ASPLe – A Methodology to Develop Self-Adaptive Software Systems with Reuse

Nadeem Abbas
nadeem.abbas@lnu.se

Jesper Andersson
jesper.andersson@lnu.se

Abstract

Advances in computing technologies are pushing software systems and their operating environments to become more dynamic and complex. The growing complexity of software systems coupled with uncertainties induced by runtime variations leads to challenges in software analysis and design. Self-Adaptive Software Systems (SASS) have been proposed as a solution to address design time complexity and uncertainty by adapting software systems at runtime. A vast body of knowledge on engineering self-adaptive software systems has been established. However, to the best of our knowledge, no or little work has considered systematic reuse of this knowledge. To that end, this study contributes an Autonomic Software Product Lines engineering (ASPLe) methodology. The ASPLe is based on a multi-product lines strategy which leverages systematic reuse through separation of application and adaptation logic. It provides developers with repeatable process support to design and develop self-adaptive software systems with reuse across several application domains. The methodology is composed of three core processes, and each process is organized for requirements, design, implementation, and testing activities. To exemplify and demonstrate the use of the ASPLe methodology, three application domains are used as running examples throughout the report.
Contents

1 Autonomic Software Product Lines 6
  1.1 Background ............................................. 6
    1.1.1 Self-Adaptive Software Systems ............ 6
    1.1.2 Software Reuse ................................. 7
  1.2 The ASPL Strategy ................................. 8
  1.3 Uncertainty Analysis ............................ 10
    1.3.1 Uncertainties due to Runtime Variability ... 10
    1.3.2 Uncertainties due to Development for Reuse . 12

2 The ASPLe Methodology 15
  2.1 ASPL Domain Engineering Process ............. 16
  2.2 Specialization Process ........................ 17
  2.3 Integration Process ............................. 17
  2.4 Running Examples ............................... 18
    2.4.1 Distributed Game Environment (DGE) .... 18
    2.4.2 News Service Product Line ............... 20
    2.4.3 PhotoShare Product Line ................ 21

3 ASPL Domain Engineering (ADE) 24
  3.1 Introduction ...................................... 24
  3.2 ASPL Requirements Engineering Process ....... 25
  3.3 ASPL Requirements Engineering – Demonstration 27
    3.3.1 General dQAS for Self-Upgradability .... 28
    3.3.2 General dQAS for Self-Optimization ....... 30
  3.4 ASPL Design Process ............................ 33
  3.5 ASPL Design – Demonstration ................. 35
    3.5.1 ASPL Design for Self-Upgradability .... 37
    3.5.2 ASPL Design for Self-Optimization ...... 40

4 Specialization Process 44
  4.1 Introduction ...................................... 44
  4.2 Requirements Specialization Process .......... 46
  4.3 Requirements Specialization – Demonstration .... 49
    4.3.1 Requirements Specialization for DGE .... 49
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.2</td>
<td>Requirements Specialization for NSPL</td>
<td>53</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Requirements Specialization for PSPL</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Design Specialization Process</td>
<td>65</td>
</tr>
<tr>
<td>4.5</td>
<td>Design Specialization – Demonstration</td>
<td>68</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Design Specialization for the DGE</td>
<td>68</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Design Specialization for the NSPL</td>
<td>71</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Design Specialization for the PSPL</td>
<td>73</td>
</tr>
<tr>
<td>5</td>
<td>Integration Process</td>
<td>80</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>80</td>
</tr>
<tr>
<td>5.2</td>
<td>Requirements Integration Process</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>Requirements Integration – Demonstration</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>DGE – Requirements Integration</td>
<td>83</td>
</tr>
<tr>
<td>5.4.1</td>
<td>NSPL – Requirements Integration</td>
<td>85</td>
</tr>
<tr>
<td>5.4.2</td>
<td>PSPL – Requirements Integration</td>
<td>86</td>
</tr>
<tr>
<td>5.5</td>
<td>Design Integration Process</td>
<td>90</td>
</tr>
<tr>
<td>5.6</td>
<td>Design Integration – Demonstration</td>
<td>93</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Design Integration for the DGE</td>
<td>93</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Design Integration for the NSPL</td>
<td>94</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Design Integration for the PSPL</td>
<td>95</td>
</tr>
<tr>
<td>Appendices</td>
<td>extended Architecture Reasoning Framework (eARF)</td>
<td>98</td>
</tr>
<tr>
<td>A.1</td>
<td>domain Quality Attribute Scenarios (dQAS)</td>
<td>101</td>
</tr>
<tr>
<td>A.2</td>
<td>domain Responsibility Structure</td>
<td>104</td>
</tr>
<tr>
<td>A.3</td>
<td>Architecture Patterns and Tactics</td>
<td>105</td>
</tr>
<tr>
<td>A.3.1</td>
<td>MAPE-K Feedback Loop Pattern</td>
<td>106</td>
</tr>
<tr>
<td>A.3.2</td>
<td>Monitoring Tactics</td>
<td>108</td>
</tr>
<tr>
<td>A.3.3</td>
<td>Execution Tactics</td>
<td>109</td>
</tr>
<tr>
<td>A.4</td>
<td>Analytical Framework</td>
<td>111</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Conceptual Architecture of a Self-Adaptive Software System</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Horizontal and Vertical Reuse</td>
<td>8</td>
</tr>
<tr>
<td>1.3</td>
<td>The ASPL Strategy</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>Uncertainties in the Development of Self-Adaptive Software Systems with and for Reuse</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Basic Structure of the ASPL Methodology</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>The ASPL Processes</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of a DGE Product</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>The NSPL Feature Model</td>
<td>19</td>
</tr>
<tr>
<td>2.5</td>
<td>The PSPL Feature Model</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>ASPL Domain Engineering (ADE) Process</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>ASPL Requirements Engineering Process Package</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>ASPL Scope Definition - An Example</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>ASPL Design Process Package</td>
<td>33</td>
</tr>
<tr>
<td>3.5</td>
<td>An eARF for Self-Upgradability and Self-Optimization</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>General dRS for Self-Upgradability</td>
<td>39</td>
</tr>
<tr>
<td>3.7</td>
<td>General dRS for Self-Optimization</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Specialization Process</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Requirements Specialization Process Package</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>DGE Domain Scope</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>NSPL Domain Scope</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Extended ASPL Scope</td>
<td>58</td>
</tr>
<tr>
<td>4.6</td>
<td>PSPL Domain Scope</td>
<td>61</td>
</tr>
<tr>
<td>4.7</td>
<td>Design Specialization Process Package</td>
<td>66</td>
</tr>
<tr>
<td>4.8</td>
<td>Self-Upgradability dRS Specialized for the DGE</td>
<td>69</td>
</tr>
<tr>
<td>4.9</td>
<td>Self-Optimization dRS Specialized for the NSPL</td>
<td>72</td>
</tr>
<tr>
<td>4.10</td>
<td>Self-Healing – General dRS</td>
<td>74</td>
</tr>
<tr>
<td>4.11</td>
<td>Self-Healing dRS Specialized for the NSPL</td>
<td>75</td>
</tr>
<tr>
<td>4.12</td>
<td>Self-Upgradability dRS Specialized for the PSPL</td>
<td>77</td>
</tr>
<tr>
<td>4.13</td>
<td>Self-Healing dRS Specialized for the PSPL</td>
<td>78</td>
</tr>
<tr>
<td>5.1</td>
<td>Requirements Integration Process Package</td>
<td>81</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.2</td>
<td>Design Integration Process Package</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>An Integrated Reference Architecture for Self-Upgradability in the DGE Domain</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>DGE Managed System - Self-Upgradability Architectural View</td>
<td>93</td>
</tr>
<tr>
<td>5.5</td>
<td>The NSPL Managed and Managing System Platforms – Design Integration</td>
<td>95</td>
</tr>
<tr>
<td>5.6</td>
<td>PSPL – Integrated dRS for Self-Healing</td>
<td>96</td>
</tr>
<tr>
<td>5.7</td>
<td>The PSPL Managed and Managing System Platforms – Design Integration</td>
<td>97</td>
</tr>
<tr>
<td>A.1</td>
<td>The extended Architectural Reasoning Framework</td>
<td>100</td>
</tr>
<tr>
<td>A.2</td>
<td>An example of a MAPE Pattern</td>
<td>107</td>
</tr>
<tr>
<td>A.3</td>
<td>Analytical framework to support rigorous reasoning in eARF</td>
<td>112</td>
</tr>
</tbody>
</table>
List of Tables

2.1 The NSPL Products - Optional Features 20
2.2 The PSPL Products – Optional Features 22

3.1 A General dQAS for Self-Upgradability 29
3.2 A General dQAS for Self-Optimization 31
3.3 Responsibilities extracted from the General dQAS for Self-Upgradability 36
3.4 Responsibilities extracted from the General dQAS for Self-Optimization 41

4.1 Self-Upgradability dQAS Specialized for the DGE 50
4.2 Self-Optimization dQAS Specialized for the NSPL 56
4.3 A General dQAS for Self-Healing 57
4.4 The Self-Healing dQAS Specialized for the NSPL 59
4.5 Self-Upgradability dQAS Specialized for the PSPL 63
4.6 Self-Healing dQAS Specialized for the PSPL 65

5.1 DGE – Self-Upgradability dQAS after Integration with the Managed System Platform 84
5.2 PSPL – Self-Upgradability dQAS after Integration with the Managed System Platform 87
5.3 PSPL – Self-Healing dQAS after Integration with the Managed System Platform 89

A.1 Quality Attribute Scenario (QAS) Template 102
A.2 domain Quality Attribute Scenario (dQAS) Template 103
Chapter 1

Autonomic Software Product Lines

Autonomic Software Product Lines (ASPL) is a systematic strategy to design and develop self-adaptive software systems [14] with systematic reuse. Developing software systems with self-adaptation properties enables software developers to mitigate uncertainties and complexities caused by runtime changes in systems’ goals, environments, and systems themselves. A vast body of knowledge on engineering self-adaptive software systems has been established. However, to the best of our knowledge, no or little work has considered systematic reuse of this knowledge. Systematic reuse enables software developers to produce systems with improved quality at a reduced cost and shorter time-to-market [22, 28]. The research gap and proven benefits of development with reuse motivated us to envision the ASPL strategy to develop self-adaptive systems with systematic reuse [2]. Before describing the strategy, we first present its background as follows.

1.1 Background

The background of the ASPL consists of self-adaptive software systems and software reuse. Understanding the background is vital to understand the ASPL strategy.

1.1.1 Self-Adaptive Software Systems

A Self-Adaptive Software Systems (SASS) is a software system that can adapt its behavior and structure in response to its perception of the environment, the system itself, and its goals [14]. Figure 1.1 depicts a conceptual architecture of a self-adaptive system. The managed subsystem abstracts the subsystem that provides core application functionality. The managing subsystem is an abstraction of the system’s adaptation logic that identifies runtime changes and
adapts the managed subsystem to maintain the system goals. Both the managed and managing subsystems are situated in an environment that refers to external world with which a self-adaptive system interacts.

The managing subsystem uses the monitor and adapt interfaces to observe changes in the managed subsystem and perform adaptive actions, respectively. The monitor interface between managing subsystem and the environment enables context-awareness. The managing system cannot affect the environment directly. However, it can perform adaptive actions on the managed subsystem that may affect the environment. The disciplined split among managed, managing and environment elements of a self-adaptive system offers an opportunity to design and develop such systems with and for reuse.

1.1.2 Software Reuse

Software reuse is a process of creating software systems from existing software components or artifacts rather than developing from scratch [28]. As shown in Figure 1.2, development artifacts can be reused both horizontally and vertically. The vertical reuse refers to the reuse of artifacts within a single application domain [34]. The set of such artifacts for vertical reuse is referred as a vertical platform. The horizontal reuse refers to reuse across several application domains, and a platform that provides artifacts reusable across several domains is called a horizontal platform.

Software reuse offers improved quality and productivity combined with a reduction in time and cost. However, a systematic approach to reuse is required to achieve the goals and claimed benefits. We consider systematic reuse approach as one that follows a repeatable and controlled process, concerned with large-scale reuse from requirements specifications to code, tests, and documentation, and supported by purposefully designed tools and infrastructure [18, 2]. The systematic reuse has a potential to improve the development process particularly in case of large and complex software systems [28]. The proven potential
Different software reuse methods have been proposed over the years, for instance, software libraries, design patterns, and frameworks. However, the most systematic, widely accepted and applied reuse method is Software Product Line Engineering (SPLE) [33, 39]. The fundamental idea in SPLE is to establish a reusable platform of core assets that meet common and variable requirements of a family of similar software systems. The software systems instantiated from traditional SPLs usually bind components statically at compile time or latest at load time. Such systems with static binding perform well in settings where changes to requirements, product environment, or other related concerns are rare. However, this is not true for the emerging world of more dynamic, open and inter-connected systems such as cyber-physical systems [35]. Such systems are characterized with more frequent changes in requirements, operating environments and systems themselves. The traditions SPLs approaches and derived software systems lack in support for runtime reconfigurations and adaptations in response to changes at runtime. The inability of traditional SPLs to deal with runtime variations and the lack of systematic reuse method for the design and development of SASS motived us to envision an Autonomic Software Product Lines (ASPL) strategy.

1.2 The ASPL Strategy

The ASPL is a multi-product lines strategy to design and develop self-adaptive software systems with reuse [2]. It exploits discipline split between managed and managing subsystems of a self-adaptive system, and as shown in Figure 1.3 defines a three steps strategy. Each of the three steps is described below.
Step 1: Establish a horizontal ASPL Platform The first step of the ASPL strategy is to establish a horizontal ASPL platform. The ASPL platform provides application domain independent artifacts for managing subsystems, i.e., adaptation logic. The artifacts span the range from requirements engineering, design, implementation, and testing. To support reuse across several application domains, the ASPL platform artifacts have abstract interfaces and hooks that need to be specialized and glued to meet requirements of a specific application domain.

Step 2: Derive a vertical Managing System Platform The second step transforms the horizontal ASPL platform into a vertical managing system platform. As shown in Figure 1.3, n number of application domain specific managing system platforms can be derived with reuse from a single ASPL platform. Each managing system platform targets adaptation logic in a particular application domain and is defined through a specialization process. The specialization process selects (reuse) artifacts from the ASPL platform and customizes them according to requirements of a specific application domain.

Step 3: Integrate Managing System and Managed System Platforms The third step integrates each managing system platform, derived in the second step, with a managed system platform for a corresponding application domain. A managed system platform provides application domain specific artifacts for managed subsystems, i.e., application logic. The development of managed system platforms is not a focus of the ASPL strategy. The managed system platforms are developed independently of the managing system platforms by following some domain engineering method such as the software product line engineering framework [33]. The ASPL
strategy takes an independently developed managed system platform and integrates it with a corresponding managing system platform. The integration is needed to make the two platforms compatible and complete a product line of self-adaptive systems.

The ASPL strategy holds potential benefits of development with reuse; however, it requires systematic process support to realize the ASPL strategy. For instance, how a horizontal ASPL platform is defined, specialized and integrated with a vertical managed system platform? What challenges are involved and how these challenges can be addressed? In our work to develop process support for the ASPL, we found that the development for and with reuse introduces additional uncertainties. To understand the factors and root causes of the uncertainty, we performed an uncertainty analysis described below.

1.3 Uncertainty Analysis

Uncertainty is an inherent property in complex systems with effects on all system development activities. Walker et al. [38] define uncertainty as “*any deviation from the unachievable ideal of complete determinism*”, that is, it refers to things which are not or imprecisely known at a specific point in time [29]. Many factors cause uncertainty, for instance, lack of knowledge, changes in user needs, market shifts, humans in the loop [29].

Garlan [21], and Esfahani and Malek [17] argued that uncertainty had been treated as a second-order concern in software engineering. However, complexity and challenges raised by uncertainty require that it should be addressed as a first-order concern. A software system developed without considering and taking care of uncertainty is more likely to suffer from risks such as technical failures, degradations, cost and schedule deviations, market and need shifts. To avoid such risks in our work to define process support for the ASPL, we analyzed uncertainty in the context of self-adaptive software systems development with reuse. We used Ishikawa fishbone diagram [25] for the analysis and identified *runtime variability* and *development for reuse* as two principal sources of uncertainty. Each of the two sources is described below.

1.3.1 Uncertainties due to Runtime Variability

Runtime variability refers to changes that occur in a system’s requirements, environment, interconnected systems and the system itself. Traditional software engineering is based on the assumption that all requirements and environmental conditions are fixed, and all changes are managed off-line in software maintenance activities [21]. However, this assumption does not hold for today’s systems that are mobile, more dynamic, interconnected and highly-customizable. The emergence of mobile computing, the internet of things, and cyber-physical systems leads to more frequent runtime variations in systems’ requirements, environments, market forces and systems themselves [14].
Runtime variability is a principal factor behind uncertainties in the development of self-adaptive systems. We analyzed the sources of uncertainty in self-adaptive systems enumerated by Esfahani and Malek [17], and found that the principal factor behind almost all the sources is runtime variability. The first source, for instance, refers to uncertainties caused by inaccuracy of the models representing managed systems. However, the cause for such inaccuracies is runtime variability which invalidates assumptions underlying the models.

The runtime variability has its roots in several areas of concern, including 1) functional and non-functional requirements 2) operating environments or context, 3) interconnected cyber-physical systems, and 4) market forces. All these areas of concerns are shown as sub-branches of the runtime variability in Figure 1.4. The knowledge about runtime variations in these areas of concern is either not available or available only partially at design time. Due to this lack of knowledge, system developers are less able to specify requirements and model design decisions [29]. For instance, runtime variations in a system’s operating environment cannot be predicted or known entirely and precisely at design time. Even if predicted, there are no guarantees whether the predicted variations will come true at runtime or not? For instance, in a znn.com exemplar [13], one may predict and specify that the load-balancer and server pool components may fail at runtime. But, in practice, none of the predicted components may fail; instead some other element, for instance, a communication link between load-balancer and clients may suffer from failure. Moreover, even for predicted variations there remain uncertainties about when a variant will come true and how may it impact the system. Such uncertainties caused by runtime variations challenge developers in each phase of development. In this study, we focus on uncertainties in requirements and design phases only.

Software development usually begins with requirements engineering in which software engineers are required to identify and specify functional and non-functional requirements. Identifying and specifying requirements completely
and precisely is a known research problem [12]. Requirements engineering is traditionally a design time activity. The knowledge about runtime variations is often not available at design time. The lack of knowledge about runtime variations complicates the requirements identification and specification even more.

Requirements engineering is usually followed by software design. Software design maps requirements to architectural elements. Mapping requirements to architectural elements is a decision making process. Requirements, either functional or non-functional, can be realized through different design alternatives. For instance, performance quality attribute can be satisfied by making design decisions based on increasing computational efficiency, removing computational overheads, adding new resources, introducing concurrency architectural tactics for performance [6]. To make best design decisions, designers are required to analyze and reason about several design alternatives [5].

Uncertainties induced by the runtime variability complicate the design space analyzed by designers to make design decisions. For systems with runtime variations, designers are required to identify variants for each design alternative. Moreover, due to no or incomplete knowledge of runtime variations, designers are less able to perceive design choices, reason about the options, make decisions and model the decisions in the form of software architecture. Designing an architecture for the znn.com exemplar [13], for instance, presents designers with an intricate design space due to runtime variations in news requests, server pool, and news content elements.

Uncertainties caused by runtime variations can be mitigated by delaying design decisions until runtime [37]. The self-adaptive system architecture is based on the delay design decisions strategy. Based on the delay design decisions strategy, we would like to investigate delaying requirement specifications strategy. The basic idea of delaying requirements specifications is to push the specifications to the point, for instance, deployment or runtime, where complete or more information about a system’s requirements is available. The requirements specification can be delayed, for example, by defining variation points with several variants, which may get bonded or unbounded according to runtime variations. The variation points and variants can be embedded inside requirement specifications or defined as a separate orthogonal variability model [33] linked with traditional software requirements specifications.

1.3.2 Uncertainties due to Development for Reuse

To support development with reuse, software developers are required to plan and design artifacts, such as requirements and design artifacts, for reuse. While designing artifacts for reuse in several applications or application domains, knowledge about target applications or application domains is either not available or available only partially. This lack of knowledge leads to uncertainties in the design and development of reusable artifacts. To analyze uncertainties in the development of the self-adaptive systems, we distinguish reuse at two levels: 1) horizontal reuse, and 2) vertical reuse.
Uncertainties due to Horizontal Reuse

Horizontal reuse refers to reuse across several application domains [34]. It is supported by establishing a horizontal platform. The horizontal platform serves as a collection of generic, i.e., domain-independent, artifacts that can be specialized for reuse in more than one application domains. While developing a horizontal platform, knowledge about target application domains is not available completely and precisely. The lack of knowledge about target application domains, their requirements, environments, stakeholders, and related concerns leads to uncertainties.

The separation of managing and managed systems, as shown in Figure 1.1, implies that a horizontal platform for managing subsystems can be established and reused to derive several vertical, i.e., application domain specific, managing system platforms. The ASPL strategy, described in Section 1.2, directs to create a horizontal managing system platform, and reuse the horizontal platform to derive several vertical managing system platforms. The objective of the horizontal managing system platform is to support reuse across several application domains of self-adaptive systems.

To support reuse across several domains, developers of the horizontal platform are provided with generic, application domain independent requirements. At this level, knowledge about applications and application domains that are derived from the horizontal platform is either not available or available only partially. The developers are often uncertain about what will be target applications, who will be the applications’ end-users, where the applications will be deployed and executed, what will be the interactions with other systems, what variations, both traditional and runtime, will be there, and how will the applications respond to the variations. All these unknown factors about the target application domains lead to uncertainties in the development of the horizontal platform.

Uncertainties due to Vertical Reuse

Vertical reuse refers to reuse within a single application domain [34]. At this level, reusable artifacts are designed and developed for a single known application domain. The conceptual architecture depicted in Figure 1.1 splits a self-adaptive software system into a managed and a managing subsystem. From reuse perspective, the separation of concerns implies that the managed and managing systems coexist and to a large degree are independent of each other. The ASPL strategy exploits this separation of managed and managing subsystems and recommends to develop the two subsystems by establishing separate platforms.

The separate development of managed and managing system platforms presents developers with uncertainties in areas that cross boundaries between the two platforms. These uncertainties are mainly due to lack of knowledge about the monitor and adapt interfaces between managed and managing systems, and the monitor interface between managing systems and environment. While de-
veloping a vertical platform for a managing system domain, knowledge about the corresponding managed system platform artifacts and their monitor adapt interfaces may not be available or available only partially. This lack of knowledge troubles designers ability to analyze and reason about design alternatives and make informed decisions. Moreover, self-adaptive systems in an application domain may differ in their requirements for self-adaptation. For instance, one system may require “introduce concurrency” tactic while another system, in the same domain, may require “reduce computational overhead” tactic for self-optimization. Such variations among systems within an application domain are referred as domain variability [33]. Variability, being a known factor to trigger uncertainty, challenges architects and designers’ ability to collect complete knowledge about all the variants and make well-informed decisions. Thus, developing managed and managing systems as two separate domains challenge developers ability to reason about managed and managing subsystems domain variability and interfaces between the managed and managing subsystems.

The above-described analysis helped us to understand uncertainties involved in the ASPL strategy. To address these uncertainties and provide developers with step-wise process support to implement the ASPL strategy, we defined an Autonomic Software Product Lines engineering (ASPLe) methodology. The ASPLe methodology is introduced in chapter 2 along with three example application domains. The application domains are used as running examples to demonstrate the ASPLe processes. The ASPLe is composed of three processes: 1) ASPL Domain Engineering, 2) Specialization, and 3) Integration. Each of the three processes is described and demonstrated in chapters 3, 4, and 5, respectively.
Chapter 2

The ASPLe Methodology

The ASPLe is a domain engineering based methodology to design and develop self-adaptive software systems with reuse across several application domains. It complements the ASPL strategy by providing documented and repeatable process support. Figure 2.1 depicts basic structure of the ASPLe. In line with the ASPL strategy, the ASPLe is composed of three principal processes:

1. ASPL Domain Engineering Process
2. Specialization Process
3. Integration Process

The three process correspond to three steps of the ASPL strategy. The ASPL Domain Engineering process maps to the first step and defines activities, work-products, and roles to establish a horizontal ASPL platform. The Specialization process maps to the second step and provides process support to transform the horizontal ASPL platform into a vertical managing system platform. The integration process maps to the third step and integrates manag-

Figure 2.1: Basic Structure of the ASPLe Methodology
ing system platform(s) with a corresponding managed system platform(s). An overview of the ASPLe processes is given below.

2.1 ASPL Domain Engineering Process

The ASPL Domain Engineering (ADE) process defines roles, activities, and work-products to establish a horizontal ASPL platform. The ASPL platform provides reusable artifacts for the development of managing systems. As shown in Figure 2.2, the ADE process structure is derived from domain engineering process of the Software Product Line Engineering framework [33]. The ADE is composed of four subprocesses. It begins with ASPL requirements engineering. The ASPL requirements engineering scopes the ASPL platform and specifies application domain independent requirements for self-adaptation. The requirements are then mapped to a reference architecture by the ASPL design subprocess. The reference architecture models a high-level architecture that can be specialized for reuse in a number of application domains. To support reuse across multiple domains, the reference architecture specifies variation points and a set of variants. The ASPL implementation subprocess realizes these variation points and variants by writing reusable code components such as libraries, classes, etc. The ASPL domain engineering ends with ASPL testing subprocess. The ASPL testing defines activities to validate and verify the reusable code components produced by the ASPL implementation.

It is important to note that the ADE is a continuous process and works incrementally. The scope of the ADE may initially be limited to one or two self-adaptation properties, for instance, self-healing, self-optimization. However, the scope can be extended to support a wide range of self-adaptation properties. See chapter 3 for further details about the ADE process.

![Figure 2.2: The ASPLe Processes](image-url)
2.2 Specialization Process

The specialization process defines roles, activities, and work-products to transform a horizontal ASPL platform into an application domain specific managing system platform. In line with the ADE, the specialization process is composed of requirements, design, implementation, and tests specialization subprocesses. Each subprocess searches the ASPL platform for reusable artifacts, and if found, customizes found artifacts according to needs of a given application domain. For instance, requirements specialization subprocess searches the ASPL platform to find requirement engineering artifacts that match to requirements of a given application domain. The found requirement specification artifacts are then customized, if needed, according to requirements of the given application domain. New requirements specifications are defined from scratch if a specialization process fails to find reusable requirement specification artifacts from the ASPL platform. The newly created artifacts are sent as feedback to the ASPL platform to support reuse in other application domains. See chapter 4 for further details about the specialization process, its subprocesses, and underlying activities.

2.3 Integration Process

The integration process defines activities, work-products and roles to align and integrate two separately developed managing and managed system platforms for a product line of self-adaptive systems. The managed and managing system platforms are developed separately to preserve the separation of concerns as envisioned by the ASPL strategy. The managing system platform is defined by following the ASPLe specialization process. The managed system platform can be developed using any traditional product line engineering framework such as a framework described in [33]. As the two platforms are defined separately, there may exist mismatches between their artifacts. The mismatches are more likely to be in artifacts and areas that cross boundaries between the managed and managing system platforms, such as the monitor and adapt interfaces. The integration process provides developers with process level support to analyze the two platforms, identify and address information and operation abstraction mismatches in requirements, design, implementation and testing artifacts.

The integration process follows the same structure as the other two ASPLe processes. It begins with requirements integration followed by design, implementation and tests integration subprocesses. Each subprocess aims to ensure that the development artifacts in managed and managing system platform are well aligned with each other, and there are no mismatches between the two platforms. See chapter 5 for details about the integration process, subprocesses, and underlying activities.
2.4 Running Examples

We use three application domains as running examples to demonstrate the ASPLe processes. All the three application domains require self-adaptation properties. Each of the three example domains is described below.

2.4.1 Distributed Game Environment (DGE)

The Distributed Game Environments (DGE) is an educational product line of distributed board games. The products or applications of the DGE are game environments for multi-player board games deployed in a distributed setting. Figure 2.3 depicts high-level view of a DGE product. Each product consists of two subsystems: 1) Operator Center (OC), and 2) Player Environment (PE). The PE represents a client-side used by a human player to play games. A new player can install and start a PE to register with the OC. The OC represents a server-side operated by a human operator to perform administrative tasks such as add, remove, or update games; register or unregister players, etc. Players can find each other and launch a game offered by the OC via the PE. Alternatively, a player can select a soft-bot, a software module, as an opponent to play a game.

![Figure 2.3: Example of a DGE Product](image)

In the initial release of the DGE, updates of the player environments are triggered and controlled by a human operator who uses the OC to push updates towards player environments. Alternatively, a player environment can request OC for an update. All update requests from player environments need to be approved by the human operator, which may take longer than expected time to approve and reply the update requests. To improve upgradability property of the DGE products, management of the DGE has decided to introduce a self-upgradability property for the DGE products. Self-upgradability is an ability...
of a software system to update itself at run-time by itself without requiring the system to restart. The DGE requirements for self-upgradability are as follows:

1. The updates to the PE will be introduced by dropping them to an updates repository, a directory or folder in a file system to store the updates.
2. The OC should get a notification as soon as a new update appears in an update repository.
3. The OC should be capable of analyzing the new updates and pushing them to target PEs. Such updates are called “push” type of updates.
4. When a new update appears, a PE gets an update notification from the OC within 5 to 60 seconds.
5. The PE may accept or ignore push type updates. However, there are some critical updates which cannot be ignored by a PE; such updates are called “critical push” updates. Critical push type updates must be performed within 120 seconds.
6. The PE can view the available updates and request OC for a particular update. Such updates that are requested by a PE are called “pull” type of updates.
7. The OC responds an update request by looking into the update repository, and providing it with the requested update if found. The response time for update requests may take 5 to 60 seconds.

The DGE domain is currently composed of four products, P1, P2, P3, and P4. The products differ in their requirements for the push, critical push, and pull type updates. The product P1 requires only push type updates, which can be either accepted or postponed by PEs. The product P2 needs both push and critical push type updates; the critical push type updates must be performed immediately, i.e., PEs can not postpone such updates. The product P4 requires only pull type update; the pull type updates are those which are requested by a PE. The product P3 wants support for all three (push, critical push, and pull) types of updates.

Figure 2.4: The NSPL Feature Model
2.4.2 News Service Product Line

The News Service Product Line (NSPL) is a product line of software applications designed for news organizations. A news organization collects, writes and sells news to those who have subscribed for news. The subscribers or end-users are newspapers, magazines, radio, and television broadcasters, government agencies and other users who are interested in news. Figure 2.4 depicts a high-level feature model of the NSPL. The mandatory features are part of all the NSPL products, whereas the optional features are part of few but not all products. The mandatory format feature refers to text and multimedia formats for the news. The mandatory language feature refers to language in which news are delivered to subscribers. It has three variants: English, Chinese and Spanish. The mandatory server-pool feature represents a collection of servers required to process and deliver the news. The mandatory topic feature refers to the news theme or subject area. It has three variants: politics, sports, and entertainment.

<table>
<thead>
<tr>
<th>Products</th>
<th>Optional Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Chinese, Sports</td>
</tr>
<tr>
<td>P2</td>
<td>Spanish, Entertainment</td>
</tr>
<tr>
<td>P3</td>
<td>Chinese, Spanish, Sports, Entertainment and Server-Pool</td>
</tr>
</tbody>
</table>

Table 2.1: The NSPL Products - Optional Features

At present, the NSPL consist of three products P1, P2, and P3. The mandatory features are part of all the three products. The optional features for the NSPL products are listed in Table 2.1. In addition to these features, the NSPL management requires two self-adaption properties, 1) self-healing and 2) self-optimization [27]. Self-healing is an ability of a software system to detect and recover from failures by itself. Self-optimization is an ability of a software system to monitor and optimize its performance by itself. Performance of the NSPL products is measured in terms of processing time taken by a product’s server-pool to collect and distribute a news item to the subscribed clients. The NSPL product-specific requirements for self-optimization and self-healing are listed below:

Self-Optimization Requirements  All the three products require to monitor server-pool’s processing time, and if the time exceeds a certain threshold, the products should self-optimize themselves as specified below.

1. Processing time threshold for product P1 is 90 seconds. If the time exceeds the threshold, the product requires to self-optimizing its performance, i.e., processing time, by adding servers to its server pool. A constraint for adding new servers is that the operating cost of the product should not exceed 2 million US dollars.

2. Processing time threshold for P2 is 180 seconds. If the time exceeds the
threshold, the P2 requires to self-optimizing its performance, i.e., processing time, by excluding multimedia contents, i.e., restricting news to text format only.

3. Processing time threshold for P3 is 60 seconds. If the time exceeds the threshold, the P3 requires to self-optimizing its performance, i.e., processing time, by first adding servers to its server pool. If the cost of adding serves exceeds 1 million US dollars, the P3 resorts to optimize its performance by excluding multimedia content and delivering news in text format only.

Self-Healing Requirements

1. Product P1 requires self-healing for its language feature. It should be able to detect and recover from failures in the language service; for instance, failure in changing current language to another. The product should not take more than 90 seconds to detect and recover from failures. Moreover, the product requires checkpoint/rollback tactic [6] for failure recovery.

2. Product P2 requires self-healing property for servers in the server-pool. If any of the servers fails, the product should be able to detect and recover from the failures. Heartbeat tactic [6] to monitor server-failures, and failures detection and recovery should not take more than 180 seconds. The failed servers should be replaced with their standby spare replicas.

3. Product P3 requires self-healing property for its language feature and servers in the server-pool. It should not take more than 240 seconds to detect and recover from any of the failures. The failure in language feature should be recovered using the checkpoint/rollback tactic, while server failures should be addressed by replacing failed servers with their standby spare replicas.

2.4.3 PhotoShare Product Line

The PhotoShare Software Product Line (PSPL) is composed of web applications that allow end-users to upload, edit and share their pictures with friends and
family through web-browsers such as firefox, safari, chrome, etc. Figure 2.5 depicts a high-level feature model of the PSPL. The mandatory features are part of all the PSPL products, whereas the optional features are part of only selected products. The mandatory “uploading” feature allows products to either upload pictures from a file system or import pictures from a partner application such as Facebook, Instagram. By default all pictures are in private mode, no one other than the owner (user) can view the pictures. The mandatory “sharing” feature enables users to public their pictures or to share with friends only. The optional “editing” feature enables users to remove or edit their pictures. At now, the PSPL consist of only two products, P1 and P2. The mandatory features are part of both products, while the optional features are listed in Table 2.2. In addition to these features, the PSPL management requires two self-adaption properties, 1) self-healing and 2) self-upgradability. The PSPL requirements for self-healing and self-upgradability are as follows:

### Self-Healing Requirements

1. Product P1 requires self-healing property for its “upload” feature. The product should be able to detect failure of the upload feature within 40 seconds. The ping/echo tactic [6] should be used for fault detection. The failure recovery should not take more than 100 seconds, and the product should use checkpoint/rollback tactic [6] to recover from the failures.

2. Product P2 requires self-healing property for its “share” feature. The product should be able to detect failure of the share feature within 60 seconds. The heartbeat tactic [6] should be used for fault detection. The failure recovery should not take more than 120 seconds, and the product should use standby spare tactic [6] to recover from the failures.

### Self-Upgradability Requirements

1. New updates for the PSPL products should be introduced through a directory or folder in a file system, which serves as a repository to store the updates.

2. All updates should be managed and controlled through a central updates management component.

3. Product P1 requires new updates to be detected and notified to it within 60 seconds. The P1 can either ignore or download and execute the notified updates.

<table>
<thead>
<tr>
<th>Products</th>
<th>Optional Features (Services)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Import</td>
</tr>
<tr>
<td>P2</td>
<td>Share with Friends, Editing</td>
</tr>
</tbody>
</table>

Table 2.2: The PSPL Products – Optional Features
4. Product P2 requires no notifications for new updates. Instead, it requires a feature that allows end-users to view available updates and request one or more updates. The updates management component is required to process update requests and respond within 120 seconds.
Chapter 3

ASPL Domain Engineering (ADE)

The ASPL Domain Engineering process provides the basis for horizontal reuse across several application domains of self-adaptive systems. An overview of the ADE process has been presented already in Section 2.1. This chapter provides a detailed description of the process and its subprocesses. To describe and document the ADE and all other ASPLe processes, we follow process modeling concepts and notations from the Software Process Engineering Metamodel (SPEM) [31]. The SPEM is process engineering metamodel that provides necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes.

3.1 Introduction

The ADE deals with adaptation logic, i.e., managing system part of a self-adaptive system. It provides developers with process level support, i.e., roles, work-products, and activities, to establish a horizontal ASPL platform. The ASPL platform does not target any specific application domain. Instead, it defines high-level development artifacts that can be configured or specialized for reuse in several application domains of self-adaptive systems.

As shown in Figure 3.1, ASPL domain engineering is composed of four subprocesses. It begins with ASPL requirements engineering that defines in-scope self-adaptation properties and application domain independent requirements for these properties. We use the term in-scope properties to refer to the self-adaptation properties supported by the ASPL platform. The requirements engineering is followed by the ASPL design that defines a reference architecture. The reference architecture models a common, high-level structure to realize requirements specified by the ASPL requirements engineering. Next, the ASPL implementation process implements a set of reusable code components or libraries to accomplish adaptation logic for the in-scope self-adaptation properties. The
ASPL implementation does not aim at developing a running application. Instead, it results in a set of loosely coupled, configurable code components, where each component is planned, designed, and realized for reuse in several managing systems of different application domains. The ASPL testing defines a set of test artifacts to validate and verify the domain components produced by the ASPL implementation.

As highlighted in Figure 2.2, the ADE and other ASPL processes, at present, focus on the requirements and design subprocesses. To illustrate the requirements and design subprocess, we use Distributed Game Environment as a running example application domain. The implementation and testing subprocesses are essential parts of the ASPL and are planned as future work.

### 3.2 ASPL Requirements Engineering Process

The ASPL requirements engineering is responsible for identifying and specifying application domain independent requirements for managing systems. Its main objective is to contribute the horizontal ASPL platform with artifacts that can be reused to specify requirements of managing systems in several product lines of self-adaptive systems.

The domain for the ASPL requirements engineering consists of self-adaptation properties determined through ASPL scope definition. To define application domain independent requirements for in-scope self-adaptation properties, the ASPL methodology recommends the use of architectural tactics. The tactics encapsulate application domain independent and commonly used design solutions to realize quality attributes [6], and behind each solution, there is a problem (requirement). Self-adaptation properties are self-oriented forms of quality attributes. Hence, tactics can be used to identify application domain independent requirements.

Figure 3.2 depicts the ASPL requirements engineering process package diagram. The process package diagram specifies three key elements: roles, work-products, and process workflow. Each of these elements is described below.
**ASPL Requirements Engineering**

- **+ Identify in-scope self-management properties**
- **+ Define a general dQAS for each in-scope self-management property**

**Domain Analyst**

**ASPL Scope**

**General dQASs**

**Workflow**

- **<<uses>>**
- **<<uses>>**

Figure 3.2: ASPL Requirements Engineering Process Package

**Roles** Only one role, a Domain Analyst, is required to perform the ASPL requirements engineering process. A main responsibility of the domain analyst is to define application domain independent requirement specification artifacts that can be specialized for reuse in a number of product lines of self-adaptive systems. The domain analyst fulfills this responsibility by defining a general dQAS [4], a requirements specification work-product, for each in-scope self-adaptation property. For instance, if there are three in-scope self-adaptation properties, the domain analyst defines three general dQASs, one for each property.

**Work-products** The following two work-products are consumed in or produced out as a result of the ASPL requirements engineering.

1. ASPL Scope: The ASPL scope defines boundaries for the ADE and resulting ASPL platform. It specifies in-scope self-adaptation properties and tactics the ASPL platform should support.

2. General dQAS: The general dQAS is an application domain independent form of a dQAS [4]. It functions as a template to specify domain independent requirements for self-adaptation properties. See section A.1 in appendix A for details about the dQAS and general dQAS.

**Process Workflow** The ASPL requirements engineering process workflow consists of two activities: ① “Scoping”, and ② “Define General dQASs”. Activity-wise description of the workflow is as follows:

**Activity 1 - Scoping** The workflow begins with a scoping activity where a domain analyst defines the ASPL scope. As stated above, the ASPL scope definition specifies self-adaptation properties and tactics supported by the ASPL platform. Defining boundaries is critical to the success of any project. For the scope definition, the domain analyst consults an extended Architectural Reasoning framework (eARF), described in appendix A. The eARF encapsulates proven best architectural practices and knowledge in the form of tactics and patterns. The tactics are used to foresee and specify application domain independent requirements for the in-scope self-adaptation properties.

26
Activity 2 - Define General dQASs This activity defines a general dQAS for each in-scope self-adaptation property. The general dQAS is a variant of a domain Quality Attribute Scenario (dQAS) [4]. Table A.2 in appendix A lists and describes the dQAS elements. The definition of a general dQAS starts from the “source” element and continues until the “variants” element. The “valid QAS configurations” and “fragment constraints” dQAS elements are left undefined. This is because definition of both these elements depends on a specific application domain which is not known until the ASPL Specialization process. The domain analyst uses tactics (from the ASPL scope definition) to identify fragments (parts) of a general dQAS elements.

3.3 ASPL Requirements Engineering – Demonstration

We develop an example ASPL platform to demonstrate the ASPL requirement engineering and the ASPL design processes. The example ASPL platform initially supports only two self-adaptation properties, self-upgradability and self-optimization.
The first activity in the ASPL RE is to define a scope of the ASPL platform. We used feature modeling to scope the example ASPL platform. The feature modeling [26] is a widely used approach in product line community to scope a product line. Figure 3.3 depicts the ASPL scope definition in the form of two feature trees, one for self-upgradability and the other for self-optimization. The feature trees model mandatory and optional (“alternative” and “or”) features supported by the example ASPL platform. The features represent distinct characteristics of a property and are derived based on tactics [6]. For instance, update detection and update delivery features group a set of distinguished system characteristics to detect and deliver updates. These features are derived based on various tactics for self-upgradability [1]. Similarly, resource demand and resource management features for the self-optimization property are derived based on commonly used tactics to satisfy software systems’ requirements for performance quality attribute.

After scoping the ASPL platform, we continued with activity 2 of the ASPL requirements engineering. In this activity, we defined a general dQAS for each in-scope self-adaptation property. The lack of knowledge about target applications and application domains raised uncertainties while defining the general dQAS elements, for instance, what stimuli conditions may trigger self-upgradability or self-optimization, what may be the sources of the stimuli conditions, and how a system may respond to the stimuli. We mitigated these uncertainties using the architectural knowledge provided by eARF part of the ASPL. We produced a general dQAS for self-upgradability and a general dQAS for self-optimization as a result of the activity 2. Below is a detailed description of how we defined the general dQASs for the in-scope self-adaptation properties, self-upgradability, and self-optimization.

3.3.1 General dQAS for Self-Upgradability

Using the general dQAS template, we started defining the general dQAS for self-upgradability with the source element. The source element specifies sources of stimuli, and to identify the source element; it is vital to identify stimuli conditions first. To identify stimuli conditions that may trigger self-upgradability, we analyzed the in-scope self-upgradability tactics. The analysis helped us to identify two stimuli fragments [ST1] and [ST2]. The fragment [ST1] was identified from “polling” and “on-demand” update detection tactics, whereas fragment [ST2] was identified from “pull” type update delivery tactic. We could not find any stimulus fragment from the update introductions tactics.

Once stimulus fragments were identified, we analyzed these fragments to determine the source fragments. By examining the stimulus fragments [ST1], we found the first source fragment, update provider which can be a repository, such as a database or a directory in a file system, a software system, system administrator or a developer who provides updates. The second source fragment, update consumer specified as fragment [SO2], was identified by analyzing the stimuli fragment [ST2].

After defining the source and stimulus, we defined the artifact element. The
“update introduction” tactics helped us to identify abstract artifacts that get stimulated as a result of the stimuli conditions. We identified two artifact fragments [A1] and [A2]. The fragment [A1] abstracts an artifact that gets simulated with the detection of an update and plans actions in response. We abstracted this artifact as an update manager that works as a managing system or system administrator to prepare and perform updates. The fragment [A2] abstracts a target (managed) system on which adaptive actions (updates) are performed by artifact fragment [A1].

<table>
<thead>
<tr>
<th>Source (SO)</th>
<th>[SO1] Update Provider - a repository, software system, system administrator or a developer that provides updates [SO2] Update Consumer - a software system or subsystem, or an end user who requests updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulus (ST)</td>
<td>[ST1] Update provider introduces a new update [ST2] Update consumer requests for an update</td>
</tr>
<tr>
<td>Artifacts (A)</td>
<td>[A1] Update Manager - a managing system, system administrator who gets stimulated by detection of an update and triggers response [A2] Target System - (managed) software system or subsystem on which updates are applied</td>
</tr>
<tr>
<td>Environment (E)</td>
<td>[E1] Runtime under normal operating conditions</td>
</tr>
<tr>
<td>Response (R)</td>
<td>[R1] Update Manager detects an update [R2] Update Manager notifies an update to a target system [R3] Update Manager responds update requests [R4] Update is applied to a target system</td>
</tr>
<tr>
<td>Response Measure (RM)</td>
<td>[RM1] New update is detected within W seconds [RM2] New update is detected and notified within X seconds [RM3] Update manager responds update requests within Y seconds [RM4] Update is applied to a target system within Z minutes</td>
</tr>
<tr>
<td>Variants (V)</td>
<td>[V1] push [V2] push critical [V3] pull</td>
</tr>
<tr>
<td>Valid QAS Configurations (VC)</td>
<td>Valid QAS configurations depend on a specific domain. Thus, we leave them undefined until the general dQAS is specialized for a specific domain.</td>
</tr>
<tr>
<td>Fragment Constraints (FC)</td>
<td>The fragment constraints can’t be defined without knowing valid QAS configurations, so we leave these until this general dQAS is specialized for a specific domain.</td>
</tr>
</tbody>
</table>

Table 3.1: A General dQAS for Self-Upgradability

There was no information in the ASPL scope about operating environment under which support for the self-upgradability property is required. Hence, we assumed that the example ASPL platform supports self-upgradability under normal operating environment, and specified this as an environment fragment [E1]. The normal operating environment is when a system is not overloaded with tasks.

In the response element, we characterized actions taken by the stimulated artifacts, [A1] and [A2], in response to the stimuli fragments. The actions were characterized as fragments, and to identify these fragments, we analyzed the
self-upgradability tactics. Based on this analysis, we identified four response
fragments [R1], [R2], [R3], and [R4]. The fragment [R1] was identified from
“update detection” tactics. It specifies behavioral requirement in response to
update detection. The fragment [R2] was identified from push type “update
delivery” tactics. It specifies a general response requirement where an “update
manager” artifact specified as fragment [A1] responds to the stimuli fragments
[ST1] and [ST2] by sending an update notification to the target managed system
artifact specified as fragment [A2]. The response fragment [R3] was identified
from the “pull” update delivery tactic in association with stimulus fragment
[ST2]. It specifies a general response requirement where an “update manager”
responds requests for updates from target systems or clients that need an up-
date. The fragments [R4] was identified to specify response action for applying
an update to target systems.

To measure realization of the desired self-upgradability property, four re-
sponse measure fragments, [RM1], [RM2], [RM3] and [RM4], were identified
in association with the response fragments. Being an application domain in-
dependent process, we had no specific requirements for the response measure
fragments. Moreover, requirements for response measure fragments may vary
from product to product and an application domain to another. Thus we used
parameters to specify variability for the response measure fragments. The re-
sponse measure fragment [RM1], for instance, [RM1] specified that a new up-
date should be detected within $W$ seconds, here $W$ is a parameter used to define
variability.

For the variants element, we identified three variants for self-upgradability:
1) Push, 2) Push Critical, and 3) Pull. These variants were identified from the
“update delivery” tactics. The push type self-upgradability variant abstracts a
scenario where an update is pushed from an update provider, artifact fragment
[A1], such as update repository, update manager or any other source, towards
update consumer, artifact fragment [A2]. The update consumer may accept or
postpone the pushed update. The push critical variant extends the push variants
such that the pushed update must be performed, i.e., the update consumer may
not delay or postpone it. The pull type variant specifies a scenario where an
update is requested (pulled) by an update consumer or managed system itself.
The three self-upgradability variants were specified as fragments [V1], [V2] and
[V3], respectively, in the variants element.

The definition of “valid QAS configurations” and “fragment constraints”
dQAS elements depends on specific application domain requirements. Thus,
we left these elements undefined. This brought us to completion of the ASPL
requirements engineering process which comes to an end with the definition of
a general dQAS for each in-scope self-adaptation property.

### 3.3.2 General dQAS for Self-Optimization

Table 3.2 displays a general dQAS template for a self-optimization property.
The template was defined by applying the ASPL requirement engineering pro-
cess. A general (independent of any application domain) requirement for self-
<table>
<thead>
<tr>
<th>Source (SO)</th>
<th>[SO1] Managed System’s response time</th>
<th>[SO2] Managed System’s workload, i.e., number of events or requests to respond</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulus (ST)</td>
<td>[ST1] Response Monitor detects increase in response time, i.e., response time exceeds specified threshold of X time units</td>
<td>[ST2] Workload Monitor detects increase in the workload, i.e., number of requests, events or tasks exceeds specified threshold of Y workload units</td>
</tr>
<tr>
<td>Artifacts (A)</td>
<td>[A1] Response Monitor - part of Managing System which monitors response time and notify Response Manager for increase in the response time</td>
<td>[A2] Workload Monitor - part of Managing System which monitors workload and notify Response Manager for increase in the workload</td>
</tr>
<tr>
<td></td>
<td>[A3] Performance Manager - a managing system which monitors and optimizes performance of a managed system</td>
<td>[A4] Target System - a managed software system or resource which is adapted for performance optimization</td>
</tr>
<tr>
<td>Environment (E)</td>
<td>[E1] Runtime under normal operating conditions</td>
<td>[E2] Runtime under overloaded operating conditions</td>
</tr>
<tr>
<td>Response (R)</td>
<td>Managing system optimizes performance by:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[R1] Increasing computation efficiency of the managed system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[R2] Reducing computational overhead of the managed system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[R3] Introduce concurrency to the managed system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[R4] Add more resources to the managed system</td>
<td></td>
</tr>
<tr>
<td>Response Measure (RM)</td>
<td>[RM1] Managed system’s response time is less than X time units</td>
<td>[RM2] The cost for adaptive actions is under the allowed budget of Z monetary units</td>
</tr>
<tr>
<td>Variants (V)</td>
<td>[V1] Optimize computation efficiency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[V2] Reduce computational overhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[V3] Concurrent processing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[V4] Add more resources</td>
<td></td>
</tr>
<tr>
<td>Valid QAS Configurations (VC)</td>
<td>Valid QAS configurations depend on a specific domain. Thus, we leave them undefined until the general dQAS is specialized for a specific domain</td>
<td></td>
</tr>
<tr>
<td>Fragment Constraints (FC)</td>
<td>The fragment constraints can’t be defined without knowing valid QAS configurations, so we leave these until the QAS configurations are defined</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: A General dQAS for Self-Optimization

31
optimization is to monitor the performance of a managed system and optimize the performance if it does not meet required criterion or threshold. Performance is a measure of responding system requests or events within specified time threshold [6]. The self-optimization property gets triggered when a managing system detects increase (or delay) in response time or workload (number of requests or events to process) of a managed system. Thus managed system’s response time and workload were identified as two source fragments, [SO1] and [SO2], and the conditions or stimulus generated by these two sources were specified as stimulus fragments [ST1] and [ST2], respectively.

For the artifact element, we identified two fragments, [A1] and [A2]. The fragment [A1] abstracts a managing subsystem that gets stimulated by the two stimuli fragments, and trigger adaptive actions in response to optimize the performance of a managed system. The fragment [A2] abstracts a managed system on which the managing system performs the triggered adaptive actions.

For the environment element, we identified two environment variants, fragments [E1] and [E2], under which the example ASPLe framework supports self-optimization. The fragment [E1] specifies run-time operating environment with a normal workload, whereas the fragment [E2] specifies run-time working environment with a high workload. The definition of normal and high workload may vary among different domains or domain products and is defined more precisely while specializing a general scenario to a domain or product specific scenario.

The performance tactics [6] outline number of actions that can be taken by a (managing) system for performance optimization. We use these tactics to identify fragments, i.e., variants, of the response element. The response fragment [R1] is derived from “increase computational efficiency” tactic. It specifies a requirement where a system is required to optimize its performance by optimizing its computational efficiency, for instance by improving the algorithms and other computational resources that are used in performance critical areas. The fragment [R2] is derived from “reduce computational overhead” tactic, and specifies a requirement variant of performance optimization by removing computational overheads, for instance by removing computations that can be avoided, or switching to best fit data structures. The fragment [R3] is derived from “introduce concurrency” tactic, and specifies a requirement where a system is required to optimize its performance through parallel processing, for instance using multiple threads. The last fragment [R4] is derived from the “increase available resources” tactic. It specifies a variant of performance optimization by adding faster additional processors, memories, communication channels and other resources.

To measure realization of the self-optimization property, we identified two response measure fragments. The fragment [RM1] requires that as a result of the adaptive actions performed by the stimulated managing system artifact, the managed system’s response time should become less than \( x \) time units, here \( x \) is a parameter used to define domain or product specific threshold for the response time. The fragment [RM2] specifies that the cost of the adaptive actions (those specified as response variants) taken for performance optimization should be less than \( y \) currency units, here \( y \) is a parameter used to define domain or product specific threshold for the cost of the adaptive actions.
3.4 ASPL Design Process

The ASPL design (sub)process defines a general reference architecture to realize self-adaptation properties. Our definition of the general reference architecture is an application domain-independent architecture that can be specialized to derive several application domains specific reference architectures. Thus, the main objective of the ASPL design process is to define architectural artifacts that can be specialized for reuse in several domains of self-adaptive systems. It uses general dQASs defined by the ASPL requirements engineering to identify requirements, analyze and reason about design alternatives, and model design decisions to realize the requirements. The design decisions are modeled as architectural elements with abstract responsibilities and interface definitions so that these elements can be specialized (through ASPL design specialization) to meet requirements of a number of application domains.

Figure 3.4 depicts a process package diagram of the ASPL design process. The package diagram depicts roles, work-products, and workflow of the ASPL design process. An overview of the roles, work-products is given below, followed by a description of the process workflow.
Roles Only one role, a Domain Designer, is needed to perform the ASPL design process. The domain designer is required to identify design options, reason about them and map the evaluated or verified design options to architectural elements. The ASPL methodology provides domain designers with proven best design practices and knowledge encapsulated in the form of extended Architectural Reasoning Framework (eARF). Details about the eARF, its elements, and usage in the design process are given in appendix A.

Work-products Three work-products, described below, are used in the process.

1. General dQAS: The domain designer uses general dQASs, defined by the ASPL requirements engineering, as an input work-product to identify required self-adaptation properties and their variants. See section A.1 in appendix A for details about the dQAS and general dQAS.

2. eARF: The eARF is used to provide domain designers with proven best design practices and knowledge to support architectural analysis and reasoning for self-adaptation properties. It is a purposefully established reasoning framework [15] to support architectural analysis and reasoning required for realization of self-adaptation properties. It helps designers to identify design alternatives, analyze and evaluate the identified alternatives, reason about the outcomes, and model design decisions.

3. General dRS: For each required self-adaptation property, a General dRS is defined as an output work-product of the ASPL design process. The General dRS is an application domain independent type of a dRS. The dRS is an architectural representation of design decisions made to realize a self-adaptation property. A general dRS models a reference architecture to realize a self-adaptation property. Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) feedback loop [27, 40] is used as a principal architectural pattern to structure the architectural element in a General dRS. See appendix A for details about the dRS and General dRS.

Process Workflow Figure 3.4 depicts workflow of the ASPL design process. The workflow comprises four activities that are performed for each in-scope self-adaptation property required by the ASPL domain. Activity by activity description of the process is as follows:

Activity 1 - Extract responsibilities with variability Beginning with a first self-adaptation property, the domain designer analyzes a general dQAS, and extracts a set of application domain independent responsibilities and their variants. The responsibilities are identified by following responsibility driven design approach [41]. Moreover, the domain designer should follow conceptual split between managing and managed subsystems of a self-adaptive software system to identify responsibilities.
Activity 2 - Identify design options The responsibilities and their variants, extracted as a result of activity 1 can be achieved in several ways. For instance, responsibilities identified for a "self-protection" can be satisfied by following different security tactics, such as user authentication, authorization, maintaining data confidentiality and integrity. Thus, as next step in the ASPL design process, the activity 2 identifies available and foreseeable design choices to model or realize the extracted responsibilities. The domain designers should use self-adaptation property driven architectural tactics (from the eARF) to identify design choices.

Activity 3 - Reason about and verify design options The identified design options are then analyzed and reasoned about their outcomes to ensure the realization of the desired self-adaptation properties. The analysis and reasoning can be achieved using any architectural analysis method which provides means to assure that the identified set of design options would comply with requirements specified in a general dQAS. The extended Architectural Reasoning Framework (eARF), described in appendix A encapsulates knowledge needed by domain designers to understand and reason about self-adaptation properties. To provide more rigorous support for architectural analysis, reasoning, and to verify the analyzed design options, we enhanced the eARF with an analytical framework [3]. The analytical framework support architects with transforming requirements to rigorous architecture models that comply with required self-adaptation properties. The models are defined using timed automata, and the properties are specified using temporal logic. The resulting models and properties allow verifying model compliance with the properties. The analytical framework uses Uppaal [9], a state-of-the-art toolbox for verification.

Activity 4 - Map design decisions to responsibility components In this activity, the design decisions are mapped to responsibility components to define a reference architecture in the form of a General dRS. The mapping to architectural elements once again requires domain designers to reason about architectural structure and interfaces among the responsibility components. The self-adaptation property driven architectural patterns from the eARF can be used once again to reason about and organize responsibility components, their structure and provide/required interfaces.

3.5 ASPL Design – Demonstration

This section demonstrates how we performed the ASPL design process for an example ASPL platform. The example ASPL platform supports two self-adaptation properties, self-upgradability and self-optimization, see Section 3.3.
<table>
<thead>
<tr>
<th>dQAS Elements</th>
<th>Responsibilities and Design Choices</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Source</strong></td>
<td>1. Updates Provider: serves as a source for the updates; may have following variants:  &lt;br&gt; i) software (sub)system  &lt;br&gt; ii) system administrators  &lt;br&gt; iii) software developers  &lt;br&gt; 2. Update Consumer: requests for updates with following variants:  &lt;br&gt; i) software (sub)system  &lt;br&gt; ii) end user</td>
</tr>
<tr>
<td><strong>Stimulus</strong></td>
<td>1. Update Provider: (same as in the source element)  &lt;br&gt; 2. Update Consumer: (same as in the source element)</td>
</tr>
<tr>
<td><strong>Artifact</strong></td>
<td>1. Following responsibilities were identified for the Updates Manager Artifact:  &lt;br&gt; a) Monitor Updates: monitor updates provider for new updates  &lt;br&gt; i) Periodic polling  &lt;br&gt; ii) Event-based  &lt;br&gt; iii) On-demand  &lt;br&gt; b) Analyze: analyze new updates  &lt;br&gt; c) Plan: plan delivery and execution of the new updates  &lt;br&gt; i) Push  &lt;br&gt; ii) Push-Critical  &lt;br&gt; d) Execute: execute the updates  &lt;br&gt; i) Quiescence  &lt;br&gt; ii) Rewriting Binary Code  &lt;br&gt; iii) Use of Proxies  &lt;br&gt; iv) Intrusion and Cooperation  &lt;br&gt; 2. The Update Manager can be realized in two different ways:  &lt;br&gt; i) Centralized Control: a central component is responsible for all responsibilities  &lt;br&gt; ii) Distributed Control: responsibilities are distributed among several components  &lt;br&gt; 3. Target System: uses updates i.e., a software system on which updates are performed.</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>Run-time operating environment with normal work-load</td>
</tr>
<tr>
<td><strong>Response</strong></td>
<td>1. Updates Monitor: monitors the source variants for new updates; may have following variants:  &lt;br&gt; i) Periodic polling  &lt;br&gt; ii) Event-based  &lt;br&gt; iii) On-demand  &lt;br&gt; 2. Analyzer: analyzes new updates: Update Manager: notifies an update to the Target System. A notification may trigger two response variants:  &lt;br&gt; i) Target System may accepts and performs the update  &lt;br&gt; ii) Target System may postpone the updates  &lt;br&gt; 2. Target System: requests Update Manager for an update  &lt;br&gt; 3. Target System: downloads an update from the Update Manager and executes it.</td>
</tr>
<tr>
<td><strong>Response Measure</strong></td>
<td>No new responsibility identified in this element.</td>
</tr>
<tr>
<td><strong>Variants</strong></td>
<td>No responsibilities but three variants for how an update is planned and performed are identified in this element: 1) Push, 2) Critical Push, and 3) Pull</td>
</tr>
</tbody>
</table>

Table 3.3: Responsibilities extracted from the General dQAS for Self-Upgradability
for details about the example ASPL platform. Following is a one by one description of how we performed the ASPL design process to model application domain independent design artifacts for the two in-scope self-adaptation properties.

### 3.5.1 ASPL Design for Self-Upgradability

Following the ASPL design process workflow, described in Section 3.4, we started with activity 1. In this activity, we analyzed the general dQAS for self-upgradability, shown in Table 3.1, and extracted a set of responsibilities by following the responsibility driven design approach [41]. We analyzed each element of the general dQAS and extracted responsibilities with their variants. For instance, from the source element, we extracted two responsibilities: 1) updates provider, and 2) updates consumer. The updates provider responsibility abstracts a component or subsystem that serves as a source of the updates. Three variants, as follows, were identified for the updates provider responsibility from the source element: i) software (sub)system, ii) system administrator and iii) software developers. All the responsibilities extracted from the general dQAS for self-upgradability are listed in Table 3.3. We

Next activity in the ASPL design process is to identify design options to realize the extracted responsibilities. We used self-upgradability tactics from the eARF instance shown in Figure 3.5 to identify and reason about the design options. The eARF recommends use of MAPE pattern [40] and the self-adaptation property specific tactics to identify design choices. The appendix A describes the MAPE pattern and commonly used monitor and execute tactics to model design decisions for self-adaptation. We used the monitor, analyze, plan and execute elements from the MAPE pattern, and upgradability tactics described in [1, 30] to identify design options for the self-upgradability property. It is important to note that the ASPLe methodology uses tactics to scope the supported self-adaptation properties, and also to assist architects in identifying
and reasoning about design alternatives. The Table 3.3 presents design options (listed in italics) identified for each responsibility as a result of this activity. From variability modeling perspective, the extracted responsibilities represent variation points, and the design options represent variants.

Continuing the ASPL design process, in activity 3, we evaluated and verified the design options with the help of an analytical framework provided by the eARF. The design choices were reconsidered and adapted based on results of the verification. We completed the ASPL design process by performing the activity 4. In this activity, we mapped the verified design options to architectural components of the form of a General dRS for self-upgradability. Figure 3.6 shows the General dRS for self-upgradability produced as a result of the ASPL design process. The responsibility components depict abstract responsibilities whereas the orthogonal variability model, i.e., variation points and variants, represent design choices and alternatives to realize the modeled responsibilities. The annotation «managed» specifies that a component, e.g., updates consumer, is a part of the managed subsystem, whereas the annotation «managing » specifies that a component, e.g., updates monitor, is a part of the managing subsystem.

The “update provider” and “update consumer” responsibility components were modeled to realize responsibilities identified in the source element of the general dQAS for self-upgradability. The update provider responsibility component abstracts a source entity through which updates are introduced. Three variants of this component were identified as a result of activity 1 of the ASPL design process. We did not have specific requirements for an application domain or application itself. Thus, we modeled the variants as optional variants under the variation point “updates provider”. The “software (sub)system” variant abstracts any software system or subsystem which serves as a pool or a database of updates. In some application domains, the updates might be introduced by system administrators or system developers. This is modeled by defining “system administrator” and “system developer” update provider variants, respectively.

The update consumer responsibility component abstracts an entity, which requests for updates. Depending on application domain requirements, the update consumer can be an end-user or a software system itself. This is modeled as a variation point “updates consumer” with two variants, software system, and end-user.

The “target system” responsibility component represents a (managed) software system or subsystem on which updates are performed. The target system can be same as the update consumer component if the target system requests for an update itself.

The “updates monitor” component is identified from the artifact element variant [A1]. It models a responsibility to detect new updates and notify updates to the updaters manager. Three design choices (variants) were identified and verified for the updates monitor. These design choices were identified using “update detection” tactics from the self-upgradability tactics described in [1]. The verified design choices are modeled as variants of the “repo. monitor” variation point in the OVM.

The “updates manager” component identified from the artifact element, has
Figure 3.6: General dRS for Self-Upgradability
a responsibility of managing system to control and perform updates in a self-adaptive manner. It uses the "update monitor" component to monitor source of the updates, i.e., "update provider" component. The update manager also has a responsibility for handling update requests from update consumers. On getting a new update notification or request, the updates manager coordinates with the "analyzer" and "planner" components to analyze and plan adaptive actions.

The analyzer, planner, and executor components are identified and structured based on the MAPE pattern. The analyzer has two variants (design choices), updates analyzer, and requests analyzer. The requests analyzer is used to analyze requests from update consumers, for instance, whether the requested update is available or not. The update analyzer variant is there to analyze types of the requested or newly detected updates, for instance, whether it is a simple push type or critical push type update. Determining an update type, helps the planner and update manager components to plan appropriate adaptive actions. For instance, critical push type updates can’t be delayed, and the managing system is required to ensure that it has been successfully applied or executed on the target system. The “push” and “critical push” plan variants represent adaptive actions that are required for push and critical push type updates, respectively. The planned actions for self-upgradability are forwarded to the “executor” component which has a responsibility to perform the planned actions (updates) on the target managed system. The four variants for the executor are modeled based on execute tactics for self-upgradability. Details about the execute tactics can be found in Appendix A, Section A.3.3.

### 3.5.2 ASPL Design for Self-Optimization

A General dRS for self-optimization was defined by applying the ASPL design process. Starting with activity 1 of the ASPL design, we analyzed the general dQAS for self-optimization, shown in Table 3.2, and extracted a set of responsibilities. The responsibilities were extracted using the responsibility driven design approach [41]. Each element of the general dQAS was analyzed, and responsibilities were extracted with their variants. For instance, response monitor and workload monitor were identified as responsibilities from the stimulus element. All the responsibilities extracted from the general dQAS for self-optimization are listed in Table 3.4.

Next, in activity 2, we identified design options to realize the extracted responsibilities. We used self-optimization tactics from the eARF instance shown in Figure 3.5 to identify and reason about the design options. The eARF recommends use of MAPE pattern [40] and self-adaptation property specific tactics to identify design options. The appendix A describes the MAPE pattern and commonly used monitor and execute tactics to model design decisions for self-adaptation properties. We used the monitor, analyze, plan and execute elements from the MAPE pattern, and performance tactics described by Bass et al. [6] to identify design options for the self-optimization property. The Table 3.4 records design options (listed in italics) identified for each responsibility as a result of this activity.
Elements | Responsibilities and Design Choices
---|---
Source | A managed system is required with following two properties:
  i) response time
  ii) workload

Stimulus | 1. A response monitor is required to detect increase in response time.
  2. A workload monitor is required to detect an increase in workload

Artifact | 1. A response monitor is required to monitor response time and notify performance manager for increase in the response time
  2. A workload monitor is required to monitor workload and to notify performance manager for an increase in the workload
  3. A performance manager (managing system) is required to optimize the performance of a managed system
  4. The managed system is required to cooperate with managing system for monitoring and optimization

Environment | No responsibility identified

Response | 1. Performance manager analyzes managed system to find out factors causing an increase in the managed system’s response time or workload properties.
  2. Performance manager plans adaptive actions to optimize the managed system’s performance. The actions can be based on following performance optimization tactics [6]:
    i) Increase computation efficiency
    ii) Reduce computational overhead
    iii) Introduce concurrency
    iv) Increase available resources
  3. Performance manager executes the planned adaptive actions on the managed system. The planned actions can be executed using one of the following tactics:
    i) Quiescence
    ii) Rewriting Binary Code
    iii) Use of Proxies
    iv) Intrusion and Cooperation

Response Measure | No new responsibility identified in this element.

Variants | No new responsibility identified in this element

Table 3.4: Responsibilities extracted from the General dQAS for Self-Optimization

Continuing the ASPL design process, in activity 3, we evaluated and verified the design options with the help of an analytical framework provided by the eARF. The design choices were reconsidered and adapted based on results of the verification. We completed the ASPL design process by performing the activity 4. In this activity, we mapped the verified design options to architectural components of in the form of a General dRS for self-optimization. Figure 3.7 depicts the General dRS for self-optimization produced as a result of the ASPL design process. The responsibility components model abstract responsibilities whereas the orthogonal variability model, i.e., variation points and variants, represent design choices and alternatives to realize the modeled responsibilities.

The “managed system” responsibility components models a basic application software system or a subsystem that requires optimizing its performance. It was identified from the source element of the general dQAS for self-optimization. The workload monitor component models a responsibility to monitor the managed system’s workload and to notify the workload to the performance manager. Similarly, the response monitor component models a responsibility to monitor
Performance Manager

Monitors and optimizes performance of a managed system

Analyzer

Analyzes root cause factors, which affect managed system's performance

Notification Analysis Results Planner

Plans adaptive actions to optimize managed system's performance

Managed System

Abstracts a base level system responsible for application logic

Response Monitor

Monitors managed system's response time and notifies the performance manager

Executor

Executes the Plan

Managed System

Abstracts a base level system responsible for application logic

Response Monitor

Monitors managed system's response time and notifies the performance manager

Executor

Executes the Plan

Managed System

Abstracts a base level system responsible for application logic

Key

- [name]
- [required]
- [provided]
- [constraint]
- [artifact]
- [variant]
- [variation point]
- [dependency]

Plan

Managed System

Abstracts a base level system responsible for application logic

Analyzer

Analyzes root cause factors, which affect managed system's performance

Notification Analysis Results Planner

Plans adaptive actions to optimize managed system's performance

Managed System

Abstracts a base level system responsible for application logic

Figure 3.7: General dRS for Self-Optimization
the managed system’s response time and to notify the time to the performance manager. Responsibilities for both the monitor components were extracted from the stimulus element of the general dQAS. The performance manager component models a managing system which is responsible for monitoring and optimizing the performance of a managed system. It is identified from the artifact element of the general dQAS.

The analyzer, planner and executor responsibility components are modeled based on the MAPE pattern and responsibilities identified against the response element of the general dQAS for self-optimization. The analyzer component analyzes root cause factors which affect performance and trigger self-optimization. The performance can be affected by a number of factors such as workload, available resources, bottlenecks, etc. The variability of these factors is modeled in the form of analyzer variation point associated with the analyzer component.

The planner component models responsibilities to plan adaptive actions for performance optimization. The response element of the general dQAS specifies four moves in response to stimuli conditions which may trigger self-optimization. The four moves are modeled as variants of a planner variation point associated with the planner component.

The performance manager executes the planned actions for self-optimization with the help of executor component. The four variants for the executor are modeled based on tactics for execution described in Appendix A, Section A.3.3.
Chapter 4

Specialization Process

The specialization process is used to transforms the horizontal ASPL platform into several vertical platforms to support the development of managing systems across several domains of self-adaptive systems. An overview of the specialization process has been presented in Chapter 2. This chapter provides a detailed description of the process including all the subprocesses, activities and involved entities.

4.1 Introduction

The ASPL platform produced by the ASPL domain engineering process provides generic reusable artifacts. The ASPL platform artifacts are purposefully designed to realize self-adaptation properties across several application domains of self-adaptive systems. These artifacts are defined independent of any application domain and are likely to have gaps between what is needed by an application domain, and what is offered by the ASPL platform. For instance, an application domain may require the use of “introduce concurrency” tactic for self-optimization, but the ASPL platform provides “increase computation efficiency” tactic based artifacts for self-optimization. The ASPL provides the specialization process to address gaps between the ASPL platform and requirements of underlying application domains.

The specialization process defines roles, activities, and work-products required to find (reuse) artifacts from a horizontal ASPL platform and specialize them to establish multiple vertical platforms. Each of the resulting vertical platforms provides application domain specific artifacts for managing subsystems of a self-adaptive system domain. Thus, such platforms are called Managing System Platforms. The derivation of multiple managing system platforms from a horizontal ASPL platform is a perfect example of development with and for reuse. Because, instead of developing from scratch, each managing system platform is defined with reuse from the horizontal ASPL platform, and each managing system platform provides artifacts for reuse in several products of an
The ASPL and application domain scopes are used as starting-points for the specialization process. The application domain scope is used to know what self-adaptation properties are required by an application domain, whereas the ASPL scope definition is used to find whether required self-adaptation properties are
supported by the ASPL platform or not. The specialization process comes to an end with a feedback activity. In the feedback activity, artifacts produced as a result of the specialization process are considered for inclusion in the ASPL platform. For instance, let’s assume that an application domain requires “push critical” variant of self-upgradability which is currently not supported by the ASPL platform. In this case, specialization process is needed to develop artifacts for the push critical variant from scratch, and the resulting artifacts are sent as feedback to the ASPL platform for potential reuse in other domains. Below is a detailed description of the requirements and design specialization processes.

4.2 Requirements Specialization Process

The requirements specialization process provides guidelines to produce application domain specific requirements engineering artifacts with reuse. Instead of defining the requirement specification artifacts from scratch, the requirements specialization process guides requirement engineers to find reusable requirement specification artifacts from the ASPL platform, and specialize the found artifacts for reuse in a specific application domain.

Figure 4.2 depicts a process package diagram of the requirements specialization process. It highlights all the work-products, activities and roles involved in the process. A brief description of the roles, work-products and process workflow is given below.

Roles Only one role, a Domain Analyst, is required to perform the requirements specialization process. For the requirements specialization, the domain analyst is required to have a good understanding of the application domain for which the requirements specialization process is being performed. Moreover, the domain analyst should have good knowledge and understanding of reusable requirements specification artifacts offered by the ASPL platform. All the responsibilities for the domain analyst role
are listed in Figure 4.2, in a box under the role “domain analyst”. These responsibilities map to the activities of the requirements specialization process.

**Work-products** Four work-products, summarized below, are used in the requirements specialization.

1. Managed (Application) Domain Scope: This work-product is used to identify what self-adaptation properties are required by an application domain for which the requirements specialization process is being performed.

2. ASPL Scope: In this process, the ASPL scope definition is used to determine whether the self-adaptation properties required by the managed application domain (for which the specialization process is being performed) are supported by the ASPL platform or not.

3. General dQAS: The General dQAS is an application domain independent, generic and reusable requirement specification artifact provided by the ASPL platform. The General dQAS is purposefully defined by the ASPL requirements engineering process for reuse in several product lines of self-adaptive systems.

4. Specialized dQAS: The Specialized dQAS is an application domain specific requirement specification artifact produced as a result of the requirements specialization process. It is defined by selecting a general dQAS from an ASPL platform and customizing it according to the needs of a specific application domain. See section A.1 in appendix A for details about the General and Specialized dQASs.

**Process Workflow** Figure 4.2 depicts workflow for the requirements specialization process. Activity-wise description of the process workflow is as follows:

**Activity 1 - Scoping** The requirements specialization process begins with a scoping activity that defines a scope of the managed application domain. The activity is an analog to scoping activity in the ASPL requirements engineering. It reuses the ASPL scope and either modifies the feature model, for example removing or adding features, variants, and constraints for an in-scope property, or creates new feature models for unsupported properties.

**Activity 2 - Identify Candidates** After defining the application domain scope, the domain analyst searches for reusable requirement specification artifacts from the ASPL platform. The search is performed for each self-adaptation property required by a given application domain. If a self-adaptation property is supported by the ASPL platform, the search activity will return a general dQAS that can be specialized for reuse in the given application domain. Currently, the search activity is performed manually by looking at all requirements engineering artifacts
in the ASPL platform; however, in future, more advanced search techniques can be added to the ASPL methodology. If the ASPL platform does not support a self-adaptation property, the domain analysts have to define requirements specification artifacts from scratch. The newly defined artifacts with potential for reuse in other domains are considered for inclusion in the ASPL platform as feedback.

Activity 3a - Analyze General dQASs The general dQASs found as a result of the activity 1 are analyzed to investigate what requirements are already specified, what are missing and what needs to be modified or adapted. The general dQASs in the ASPL platform are defined without knowing specific application domain requirements. Thus, it is more likely that there exist gaps between what is required by a specific application domain and what is specified in a general dQAS. For instance, an application domain may require the use of “introduce concurrency” tactic for performance optimization which may not be specified in the general dQAS provided by the ASPL platform. Similarly, a general scenario may have fragments (parts of a dQAS) that are not needed by an application domain. Such gaps are identified in this activity and are addressed in the next activity.

Activity 3b - Specialize the General dQASs This activity specializes general dQASs to application domain specific dQASs called Specialized dQASs. The specialization is performed to address gaps identified as a result of activity 2 and to define the dQAS elements that were left undefined in the general dQASs. The “valid QAS configurations” and “fragment constraints” dQAS elements are left undefined in general dQASs. The definition of these elements requires knowledge of a target application domain which is unknown in the ASPL domain engineering process. This knowledge is, however, available in the specialization process, and the requirements specialization process uses this knowledge to complete the definition of “valid QAS configurations” and “fragment constraints” elements as a part of this activity.

Activity 4 - Define New dQASs If the ASPL platform does not support a self-adaptation property, the domain analysts have to define new domain-specific dQASs from scratch. The process workflow for defining a new application domain specific dQAS is same as the activity 2 in the ASPL requirements engineering process workflow.

Activity 5 - Feedback The newly defined dQASs artifacts, with potential for reuse in other domains, are considered for inclusion in the ASPL platform as feedback.

The requirements specialization process comes to an end by adding the specialized dQASs to an application domain specific Managing System Platform.
4.3 Requirements Specialization – Demonstration

This section demonstrates how a requirements specialization process is performed in practice. We use three application domains described in section 2.4 as running examples to demonstrate the specialization and integration processes. The three example application domains are: 1) Distributed Game Environment (DGE), 2) PhotoShare Software Product Line (PSPL) and NewsService Software Product Line (NSPL). Details about how we performed the requirements specialization process for the example domains are given below.

4.3.1 Requirements Specialization for DGE

We used the requirements specialization process to specify the DGE domain requirements for self-adaptation. Following the requirements specialization process workflow, we (in a role of domain analyst) began with activity 1. In this activity, we reused the ASPL scope definition from example ASPL platform, see Section 3.3, to define the DGE domain scope. Figure 4.3 depicts the DGE domain scope definition. The DGE domain does not require self-optimization; thus, we excluded feature tree for the self-optimization in the DGE scope definition. The DGE requires self-upgradability property, so feature tree for the
| Source (SO) | [SO1] Update Repository is used to introduce new updates  
| [SO2] Players who requests updates for their Player Environments (PEs) |
| Stimulus (ST) | [ST1] A new update appears in an updates repository  
| [ST2] Player requests an update |
| Artifacts (A) | [A1] Operator Center (OC) - an update manager that handles new updates, update requests, and triggers actions to deliver and perform updates  
| [A2] Player Environment (PE) - target managed system on which updates are performed |
| Environment (E) | [E1] Runtime under normal operating conditions |
| Response (R) | [R1] New update is detected and notified to the OC  
| [R2] OC notifies an update to player environments  
| [R3] OC responds update requests  
| [R4] Update is applied to player environments |
| Response Measure (RM) | [RM1] New update is detected and notified to the OC with no delay after it has been placed in the Updates Repository  
| [RM2] OC notifies new updates to target PEs within X seconds, with \(range(X) = [5..60] \)  
| [RM3] OC responds an update request within Y seconds, with \(range(Y) = [5..60] \)  
| [RM4] Update is applied to target PEs within Z minutes, with \(range(Z) = [1..10] \) |
| Variants (V) | [V1] push  
| [V2] push critical  
| [V3] pull |
| Valid QAS Configurations (VC) | [VC1] V1  
| [VC2] V1 \& V2  
| [VC3] V1 \& V2 \& V3  
| [VC4] V3 |
| Fragment Constraints (FC) | [FC1] Mandatory Fragments: { A1, A2 } \& { E1 } \& { R4 } \& { RM4 }  
| [FC2] Configuration Specific Fragments:  
| [Variants VC1] { SO1 } \& { ST1 } \& { R1, R2 } \& { RM1, RM2 }  
| [Variants VC2] { SO1 } \& { ST1 } \& { R1, R2 } \& { RM1, RM2 }  
| [Variants VC3] { SO1, SO2 } \& { ST1, ST2 } \& { R1, R2, R3 } \& { RM1, RM2, RM3 }  
| [Variants VC4] { SO2 } \& { ST2 } \& { R3 } \& { RM3 }  
| [FC3] Bindings:  
| [Bindings VC1] V1.( RM2.bind(X) + RM4.bind(Z) ) | X+Z \leq 10 \text{ minutes}  
| [Bindings VC2] ( V1.( RM2.bind(X) + RM4.bind(Z) ) | X+Z \leq 10 \text{ minutes} )  
| \& ( V2.( RM2.bind(X) + RM4.bind(Z) ) | X+Z \leq 120 \text{ seconds} )  
| [Bindings VC3] ( V1.( RM2.bind(X) + RM4.bind(Z) ) | X+Z \leq 10 \text{ minutes} )  
| \& ( V2.( RM2.bind(X) + RM4.bind(Z) ) | X+Z \leq 120 \text{ seconds} )  
| \& ( V3.( RM3.bind(Y) + RM4.bind(Z) ) | Y+Z \leq 10 \text{ minutes} )  
| [Bindings VC4] V3.( RM3.bind(Y) + RM4.bind(Z) ) | Y+Z \leq 10 \text{ minutes} |

Table 4.1: Self-Upgradability dQAS Specialized for the DGE
self-upgradability was retained and specialized according to needs of the DGE domain. The red “X” marks in the scope definition specify out of scope properties and features.

Next, for activity \( j_2 \), we searched the ASPL platform and found a general dQAS for self-upgradability. The general dQAS was defined without knowing the DGE domain requirements, so we analyzed it in activity \( j_3a \). The analysis was performed to identify gaps between what is required by the DGE and what is specified in the general dQAS. The gaps in the general dQAS were addressed in activity \( j_3b \), and a specialized dQAS for self-upgradability property required by the DGE was produced. The DGE does not require a property, which is not supported by the example ASPL platform. Thus, there was no need to perform the activities \( j_4 \) and \( j_5 \).

The specialized dQAS produced as a result of the requirements specialization for the DGE is shown in Table 4.1. An element by element description of how the specialized dQAS was produced by specializing a general dQAS is as follows.

Beginning with source element of the general dQAS for self-upgradability, we observed that the source fragments, \([SO1]\) and \([SO2]\), satisfy the DGE requirements in general. However, both fragments required specialization to specify the DGE requirements precisely. In the DGE domain, an update repository and a player are the source entities which may trigger self-upgradability. The update repository serves as an update provider, so we specified it as source fragment \([SO1]\). The player may request operator center for updates, so we considered a player as an update consumer and specified this requirement by specializing the source fragment \([SO2]\).

In the DGE domain, self-upgradability is triggered either when a new update appears in an update repository or when a player requests for an update. We specified these requirements as stimulus fragments \([ST1]\) and \([ST2]\).

For the artifact element, Operator Center (OC) and Player Environment (PE) are the two DGE artifacts that get stimulated for the self-upgradability property. Thus, we specialized the artifact fragments \([A1]\) and \([A2]\) to specify OC and PE as stimulated fragments, respectively.

No changes were made in the environment element. This is because of the requirements specified in the environment element of the general dQAS matched with the DGE requirements for the environment.

The response fragment \([R1]\) was specialized to specify the DGE requirement that the OC should get a notification as soon as a new update appears in the updates repository. For all other response fragments, there were no significant changes other than renaming the artifacts that respond to the stimulus conditions. For response fragments \([R2]\), for instance, the “Update Manager” was renamed to “OC”. This specialization was made because, in the DGE application domain, it is the operator center which works as an update manager.

In the general dQAS for self-upgradability, abstract parameters were used to specify variability in Response Measure fragments. We specialized the response measure fragments by defining the parameter values according to the DGE domain requirements. For instance, we specialized the response measure fragment \([RM2]\) by defining a function \( \text{range}(X) = [5..40] \). This function specifies lower
and upper bound values for the parameter $X$.

The DGE domain requires three variants of self-upgradability: (1) push, (2) critical push, and (3) pull. These three variants were already specified as application domain independent variants of self-upgradability in the general dQAS. Thus, no changes were made in the variant element.

The last step in the specialization of a general dQAS is the definition of “valid QAS configurations” and “fragment constraints” elements.

The DGE domain is composed of four products that vary in their requirements for the self-upgradability variants. We used the self-upgradability variants required by the DGE domain to determine the valid QAS configurations. Four valid QAS configurations, \([\text{VC1}], \text{[VC2]}, \text{[VC3]}, \text{and [VC4]},\) one for each DGE product, were defined as fragments of the “valid QAS configuration” element. The fragment \([\text{VC2}],\) for instance, was defined to specify a product configuration that requires the push and push critical variants of self-upgradability.

The “fragment constraints” element was defined to specify constraints for each valid QAS configuration. We distinguished three types of fragment constraints: 1) mandatory fragments, 2) configuration specific fragments and 3) bindings. The mandatory fragments type was defined to specify fragments that are needed by all valid QAS configurations. For instance, the OC and PE artifacts get stimulated for self-upgradability needs of all the DGE products, i.e., valid QAS configuration. Hence, the artifact fragments \([\text{A1}]\) and \([\text{A2}]\) for the OC and PE, respectively, were specified as mandatory fragments along with fragments \([\text{E1}], \text{[R4]}, \text{and [RM4]}\).

The configuration specific fragments type was defined to specify fragments required by only a particular configuration and not by all. As there were four valid QAS configurations for the DGE, so four valid QAS configuration specific constraints were defined. The constraint Variants VC2, for instance, was set to specify that the fragments \([\text{SO1}], \text{[ST1]}, \text{[R1]} \text{and [RM1]}\) need to be selected together with mandatory fragments to derive a valid configuration VC2 specific QAS from the specialized dQAS for self-upgradability.

The bindings constraint type was defined for each valid QAS configuration to constrain values of the parameters set for the response measure fragments. The DGE domain products with push type of self-upgradability were specified as a valid QAS configuration fragment \([\text{VC1}]\). The bindings constraints for the \([\text{VC1}]\), for instance, were defined as Bindings VC1. This binding constraint requires that the sum of the parameter $X$ (for update notification) and the parameter $Z$ (for update execution) should be less than or equal to 10 minutes. The binding constraints for all other valid QAS configurations were defined according to the DGE products’ requirements. The specialization of a general dQAS was completed with the definition of the fragment constraint element. And this brought us to the end of the requirements specialization process for the DGE domain.
4.3.2 Requirements Specialization for NSPL

Following the requirements specialization process, we began with activity 1. As instructed in the activity 1, we reused ASPL scope definition from the example ASPL platform, described in Section 3.3, to define the NSPL domain scope. Figure 4.4 depicts the NSPL domain scope definition. The NSPL domain does not require self-upgradability; thus, we excluded the self-upgradability in the NSPL scope definition. Further, we added a feature tree for self-healing as it was missing in the ASPL scope definition, and specialized self-optimization feature tree according to needs of the NSPL domain. The red “X” marks in the scope definition specify out of scope properties and features.

Next, in activity 2, we searched the example ASPL platform to find reusable requirements specification artifacts for self-optimization and self-healing. A gen-
eral dQAS for self-optimization was found, so we specialized it according to needs of the NSPL domain, see section 4.2 for details. The self-healing property was not supported by the example ASPL platform, so we could not find a general dQAS for self-healing. Thus, we defined self-healing dQAS from scratch, see section 4.3.2 for details.

NSPL – dQAS for Self-Optimization

The general dQAS for self-optimization, found in activity 2, was defined without knowing the NSPL domain requirements. Thus, we analyzed it in activity 3 to identify gaps between what is required by the NSPL and what is defined in the general dQAS. The gaps in the general dQAS were addressed in activity 3b, and a specialized dQAS for self-optimization was produced. The specialized dQAS produced is shown in Table 4.2. An element by element description of how the specialized dQAS was produced is given below.

Beginning with the source element, we analyzed source fragments [SO1] and [SO2]. The NSPL requires its products to self-optimize time to collect and distribute news. Thus, we specialized the source fragment [SO1] to specify this requirement. We removed the source fragment [SO2] and stimulus fragment [ST2] because the NSPL does not require monitoring of system workload to trigger self-optimization. Further, we specialized the stimulus fragment [ST1] to specify the NSPL domain specific condition to trigger self-optimization.

For the artifact element, we removed the fragment [A2] (workload monitor) as it was not needed in the NSPL. For fragments [A1], [A3] and [A4], we renamed and rephrased them according to requirements of the NSPL. The environment element was adopted without any changes, as it satisfies the environment requirements under which self-optimization property is needed in the NSPL domain.

The response element was specialized by removing fragments [R1] and [R3] because the NSPL does not require these actions. Instead, the NSPL domain requires following two adaptive actions for performance optimization:

1. Add more servers to the server-pool
2. Exclude news in multimedia format

We specified the above actions as fragments [R1] and [R2] in the specialized dQAS. Here, the fragment [R1] is a specialized form of the “add more resources” fragment [R4] of the general scenario; and the fragment [R2] is a specialized form of the “reduce computational overhead” response fragment [R2] of the general scenario.

There were no major changes in the response measure element, except changing names of the involved entities. The NSPL requires two main variants of self-optimization: (1) optimization by adding servers, and (2) optimization by excluding multimedia contents. The variant element was specialized to specify these two variants as fragments [V1] and [V2]. These variants lead to valid products and QAS configurations.
The NSPL domain consists of three products that vary in their requirements for self-optimization variants specified in the variant element. The self-optimization variants required by these products were specified in the form of valid QAS configuration fragments [VC1], [VC2] and [VC3]. The three fragments correspond to the NSPL products P1, P2, and P3, respectively.

The “fragment constraints” element was added to specify constraints on how the first six QAS elements and their fragments can be combined to derive product specific QASs from the specialized dQAS. The mandatory constraint type specifies fragments that are required for all the valid QAS configurations. In other words, this constraint specifies fragments that need to be included in all product specific QASs derived from the specialized dQAS. For instance, the source fragment [SO1] is required for all three valid QAS configurations. Thus, it is specified as a mandatory fragment. The fragments that need to be included only for a specific valid QAS (product) configuration were specified individually for each valid QAS configuration. For instance, the constraint Variants VC1 specifies that fragments R1, RM1, and RM2 must be selected, together with the mandatory fragments. Moreover, for each valid QAS configuration, binding constraints were defined to constrain values of the parameters X and Y defined for the response measure fragments.

**NSPL – dQAS for Self-Healing**

To specify self-healing requirements of the NSPL, we could not find a general dQAS for self-healing in the requirement specialization activity (2). Thus, we started specifying the requirements using a dQAS template from scratch. While specifying the requirements, we decided to first define a general dQAS for self-healing and then specialize it according to requirements of the NSPL domain. The decision was made to support horizontal reuse. The general dQAS was defined using the ASPL requirements engineering process described and demonstrated in Chapter 3. The resulting General dQAS for self-healing is shown in Table 4.3. Following the requirement specialization activity (5), the general dQAS was added to the example ASPL platform as feedback. With the addition of the general dQAS for self-healing, the example ASPL platforms’ scope definition was extended, as shown in Figure 4.5, to include self-healing as an in-scope property.

The general dQAS for self-healing was defined independently of the NSPL requirements for self-healing. To specialize the general dQAS according to requirements of the NSPL, we performed requirements specialization activities (3a) and (3b). During these activities gaps between what is specified in the general dQAS and what is required by the NSPL were identified and addressed. A specialized dQAS for self-healing produced as a result of these activities is shown in Table 4.4. Following is an element by element description of how the specialized dQAS was defined by specializing the general dQAS for self-healing.

The sources of failures in the NSPL domain are two components, language feature, and server-pool. Thus, we specialized the source element by adding two fragments [SO1] and [SO2], one for each source of failures. In the stimulus
Processing time taken by the NSPL products’ server-pool to collect and distribute a news item

Processing time exceeds certain threshold

Processing Time-Monitor - part of managing system, which monitors and notifies processing time to performance manager

Performance Manager - part of managing system, which adapts a target system for performance optimization

Target System - a managed system, which abstracts an NSPL product and a server-pool

Runtime under normal operating conditions

Runtime under overload operating conditions

Add servers to the server-pool (Add more resources to the managed system)

Exclude news in multimedia format (reduce computational overhead)

The NSPL products’ processing time is \( \leq X \) seconds

The server-pool cost is \( \leq Y \) $

Add servers to the server-pool

Exclude news in multimedia format

Add servers to the server-pool

Exclude news in multimedia format

Add servers to the server-pool

Exclude news in multimedia format

\[ \text{(RM1)} \quad \text{RM1}.bind(X) \mid X \leq 90 \text{ seconds} \wedge (\text{RM2}.bind(Y) \mid Y \leq 2 \text{ million US dollars}) \]

\[ \text{VC1} \quad \text{V1} \wedge \text{V2} \]

\[ \text{VC2} \quad \text{V1} \wedge \text{V2} \]

\[ \text{VC3} \quad \text{V1} \wedge \text{V2} \]

Table 4.2: Self-Optimization dQAS Specialized for the NSPL element, we excluded fragments [ST1] and [ST3] as the stimuli condition specified in these fragments did not match with the NSPL requirements to trigger self-healing. The fragment [ST2] was rephrased to specify that the self-healing property is triggered when one or more servers in the server-pool fail to send heartbeats. We also added a new fragment [ST1] to specify a stimulus for a failure of the language feature.

For the artifact element, we removed fragments [A2] and [A3] as they do not specify artifacts which get stimulated in the NSPL. The fragments [A1] and [A4] were rephrased according to requirements of the NSPL. A failure may occur both under normal or overloaded operating conditions. Thus, a new fragment [E2] was added, along with the existing fragment [E1] in the environment element.

In the response element, we specialized the response fragment [R1] to specify the action by a fault monitor on detecting failure of the language feature. We removed the fragment [R2] from the general dQAS as the NSPL does not require exception detector to report a fault by throwing an exception. The NSPL uses
Failed System - a Managed System or subsystem with failure

Stimulus (ST)
- [ST1] Managed system or subsystem fails to reply ping messages
- [ST2] Managed system or subsystem fails to send heartbeat messages
- [ST3] Managed system throws an exception

Artifacts (A)
- [A1] Fault Monitor - part of Managing System - detects and report faults in a managed system to a system manager
- [A2] Exception Detector - part of Managing System - detects and throws exceptions
- [A3] Exception Handler - part of Managing System - a variant of the system manager. It handles exceptions thrown by the managed system
- [A4] System Manager – Managing System - performs adaptive actions to recover and restore the failed system

Environment (E)
- [E1] Runtime under normal operating conditions

Response (R)
- [R1] Fault Monitor detects a failure and notify it to the system manager
- [R2] Exception Detector throws an exception
- [R3] System manager replaces failed system with its standby replica or backup
- [R4] System manager rollbacks failed system to a previous checkpoint with consistent state and desired behavior
- [R5] Exception handler addresses exceptions thrown by the exception detector

Response Measure (RM)
- [RM1] Failed system is recovered and restored in \( \leq X \) time units

Variants (V)
- [V1] Ping/echo tactic [6] based fault detection
- [V2] Heartbeat tactic based fault detection
- [V3] Exceptions tactic based fault detection
- [V4] Standby tactic based fault recovery
- [V5] Checkpoint/rollback tactic based fault recovery

Valid QAS Configurations (VC)
Valid QAS configurations depend on a specific domain. Thus, we leave them undefined until the general dQAS is specialized for a specific domain

Fragment Constraints (FC)
The fragment constraints can’t be defined without knowing valid QAS configurations, so we leave these until the QAS configurations are defined

Table 4.3: A General dQAS for Self-Healing
heartbeat tactic [6] to detect and report failures in the server-pool component. This was specified by adding a new response fragment [R2]. The fragments [R3] and [R4] were specialized by changing names of the components which are restored or recovered. The fragment [R5] was excluded as it was not required for failure recovery in the NSPL.

No gap was identified in the response measure element, so the element was kept unchanged. Next, in the variant element, we rephrased fragments [V1], [V2], [V4] and [V5] according to fault detection and recovery requirements of the NSPL products. The fragment [V3] was dropped as it did not match with the NSPL requirements.

The NSPL domain consists of three products that vary in their requirements for self-healing fragments (variants) specified in the variant element. The self-
Table 4.4: The Self-Healing dQAS Specialized for the NSPL

healing variants required by these products were specified in the form of valid QAS configuration fragments [VC1], [VC2] and [VC3]. The three fragments correspond to self-healing requirements of the NSPL products P1, P2, and P3, respectively.

The “fragment constraints” element was added to specify constraints on how the first six QAS elements and their fragments can be combined to derive product specific QASs from the specialized dQAS. The mandatory constraint type was defined to specify fragments that are required by all valid QAS configurations. For instance, the artifact fragments [A1] and [A2] are required for all three valid QAS configurations. Thus, these fragments are specified as mandatory fragments. The fragments that are needed only for a specific valid QAS (prod-
uct) configuration were specified individually for each valid QAS configuration. Moreover, binding constraints were defined, for each valid QAS configuration, to constrain values of the parameter \( X \) used in the response measure fragment [RM1].

### 4.3.3 Requirements Specialization for PSPL

We used the requirements specialization process to specify requirements of the PSPL domain. Following the requirements specialization process workflow, we began with activity 1. In this activity, we reused the extended ASPL scope definition from the example ASPL platform to define the PSPL domain scope. Figure 4.6 depicts the PSPL scope definition. The PSPL did not require Self-optimization, so feature tree for the self-optimization was excluded from the PSPL scope definition. The PSPL requires self-upgradability and self-healing, so feature trees for these properties were reused and specialized according to needs of the PSPL domain. The red “X” marks in the scope definition specify out of scope properties and features.

Next, in activity 2, we searched the extended ASPL platform to find reusable requirements specification artifacts (general dQASs) for self-upgradability and self-healing properties. For both the properties, we found a general dQAS and specialized it according to needs of the PSPL domain. Below are the details about specialization of the general dQASs for both the required properties.

**PSPL – dQAS for Self-Upgradability**

The general dQAS for self-upgradability, found in activity 2, was defined without knowing the PSPL domain requirements. Thus, we analyzed it in activity 3a to identify gaps between what is required by the PSPL and what is specified in the general dQAS. The gaps in the general dQAS were addressed in activity 3b, and a specialized dQAS for self-upgradability was produced. The specialized dQAS produced is shown in Table 4.5. Below is an element by element description of how the specialized dQAS was produced.

Beginning with source element of the general dQAS for self-upgradability, we observed that the source fragments, [SO1] and [SO2], satisfy the PSPL requirements in general. However, both fragments required specialization to specify the PSPL requirements precisely. In the PSPL domain, an update repository and end-user of the PSPL products are the two source entities which may trigger self-upgradability. Thus, we rephrased the source fragments [SO1] and [SO2] to specify update repository and end-user as entities which may trigger self-upgradability in the PSPL domain.

The self-upgradability is triggered either when a new update appears in an update repository or when an end-user requests an update. The two condition to trigger self-upgradability were specified as stimulus fragments [ST1] and [ST2].

For the artifact element, fragment [A1] matched with the PSPL requirements, so no changes were made in this fragment. In fragment [A2], we replaced the target managed system with “photoshare application” to specify the
Figure 4.6: PSPL Domain Scope
managed system which requires self-upgradability.

No changes were made in the environment element; because, the requirements specified in the environment element of the general dQAS matched with the PSPL requirements for the environment.

The response fragments [R1] and [R2] were merged into a single [R1] fragment. The merge decision was made as the response actions specified in the two fragments were required to be performed simultaneously. The PSPL products require both push and pull type updates. The adaptive actions needed to support push and pull type updates were specified as fragments [R2] and [R3]. The fragment [R4] was removed because the PSPL does not require push critical updates.

The response measure fragments were specialized according to changes made in the response fragment. For instance, the fragments [RM1] and [RM2] were merged to reflect the merger of response fragment [R1] and [R2]. Further, the parameters defined in the response measure fragments were specialized according to the PSPL domain requirements.

The PSP domain is composed of three products P1, P2, and P3. However, the three products require only two type of self-upgradability: 1) push and 2) pull. The required types were specified as fragments [V1] and [V2] in the variant element. The self-upgradability variants required by the PSPL products were specified in the form of valid QAS configuration fragments [VC1], [VC2] and [VC3]. The three fragments correspond to self-upgradability requirements of the PSPL products P1, P2, and P3, respectively.

The “fragment constraints” element was added to specify constraints on how the first six QAS elements and their fragments can be combined to derive product specific QASs from the specialized dQAS. The mandatory constraint type was defined to specify fragments that are required by all valid QAS configurations. For instance, the artifact fragments [A1] and [A2] are required for all three valid QAS configurations. Thus these fragments were specified as mandatory fragments. The fragments that are needed only for a specific valid QAS (product) configuration were specified individually for each valid QAS configuration. Moreover, binding constraints were defined, for each valid QAS configuration, to constrain values of the parameters X, Y, and Z. These parameters are used in the response measure element to constrain actions specified in the response element.

### PSPL – dQAS for Self-Healing

The general dQAS for self-healing, found in activity2 was defined independent of the PSPL requirements for self-healing. To specialize the general dQAS according to requirements of the PSPL, we performed requirements specialization activities3 and4. During these activities gaps between what is specified in the general dQAS and what is required by the PSPL were identified and addressed. A specialized dQAS for self-healing produced as a result of these activities is shown in Table 4.6. Following is an element by element description of how the specialized dQAS was defined by reusing the general dQAS for self-healing.
| Source (SO) | [SO1] Updates Repository - a directory in a file system used to introduce new updates  
|            | [SO2] End-user who requests updates  |
| Stimulus (ST) | [ST1] A new update appears in the updates repository  
|              | [ST2] An end-user requests an update  |
| Artifacts (A) | [A1] Updates Manager - a managing system that handles new updates, update requests, and triggers actions to deliver and perform updates  
|               | [A2] PhotoShare Application - a managed system on which updates are applied  |
| Environment (E) | [E1] Runtime under normal operating conditions  |
| Response (R) | [R1] New update is detected and is notified to the Updates Manager  
|              | [R2] The Updates Manager pushes an update to end-users of the PhotoShare application  
|              | [R3] The Updates Manager responds an update request  |
| Response Measure (RM) | [RM1] New update is detected and notified to the Updates Manager within $X$ seconds, with $X \leq 30$  
|                      | [RM2] The Updates Manager pushes an update to end-users of the PhotoShare application within $Y$ seconds, with $Y \leq 60$  
|                      | [RM3] The Updates Manager responds an update request within $Z$ seconds, with $Z \leq 120$  |
| Variants (V) | [V1] push  
|              | [V2] pull  |
| Valid QAS Configurations (VC) | [VC1] V1  
|                       | [VC2] V2  
|                       | [VC3] V1 $\land$ V2  |
| Fragment Constraints (FC) | [FC1] Mandatory Fragments: $\{ A1, A2 \} \land \{ E1 \}$  
|                        | [FC2] Configuration Specific Fragments:  
|                        | $\{ \text{Variants VC1} \} \land \{ \text{ST1} \} \land \{ \text{R1, R2} \} \land \{ \text{RM1, RM2} \}$  
|                        | $\{ \text{Variants VC2} \} \land \{ \text{ST2} \} \land \{ \text{R3} \} \land \{ \text{RM3} \}$  
|                        | $\{ \text{Variants VC3} \} \land \{ \text{SO1, SO2} \} \land \{ \text{ST1, ST2} \} \land \{ \text{R1, R2, R3} \}$  
|                        | $\land \{ \text{RM1, RM2, RM3} \}$  
|                        | [FC3] Bindings:  
|                        | $\{ \text{Bindings VC1} \} \land \text{RM1.bind}(X) + \text{RM2.bind}(Y) \land X+Y \leq 90$ seconds  
|                        | $\{ \text{Bindings VC2} \} \land \text{RM3.bind}(Z) \land Z \leq 120$ seconds  
|                        | $\{ \text{Bindings VC3} \} \land \text{RM1.bind}(X) + \text{RM2.bind}(Y) \land X+Y \leq 90$ seconds,  
|                        | $\land \text{RM3.bind}(Z) \land Z \leq 120$ seconds  |

Table 4.5: Self-Upgradability dQAS Specialized for the PSPL
Beginning with source element, we found that the general dQAS specifies a generic failed system as a source of failures. We identified this as a gap and addressed it by adding two fragments \([SO1]\) and \([SO2]\). Both the fragments specify PSPL domain specific sources of failure.

Next, in the stimulus element, no significant changes were made except renaming the source artifacts which trigger self-healing. The fragment \([ST3]\) was excluded as the condition specified there did not match to requirements of the PSPL.

The PSPL domain requires having some artifacts to detect and recover from failures. Requirements for these artifacts were specified by specializing artifact fragments \([A1]\) and \([A4]\). The fragments \([A2]\) and \([A3]\) were excluded as they did not specify the PSPL domain artifacts that may get stimulated and trigger actions to recover from the failure.

In the PSPL domain, a failure may occur both under normal or overloaded operating conditions. Thus, a new fragment \([E2]\) was added, along with the existing environment fragment \([E1]\).

In the response element, we specialized the response fragment \([R1]\) into two new fragments, \([R1]\) and \([R2]\). The specialization was made to specify that the PSPL requires the use of ping/echo and heartbeat tactics to detect failures in upload and share services, respectively. We removed the fragment \([R2]\) from the general dQAS because the PSPL does not require exception detector to report a failure. The fragments \([R3]\) and \([R4]\) were specialized by renaming the restored or recovered components. We excluded fragment \([R5]\) as it was not needed for failure recovery in the PSPL.

The PSPL products distinguish between the time required to detect and the time needed to recover from a failure. In the general dQAS, the response measure element does not differentiate between the time for detection and the time for recovery. Moreover, the products vary in the time requirements for failure detection and failure recovery. Thus, we specialized the response measure fragment \([RM1]\) into four new fragments, two to specify failure detection threshold and two to define failure recovery threshold.

Next, in the variant element, fragments \([V1]\), \([V2]\), \([V4]\) and \([V5]\) matched to the PSPL requirements and were adopted with no changes. We removed the fragment \([V3]\) as it did not match with the PSPL requirements.

The PSPL domain consists of two products, which vary in their requirements for fragments specified in the variant element. We defined variant fragments for the products P1 and P2 in the form of valid QAS configuration fragments \([VC1]\) and \([VC2]\), respectively.

The “fragment constraints” element was defined to constrain how the first six QAS elements and their fragments can be combined to derive product specific QASs from the specialized dQAS. The mandatory constraint type was defined to specify fragments that are required by all valid QAS configurations. For instance, the artifact fragments \([A1]\) and \([A2]\) are required for both \([VC1]\) and \([VC2]\); thus, these fragments were defined as mandatory fragments. The fragments that are needed only for an individual valid QAS (product) configurations (and not for all) were specified individually for each valid QAS configuration.
Moreover, binding constraints were defined, for each valid QAS configuration, to constrain values of the parameter $X$ and $Y$ defined for the response measure fragments.

| Source (SO) | [SO1] Upload Service - part of a Managed System  
|            | [SO2] Share Service - part of a Managed System |
| Stimulus (ST) | [ST1] Upload service fails to reply ping messages  
|              | [ST2] Share service fails to send heartbeat messages |
| Artifacts (A) | [A1] Fault Monitor - part of Managing System - detects failure of upload and share services, and report failures to system manager  
|              | [A2] System Manager – Managing System - performs adaptive actions to recover and restore upload and share services |
| Environment (E) | [E1] Runtime under normal operating conditions  
|               | [E2] Runtime under overloaded operating conditions |
| Response (R) | [R1] Fault Monitor uses ping/echo tactic [6] to detect failure of upload service. The fault monitor notifies failures to the recovery manager  
|              | [R2] Fault Monitor uses heartbeat tactic [6] to detect failure of share service. The fault monitor notifies failures to the recovery manager  
|              | [R3] The system manager rollbacks the upload service to a checkpoint with consistent system state and behavior  
|              | [R4] The system manager replaces the component providing share service with its standby replica or backup |
| Response Measure (RM) | [RM1] Fault Monitor detects a failed upload service and notifies to the recovery manager within $X$ seconds  
|                  | [RM2] Fault Monitor detects a failed share service and notifies the failure to the recovery manager within $X$ seconds  
|                  | [RM3] The system manager rollbacks the upload service to a checkpoint with consistent system state and behavior within $Y$ seconds  
|                  | [RM4] The system manager replaces the component providing share service with its standby replica within $Y$ seconds |
| Variants (V) | Variants for Fault Detection:  
|               | [V1] Ping/echo tactic [6] based fault detection  
|               | [V2] Heartbeat tactic based fault detection  
|               | Variants for Recovery:  
|               | [V3] Checkpoint/rollback tactic based fault recovery  
|               | [V4] Standby spare tactic based fault recovery |
| Valid QAS Configurations (VC) | [VC1] $V_1 \land V_3$  
|                             | [VC2] $V_2 \land V_4$ |
| Fragment Constraints (FC) | [FC1] Mandatory Fragments: $\{ A_1, A_2 \} \land \{ E_1, E_2 \}$  
|                          | [FC2] Configuration Specific Fragments:  
|                          | $\{ Variants V_1 \} \land \{ SO_1 \} \land \{ ST_1 \} \land \{ R_1, R_3 \} \land \{ RM_1, RM_3 \}$  
|                          | $\{ Variants V_2 \} \land \{ SO_2 \} \land \{ ST_2 \} \land \{ R_2, R_4 \} \land \{ RM_2, RM_4 \}$  
|                          | [FC3] Bindings:  
|                          | $\{ Bindings V_1 \} \land RM_1.bind(X) + RM_3.bind(Y) \land X+Y \leq 140$ seconds  
|                          | $\{ Bindings V_2 \} \land RM_2.bind(X) + RM_4.bind(Y) \land X+Y \leq 180$ seconds |

Table 4.6: Self-Healing dQAS Specialized for the PSPL

4.4 Design Specialization Process

The design specialization process defines a workflow for architectural analysis and design activities to realize self-adaptation properties with reuse. Instead of
designing architecture from scratch, the design specialization process adopts a high-level reference architecture from the ASPL platform and specializes it for an application domain.

Figure 4.7 depicts the design specialization process package. Following is an overview of the process roles, work-products, and workflow.

Roles  The design specialization process is performed by a role called Domain Designer. The domain designer is required to analyze an application domain’s requirements for self-adaptation, and model design decisions to realize these requirements. The design specialization process helps domain designer to fulfill this responsibility by reusing design artifacts from the ASPL platform, and proven best design practices and knowledge from the eARF reasoning framework described in appendix A.

Work-product  Four work-products, described below, are used in the process.

1. Specialized dQASs: A Specialized dQAS is an application domain specific requirement specification artifact produced as a result of the requirements specialization process. In this process, specialized dQASs are used as input work-product to identify self-adaptation requirements of a given application domain. See section A.1 in appendix A for details about the specialized dQASs.

2. General dRSs: A General dRS is an application domain independent type of a dRS. It models a reference architecture to realize a self-adaptation property. Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) feedback loop [27, 40] is used as a primary architectural pattern to structure the architectural element in a General dRS. See appendix A for details about the dRS and General dRS.

3. Specialized dRS: A Specialized dRS is an application domain specific type of a dRS produced as a result of the design specialization process.
It is defined by finding a general dRS from the ASPL platform and customizing it according to the needs of a specific application domain.

4. eARF: The eARF is a purposefully established reasoning framework [15] to support architectural analysis and reasoning required for realization of self-adaptation properties. It helps designers to identify design alternatives, analyze and evaluate the identified alternatives, reason about the outcomes, and model design decisions.

**Process Workflow** The design specialization process workflow comprises five activities performed for each self-adaptation property required by an application domain. Activity-wise description of the process workflow is as follows:

**Activity 1 - Analyze dQASs** The design specialization process begins with an analysis activity where domain designer analyzes a set of specialized dQASs, and finds corresponding general dRSs for reuse from the ASPL platform. For each self-adaptation property needed by an application domain, a specialized dQAS for corresponding self-adaptation property is used as an input work-product. The analysis helps to determine domain requirements and to find a corresponding general dRS from the ASPL platform.

**Activity 2a - Analyze General dRSs** The general dRSs got as a result of activity 1 are defined without knowing a specific application domain and its requirements. Thus, it is more likely that there exist gaps between what is required by a particular application domain and the design decisions modeled in the general dRSs. For instance, an application domain may demand use of “event based” monitoring tactic to monitor its performance, but the found general dRS does not model any architecture element (design decision) to realize event-based performance monitoring. Moreover, being a reference architecture, a general dRS, may have some architectural elements, such as responsibility components, variation points, and variants, which are not needed by the application domain for which design specialization process is being performed. Such gaps between what is required and what is offered by a reference architecture are identified in this activity and are addressed in activity 2b.

**Activity 2b - Specialize the General dRSs** This activity transforms a General dRS into an application domain specific dRS called Specialized dRS. The specialization is performed to address the gaps identified as a result of the activity 2a. As shown in Figure 4.7, the analysis and specialization activities are performed with the help of self-adaptation property specific architectural tactics and patterns provided by the eARF. The architecture patterns are used to identify and reason about structural organization and relationships among architectural elements. And the architectural tactics are used to analyze and argue about design alternatives modeled as variation points.
and variants in the orthogonal variability model. For instance, monitoring and execution tactics, described in appendix A.3, are used to reason about different options for monitoring of a managed system, and execution of adaptive actions, respectively. Tactics for other quality attributes such as performance, availability, and security, can be used for self-adaptation properties associated with these quality attributes.

Activity 3 - Define Application Domain Specific dRSs If a general dRS is not found as a result of the activity 1, the activity 3 defines a new application domain specific dRS. The process workflow for creating the new dRS is same as the workflow for the general dRS described in Section 3.4.

Activity 4 - Verify the Domain Specific dRSs This activity is performed to assure that domain-specific dRSs produced as a result of activities 2b and 3 comply with requirements of a given application domain. The eARF provides domain designers with formal methods based properties specification and verification mechanism to verify the design decisions modeled in a specialized dRS. For details about verification and reasoning support using the eARF, please see [3]. Based on domain designer’s preferences and application domain requirements, any other evaluation method or theory that provides analytical means to verify the design decisions can be used to verify the specialized dRS.

Activity 5 - Feedback This activity considers new dRSs defined in activity 3 for inclusion in the ASPL platform as feedback. The feedback helps to expand the ASPL platform, which in turn improves the development of self-adaptive systems with reuse.

4.5 Design Specialization – Demonstration

This section demonstrates how a design specialization process is performed in practice. Continuing with running examples application domains, following is a description of how we performed design specialization for each of the example domains.

4.5.1 Design Specialization for the DGE

The DGE requires only one self-adaptation property, self-upgradability. Thus, beginning with the design specialization activity 1, we analyzed a self-upgradability specialized dQAS for the DGE, shown in Table 4.1. The self-upgradability property is supported by the example ASPL platform; thus, we found a general dRS for self-upgradability as a result of the activity 1. Following the process workflow, in activity 2a, we analyzed the found general dRS to identify gaps between what has been modeled in the general dRS and what is needed by the DGE.
Figure 4.8: Self-Upgradability dRS Specialized for the DGE
Few gaps were identified, mainly in the orthogonal variability model part of the General dRS.

The analysis was followed by a specialization activity in which we addressed the identified gaps by specializing the General dRS elements according to the DGE domain requirements. For instance, there were three optional variants defined for the “updates provider” variation point in the general dRS. However, the DGE required only one of these variants. This gap was addressed by keeping only one variant and removing the other two variants. Figure 4.8 depicts the Specialized dRS produced as a result of the activity 2b. The red “X” mark denotes the removed variants. Following is a summary of the specialization activities, and resulting elements of the specialized dRS.

The “updates manager” responsibility component from the reused General dRS was specialized by renaming it to an “operator center” component. This specialization was made because, in the DGE domain, it is the operator center which has a responsibility of managing and coordinating updates. The “updates manager” variation point associated with the updates manager component was specialized by excluding the optional variant “distributed” and changing “centralized” variant from optional to mandatory. This specialization was made because the DGE requires a central operator center (updates manager) to manage and coordinate updates.

In the DGE, updates are introduced through an updates repository (a folder or directory in a file system). Thus, we specialized the “Updates Provider” responsibility component by renaming it to “Updates Repository”. The variation point associated with the updates provider component was also specialized by excluding the unwanted “system administrator” and “system developer” variants.

The “Updates Monitor” and “Updates Consumer” components were specialized by adapting their names according to the DGE domain requirements. The “Updates Monitor” was renamed to “Repository Monitor”, and “Updates Consumer” was renamed to “Player Environment”. The “Repo. Monitor” variation point associated with updates monitor component was specialized by excluding the “periodic polling” variant. This specialization was made because all the three update types (push, push critical, and pull) can be detected using “event based” and “on-demand” monitoring tactics. To limit the number of repository monitor variants selected for a valid product configuration, we added an alternative choice variability constraint, [1..2] to the “Repo. Monitor” variation point. The constraint specifies at least one monitoring variant must be selected, but no more than two variants can be selected for any design artifact derived from the specialized dRS.

The specialized dRS has two instances of “Player Environment” component. The instance associated with “updates consumer” variation point models pull type updates that enable a player environment to request updates from the operator center. The “updates consumer” variation point was also specialized by excluding “software system” variant and renaming the “end user” variant to “player”. This specialization was made because, in the DGE, the players request OC for updates. The “player environment” instance associated with “executor” component models a target managed system on which updates are performed.
by the OC with the help of executor component. There were no requirements found for how updates are performed or executed. Thus, any one of the executor variants defined under the executor variation point can be selected. We added an alternative choice variability constraint [1..1] to the "executor" variation point to limit the number of executor variants that can be selected for a valid product configuration. The analyzer and planner component matched with requirements of the DGE domain; thus, no changes were made to these components.

For activity 4, we have not yet verified the specialized dRS, as we plan it as future work. The steps for verification are described and illustrated in [3]. Moreover, the specialized dRS is derived from a verified general dRS. Thus, it is more likely that an architecture derived from a validated architecture will comply with the specific application domain requirements.

4.5.2 Design Specialization for the NSPL

The NSPL requires two self-adaptation properties, self-optimization, and self-healing. Thus, beginning with the design specialization activity, we analyzed self-optimization and self-healing dQASs specialized for the NSPL domain. The analysis helped us to understand the NSPL requirement for self-optimization and self-healing. We searched the example ASPL for reusable design artifacts and found a general dRS for self-optimization. The general dRS was specialized for the NSPL domain. Details about the specialization and resulting self-optimization dRS are given below in Section 4.5.2.

The example ASPL platform did not initially support self-healing property. Thus, we could not find a general dRS for self-healing. Following the design specialization process activity, we defined a self-healing dRS from scratch. Details about how the self-healing dRS was defined are given below in Section 4.5.2.

NSPL – Self-Optimization dRS

We specialized the general dRS for self-optimization to produce a self-optimization dRS for the NSPL. The specialization was done by following the design specialization process activities. In activity, we analyzed the general dRS and identified several gaps. The gaps were addressed in activity, according to requirements of the NSPL. Some of the gaps and how we addressed them, for example, are described below.

The NSPL requires triggering self-optimization when processing time exceeds a certain threshold. It does not require to monitor its workload to trigger self-optimization. Thus, we excluded the workload monitor component and associated variation point, variants and interfaces. Moreover, there is no requirement specified for analyzer component, so we removed it too together with associated variation point, variants and interfaces. We also eliminated the "increase efficiency" and "add threads" variants of planner variation point. The elimination was made because the NSPL does not require tactics behind these variants to plan and execute adaptive actions for performance optimization.
Figure 4.9: Self-Optimization dRS Specialized for the NSPL
Moreover, we specialized the resp. monitor and executor variation points to constrain variants. We constrained the variants by specifying the minimum and the maximum number of allowed variants in the form of \([\text{min..max}]\) constraint. Figure 4.9 depicts self-optimization dRS produced as a result of the specialization activity \(\text{k} \_2\text{b}\). The red “X” mark, in the figure, specify the removed elements such as components, variation points, and variants.

**NSPL – Self-Healing dRS**

While defining requirements for the NSPL, we extended the scope of the example ASPL platform to support self-healing property. However, we did not add the general dRS for self-healing to the platform. Thus, we decided first to define the general dRS and then specialize it for the NSPL. The general dRS was defined using the ASPL design process described and demonstrated in Chapter 3. Figure 4.10 depicts a resulting general dRS for self-healing. Following the design specialization activity \(\text{k} \_5\), we added the self-healing general dRS as feedback to the ASPL platform.

The self-healing general dRS was defined independently of the NSPL domain. To specialize the general dRS for the NSPL domain, we performed design specialization activities \(\text{k} \_2\text{a}\) and \(\text{k} \_2\text{b}\). In activity \(\text{k} \_2\text{a}\), we analyzed the general dRS and identified gaps between the dRS and self-healing requirements of the NSPL. The gaps were identified mainly in the orthogonal variability model. For instance, there were three variants for the fault monitor variation point in the general dRS. However, the NSPL requirements for self-healing can be satisfied using only two of the variants. This and other such gaps were addressed in activity \(\text{k} \_2\text{b}\), for instance, by removing the third variant of the fault monitor variation point. Furthermore, the NSPL requires standby spare and checkpoint/rollback tactics [6] based adaptations to recover from failure. Thus, we kept these variants for the planner and execute components and removed the “exception handler” variant. Further, as a specialization activity \(\text{k} \_2\text{b}\), we constrained variants by specifying minimum and maximum bounds in the form of \([\text{min..max}]\) variability constraint.

The self-optimization dRS for the NSPL produced as a result of the design specialization process is shown in Figure 4.11. The red “X” mark, in the figure, specify the elements, such as components, variation points and variants, which were removed as a result of the design specialization activity \(\text{k} \_2\text{b}\).

The design specialization process ends with verification activity \(\text{j} \_4\). We did not verify the specialized dRSs for the NSPL, and plan the verification as future work. The verification activities are described and demonstrated in [3].

**4.5.3 Design Specialization for the PSPL**

The PSPL requires two self-adaptation properties, self-upgradability, and self-healing. Thus, beginning with the design specialization activity \(\text{1}\), we analyzed self-upgradability and self-healing dQASs specialized for the PSPL domain. The analysis helped us to understand the PSPL requirement for self-upgradability
Fault Monitor <<managing>>
Monitors and reports failed systems or components

System Manager <<managing>>
Takes actions to recover from failure and restore the failed system

Analyzer <<managing>>
Analyzes a failure

Planner <<managing>>
Plans adaptive actions to recover and restore the failed system

Executor <<managing>>
Executes the Plan

Software System <<managed>>
Base-level application software system that requires self-healing

Key
- [name] [responsibility] [Variant] [Variation Point] [Alternative Choice]
- [name] [min.. max]
- [name] [requires vp_vp]
- [name] [provides requires]
- [name] [Dependency Analyzer]
- [name] requires vp_vp

Figure 4.10: Self-Healing – General dRS
Fault Monitor
<<managing>>
Monitors and reports faults
System Manager
<<managing>>
Takes actions to recover from failure and restore the failed system
Analyzer
<<managing>>
Analyzes a failure
Planner
<<managing>>
Plans adaptive actions to recover and restore the failed system
Executor
<<managing>>
Executes the Plan

Software System
<<managed>>
Abstracts language feature and server-pool subsystems that require self-healing
Fault Monitor
<<managing>>
Monitors and reports faults
System Manager
<<managing>>
Takes actions to recover from failure and restore the failed system
Analyzer
<<managing>>
Analyzes a failure
Planner
<<managing>>
Plans adaptive actions to recover and restore the failed system
Executor
<<managing>>
Executes the Plan

Software System
<<managed>>
Base-level application software system that requires self-healing

Figure 4.11: Self-Healing dRS Specialized for the NSPL
and self-healing. As a part of the activity $\text{1}$, we searched the example ASPL to find reusable design artifacts for the self-healing and self-upgradability. We found a general dRS for each of the two properties and specialized it according to needs of the PSPL domain. Following is a one by one description of how we specialized general dRSs for self-upgradability and self-healing.

Self-Upgradability dRS Specialized for the PSPL

The general dRS for self-upgradability, found in activity $\text{1}$, was defined without knowing the PSPL domain requirements. Thus, following the design specialization process activity $\text{2a}$, we analyzed the general dRS to identify gaps between what is required by the PSPL and what was modeled in the general dRS. Few gaps were identified and addressed in activity $\text{2b}$. Some of the gaps and how we addressed them, for example, are described below.

The general dRS models three variants of “updates provider” variation point. However, in the PSPL domain, all the updates are provided through a directory in a file system. That is, in the PSPL domain there are no variants of the updates provider other than a directory. We addressed this gap by renaming the “software system” variant to “directory” and removing the other two updates provider variants. Similarly, the general dRS models three variants for the planner variation point. These variants were derived based on three variants of self-upgradability, push, pull and push critical [1]. The PSPL, however, requires only push and pull type variants. Thus, we specialized the planner variation point by removing the push critical variant. Further, as a specialization activity $\text{2b}$, we constrained variants by specifying minimum and maximum bounds in the form of $[\text{min..max}]$ variability constraint.

Figure 4.12 displays a self-upgradability dRS for the PSPL produced as a result of the design specialization process. The red “X” marks, in the figure, specify elements, such as components, variation points and variants, which got removed as a part of the design specialization activity $\text{2b}$.

Self-Healing dRS Specialized for the PSPL

The general dRS for self-healing, found in activity $\text{1}$, was defined without knowing the PSPL domain requirements. Thus, following the design specialization process activity $\text{2a}$, we analyzed the general dRS to identify gaps between what is required by the PSPL and what was modeled in the general dRS. We identified few gaps and addressed them in activity $\text{2b}$ based on self-upgradability requirements of the PSPL. Some of the gaps and how we addressed them, for example, are described below.

The general dRS models three variants for the fault monitor variation point. These variants were modeled based on ping/echo, heartbeat and exception tactics [6] for failure detection. The PSPL domain, however, requires only the ping/echo and heartbeat tactics to detect failures in the upload and share services. We addressed this gap by removing the “exception” variant. Moreover, the PSPL requires standby spare and checkpoint/rollback tactics [6] based adapta-
Figure 4.12: Self-Upgradability dRS Specialized for the PSPL
Fault Monitor

<<managing>>

Monitors and reports
Failed services

System Manager

<<managing>>

Takes actions to recover from failure and restore the failed system

Analyzer

<<managing>>

Analyze

s a failure

Notification

Analysis

Results

Planner

<<managing>>

Plans adaptive actions to recover and restore the failed system

Executor

<<managing>>

Executes the Plan

Software System

<<managed>>

Abstracts Upload and Share Services of the PhotoShare Application

Figure 4.13: Self-Healing dRS Specialized for the PSPL
tions to recover from failure. Hence, we kept these variants for the planner and execute components and removed the “exception handler” variant. Further, as a specialization activity, we constrained variants by specifying minimum and maximum bounds in the form of $[\text{min..max}]$ variability constraint. Figure 4.13 depicts the self-healing dRS produced as a result of the design specialization process.

The design specialization process ends with verification activity 4. We have not yet verified the specialized dRSs for the PSPL. However, we plan the verification as future work.
Chapter 5

Integration Process

The integration process defines activities, work-products and roles to align and integrate the separately developed managing and managed system platforms. The process aims to make managing and managed system platforms compatible and ready for use in application engineering projects [33]. This chapter provides a detailed description of the process including all the subprocesses, activities and related entities.

5.1 Introduction

Based on the ASPL strategy, the ASPLe methodology supports the development of self-adaptive systems with reuse by splitting the development of managing and the managed subsystems into separate processes. The development of managing subsystem is supported by establishing a horizontal, application domain independent ASPL platform, and reusing it to derive several application domains specific Managing System Platforms. Whereas, the development of the managed subsystem is supported by establishing a separate Managed System Platform.

As the two platforms, Managing System Platform and Managed System Platform, are defined separately, there may exist mismatches between the two platforms. To derive a product line of self-adaptive systems from two independently developed managing and managed system platforms, we need to address the mismatches between the two platforms. The mismatches are more likely to be in artifacts and areas that cross boundaries between the managed and managing system platforms, such as the monitor and adapt interfaces.

The managed system platform artifacts are required to provide “probe” and “adapt” interfaces to support communication and operations between managing and managed subsystems. The probe interfaces allow the managing system artifacts to monitor states and relevant properties of the underlying managed system artifacts and their environment. The adapt interfaces allow the managing system artifacts (execute component) to perform adaptive actions required
to adjust behavior and structure of the underlying managed system. If there are no probe and adapt interfaces and supporting components defined in the managed system platform, or the interface definitions does not match with their counterparts, i.e., monitor and execute interfaces and components in the managing system platform, one can’t reuse the artifacts from the two platforms to develop self-adaptive systems. To that end, the ASPLe methodology provides the integration process which ensures that the artifacts from managing and managed system platforms can be used together to derive a complete self-adaptive system. Comparing the ASPLe methodology to the Rainbow framework [20], the integration process corresponds to the translation infrastructure layer which helps to mediate different levels of information and operation abstractions between system layer (managed system platform) and architecture layer (managing system platform).

The integration process follows the structure of the other two processes in the ASPLe methodology. It starts with requirements integration followed by design, implementation and tests integration processes.

### 5.2 Requirements Integration Process

The requirements integration process provides guidelines for integration of the requirements specification artifacts in managed and managing system platforms. It ensures that requirement artifacts in the two platforms are well aligned and there are no mismatches between requirement specifications.

Figure 3.2 depicts the ASPL requirements engineering process package diagram. The package diagram specifies roles, work-products and workflow element of the requirements integration process. Each of these elements is described below.

**Roles** The requirements integration process is performed by a role called Domain Analyst. The domain analyst is required to have good knowledge of requirement specification artifacts in both managed and managing system platforms. The domain analyst’s responsibilities are listed, in Figure 5.1,
under the domain analyst role. These responsibilities map to activities of the requirements integration process workflow.

**Work-products** Four work-products, described below, are used in the process.

1. Managing System Requirements Engineering (RE) Artifacts: This work-product represents specialized dQASs in the managing system platform defined for a specific application domain. The Specialized dQASs specify managing system requirements and are defined as a result of the requirements specialization process.

2. Managed System Requirements Engineering (RE) Artifacts: This work-product represents requirement specifications in the managed system platform defined for a specific application domain. The requirement specifications in the managed system platform specify managed system requirements and are defined by following traditional requirement engineering methods such as software requirement specifications, use cases.

3. Managing System Integrated RE Artifacts: This work-product represents the RE artifacts in a managing system platform, i.e., specialized dQASs, which are aligned and integrated with a managed system platform as a result of the requirements integration process.

4. Managed System Integrated RE Artifacts: This work-product represents the RE artifacts in a managed system platform which are aligned and integrated with a managing system platform as a result of the requirements integration process.

**Process Workflow** As depicted in Figure 5.1, the requirements integration process workflow consists of two activities. The activities are performed for each self-adaptation property required by a given application domain. Activity-wise description of the workflow is as follows:

**Activity 1 - Analyze RE Artifacts in the Managing and Managed System Platforms:** The requirements integration process begin by analyzing requirement engineering (RE) artifacts in the managed and managing system platforms. The analysis is performed to make sure that requirements for all interfaces and components that cross boundaries or have provide/required dependencies between the two platforms are well aligned with no mismatches. These requirements are typically concerned with probe and adapt component interfaces in a managed system platform and their respective monitor and execute component interfaces in a managing system platform.

The requirements specification artifacts in a managed system platform are mainly defined for the base-level application logic. These artifacts may lack in requirements for the probe and adapt interfaces needed by the monitor and execute components in a managing system platform. For instance, requirement specification artifacts in a managed system platform
may lack in specifying requirements for the probe and adapt interfaces needed to detect and perform updates for the self-upgradability property.

The analysis activity also analyzes requirement specification artifacts in a managing system platform. Following the requirements specialization process, these artifacts are already specialized for a specific application domain. Thus, the managing system platform artifacts are more likely to be in-line with their counterparts in a managed system platform. However, the integration process double checks the requirements specifications in a managing system platform to ensure that there are no mismatches or inconsistencies.

**Activity 2 - Address the Mismatches:** This activity addresses mismatches identified as a result of the activity 1. The mismatches are addressed, for instance, by adding missing interface or component requirements, modifying requirements or parts of the requirements as needed, and in some cases by removing requirements or parts of requirements which are not needed for a specific application domain. The activities 1 and 2 are iterated with several subsequent steps where all requirement artifacts are analyzed for inconsistencies, refined and integrated. The integration results in a set of integrated requirement artifacts ready to derive requirement specifications for a product line of self-adaptive systems.

The results from the analysis activity of requirements integration are useful and should be considered for the subsequent design, implementation, and tests integration processes. This is because all these subsequent processes are defined according to what is required and specified by requirements engineering (sub)process. So if there are mismatches or inconsistencies identified in requirements specification artifacts, it is more likely that there will be mismatches in the artifacts produced by subsequent design, implementation and testing processes.

### 5.3 Requirements Integration – Demonstration

This section demonstrates requirements integration process in practice. We use the three running example application domains, DGE, PSPL, and NSPL, for the demonstration. Following is a one by one description of how we performed the requirements integration process for each of the example domains.

#### 5.4 DGE – Requirements Integration

Beginning with the requirements integration process activity 1, we analyzed RE artifacts in the managed and managing system platforms for the DGE. The RE artifacts in the managing system platform were developed using requirements specialization process, see Section 4.3.1 for details. The RE artifacts in the
Update Repository is used to introduce new updates.

Players who requests updates for their Player Environments (PEs).

A new update appears in an updates repository.

Player requests an update.

Operator Center (OC) - an update manager that handles new updates, update requests, and triggers actions to deliver and perform updates.

Player Environment (PE) - target managed system on which updates are performed.

Update Repository - used by the OC and its operator to introduce and store updates.

Runtime under normal operating conditions.

New update is detected and notified to the OC.

OC notifies an update to player environments.

OC responds update requests.

Update is applied to a target system.

New update is detected and notified to the OC with no delay after it has been placed in the Updates Repository.

OC notifies new updates to target PEs within X seconds, with range(X) = [5..60].

OC responds an update request within Y seconds, with range(Y) = [5..60].

Update is applied to target PEs within Z minutes, with range(Z) = [1..10].

Push

Push critical

Pull

Table 5.1: DGE – Self-Upgradability dQAS after Integration with the Managed System Platform
DGE managed system platform was established independently using traditional requirements engineering methods. The managed system platform for the DGE and the other two example domains, NSPL and PSPL, can be downloaded at http://homepage.lnu.se/staff/janmsi/ms-platforms/.

Beginning with the requirements integration process activity 1, we analyzed RE artifacts in the managed and managing system platforms. In the managing system platform, we analyzed a specialized dQAS for self-upgradability defined in Section 4.3. The analysis showed that the specialized dQAS was mostly aligned with requirement specification document in the managed system platform. The only mismatch identified was a missing artifact fragment to specify an update repository component used in the DGE to store updates. The mismatch was addressed, as an activity 2, by adding a new fragment [A3] to the artifact element and declaring [A3] as a mandatory fragment in the fragment constraints element. Table 5.1 depicts an integrated dQAS, an updated version of a specialized dQAS, for self-upgradability.

In the managed system platform, we analyzed a requirement specification document. In the requirement specification document, the DGE domain requirements for self-upgradability were defined along with other functional and non-functional requirements. The self-upgradability was specified as a functional requirement by defining several views. The views described step by step requirements, similar to a use case, to specify the push and pull type updates, and how these updates are triggered and performed by an operator center component. However, considering the architectural split between managing and managed subsystems, the requirements for probe and adapt interfaces between the two subsystems were missing. This lack of requirements for the probe and adapt interfaces was addressed, as an activity 2 of the integration process, by adding following requirements to the requirement specification document.

1. Updates repository component should provide an interface for the operator center to monitor or probe updates.
2. The player environment should provide an interface to get update notifications and updates.
3. The operator center should provide an interface for player environments to probe and pull updates.
4. The player environment should also provide an interface for the operator center to probe and retrieve runtime information about current status of the player environment and its operating environment.
5. The player environment should also provide an interface for the operator center to push and perform updates.

5.4.1 NSPL – Requirements Integration

Beginning with the requirements integration process activity 1, we analyzed RE artifacts in the managed and managing system platforms for the NSPL. The RE
artifacts in the managing system platform were developed using requirements specialization process, see Section 4.3.2 for details. The RE artifacts in the NSPL managed system platform was established independently using traditional requirements specification methods.

In the managing system platform, we analyzed specialized dQASs for self-optimization and self-healing properties needed by the NSPL. The analysis showed that specialized dQASs for both self-optimization and self-healing were well aligned with requirement specification document in the managed system platform. No mismatch was identified, so no changes were made to the specialized dQASs for both the properties.

In the managed system platform, we analyzed a requirements specification document. The requirements document specifies functional requirements using use case scenarios. For self-optimization, the requirements document lacked in requirements for probe and adapt interfaces needed by managing system components to monitor and optimize managed system’s performance. We addressed this, as an activity of the integration process, by adding following requirements to the managed system’s requirements specification document.

1. The NSPL products should provide an interface for the managing system to monitor processing time to collect and distribute a news item.

2. The server-pool should provide an interface for adding servers at runtime.

For the self-healing property, the failures of the upload and share services were specified as alternative flows in use case scenarios. The scenarios, however, lacked in requirements for probe and adapt interfaces needed by the managing system components to retrieve runtime information and perform adaptive actions to recover from failures. Thus, we added following requirements to the requirements specification document.

1. The component providing “language” feature should provide a monitor interface to detect failures. It should also provide interfaces for the managing system to record checkpoints and rollback the component, in case of failure, to a checkpoint with consistent state and behavior.

2. The managed system should provide an interface to replace a failed server, in the server-pool, with a standby spare replica of the failed server.

3. Each server in the server-pool is required to send heartbeat messages to fault monitor component of the managing system.

5.4.2 PSPL – Requirements Integration

Beginning with the requirements integration process activity, we analyzed RE artifacts in the managed and managing system platforms for the PSPL. The RE artifacts in the managing system platform were developed using requirements specialization process, see Section 4.3.3 for details. The RE artifacts in the
PSPL managed system platform was established independently using traditional requirements specification methods.

In the managing system platform, we analyzed specialized dQASs for self-upgradability and self-healing. The analysis showed that the specialized dQAS for self-upgradability was mostly aligned with requirement specification document in the managed system platform. Only a couple of mismatches were identified. Both the mismatches were addressed as activity $\text{2}$ of the requirements integration process. The first mismatch identified was a use of the term “application” and “product” in the two platforms. We addressed this mismatch by replacing “application” with “product” in the specialized dQAS. The second
mismatch was the missing updates repository fragment in the artifacts element. The PSPL requires an updates repository, directory in a file system, to introduce and store updates. We addressed this mismatch by adding a new artifact fragment [A3] and specifying it as a mandatory fragment in the fragment constraints element. Table 5.2 depicts an integrated dQAS, an updated version of the specialized dQAS, for self-upgradability. The mismatches are highlighted using bold text.

The analysis of the specialized dQAS for self-healing showed several mismatches between the managing and the managed system platforms. The mismatches were mainly caused due to a misunderstanding of how failures are detected and handled by managing system. The managed system platform considers and treats feature failures as exceptions. Whereas, the managing system considers the failures as node failures and specifies the use of ping/echo and heartbeat tactics [6] to detect and recover from failures. We addressed this mismatch by rephrasing the stimulus, artifacts, response, response measure and variants fragments. For instance, while integrating requirements with managed system platform, we found that failures of the upload and share feature can be detected using exception tactic [6]. Thus, we removed the variant fragment [V2] (Heartbeat) and changed the variant fragment [V1] from Ping/Echo to Exception. The changes in the variants and other elements led to changes in the valid QAS configurations and fragment constraint elements. Table 5.3 highlights all the changes made to address the gaps and to make the self-healing dQAS consistent with the PSPL domain requirements specified in the managed system platform.

In the managed system platform, we analyzed a requirements specification document. The requirements document specifies functional requirements using use case scenarios. For the self-upgradability, the requirements document lacked in specifications of the probe and adapt interfaces needed to push and pull updates between managed and managing system components. We addressed this lack of requirements, as activity 2 of the integration process, by adding following to the managed system’s requirements specification document.

1. The PSPL products should provide an interface to get update notifications and updates from the Updates Manager, which is a managing system component.

2. The Update Manager should provide an interface for the PSPL products to view and request updates.

For the self-healing property, the failures of the upload and share services were specified as alternate scenarios. The requirements document, however, lacked in requirements for probe and adapt interfaces needed by the managing system components to retrieve runtime information and perform adaptive actions to recover from failures. Thus, we added following requirements to the requirements specification document.

1. The PSPL products should support failure detection and recovery using exception tactic [6], i.e., by throwing and managing exceptions.
| Source (SO) | [SO1] Upload Service feature - part of a Managed System  
[SO2] Share Service feature - part of a Managed System |
| Stimulus (ST) | [ST1] Failure Monitor detects that the upload service is not responding  
Upload feature fails to display “Select Pictures” dialogue box  
[ST2] Failure Monitor detects that the share service is not responding  
Share feature fails to share pictures with friends or to make the pictures available for general public  
[ST3] Upload feature fails to upload the selected pictures |
| Artifacts (A) | [A1] Failure Monitor Exception Detector - part of Managing System  
detects and reports service failures to a system manager  
[A2] System Manager Exception Handler - Managing System  
performs adaptive actions to handle exceptions, i.e., recover and restore the failed services |
| Environment (E) | [E1] Runtime under normal operating conditions |
| Response (R) | [R1] Failure Monitor detects a “not responding” upload service and notifies the failure to the recovery manager  
Exception handler displays error message “Upload feature is temporarily out of order, wait please!”, and notifies system manager  
[R2] Failure Monitor detects a “not responding” share service and notifies the failure to the recovery manager  
Exception handler displays error message “Share feature is temporarily out of order, wait please!”, and notifies system manager  
[R3] The system manager rollbacks the upload service feature to a checkpoint with consistent system state and behavior  
[R4] The system manager replaces the component providing share service feature with its standby replica or backup |
| Response Measure (RM) | [RM1] Failure Monitor detects a “not responding” upload service  
Exception handler displays error message and notifies the system manager within X seconds  
[RM2] Failure Monitor detects a “not responding” share service  
and notifies the failure to the recovery manager within Y seconds  
[RM3] [RM2] The system manager rollbacks the upload service feature to a checkpoint with consistent system state and behavior within Y seconds  
[RM4] [RM3] The system manager replaces the component providing share service feature with its standby replica or backup within X seconds |
| Variants (V) | Variants for Fault Detection:  
[V1] Ping/echo Exception tactic [6] based fault detection  
[V2] Heartbeat tactic based fault detection  
Variants for Recovery:  
[V2] [V1] Checkpoint/rollback tactic based fault recovery  
[V4] [V3] Standby tactic based fault recovery |
| Valid QAS Configurations (VC) | [VC1] V1 → V4 V1 ∧ V2  
[VC2] V2 → V4 V1 ∧ V3 |
| Fragment Constraints (FC) | [FC1] Mandatory Fragments:  
{ A1, A2 } ∧ { E1 }  
[FC2] Configuration Specific Fragments:  
{ Variants VC1 } { SO1 } ∧ { ST1, ST3 } ∧ { R1, R3 }  
{ Variants VC2 } { SO2 } ∧ { ST2 } ∧ { R2, R4 } ∧ { RM2, RM4 RM1, RM3 }  
[FC3] Bindings:  
{ Bindings VC1 } RM1.bind(X) + RM3.bind(Y) RM2.bind(Y)  
X+Y ≤ 140 seconds  
{ Bindings VC2 } RM2.bind(X) + RM4.bind(Y) RM1.bind(X) + RM3.bind(Y)  
X+Y ≤ 180 seconds |

Table 5.3: PSPL – Self-Healing dQAS after Integration with the Managed System Platform
2. The component providing “upload” feature should provide interfaces for the managing system to record checkpoints and rollback the failed component to a checkpoint with consistent state and behavior.

3. The PSPL products should provide an interface for managing system to replace a failed “feature” component with its standby replica.

5.5 Design Integration Process

The design integration process is concerned with the synthesis of design artifacts in the separately established managed and managing system platforms. The design artifacts are modeled to achieve what is specified in requirement specification artifacts. The mismatches or inconsistencies among the requirement artifacts, if any, may lead to similar mismatches in design artifacts. For instance, a domain architecture for a managed or managing subsystem will not define a component or component interface, if it is not specified in a requirement specification artifact. Moreover, as the two platforms are defined by separate roles and processes, the design artifacts may differ in their levels of abstraction for the design components and provide/required interfaces. If not identified and addressed, such differences among design artifacts may lead to problems in subsequent development with reuse activities. The design integration process makes sure that design artifact in the two platforms are well aligned with each other, and there are no architectural mismatches [19].

Figure 5.2 depicts roles, work-products and workflow elements of the design integration process. Each of the process elements is described below.

Roles The design integration process is performed by a role called Domain Designer. The domain designer is required to have good knowledge and understanding of design artifacts in the managed and managing system platforms. Further responsibilities of the domain designer are listed in Figure ???. These responsibilities map to activities of the design integration process.
Work-products Four work-products, described below, are used in the process.

1. Managing System Requirements and Design Artifacts: This work-product represents specialized dQASs and dRSs in a managing system platform defined for a specific application domain. The Specialized dRSs map requirements specified in specialized dQASs to design decisions.

2. Managed System Requirements and Design Artifacts: This work-product represents requirement specifications and design artifacts in a managed system platform defined for a specific application domain.

3. Managing System Integrated Design Artifacts: This work-product represents design artifacts in a managing system platform, i.e., specialized dRSs, which are aligned and integrated with a managed system platform as a result of the design integration process.

4. Managed System Integrated Design Artifacts: This work-product represents design artifacts in a managed system platform which are aligned and integrated with a managing system platform as a result of the design integration process.

Process Workflow As depicted in Figure 5.2, the design integration process workflow consists of two activities, which are performed for each self-adaptation property required by a given application domain. Activity-wise description of the workflow is as follows:

Activity 1 - Analyze Requirements and Design Artifacts: The design integration process begins by analyzing requirements artifacts to more easily pinpoint gaps in design artifacts. The analysis activity attempts to identify architectural mismatches, for instance, mismatches in provide/required interfaces. The analysis focuses on architectural components and interfaces that cross boundaries between managed and managing subsystem. These components and interfaces are mainly concerned with monitor and execute responsibilities in a managing system and their respective probe and adapt responsibilities in a managed system. As the design artifacts in the two platforms are defined separately, it is more likely to have mismatches or gaps among design artifacts. A list of mismatches is produced as a result of the analysis.

Activity 2 - Address the Mismatches: The architectural mismatches between managed and managing platforms are addressed iteratively in this activity, which integrates, analyzes, and refines design artifacts. Refinement examples include reconsideration of design decisions, for example, adding, removing, and changing design components, their interfaces, variation points and variants. The resulting set of design artifacts represents an integrated design for a product line of self-adaptive systems.
Figure 5.3: An Integrated Reference Architecture for Self-Upgradability in the DGE Domain
Figure 5.4: DGE Managed System - Self-Upgradability Architectural View

5.6 Design Integration – Demonstration

Continuing with running examples for demonstration, this section exemplifies how we performed the design integration process for the three product lines that we use as example application domains.

5.6.1 Design Integration for the DGE

The first activity in the design integration process is to analyze design artifacts in managed and managing system platforms. The design artifacts in the managing system platform were developed using design specialization process, see Section 4.5.1 for details. The design artifacts in the managed system platform were defined in the form of architectural views.

Beginning with the design integration process activity 1, we analyzed integrated requirements and design artifacts in the managed and managing system platforms. In the managing system platform, we analyzed the specialized dRS for self-upgradability produced as a result of design specialization process in Section 4.5. The only main gap identified in the dRS was a missing interface between the “Operator Center” and the “Player Environment” components. The interface was required by the OC to probe and retrieve runtime information about the PEs. We addressed the gap in activity 2 by adding provide/required interfaces between the OC and PE components. Moreover, the interface between “execute” and “player environment” components was redefined to make it clear that this interface is used both to notify and deliver updates to player environments. Figure 5.3 highlights the changes made to the Specialized dRS after integration with the managed system platform.

In the managed system platform, we analyzed a “player environment update”
architectural view. The “player environment update” view models how updates are performed in the DGE. We found that the view did not distinguish between managed and managing subsystems. The OC and PE components were connected through “data flows”, no provide/required interfaces were defined for the OC, PE and other design components. We accounted this lack of interface definition and no distinction between managed and managing subsystem as an architectural mismatch. The mismatch was addressed by defining provide/required interfaces between the OC, PE and other design components. Moreover, we annotated the OC as managing, the PE as managed subsystem components, and “Bundle Storage” as data store component. The “bundle storage” component in the OC was renamed to “Updates Repository”, and a monitor interface was defined between the update repository and OC components. The “bundle storage” artifact in the PE was modeled as a component with read and write interfaces. All these changes were made to follow standard design notations, UML in this case, and to better align the design artifacts with their counterparts in the managing system platform. Figure 5.4 depicts the managed system’s self-upgradability architectural views before and after integration. The integration resulted in well aligned architectural artifacts in the managed and managing system platforms to design the DGE products with the self-upgradability property.

5.6.2 Design Integration for the NSPL

Beginning with the design integration process activity 1, we analyzed design artifacts in the managed and managing system platforms for the NSPL. The design artifacts in the managing system platform were developed using design specialization process, see Section 4.5.2 for details. The design artifacts, in the NSPL managed system platform, were defined in the form of architectural views. In the managing system platform, we analyzed specialized dRSs for self-optimization and self-healing. The analysis showed that the design decisions in the specialized dRSs were modeled with respect to managed system artifacts. We did not find any mismatch, so no changes were made to the specialized dRSs for both the properties.

In the managed system platform, we analyzed the architectural views which model design decisions concerned with the NSPL application logic. Similar to the requirement artifacts, we found that for both self-optimization and self-healing, no design decisions were modeled for monitor and adapt operations that cross boundaries between managed and managing system artifacts. The design artifacts, i.e., dRSs, in the managing system platform, however, require the managed system artifacts to provide with the probe and execute interfaces that can be used by the managing system artifacts to monitor and adapt the managed system. To address this mismatch between managed and managing system design artifacts, we added self-optimization and self-healing architectural views to the managed system platform. Both the architectural views model explicit interfaces for monitor and adapt operations between the managed and managing systems. The self-optimization and self-healing views are shown in
5.6.3 Design Integration for the PSPL

Following the design integration process workflow we started with activity ①. In this activity, we analyzed design artifacts in the managed and managing system platforms for the PSPL. The design artifacts in the managing system platform were developed using design specialization process, see Section 4.5.3 for details. The design artifacts, in the PSPL managed system platform, were defined in the form of architectural views.

In the managing system platform, we analyzed specialized dRSs for self-upgradability and self-healing. The design decisions in the dRS for self-upgradability were modeled with respect to managed system artifacts. No mismatch was identified, so no changes were made in the specialized dRS for self-upgradability. The analysis of the specialized dRS for self-healing revealed several mismatches between the managing and the managed system platforms. These mismatches map to mismatches identified during requirements integration. The primary cause of the mismatches was inconsistency in the use of tactics to detect and address failures. The managed system design required the use of exception tactic [6] to identify and restore failed components. Thus, we remodeled the fault monitor, planner and executor variation points to make the dRS consistent with the managed system artifacts. The updated dRS for self-healing produced as a result of the design integration process is shown in Figure 5.6.
Figure 5.6: PSPL – Integrated dRS for Self-Healing
In the managed system platform, we analyzed software architecture document that models and describes design decisions concerned with the PSPL application logic. The architecture document uses architectural views to model and document the PSPL managed system components. There were no architectural views defined, for both self-upgradability and self-healing, to model design decisions for monitor and adapt operations that cross boundaries between managed and managing system artifacts. The integrated dRSs for both self-upgradability and self-healing require the managed system’s design artifacts to provide the probe and execute interfaces. These artifacts are used by the managing system artifacts to monitor and adapt the managed system components. To address the lack of interfaces for probe (monitor) and adapt (execute) operations, we defined self-healing and self-upgradability architectural views shown in figures 5.7(a) and 5.7(b), respectively. Both the views were added to design artifacts in the managed system platform. As depicted in the figures, both the architectural views model explicit interfaces for monitor and adapt operations between managed and managing systems.
Appendices
A reasoning framework (RF) is an abstraction to encapsulate architectural knowledge and methods. It helps software architects to identify design alternatives, analyze and evaluate the identified alternatives, reason about the outcomes, and model design decisions. Diaz-Pace and Bass et al. [15, 7] proposed use of reasoning framework to realize quality attributes. In original formulation, a reasoning framework is composed of six elements [7]:

1. **Problem description** specifies a quality attribute for which the reasoning framework is defined and is useful.

2. **Analytic theory** is an established discipline, such as queuing theory, rate monotonic scheduling theory, temporal logic, etc., that provides basis for reasoning.

3. **Analytic constraints** are imposed by the analytic theory to make sure that all the assumptions for use of the analytic theory are satisfied.

4. **Model representation** is a model of system aspects that are relevant to the analytic theory.

5. **Interpretation** is a mapping from architecture to the model representation.

6. **Evaluation procedure** consists of algorithms or formulae used to evaluate the model representation by computing specific measures of the quality attribute.
The ASPL strategy requires architects to define reference architecture artifacts that can be specialized for reuse in a number of product lines of self-adaptive systems. To define such reference architectures, the architects are required to identify number of design options, reason about them and then map them to design decisions with abstract interfaces that can be specialized for reuse in a number of application domains and contexts. While developing design support as part of the ASPL methodology, we found the concept of reasoning framework useful to address challenges faced by domain designers. However, existing reasoning frameworks lacked in analysis and reasoning support required for defining reference architectures to realize self-adaptation properties with and for reuse. To that end, we extended a reasoning framework presented by Diaz-Pace et al. [15] to develop an extended Architectural Reasoning Framework (eARF) [1, 5].

The eARF encapsulates proven best architectural practices and knowledge to support architectural analysis and reasoning in context of the ASPL design process. Figure A.1 depicts roles, work-products, and workflow to use the eARF. The eARF is used by domain analysts and designers to analyze requirements and map them to design decisions. The eARF’s workflow involves four work-products: i) dQAS, ii) dRS, iii) Architectural Tactics and iv) Patterns. An overview of the work-products is given below followed by detailed description.

The identification and characterization of domain requirements and their variability is a prerequisite to architectural reasoning and design. The eARF recommends use of domain Quality Attribute Scenarios (dQAS) to specify requirements for the in-scope self-adaptation properties. A dQAS is an extended form of quality attribute scenario template [6]. It provides purposefully defined constructs and elements to specify variability at domain level, see section A.1 for details about the dQAS. Specifying self-adaptation requirements using dQASs is equivalent to the “problem description” element of the reasoning framework structure proposed by Bass et al. [7].

To design an architecture, the architects need to map requirements to architectural elements. Being quality attributes, self-adaptation properties are

Figure A.1: The extended Architectural Reasoning Framework
difficult to localize, interpret and model as architectural elements. To that end, the eARF recommends to follow responsibility-driven design [41] to extract a set of responsibilities from a dQAS (domain requirements) and map the extracted responsibilities to architectural elements in the form of a domain Responsibility Structure (dRS). The dRS is a modular representation of analyzed, reasoned about and verified domain requirements for a self-adaptation property. It is equivalent to the model representation element in the original reasoning framework formulation. Details about the domain responsibility structure are given in Section A.2.

The process of defining a dRS presents architects with identifying and reasoning about number of design alternatives. To identify various design alternatives, reason about them and come up with best design decisions, the eARF provides architects with proven best architectural practices and knowledge in the form of architectural patterns and tactics. Tactics and patterns encapsulate proven design decisions that are being used for years to satisfy quality attributes. Details about tactics and patterns and their role in defining responsibility structures are given in Section A.3.

The eARF initially lacked in analytic theory and analytic constraint elements that enable architects to evaluate and verify the modeled design components. To fill the gap, an analytical framework was added to the eARF. The analytical framework provides domain designers with formal means to model responsibilities and associated design options as a network of timed automata, specify desired self-adaptation properties in a timed computation-tree-logic based query language, and a model checking tool to verify specified properties. A brief description of the analytical framework is given in Section A.4. More information about the analytical framework and how it can be used for architectural analysis and reasoning is given in [3].

A.1 domain Quality Attribute Scenarios (dQAS)

A quality attribute scenario (QAS) is a requirements specification template purposefully designed to characterize quality attributes [6] of a software system. A standard QAS consists of six elements: stimulus, source of stimulus, environment, stimulated artifact, response, and response measure. These elements together characterize a quality attribute in the form of a scenario. Table A.1 provides an overview of the QAS elements. Details about these elements and example quality attribute scenarios can be found in [6].

Pohl et al. [33] suggested three questions to identify an application domain requirements with variability:

1. What does vary? - e.g., an algorithm or a configuration parameter value.
2. Why does it vary? - e.g., to support a new execution environment or requirement.
3. How does it vary? - e.g., by dynamic linking or by setting a parameter at compile time.

101
Elements | Description
--- | ---
Source (SO) | This is some entity (a human, a computer system, or any other actuator) that triggers a stimulus.
Stimulus (ST) | The stimulus is a condition or an event, for instance, a user request, a message, that must be considered on arrival at a system.
Artifacts (A) | Artifacts are parts of a system which get stimulated by a stimulus and trigger an activity or action in response.
Environment (E) | The environment represent operating conditions under which a stimulus occurs. The stimulus may occur when a system is operating in an ‘over-load’ or in a ‘normal’ mode.
Response (R) | The response abstract one or more activities taken by a system to respond the stimulus.
Response Measure (RM) | The response measure specifies measurable constraints on the response so that the quality attribute can be tested.

Table A.1: Quality Attribute Scenario (QAS) Template

Abbas et al.[4] argued that these questions can be answered by defining a quality attribute scenario. The first question, “what does vary?”, can be answered by defining artifact and environment elements of the QAS. The second question, “why does it vary?” can be answered using the stimulus and the source elements. And the third question, “how does it vary” can be answered using the response and response measure elements. Thus, the QAS provides basic structure to characterize quality attribute including self-adaptation properties with variability. However, the QAS elements lack in explicit support for variability specification, i.e., there are no specific constructs to specify variation points and variants. To address this lack of support for variability specification, Abbas et al.[4] extended the QAS template with additional elements to specify variability at domain level. This extension effort resulted in a new template named domain Quality Attribute Scenarios (dQAS).

As shown in Table A.2, the dQAS is an extended form of quality attribute scenario. The first six elements of the dQAS are same as originally defined in a standard QAS. However, contents of these elements are extended with fragments and parameters to specify domain variability. The fragments are variants of scenario elements that are used to specify variations in the dQAS elements. For example, different products in an application domain may differ in their requirements for the source, stimulus and other dQAS elements. Such variations in a domain requirements are specified as fragments of the dQAS elements. The parameters are used to express more fine-grained variations inside fragments. The parameters have to be bound to specific values for a product specific scenario, and constraints for the parameter values are specified in the newly added fragment constraints element.

Three new elements were introduced in a dQAS: Variants, Valid QAS Con-
The first six elements are same as originally defined in the QAS, see Table A.1 and [6] for details. The dQAS extends these elements with fragments and parameters to specify variability.

<table>
<thead>
<tr>
<th>Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td></td>
</tr>
<tr>
<td>Stimulus</td>
<td></td>
</tr>
<tr>
<td>Artifacts</td>
<td></td>
</tr>
<tr>
<td>Environment</td>
<td></td>
</tr>
<tr>
<td>Response</td>
<td></td>
</tr>
<tr>
<td>Measure</td>
<td></td>
</tr>
<tr>
<td>Variants (V)</td>
<td>This element specifies variants to realize a quality attribute or a self-adaptation property.</td>
</tr>
<tr>
<td>Valid QAS Configurations (VC)</td>
<td>Valid QAS Configurations define allowed combinations of the variants to derive an application domain or product specific scenarios from a general dQAS.</td>
</tr>
<tr>
<td>Fragment Constraints (FC)</td>
<td>Fragment constraints express constraints on fragments of a standard QAS elements (i.e., Source, Stimulus, Artifact, Environment, Response, and Response Measure). These constraints are organized in three types: 1) Mandatory constraints define fragments that need to be included in all domain or product specific scenarios derived from a general dQAS. 2) Variants Specific constraints define additional fragments that need to be included for only for a domain or product specific scenarios derived from a general dQAS. 3) Binding constraints define restricts or constrains values of parameters defined to support variability in the first six (standard QAS) elements.</td>
</tr>
</tbody>
</table>

Table A.2: domain Quality Attribute Scenario (dQAS) Template

*figurations, and Fragment Constraints.* These elements together define domain variants of a self-adaptation property, and specify constraints on how the scenario elements, their fragments and parameters can be combined to derive product or domain specific scenarios. The “variants” element specifies domain variants for a specified self-adaptation property. For example, the DGE application domain, see appendix 2.4.1, requires three variants for self-upgradability: push, pull, and critical push. So the variants element of a self-upgradability dQAS for the DGE domain should specify three fragments: [v1] push, [v2] pull, and [v3] critical push. The “Valid QAS configurations” element specifies how the variants can be combined to derive product specific scenarios from a dQAS. For instance, variant v1 can be combined with variant v2 to derive a self-upgradability scenario for DGE product P2. The “fragment constraints” element specifies constraints on fragments and parameters defined in the first six (standard QAS) elements.
To support reuse across several domains, a dQAS can be defined independent of any application domain. A dQAS which has been defined independent of any application domain is called a General dQAS. The domain analysts may use self-adaptation property specific tactics to identify general (application domain independent) variants and specify these as fragments of the variants element. However, definition of “valid QAS configurations” and “fragment constraints” depends on a specific application domain requirements, so these two elements cannot be defined without knowing the target application domain. Thus, the valid QAS configurations and fragment constraints elements are left undefined in a general dQAS.

A general dQAS can be used to derive several application domain specific scenarios called specialized dQAS. Deriving a specialized dQAS from a general dQAS requires to customize the general dQAS according to the needs of the application domain targeted by a specialized dQAS. The ASPLLe methodology provides a requirements specialization process, described in Section 4.2, to transform a general dQAS into a specialized dQAS.

A.2 domain Responsibility Structure

The domain Responsibility Structure (dRS) is a modular representation of design decisions made to realize a self-adaptation property. It is defined by following responsibility-driven design [41] method. Basic idea of the responsibility-driven design is to interpret requirements as responsibilities and model them as architectural elements called “responsibility components”. Both responsibility and responsibility components are abstract concepts. A responsibility is an abstraction of activities and/or operations that a software system or subsystem is required to perform for desired functionalities and/or quality attributes. A responsibility component abstracts a system unit, such as subsystem, component, package, or a class, with abstract responsibility definition. In other words, a responsibility component is a basic unit of a system architecture with well defined responsibilities. Each responsibility component has two parts; the upper part specifies a unique identifier for the component, whereas the bottom part specifies one or more responsibilities assigned to the component. The responsibility components coordinate together through “provide” and “required” interfaces to realize a self-adaptation property or a quality attribute.

To support different dimensions of variability and associated uncertainty, the dRS uses an Orthogonal Variability Model (OVM) [33]. Instead of defining variability internal to the responsibility structure, the dRS defines a separate variability model, the OVM. Defining a separate external variability model provides separation of concerns and helps in mitigating complexity of the responsibility components and their provide/required interfaces. The OVM defines domain and cross-domain variability for the modeled responsibility components by defining variation points and variants. A variation point represents a design decision which may vary from domain to domain or product to product either at design time or run-time. The supported variations of each variation
point are modeled as a set of variants. For instance, “caching” and “introduce concurrency” can be two variants of a variation point defined for a component responsible for planning and performing performance optimization. The bindings between variation points and variants may change at run-time to support run-time variability. For instance, a product initially launched with “caching” variant for performance optimization may switch to “introduce concurrency” variant at run-time. The run-time rebinding of variants and variation points are supported by defining abstract monitor and adapt interfaces between the managing and managed subsystems and their operating environment. The monitor and adapt interfaces are defined based on the MAPE-K feedback loop [27, 40] pattern.

To support domain design with and for reuse, the dRS has two sub-types: 1) General dRS, and 2) Specialized dRS. The general dRS is independent of any application domain, and is defined by following ASPL Design process described in Section 3.4. The specialized dRS is defined for a specific application domain by following Design Specialization process described in Section 4.4. Both types of the dRS model reference architectures to realize a self-adaptation property. Difference between the two is that the general dRS models an application domain independent reference architecture, and the specialized dRS models an application domain specific reference architecture. The general responsibility structure is defined as a result of the ASPL Design process described in Section 3.4. The ASPL design maps requirements from a general dQAS to responsibility components in the form of a general dRS. The resulting general dRS is reused by the Design Specialization process, see Section 4.4 to derive a number of specialized domain responsibility structures. Each specialized dRS a domain specific reference architecture that maps requirements from a specialized dQAS to the components of a specialized reference architecture. The resulting specialized dRS is reused by application engineering process, which is out of scope here, to derive application or product specific architectures.

A.3 Architecture Patterns and Tactics

The eARF provides design assistance in the form of architectural patterns and tactics. Architectural patterns and tactics encapsulate best design practices and knowledge that architects have been using for years to improve and simplify the design process [6]. An architecture pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined sub-systems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [11]. A large number of architectural patterns have been proposed over the time. These include client-server, piper and filter, layers, and blackboard among few others. Architecture patterns encapsulate high-level structure and design options to realize multiple system requirements, whereas tactics encapsulate more fine grained design options for individual quality attribute concerns [23].

Tactics are quality attribute specific design options that architects have been using for years to realize a quality attribute. For instance, “heartbeat” is a widely
used tactic to detect failed components, which is a primary concern for realization of the self-healing attribute. Bass et al. [6] described a collection of tactics for prominent quality attributes, such as availability, performance, modifiability and security. The eARF uses these tactics to identify, analyze, reason about and realize design decisions for self-adaptation properties. In addition to identify design options, the ASPL methodology also uses tactics to identify application domain independent requirements for in-scope self-adaptation properties in the ASPL requirements engineering process.

Patterns and tactics are closely related to each other as both assist architects in architectural reasoning and decision making. A pattern may use a set of quality attribute specific tactics to realize different quality attributes. For instance, the layered architecture pattern provides for performance, modifiability, and reusability quality attributes. This implies that a pattern can be used to realize multiple quality attributes, while a tactic is typically used to realize one quality attribute at a time.

The domain for the ADE and other ASPL design processes consists of self-adaptation properties which in general require monitoring, analysis, planning, and adapt functions [27]. Thus, responsibilities extracted from a dQAS for a self-adaptation property are more likely to fall in monitoring, analysis, planing, and effect or adapt categories. Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) feedback loop [27, 40] is one of the widely used architectural patterns to realize responsibilities for self-managing properties. Hence, the eARF recommends use of the MAPE-K feedback loop as a primary architectural pattern to structure architectural elements in a dRS.

A.3.1 MAPE-K Feedback Loop Pattern

Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) feedback loop [27] was proposed by IBM to realize self-managing systems. The MAPE-K feedback loop has established itself as a widely used approach to realize self-adaptation properties. The realization of self-adaptation properties typically involves feedback loop with five responsibilities: monitor (collect or detect), analyze (determine), plan (decide), execute (act), and knowledge (share or coordinate) [10]. These responsibilities correspond to the five elements of a MAPE-K feedback loop. A brief introduction of the MAPE-K loop elements is given below; more details can be found in [27].

Monitor: element is responsible for collecting and reporting data from a managed subsystem to the analyze element. Monitoring is usually done through probe interfaces provided by the managed system or some middleware in between the managed and managing systems.

Analyze: element analyzes data reported by the monitor element. The analysis is performed to detect whether the managed system under monitoring is in a desired state or not, and is there a need for adaptive actions or not? The planner element is invoked to plan adaptive actions if the managed system is not in a desired state.
Plan: element decides adaptation actions on the basis of the results or findings reported by the monitor and analyze elements. The adaption actions are generally an ordered set of measures planned to transform the managed system from its current state to a desired state.

Execute: element receives an adaptation plan and performs it on the managed system with the help of effector or adapt interfaces provided by some middleware or the managed system itself.

Knowledge: element serves as a knowledge base. The MAPE elements use the knowledge base to exchange knowledge including monitored data, analysis results, and planned adaptive actions. To support different functions of MAPE elements, the knowledge base may include additional knowledge such as architectural models, goal models, policies, and change plans or strategies based on proven architectural tactics and patterns.

Weyns et al. [40] argued that a single MAPE-K loop may not be suffice to manage all adaptation concerns in systems that are large, distributed, complex, and heterogeneous. For such systems, multiple MAPE-K loops should be applied to manage different parts of these systems. For some systems, realization of a single self-adaptation property may require multiple monitor, analyze or some other MAPE element. However, using multiple MAPE-K loops raises a question of how the elements from multiple MAPE loops coordinate and work together?
do they coordinate in a centralized, decentralized or hierarchical manner. To that end, Weyns et al. [40] presented a set of MAPE patterns that demonstrate how multiple interacting MAPE loops with different degrees of decentralization can be organized to achieve self-adaptation properties. These patterns don’t include the “knowledge” element in the feedback loop, that’s why the patterns are simply called MAPE patterns. The authors consider knowledge exchange as an important part of the loop, but they intentionally exclude the knowledge element to simplify complexity of the design space, and to avoid dependencies of the knowledge element on system domains and underlying infrastructure. Figure A.2 depicts an example of a MAPE pattern derived from [40]. According to this MAPE pattern, there can be only one occurrence of the MAPE loop with four elements (the one shown at the top), however, there can be multiple occurrences of the feedback loop (at the bottom) having the M (monitor) and E (execute) elements only. Details about the MAPE patterns and examples are out of scope here and can be found in [40].

A lesson learned from multiple interacting MAPE loops [40] is that the architects are not bound to use all the five elements of a MAPE-K control loop. Instead, based on domain or system requirements, any number of MAPE-K elements can be used individually or in combinations, for instance, analyze and plan element can be combined to a single element, as well as responsibilities of a plan element can be split across several plan elements.

As shown in the example MAPE pattern, Figure A.2, the MAPE feedback loop requires interactions between managed and managing subsystems. These interactions are made through monitor and execute elements which are responsible to collect details about the managed subsystem and perform adaptation actions, respectively. The eARF framework encapsulates monitoring and execute tactics. As described below, these tactics assist architects to model design decisions concerned with such interactions between managed and the managing subsystems.

### A.3.2 Monitoring Tactics

The monitoring tactics encapsulate state-of-the-art design options and measures to monitor software systems, their states and properties of interest. Following is a non-exhaustive list of such tactics which can be extended to incorporate new tactics.

(i) **Data Logging** A component, known as data logger, records, filters, and processes data to mine or retrieve useful information, for instance, when there is a change in a system state or property. The IBM’s “Log and Trace Analyzer (LTA)”\(^1\) tool uses the data logging tactic to analyze log files and troubleshoot the reported errors and exceptions.

(ii) **Profiling** Software profiling is a type of dynamic program analysis which is generally used for program analysis and optimization. It can be followed

as a program monitoring tactic to collect details about program behavior, such as resource utilization, processing time, etc. [36]. There are two common profiling strategies: 1) sampling and 2) instrumentation. A sampling based profiler interrupts program execution at specified intervals, and logs state of the program’s call stack. A problem with sampling based profiler is that some function calls may fall down through ‘holes’ of a sampling grid, and may not be seen in a profile. Instrumentation is often considered as a more ‘precise’ approach to profiling. This approach works by inserting a special code that performs analysis specific tasks, for example, monitor program state or relevant properties.

(iii) **Fault Detection** Ping/echo and heartbeat tactics are used to monitor whether a system or a system component is alive (operating) or not [6]. The ping/echo tactic works by sending a ping signal to the monitored system, and expecting an echo signal in return within a predefined time. If no echo signal is received within the predefined time, it is assumed that the monitored component has failed or not available at the moment. The heartbeat tactic works by requiring the monitored component to regularly emit heartbeat signals. The monitoring system keeps on listening to the heartbeats, and if there is no heartbeat within a specified time threshold, the component emitting heartbeats is assumed to be no more operational.

(iv) **Periodic Polling:** One component periodically, for instance, every 10 seconds, minutes, or hours, inspects or probes the underlying managed system for a property of interest, for instance if there is any change in the managed system’s state or behavior.

(v) **Event Monitoring:** An event monitor registers itself with the part of a managed system that serves as a source for changes or other events of interest. The event monitor is notified for the registered event when it happens, for instance, appearance of new update in an update repository.

### A.3.3 Execution Tactics

The execution tactics encapsulate design decisions and measures that support realization of the execute element in a MAPE-K feedback loop. The execute element performs adaptive actions that results in updating or changing parts or states of a managed subsystem. Thus, the execution tactics selected in the eARF are based on the concept of dynamic software updating [30]. Dynamic software updating technique enables a software system to update itself at runtime without halting and requiring a system to restart.

(i) **Quiescence** tactic suggests that before executing adaptive actions on a managed system, the managing system (execute element) should ensure that the managed system is in a stable state and the managing system mechanism does not interrupt an ongoing or running process in the managed system. Different mechanisms can be used to check if a system or
a part of a system is in a stable state or not. For instance, execution stack of a system can be inspected to know if it the target function or subsystem for adaptation is currently in execution or in stable state. If the execution stack does not have any reference to the target function, it is safe (stable) to perform adaptive actions. Further details about the tactic and techniques to check if a program or a component is in stable state or not can be found in [30].

(ii) **Rewriting Binary Code** tactic suggests that adaptive actions can be executed by rewriting binary code of a managed system. One such technique that allows rewriting of binary code is *binary redirection*. Binary redirection is performed by dynamically modifying binary code in a way that the code instructions that point to a function or procedure are changed to point to an updated version of that function or procedure [30].

(iii) **Use of Proxies, Intermediaries and Indirection Levels** tactic is based on the concept of proxies and suggests to introduce a proxy between a managed system and its clients. Instead of calling and using the managed system directly, the clients call an intermediary (proxy, middleware, etc.) that can dynamically direct or redirect the client requests to an adapted (updated) implementation of the managed system.

(iv) **Intrusion and Cooperation** tactic is based on the concept of *intrusion* by execute element of a managing system, and *cooperation* by a managed subsystem which is being updated or adapted. It means the managed system is aware of the execute element and provides support for it in the form of purposefully designed constructs, such as getter, setter methods or interfaces. Miedes et al. [30] specified three types of intrusion and cooperation. The first type defines special functions or procedures in both the managing, and the managed subsystems. The special functions defined in the managed system allows the managing system to get information about current state of the managed system, and modify it if needed. The managing system also defines functions which the managed system may invoke, for instance, to notify a change in its state or request an update. An example of this kind of intrusion and cooperation is the use of getter, setter methods. In the second type of intrusion and cooperation, the managing system requires the managed system to follow specific architecture, design principles, programming language, development infrastructure, and/or other constraints that force the managed system to be developed or behave in a specific manner. Requiring an application to follow Open Service Gateway initiative (OSGi ) [32] framework to support dynamic updates is an example of this type of intrusion and cooperation. The third type of intrusion and cooperation is based on meta-information. It requires the managed system to provide information about itself, its environment, goals, and related properties, which the managing system may use to plan and perform the adaptive actions.
Dynamic Linking or late binding suggests an adaptation mechanism that works by binding and rebinding system components or modules dynamically while a system’s source code is being linked and loaded. Modern programming languages support dynamic linking at run-time while a system is executing. Unlike static linking or binding that links all the translation units or object files of a program into a single executable file, the dynamic linking defers much of the linking process until a program starts executing. The modules or libraries are linked to the main executable file using a linking program known as dynamic linker. The dynamic linker is often a part of underlying operating system, otherwise a software system may define its own dynamic linker. A large number of object-oriented languages, most notably Java, and C++ support dynamic linking. C++ supports dynamic linking in the form of Dynamic Link Libraries (DLLs) based on the shared library concept. In Java, the dynamic linking process is implicit [16]. The java development environment does not require all the program elements (classes and interfaces) to be loaded and linked at compile and deployment time; new program elements can be loaded and linked on demand at run-time. The reflection API for the java language provides more advanced support for dynamic linking.

A.4 Analytical Framework

Figure A.3 gives a schematic overview of the analytical framework that enhances the eARF with support for rigorous reasoning. The framework defines artifacts and activities, some tool supported, that provide guidelines for architects to transform domain requirements to verified architecture models. The analytical framework models a system as a network of timed automata (NTA), which can be verified for a set of desired properties such as self-healing, self-optimization. The properties are expressed as queries in a temporal logic and verified using a model checker such as UPPAAL [8]. A timed automaton is a finite-state machine extended with clock variables that allow modeling timing aspects. A network of timed automata is a set of automata that can communicate through channels.

To model and verify design decisions, the analytical framework describes four core activities: 1) Identification, 2) Modeling, 3) Verification, and 4) Interpretation. The identification and modeling activities are supported by MAPE-K templates [24]. These activities derive a model which is a specification of architecturally significant requirements and quality attribute model properties. The MAPE-K templates encode design knowledge derived from modeling feedback loops of different self-adaptive systems in the form of a set of reusable templates that are composed as NTAs. The reuse of templates reduces the effort of transforming dQAS specifications to verifiable timed automata models.

The third activity, verification, simulates models and verifies if a model satisfies QA Model properties or not. Interpretation, as shown in Figure A.3, refers to post processing of the verification results. During interpretation, ar-
Figure A.3: Analytical framework to support rigorous reasoning in eARF

Architects identify design flaws in the model under verification, or compare results of candidate models. The interpretation activity feeds back to identification and modeling activities to refine the models. The modeling, verification and interpretation feedback loop continues until a verified architecture model is produced. A verified architecture model is one that satisfies all desired properties (requirements). Details about the above outlined four activities can be found in [3] with an example demonstration.
Bibliography


