structure and compose these with confidence. We focus instead on challenges that arise due to the second and third type of composition.

When composing assurances from two different systems that work together (type 2), each may have its own unique goals. Consider the case where we introduce a second system, Zbay, to work in concert with Znn that requires a higher security profile than the original to permit financial transactions. Zbay runs over an https connection and has a less stringent QoS goal from the original insecure Znn — the response time must be less than five seconds. It also has additional goals not found in Znn, related to its security requirements. If we want to assure that these two systems can work in coordination, we will need to compose their assurance cases. The assurance cases for G2 and G3 from Znn and similar ones for Zbay now have different sub-goals, and their composition depends on which system they are assured under. If, for instance, we assure G2 under Znn and G3 under Zbay, there is no guarantee that these will still satisfy G1 for Znn when combined. However, this composition should suffice for Zbay and in fact, the composition of G1 and G2 from Znn could be argued to be sufficient for the whole system composition, if the argumentation structure can show that the http connection is always at least as slow or slower than https. If we now consider the context further, the dispatcher, which was previously assumed to be independent under the Znn argumentation structure, may no longer be independent in this larger system. It is possible that dispatching across http protocols changes some global assumptions and a new argumentation structure may be needed. Additionally, for the security goals, they are found only in the Zbay system, but since Zbay now can send information through Znn, the Znn goals may need to be revised to assure that sensitive information cannot flow from Znn to Zbay. We may also find dependencies between systems and goals that must be added to the argumentation structure. If, for instance, we are under https, it is possible that another aspect of the system is disallowed (such as ftp). Arguments from Znn that include this protocol must now be revisited in this larger context.

Composing two individual systems has its challenges, but as we allow for an arbitrarily large number of systems, the challenges increase. In general we see this as a systems of systems view of composition (type 3). Under this scenario, we may have multiple variants of Znn and Zbay, such as Zmazon, Zxpedia, etc. Each of these systems has a set of common goals, and may even share components such as the dispatcher. Yet they each also have their own unique requirements, working environments and constraints. To ensure that these systems can work together, we must combine assurances across the entire system. This leads to a potential combinatorial explosion in the number of compositions that can occur between systems. Not only do we now have to face the problem of combining two assurances, we may find complex interactions between three or more assurances, and it will become infeasible to validate all compositions. There may be dependencies as well, either within the systems themselves, or ones that are global. Unlike the type 2 constraints these may now span the entire system. To assure systems of this type, we may need to resort to a sampling scheme (such as that used

in combinatorial testing [20]), and accept that our argumentation only provides a certain level of assurance across the system, rather than a comprehensive one. For instance, we may argue that we know all combinations of pairs of assurances can be composed, but we may not be able to guarantee that combinations of a higher arity of assurance is still valid. Another issue that arises in systems of systems is that of competing assurances. For instance, in Zbay and Zmazon the need for security may be more important than the goal of a low response time, however in Zxpedia and Znn, response time may be paramount. Some sort of weighted utility is possible, or we may to allow for multiple solutions for a goal and view this as a Pareto front to understand the trade-offs in time and security goals.

One possible way to model and simplify the view of a self-adaptive software system is as a set of *features* that are added and removed as the system adapts [23, 29]. Elkhodary et al. first presented the notion of using features for directing adaptation for QoS aspects of a system [23]. Garvin et al. also suggested using this view of adaptation, but from a more traditional functional view of features [29]. Software product line engineering [23] provides many tools that may help our reasoning and analysis, both from a goal based and from an evidence-based view. We can then use these models to describe composition and sampling and to guide our argumentation structure.

Challenges. In the above, we outlined some challenges particular to different types of composition. We now summarize the challenges for composing assurances in general.

Time-based Composition: The time (or the state at which an assurance is obtained) can change the outcome of the evidence, or may change the type of evidence to be gathered. If Znn and Zbay are implemented as services, then the types of evidence available for composition can vary at run time, depending on which services are currently active. For instance, the dispatch service may have different variants and within those variants use different mechanisms. If we assure the system under one variant of the dispatcher but later compose our system using a different variant of the dispatcher, the original assurances may not hold. This problem can occur in all three types of composition (1-3). This dynamic view of composition leads to a new level of complexity and may require a new argumentation structure; one that was not considered during decomposition.

Assurance dependencies: In related work on software testing for component-based or configurable systems, determining which features are dependent on others has proven to be challenging [21]. Documentation is often lacking and therefore this must be performed by domain experts, and/or via program analysis. For assurance case composition this is even more challenging. Which strategies depend on particular evidence types and how does that relate to the overall assurance case? If a composition results in reusing evidence in multiple assurances cases, how do we keep track when those goals change? How do we find the dependencies and goals that need to be added to argumentation structures?

Evidence reuse: In non-adaptive systems, evidence may be reused to support multiple assurance cases. The same kind of reuse is less obvious when self-adaptation is involved. For example, if evidence for load balancing under a certain set of activated self-adaptations is collected, it may apply under a different set of adaptations, or it may

not. The larger space of potential system states makes reusing assurances more challenging.

5 Applying Assurance Cases to Self-Adaptation

In the previous sections, we outlined how we might structure decomposition and composition of assurances for self-adaptive systems, and highlighted some of the challenges with each of these. In this section we describe some emerging work in assurance cases composition and decomposition that could be applied to help make assurance cases more useful for self-adaptive systems.

5.1 Assurance Case Decomposition and Composition Research

Safety Case Patterns. While reasoning about satisfaction of individual goals using assurance cases is useful, a means of reusing and combining assurance cases is required. Some studies in developing goal-based approaches aim to support the reuse and modularization of safety cases so that safety arguments for sub-systems/components can be re-used in other contexts. The notion of safety case patterns [33] can be applied in order to explicitly model common elements found between various safety cases created for particular applications. Patterns in arguments can emerge, for example, typical combinations of arguments and accepted interpretations of specific types of evidence. These can be documented by means of a safety pattern language. This solution promotes a structured reuse of the safety case rationale instead of its informal material reuse.

More recently, the work by Hawkins et al. [31] has defined a safety argument pattern catalogue in order to guide developers in structuring *maintainable* safety arguments. The idea is to provide evidence for low-level claims, considering different levels of abstraction suitable for different stakeholders of the system. The assumption is that as the software system moves through the development lifecycle, there are numerous assurance considerations against which evidence must be provided.

Such patterns could help guide the kind of evidence that needs to be collected, or how to structure assurance arguments for particular gaols of the system.

Modularization and Contracts. The work by Ye and Kelly [51] proposes the use of contracts to modularize safety cases in order to capture application-specific safety requirements, and corresponding assurance requirements derived for a potential COTS (common-off-the-shelf) component. This contract can be used to form the basis of a safety case module for the component. The notion of compositional safety case construction proposed by Kelly [36] is used for modelling the safety case of the application separated from the assurance requirements of the component. More generally, system safety cases are often decomposed into sub-system safety cases to cope with their complexity. As a consequence, GSN was extended with the notion of UML packages and "Away Goals" in order to support the notion of modular safety case construction. Moreover, as pointed out in [36], the need for a modular safety approach is becoming more apparent when considering new types of modern systems that are emerging, such as systems of systems [3]. For self-adaptive systems, the notion of contracts could be useful in reasoning about the composition of assurance cases.

Decomposition and Composition. The support for claim decomposition and structuring is very informal and argumentation is seldom explicit [4]. In practice, the emphasis is on communication and knowledge management of the safety cases, with little guidance on what claim or claim decomposition should be performed. Some studies are developing more rigorous approaches to claim decomposition in order to demonstrate that the decomposition is complete, that is, that the sub-claims demonstrate the higher claim [4]. Furthermore, the authors highlight the importance of (i) more efficient means for modelling safety cases since they are costly to develop, and (ii) improving safety case structuring to provide safety case modularization, to use diverse arguments and evidence, and to exploit the relationship between the argument structure and the architecture of a system.

The work by Voss et al. [47] also explores the idea of modular certification when reusing components from one system to the next, that is, when reusing a system element, engineers can (in)formally reuse the associated safety arguments of the element. This solution supports a component-based development process and a model-based tool to specify the system's architecture at different layers of abstraction and it integrates the construction of the system and the argumentation about its functional safety. Of course, increased rigour in claim decomposition could be exploited for assuring self-adaptive systems.

5.2 Challenges Applying Assurance Cases to Self-Adaptation

This chapter has taken the position that work in assurance cases can be used to guide the decomposition and composition of assurances for self-adaptive systems. While we believe that this is a good approach, there are distinct challenges with applying assurance cases to self-adaptive systems.

Uncertainty: Self-adaptive systems are often self-adaptive because they are deployed in environments with uncertainty. This uncertainty affects the types of evidence that can be collected to support assurances, the ways in which the evidence can be collected, and even the specification of the assurance case itself. For example, goals in assurances cases need to specify the environmental assumptions under which they are valid but for self-adaptive systems we need some way to make uncertainty about these assumptions first-class.

Adaptation assurances: When conditions change and the system adapts, an assurance may describe how quickly or how well it adapts. For example, with Znn, an increased demand may trigger the addition of a web server. An assurance may state that when the per-server load exceeds a threshold, the system adapts within two minutes by adding web servers and the per-server load falls below the threshold within five minutes. This assurance may hold at all times, or may be expected to hold only when the demand increases but then remains constant.

Automatable assurance cases: As mentioned in Section 2.2, assurance cases rely on human judgement to discern whether the argument and rationale actually makes the case given the evidence. One of the aims of self-adaptation is to eliminate or at least reduce the involvement of humans in the management of a software system. To accomplish this, self-adaptation requires ways to computationally reason about assurance cases,

and a logic to judge whether an assurance case is still valid, what changes must be made to it in terms of additional evidence, etc.

Adaptive assurances: As just alluded to, self-adaptation may require the assurance cases themselves to adapt. For example, replacing a new component into the system may require replacing evidence associated with that component in the assurance case. Changing goals of the system based on evolving business contexts will likely involve changes to the assurance cases for those goals. Automatable assurance cases are an initial step to addressing this challenge, but approaches, rules, and techniques for adapting the assurance cases themselves are also needed.

Assurance processes for self-adaptive software systems: One overarching challenge is the design of adequate assurances processes for self-adaptive systems. Such a process connects the system's goals, the architecture, and implementation realizing the goals to the assurance cases' argumentation structures, its strategies, evidence types, and assurance techniques. This challenge requires that parts of the design and assurance process that was previously performed off-line during development time must move to run time and carried out on-line in the system itself. The assurance goals of a system are dependent on a correct, efficient and robust assurance process, which employs on-line and off-line activities to maintain continuous assurance support throughout the system life cycle. Currently, such processes are not sufficiently investigated and understood.

Reassurance: If we are able to move the evaluation of assurance cases to run time, then challenge arises in how to reassure the system when things change. Reassurance may need to happen when environment states, or the state of the system itself, change. Which part of the assurances case needs to be re-evaluated? For composition, where the composition itself is dynamic, we need ways to identify the smallest set of claims (goals) that have to be reassured when two systems are composed? Which evidence needs to be re-established, and which can be reused?

6 Conclusions

We have considered the challenges associated with decomposing and composing assurances for self-adaptive systems. While there is a large body of work in software assurance that is beginning to address this for general software systems, self-adaptive systems raise further inherent challenges.

We have discussed assurance cases as an approach to reasoning about composing and decomposing assurances for self-adaptive systems. Assurance cases provide a discipline for decomposing assurances in a principled way. Furthermore, there is some work related to assurance cases also addresses assurance composition, meaning that assurance cases may also be suitable for reasoning about composition of assurances for self-adaptive systems, and also for composing assurances in self-adaptive systems of systems. We believe that applying assurance case approaches to the problem of assuring self-adaptive systems shows great promise.

At the same time, there are many aspects of self-adaptive systems that present challenges to assurance case research. We provided some of these challenges in Section 5.2. Most of these challenges arise from the need for self-adaptive systems to respond to

changes in the environment or the requirements of the system, and so we need ways to assess elements of assurance cases automatically, and evolve them, at run time.

References

1. R. Ali, A. Griggio, A. Franzén, F. Dalpiaz, and P. Giorgini. Optimizing monitoring requirements in self-adaptive systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 362–377. Springer, 2012.
2. C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu. Mitigating dos attacks using performance model-driven adaptive algorithms. *ACM Transactions on Autonomous and Adaptive Systems*, 9(1):3:1–3:26, Mar. 2014.
3. I. Bate and T. Kelly. Architectural considerations in the certification of modular systems. In *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security*, SAFECOMP '02, pages 321–333, London, UK, UK, 2002. Springer-Verlag.
4. R. Bloomfield and P. Bishop. Safety and assurance cases: Past, present and possible future– an adelard perspective. In *Making Systems Safer*, pages 51–67. Springer London, 2010.
5. R. Bloomfield, B. Peter, C. Jones, and P. Froome. *ASCAD — Adelard Safety Case Development Manual*. Adelard, 3 Coborn Road, London E3 2DA, UK, 1998.
6. Y. Brun, J. Y. Bang, G. Edwards, and N. Medvidovic. Self-adapting reliability in distributed software systems. *IEEE Transactions on Software Engineering (TSE), in press*, 2015.
7. Y. Brun, G. Edwards, J. Y. Bang, and N. Medvidovic. Smart redundancy for distributed computation. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 665–676, Minneapolis, MN, USA, June 2011. DOI: 10.1109/ICDCS.2011.25.
8. Y. Brun and N. Medvidovic. Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In *Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS)*, pages 38–43, Dubrovnik, Croatia, September 2007. DOI: 10.1145/1316550.1316557.
9. Y. Brun and N. Medvidovic. An architectural style for solving computationally intensive problems on large networks. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Minneapolis, MN, USA, May 2007. DOI: 10.1109/SEAMS.2007.4.
10. Y. Brun and N. Medvidovic. Keeping data private while computing in the cloud. In *Proceedings of the 5th International Conference on Cloud Computing (CLOUD)*, pages 285–294, Honolulu, HI, USA, June 2012. DOI: 10.1109/CLOUD.2012.126.
11. Y. Brun and N. Medvidovic. Entrusting private computation and data to untrusted networks. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 10(4):225–238, July/August 2013. DOI: 10.1109/TDSC.2013.13.
12. Y. Brun and D. Reishus. Path finding in the tile assembly model. *Theoretical Computer Science*, 410(15):1461–1472, April 2009. DOI: 10.1016/j.tcs.2008.12.008.
13. P. Casanova, D. Garlan, B. Schmerl, and R. Abreu. Diagnosing architectural run-time failures. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 20-21 May 2013.
14. P. Casanova, D. Garlan, B. Schmerl, and R. Abreu. Diagnosing unobserved components in self-adaptive systems. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Hyderabad, India, 2-3 June 2014.
15. B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In A. Schürr and B. Selic, editors, *Proceedings of the 12th International Conference on Model Driven*

Engineering Languages and Systems (MODELS), Denver, CO, USA, volume 5795 of *Lecture Notes in Computer Science*, pages 468–483. Springer, 2009.
16. B. H. C. Cheng and al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, pages 1–26, 2009.
17. S.-W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Shanghai, China, 21-22 May 2006.
18. S.-W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS09)*, Vancouver, BC, Canada, May 2009.
19. S.-W. Cheng, D. Garlan, B. Schmerl, J. a. P. Sousa, B. Spitznagel, and P. Steenkiste. Using architectural style as a basis for self-repair. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, pages 45–59. Kluwer Academic Publishers, 25-31 August 2002.
20. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.
21. M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engingeering*, 34(5):633–650, 2008.
22. M. Cordy, A. Classen, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking adaptive software with featured transition systems. In J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 1–29. Springer Berlin Heidelberg, 2013.
23. A. Elkhodary, N. Esfahani, and S. Malek. FUSION: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 7–16, 2010.
24. A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *33rd International Conference on Software Engineering (ICSE)*, pages 341–350, May 2011.
25. A. Filieri and G. Tamburrelli. Probabilistic verification at runtime for self-adaptive systems. In J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 30–59. Springer Berlin Heidelberg, 2013.
26. J. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, and D. Garlan. Improving self-adaptation through software architecture-based stochastic modeling. *Journal of Systems and Software*, 2016.
27. E. M. Fredericks, B. DeVries, and B. H. C. Cheng. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 17–26, New York, NY, USA, 2014. ACM.
28. D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.
29. B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Failure avoidance in configurable systems through feature locality. In J. Cámara, R. Lemos, C. Ghezzi, and A. Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 266–296. Springer-Verlag, 2013.
30. Goal Structuring Notation (GSN) community standard version 1, November 2011. Available at http://goalstructingnotation.info.
31. R. Hawkins, K. Clegg, R. Alexander, and T. Kelly. Using a software safety argument pattern catalogue: Two case studies. In *SAFECOMP*, volume 6894, pages 185–198, 2011.

32. J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
33. K. M. High, T. P. Kelly, and J. A. Mcdermid. Safety case construction and reuse using patterns. In *16th International Conference on Computer Safety and Reliability (SAFECOMP 1997*, pages 55–69. Springer-Verlag, 1997.
34. N. Huber, A. Hoorn, A. Koziolek, F. Brosig, and S. Kounev. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *Service Oriented Computing and Applications*, 8(1):73–89, Mar. 2014.
35. N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, April 2001.
36. P. Kelly. Managing complex safety cases. In *11th Safety Critical System Symposium (SSS'03*, pages 99–115. Springer-Verlag, 2003.
37. T. Kelly and R. Weaver. The goal structuring notation a safety argument notation. In *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.
38. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.
39. M. Litoiu. A performance analysis method for autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 2(1), Mar. 2007.
40. G. Püschel, S. Götz, C. Wilke, and U. Aßmann. Towards systematic model-based testing of self-adaptive software. In *ADAPTIVE 2013, The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications*, pages 65–70, 2013.
41. M. Puviani, G. Cabri, and F. Zambonelli. A taxonomy of architectural patterns for self-adaptive systems. In *International C* Conference on Computer Science & Software Engineering, C3S2E13*, pages 77–85, Porto, Portugal, July 2013.
42. M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 2009.
43. B. Schmerl, J. Cámara, J. Gennari, D. Garlan, P. Casanova, G. A. Moreno, T. J. Glazier, and J. M. Barnes. Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations. In *HotSoS 2014: 2014 Symposium and Bootcamp on the Science of Security*, Raleigh, NC, USA, 8-9 April 2014.
44. V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness requirements for adaptive systems. In *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS '11)*, pages 60–69, New York, NY, USA, 2011. ACM.
45. G. Tamura, N. M. Villegas, H. A. Müller, J. a. P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezz, W. Schfer, L. Tahvildari, and K. Wong. Towards practical runtime verification and validation of self-adaptive software systems. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 108–132. Springer Berlin Heidelberg, 2013.
46. A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–262, Washington, DC, USA, 2001. IEEE Computer Society.
47. S. Voss, B. Schätz, M. Khalil, and C. Carlan. Towards modular certification using integrated model-based safety cases. In *Proc. of VeriSure: Verification and Assurance*, 2013.
48. T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 306–315, 1988.
49. D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 67–79, New York, NY, USA, 2012. ACM.

50. D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goeschka. On patterns for decentralized control in self-adaptive systems. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475, pages 76–107. Springer-Verlag, 2012.
51. F. Ye and T. Kelly. Contract-based justification for cots component within safety critical applications. In T. Cant, editor, *Ninth Australian Workshop on Safety-Related Programmable Systems (SCS 2004)*, volume 47 of *CRPIT*, pages 13–22, Brisbane, Australia, 2004. ACS.