Rigorous Architectural Reasoning for Self-Adaptive Software Systems

Nadeem Abbas*, Jesper Andersson†, Muhammad Usman Iftikhar‡, and Danny Weyns§
AdaptWise, Department of Computer Science, Linnaeus University, Sweden
Email: *nadeem.abbas@lnu.se, †jesper.andersson@lnu.se, ‡usman.iftikhar@lnu.se, §danny.weyns@lnu.se

***Abstract*—Designing a software architecture requires architectural reasoning, i.e., activities that translate requirements to an architecture solution. Architectural reasoning is particularly challenging in the design of product-lines of self-adaptive systems, which involve variability both at development time and run-time. In previous work we developed an extended Architectural Reasoning Framework (eARF) to address this challenge. However, evaluation of the eARF showed that the framework lacked support for rigorous reasoning, ensuring that the design complies to the requirements. In this paper, we introduce an analytical framework that enhances eARF with such support. The framework defines a set of artifacts and a series of activities. Artifacts include templates to specify domain quality attribute scenarios, concrete models, and properties. The activities support architects with transforming requirement scenarios to architecture models that comply to required properties. Our focus in this paper is on architectural reasoning support for a single product instance. We illustrate the benefits of the approach by applying it to an example client-server system, and outline challenges for future work.**

I. Introduction

A software system's ability to address its requirements depends on the quality of its architecture [1]. The software architecture is a result of a series of complex design decisions where architects analyze, reason about, and select among design options. In line with [2], we refer to these decision activities as architectural reasoning. Architectural reasoning challenges architects with selecting the best fit decisions for a combination of quality attribute model properties, frequently trading locally optimal selections for global optimality.

A Self-Adaptive Software System (SASS) is a system that is capable of changing its behavior and structure in response to changes in the environment, its requirements, and itself [3]. A self-adaptive system essentially comprises of two parts: a *managed subsystem* that deals with the domain concerns providing features to users, and a *managing subsystem* that monitors and adapts the managed system to achieve particular goals. As an example of self-optimization in a web service system, the managing system may track the performance of the service configuration selected by the workflow and change services of the configuration when the performance drops below a certain threshold. A common approach to realize the managing system is by means of a feedback loop that comprises four components: Monitor, Analyze, Plan, and Execute, that share common Knowledge (usually referred to as MAPE-K [4]). In our research, we study product lines of SASS, supporting strategic reuse in the design of SASS [5]. Self-adaptation combined with product-line engineering introduces two variability dimensions, development time variability and run-time variability, which add to the complexity of architectural reasoning. In previous work, we introduced an extended Architectural Reasoning Framework (eARF) [6] which encapsulates proven best practices and architectural knowledge to support architectural reasoning for SASS. However, a number of case studies showed that applying the eARF lacked verification mechanisms for more rigorous reasoning to ensure that the architectural design complied to the system requirements [6].

Formal methods are mathematically based languages, techniques, and tools for specification, design and verification of system properties [7]. The state of the art in architecture-based self-adaptation advocates use of formal methods to provide guarantees for desired system properties. However, formal methods are generally believed to be too difficult to apply [8]. Lamsweerde [9] points at lack of guidance and professional specialization requirements as root causes.

In this paper, we introduce an analytical framework that enhances eARF to deal with the lack of rigor in architectural reasoning. The analytical framework defines a set of artifacts and series of activities. Artifacts include a template to specify domain quality attribute scenarios, and templates to specify concrete models and properties. The activities support architects with transforming requirement scenarios to rigorous architecture models that comply to required properties. The models are specified in timed automata and properties in a temporal logic. The resulting models and properties allow verifying model compliance with the properties. We use Uppaal [10], a state-of-the-art toolbox for verification. Our focus in this paper is on architectural reasoning support for a single product instance. We illustrate the benefits of the rigorous reasoning process by applying it to an example client-server system, and outline challenges for future work.

The remainder of this paper is organized as follows. Section II gives the study's context: architectural reasoning and the eARF framework. This section also introduces the example application that we use for evaluation. Section III presents the analytical framework that defines a step-by-step process for modeling self-adaptive systems and verifying required properties, using a set of reusable templates. Section V positions our contribution with respect to related work. We conclude and discuss future work in Section VI.

II. Context of the Analytical Framework

Architectural reasoning is made up of activities that eventually produce a system's software architecture. The reasoning activities aim at finding a design that complies with the system's requirements (design properties). For a single software system, architects analyze, reason about, and decide about design options in the design space where quality requirements are the primary drivers for decisions.

A. eARF

In our research, we study architectural reasoning for product lines of self-adaptive systems, which introduces two variability dimensions – development time variability and run-time variability – creating a higher-order design space. Development time variability refers to traditional product line variability, which defines the scope of products that are supported by a product line. Run-time variability refers to mechanisms that can change a system's behavior and structure in response to changes in its environment, its requirements, and in the system itself [3]. Designing an architecture for a product-line of self-adaptive systems requires that architects are able to manage cross-domain and domain specific product requirements for a system that may adapt at run-time [5]. To support such complex analysis and reasoning, we developed an extended Architectural Reasoning Framework (eARF) [6].

The eARF leverages on existing frameworks [2] and provide tailored models, methods, and guidelines for architecture reasoning. The eARF uses domain Quality Attribute Scenarios (dQAS) [11] to specify domain requirements for self-adaptation. A dQAS extends a quality attribute scenario [1] with explicit support to model variability. The eARF supports architects with identifying product line variability and run-time variability, and mapping these to design solutions. For product line variability, the domain quality attribute scenarios are analyzed to find out design alternatives, which are mapped to a domain responsibility structure, i.e., a product-line architecture. To support run-time variability, the eARF offers a set of architecture patterns and tactics that guide architects to analyze domain quality attribute scenarios, identify architecturally significant adaptation requirements and how they can be mapped to architectural elements realizing self-adaptation.

We applied eARF and evaluated it in several case studies with final year Master students in Software Technology at Linnaeus University, Sweden. These case studies showed that the novice architects found it difficult to compare design alternatives. They also lacked confidence that the final architectural design complied to the system requirements. Hence, an analytical framework that enables a rigorous reasoning process, supported by objective measures, would increase the architects' confidence and enable them to provide evidence that design decisions result in models that comply to required properties. For more background information of the case studies we refer the interested reader to [12]. Based on this conjecture, we decided to enhance the eARF with the formally founded analytical framework that we present in this paper.

B. Znn.com

In this paper, we use znn.com[1], an exemplar system widely used in the self-adaptive systems community, as an example case to illustrate the analytical framework for architecture reasoning [13]. Znn.com is a web-based news service system serving multimedia news content; it uses a load-balancer to balance requests across a pool of replicated servers. The server pool size is dynamically adjusted to balance server utilization against service response time.

The business objective at znn.com is to serve news content within a specified maximum response time, while keeping the cost of the server pool within its operating budget. At times, the znn.com experiences sudden increase in the number of requests that it cannot serve adequately, even with a maximum pool size. To prevent the system from degradation or shutdown due to an increased load, znn.com opts to provide less resource demanding content during such peak periods.

Instead of making administrative staff responsible for the business objective by adapting the server pool size and the news content mode, znn.com introduces a self-optimization capability, which enables it to monitor the average response time and optimize the system through four adaptive actions:

1) Switch content mode from multimedia to text
2) Switch content mode from text to multimedia
3) Increment the server pool size, and
4) Decrement the server pool size

Table I shows a dQAS for znn.com to handle changes in the average response time. The dQAS uses fragments, parameters and constraints to specify product-line requirements with variability. The focus of this paper is on rigorous reasoning, hence we consider only one product, i.e., valid configuration [VC3] in Table I, which includes both adaptation variants [V1] (adapt the server pool) and [V2] (adapt content mode). In Section IV we zoom out and discuss implications on rigorous reasoning for multiple products of a software product-line.

III. Rigorous Architectural Reasoning

Figure 1 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 a dQAS 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. A timed automaton is a finite-state machine (comprising states and transitions that can be annotated, e.g., with labels and constraints) extended with clock variables that allow modeling timing aspects. A NTA consists of a set of automata that can communicate through channels. Properties can be expressed as queries in a temporal logic allowing verification of particular states of a model as well as paths of the model state. The analytical framework supports architects

[1] https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com/

Source (SO)	**[SO1]** Average response time
Stimulus (ST)	**[ST1]** Increase in the average response time **[ST2]** Decrease in the average response time
Artifacts (A)	**[A1]** Load-balancer
Environment (E)	**[E1]** Normal load **[E2]** High load
Response (R)	**[R1]** Increment the server pool size **[R2]** Decrement the server pool size **[R3]** Switch the server content mode from multimedia to textual **[R4]** Switch the server content mode from textual to multimedia
Response Measure (RM)	**[RM1]** Serve news content to customers within a response time of "x" seconds **[RM2]** Keep the server pool cost an operating budget of "y" MUs.
Variants (V) & Valid QAS Configurations (VC)	**[V1]** Adapt the server pool size **[V2]** Adapt content mode **[VC1]** V1 **[VC2]** V2 **[VC3]** V1 $\wedge$ V2
Fragment Constraints (FC)	**[Mandatory]** {SO1} $\wedge$ {ST1, ST2} $\wedge$ {A1} $\wedge$ {E1, E2} $\wedge$ {RM1} **[Variants VC1]** {R1, R2} $\wedge$ {RM2} **[Variants VC2]** {R3, R4} **[Variants VC3]** {R1, R2, R3, R4} $\wedge$ {RM2} **[Bindings VC1]** RM1.bind(x) $\wedge$ RM2.bind(y) **[Bindings VC2]** RM1.bind(x) **[Bindings VC3]** RM1.bind(x) $\wedge$ RM2.bind(y)

TABLE I: Self-Optimization dQAS for the Example Product-Line

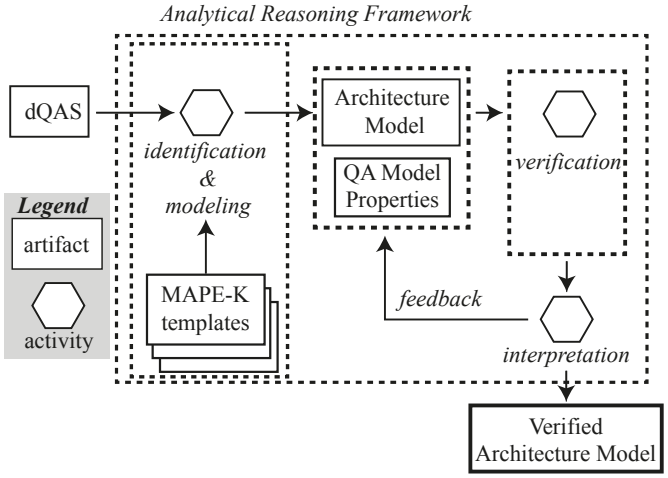


Fig. 1: Rigorous Reasoning in eARF

with transforming dQAS to NTA that can be verified for quality attribute properties.

The analytical framework describes four core activities: 1) Identification, 2) Modeling, and 3) Verification, and 4) Interpretation. Identification and modeling, are supported by MAPE-K templates [14]. These activities derive a model for the self-adaptive system, 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 the model satisfies QA Model properties. Interpretation, as shown in Figure 1, refers to post processing of the verification results. During interpretation, architects identify design flaws in a single model, or compare results of candidate models with each other. Interpretation feeds back to model refinements. Eventually a verified architecture model is complete. We discuss now the first three activities in detail. Interpretation of verification results is discussed in Section III-C.

A. Identification

Each automaton of an NTA models a subsystem by defining a template that can be customized and reused for different systems. In other words, a system model consists of a network of instantiated templates. Hence, the first step to model should identify such a set of subsystems (templates) from the requirements specification.

For the requirement analysis, we first make a pass through the first six dQAS compartments and identify functional requirements and quality properties. The purpose is to establish an initial subsystem candidate set. We exclude the variants & valid configurations and fragment constraints compartments because these are only concerned with variability. The challenge lies in finding the right subsystems. A dQAS is written in a natural language and hence the syntax and semantics are open, which complicates correct objective parsing. To provide guidance for identification, we use the structure of the quality attribute scenario. A self-adaptive system is expected to react when it detects that it is not accomplishing its goals. A quality attribute scenario is conceptually similar to a self-adaptive system, a stimulus (change) that trigger responses (adaptation). Responses are derived by an artifact (managing system) and

qualified by pre-conditions, post-conditions, and invariants. Below we describe how each part of a dQAS specification may be analyzed for subsystem candidates.

1) Analysis: The first two compartments, source and stimulus, are concerns for the monitor and analyzer in a MAPE-K loop. The stimulus is a condition that needs to be considered when it arrives at a system [1]. Stimulus conditions are typically written in two ways, directly as a quality attribute or indirectly in terms of quality attribute measures, so we should look for both. From these two compartments, we identify the quality property *response time*. This will be the trigger where our self-adaptive system will respond to changes.

Next, we look at the response the self-adaptive system executes if certain conditions hold. "The response is the activity undertaken after the arrival of the stimulus" [1]. In the response compartment, we find two main response variants, change the server pool size or switch server content mode. Our model will include subsystems that represents these variants.

The final part of the analysis focuses on conditions. To identify conditions, we focus on the environment and response measure compartments in a dQAS. The environment compartment describes conditions under which a specific stimulus occurs. It constitutes a set of pre-conditions that must hold for a stimulus. The dQAS specifies two environment conditions, normal and overloaded, both derived from the current number of client requests. The response measure compartment contains several candidates; a response time threshold, server pool cost, and a budget threshold. There is also a functional requirement concerned with serving news content.

The "artifact" is the final compartment, and it specifies the artifact that get stimulated. In the context of a dQAS and self-adaptive software systems, this corresponds to the managing system. The artifact decides if conditions hold that motivate adaptations to the underlying system it manages, derives the adaptations and executes them. The load-balancer is the artifact in the example dQAS. This completes the initial analysis.

2) Pruning: This step aims at making the initial candidate set more coherent and consistent. The initial candidate set may contain duplicates, subsystems identified more than once, and subsystems that can be combined or divided into other subsystems. Moreover, some of the identified subsystems might be renamed for consistency reasons. For the znn.com example dQAS, for example, we rename "Budget Threshold" to "Cost Threshold", and also remove duplicates of the "response time" subsystem. The pruning step for the znn.com results in the following set of subsystems: {Response Time, Load-Balancer, Client Requests, Server Pool Size, Content Mode, Response Threshold, Server Pool Cost, Cost Threshold}.

3) Allocation: In this step, the architects identify what is needed to realize the specified behavior. A self-adaptive system includes two types of primary subsystems, managed subsystem and managing subsystem. Besides the behavior, the self-adaptive system includes knowledge components in the managing system. The allocation step is intended to pinpoint where subsystems belong in the self-adaptive system. In the managed system, e.g., znn.com, we have probes and effectors to collect information and enact changes. In our candidate set, we have one probe subsystem, the Response Time, and two effector subsystems representing the Server Pool Size and Content Mode. The managing system is represented by the Load-balancer. The remaining subsystems belong to the knowledge part. This result in the following subsystem classification that we will model as timed automata.

- *Managed:* Response Time, Server Pool Size, Content Mode
- *Managing:* Load-Balancer
- *Knowledge:* Response Threshold, Server Pool Cost, Cost Threshold

B. Modeling

The next step is modeling the subsystems in each category as timed automata.

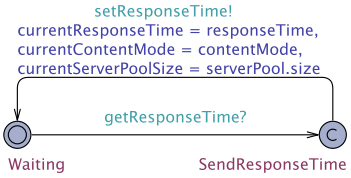
1) Managed subsystem models: We use four types of managed subsystem models that are based on the FORMS reference model for self-adaptive systems [15]: probe, effector, system state and system behavior, which results in four types of NTA templates. It is important to note that we do not model complete state or behavior of the managed subsystem, but only abstractions relevant to self-adaptation.

1) **Probe:** A Probe template models the managed system behavior that accesses state in managed system models. Probes connect to monitor templates of the managing system.
2) **Effector:** An Effector template models the behavior that manipulates state in managed system models. Effectors are invoked by the managing system's execute templates.
3) **SystemState:** A SystemState template is a representation of (a part of) the managed system's state.
4) **SystemBehavior:** A SystemBehavior template models behavior, which either represents events internal to the managed system or events in the system's environment.

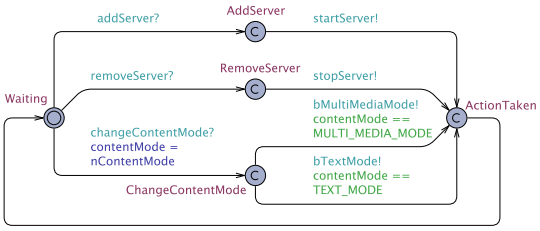
We start modeling by creating new templates for each subsystem in this category. We separate concerns according to the template types enumerated above. We name each template according to which subsystem it models. Continuing, we define (design) each template's control structure, i.e., its locations and edges. The control structure is defined by identifying what states a subsystem may have, and what triggers a transition from one state to another. For the example dQAS, a server in a server pool can be either in "executing" (i.e., added to the server pool) or "stopped" (i.e., removed from the server pool) state. A change in a system state is either triggered by an event, for example, increase in average response time, or by a clock. Event triggered transitions are modeled with a hand-shaking synchronization mechanism. A transition (an event) in one template triggers a transition in another template through a synchronization channel. Time triggered transitions are modeled using state invariants and time conditions [10]. For example, a system changes its state every 30 seconds.

Figure 2 depicts the resulting automata models for the managed subsystems. We use templates of all four types. Let us have a closer look at the datapump, Response time

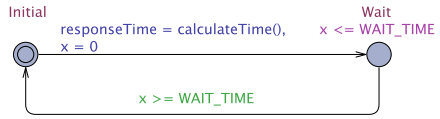
in Figure 2c. In the dQAS, there are two possible states representing an "increase" or a "decrease" in the average response time. Figure 2c depicts an automata that models these variations using the "calculateTime()" function. It simulates increase and decrease in the average response time by varying it cyclically between a minimum and maximum value. The calculateTime() function is invoked periodically using an invariant condition `x <= WAIT_TIME`, here "x" is a clock variable and "WAIT_TIME" is a constant time period defined to call the calculateTime() function. The probe template in Figure 2a accesses the currentResponse time.



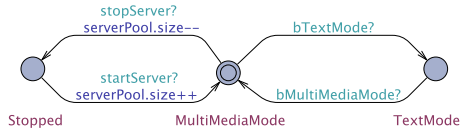
(a) The Probe Template for the Managed System



(b) The Effector Template for the Managed system



(c) The SystemBehavior Template for the Average Response Time



(d) The SystemState Template for the Managed system

Fig. 2: Managed Subsystem Models

The server pool is a collection of servers and their states. Thus, as shown in Figure 2d, we replicate "server" subsystem to model the server pool. A server may be in one of three states according to the dQAS; multi-media mode, text mode, and stopped. "Multi-media" is the default state a server enters to the server pool. In this state, it can transit to either "text" or "stopped" state, depending on the adaptive actions performed by the effector depicted in Figure2b.

Before moving to the managing subsystems, we model the knowledge subsystems. The managing subsystems templates will make use of this knowledge, for instance properties of the managed system, the environment and self-adaptation constraints in the dQAS.

2) Knowledge models: The knowledge subsystems are information abstractions required by managing subsystems to perform the adaptive actions. These subsystems are pure data abstractions, that is, they model no behaviors, thus they are modeled as data types.

The knowledge abstractions include information about managed system, such as response time threshold, server pool size. The managing subsystems may create additional working knowledge as a result of adaptive actions. Declaration 1 specifies a knowledge excerpt for the example dQAS. The server pool cost is abstracted as a maximum number of servers allowed in that budget and modeled as a constant declaration: `const int MAX_SERVERS = 10;`. The constant declaration is used because the budget will not change at run-time. The operating cost of the server pool will, however, change at run-time, depending on the number of active servers in the server pool. This is represented by the type "ServerPool" and corresponding scalar.

```
const int RESPONSE_THRESHOLD = 5;
const int TEXT_MODE=2,MULTI_MEDIA_MODE=1;//Content Mode
const int MAX_SERVERS = 10; //Server Pool Cost
typedef int[1, MAX_SERVERS] s_id; //Server id
typedef struct{
    int size;
}ServerPool;
ServerPool serverPool = {MAX_SERVERS};
```

Declaration 1: Managed System Knowledge

3) Managing system models: A self-managing system collects information from the system it manages and the environment. The information is analyzed to determine if an adaptation plan should be derived and executed [4]. The responsibilities for these activities are assigned to Monitor, Analyze, Plan, Execute, and Knowledge subsystems that form MAPE-K feedback loop. We use a set of MAPE-K templates to model managing systems [14]. We explain the step-by-step process of how we reuse the MAPE-K templates to model the "load-balancer" managing system for the example dQAS.

a) Monitor: The monitor template includes five steps; (1) monitor triggering, (2) collecting data, (3) preprocessing data, (4) updating working data, and (5) invoking analyze behavior(s). Figure 3a depicts specialization of the monitoring template. "Monitor triggering" is modeled as an invariant condition defined at the initial "waiting" state. The transition from "Waiting" to "RequestResponseTime" calls the probe in the managed system (see Figure 2a) to collect information about the current system response time, content mode, and server pool size. The monitor template then invokes the "updateKnowledge()" method to update the managing system knowledge. The monitor activity is finalized by invoking the "analysis" template by sending "analysis!" signal.

b) Analyze: The analyze template comprises four steps; (1) analyze triggering, (2) data collection, (3) analysis process, and (4) invoke planning. Figure 3b depicts the specialized analyze template which waits for an "analysis?" signal from the monitor automaton. The monitor template has already collected knowledge, so the analyze template directly invokes the "analyze()" function to determine whether an adaptation is

required or not. If an adaptation is required, it invokes planning and returns to the waiting state.

c) Plan: The plan template consists of two distinct steps; identification of the required type of plan, and plan creation, besides states for accepting invocation and signaling the execute subsystem. Figure 3c depicts the resulting automaton. The automata waits until it receives the "planning?" signal from an analysis template. It uses the available knowledge to identify the correct adaptation plan. In the example dQAS, the desired plan options are specified in the response element, along with desired measures in the "response measure" element. There are four plan options: (1) add server, (2) remove server, (3) switch to text mode, and (4) switch to multi-media mode. Depending on current response time, number of servers in the server pool, and the content mode, one of the plan variants is selected and the managing system knowledge is updated. The plan template completes by sending the "execute!" signal to the execute subsystem.

d) Execute: The execute template prepares execution, invokes the effector and performs a post-adaptation assessment. Figure 3d depicts the execute template specialized for the example dQAS. The execute automaton waits for the "execute?" signal and calls the "readPlan()" method to determine the planned adaptive action. The plan is then executed through the "effector" in the managed system, depicted in Figure 2b. For example, the executor signals "removeServer!" to the effector. Continuing, the effector receives the signal and sends a "stopServer!" signal to the "server" subsystem. On receiving the "stopServer?, signal, the server subsystem, which models the managed system state (depicted in Figure 2d), stops and removes a server from the server pool.

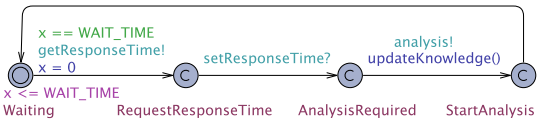
The modeling of the managing system completes the modeling of the self-adaptive software systems. We have now prepared our architectural model for verification. We may, for instance, use verification to determine if the design decisions we made are correct with respect to the system properties in the dQAS specification.

C. Verification

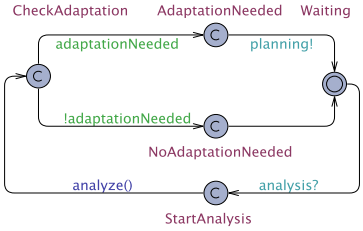
Once a dQAS has been transformed into a formal model, the architects can use a model checking tool to verify and reason about the architecture. We use the UPPAAL tool suite [10] for specification and verification. Architects may, however, use any other tool or method that supports verification of timed automata. For verification, architects must express properties of the quality attribute model as verification properties in a timed computation-tree-logic based query language [10]. The query language consists of state formulae and path formulae. A state formula is an expression that can be evaluated for a particular state of a model, whereas a path formula quantifies over paths in a model's state space. The state and path formulae are used to verify a model for deadlock-freeness, reachability, safety, and liveness properties.

The *deadlock free* property is used to verify that a model has no deadlocks. This is done by using a special state formula:

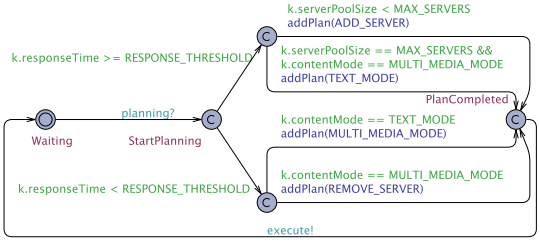
$$P1 : A[]\ not\ deadlock$$



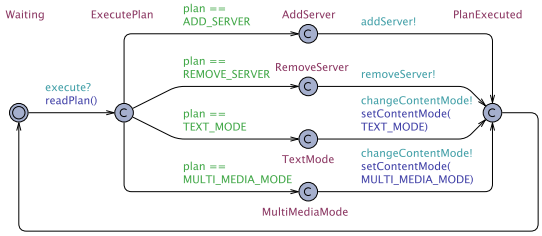
(a) Monitor Template Specialized for Load-Balancing Managing System



(b) Analyze Template Specialized for Load-Balancing Managing System



(c) Plan Template Specialized for Load-Balancing Managing System



(d) Execute Template Specialized for Load-Balancing Managing System

Fig. 3: Managing Subsystem Models

The *reachability* property verifies that a desired property is satisfied by any reachable state, i.e., there exists a path, starting at the initial state, that satisfies the property. The general syntax for this property is: $E <> \phi$ Where ϕ is a state formula for the property. Reachability properties are considered weak, however, safety properties can often express equivalent properties. A *safety* property verifies model invariants, i.e., a condition is true and reachable in all states. The general syntax for this property is: $A[]\,\phi$. For the example dQAS, we specify a safety property to verify that there will be at least one server running in the server pool at all times, and that the server pool size will never exceed the limit given by the available budget. The path formulae for this property in the query language is:

$$P2 : A[] \quad (serverPool.size >= 1 \\ \&\& serverPool.size <= MAX\_SERVERS)$$

The *liveness* property is used to verify that some criteria

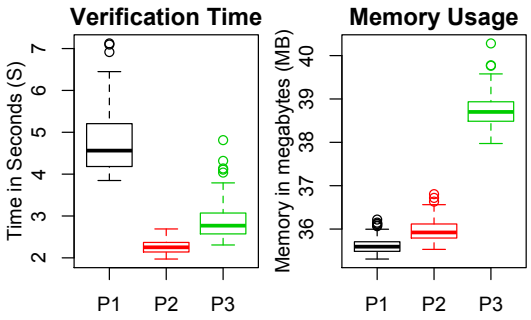


Fig. 4: Verification overhead.

will eventually be satisfied. Such properties are specified as $\phi - - > \psi$ to verify that whenever ϕ is satisfied, ψ will eventually be satisfied. For the example dQAS, we use below specified liveness property, P3, to verify that if the execution component of a load-balancing managing system executes the multi-media mode adaptation plan, every server in the server pool will eventually change its state to multi-media mode.

$$P3: \quad Executor.MultiMediaMode - - > forall(s : s\_id)Server(s).MultiMediaMode$$

A successful verification indicates that the behavior for that property has been correctly specified. Such evidence will increase architects' confidence in the model's correctness. An unsuccessful verification informs architects that a model does not satisfy the property. This enables architects and developers to detect problems earlier, which results in improved model quality. Moreover, a failed verification triggers architects to analyze root-causes, reason about, and reconsider the architectural model and design decisions. Tracing flaws in models can be tedious and time-consuming. UPPAAL includes a simulator that enables architects to visually examine the possible dynamic executions of the modeled system. The visual examination enables architects to comprehend a model and pin-point potential problem spots by inspecting subsystems, paths, and states.

In Figure 4, we plot two graphs representing time and resource consumption for 100 verifications of the above described properties ($P1, P2$ and $P3$) on a 2,5 GHz Intel Core i7 processor, and 16 GB 1600 MHz DDR3 RAM. Verification times (seconds) and memory usages (10s of MB) for the example case are low; however, these properties may increase significantly with a growing size of the state space, e.g., for a higher number of variation points and variants.

IV. Rigorous Reasoning for Product-lines

The research presented in this paper is part of a long-term research effort to study and develop a comprehensive methodology for the design of software product-lines of self-adaptive systems. In this paper, we proposed a rigorous reasoning approach and used a single product instance as illustration. Reasoning about a product-line and its assets requires some additional support compared to a single product.

In particular, two principal mechanisms that need to be added to a domain reasoning framework are fragment specification and fragment composition. In domain engineering activities architects will apply the rigorous reasoning to derive the product-line architecture and variants, not for products. In a dQAS, fragments model variants and constraints control how they are combined into products. The reasoning framework must mirror this approach allowing for specification of fragments and providing support for composition of fragments into valid model configuration.

In a product-line setting, domain architects need to define fragments of automata that match the product-line's feature models. The automata fragments are composed and combined with product specific automata designed by product architects. The domain engineers may still verify properties for domain assets, while product specific assets and properties will be verified by product architects. Fragment models and quality attribute model properties are product-line assets and will be managed as such. An indirect effect of the proposed approach is reuse, which will reduce time and increase quality for rigorous architecture reasoning.

V. Related Work

Bass et al. [2] proposed reasoning frameworks to analyze and reason about behavior of a system. The proposed reasoning framework structure uses analytic theories to analyze and reason about the behavior of a system with respect to specific quality attributes. This work proposes an analytic theory for self-adaptation properties, which includes support for formal verification both at design-time and at run-time. This theory may be used as the fundamental theory to reason about different quality properties realized through self-adaptation.

Our work also addresses reasoning across products and product domains. Dynamic Software Product Lines [16] is an emerging research theme that studies design and development of self-adaptive systems using product-line engineering approach. The use of formal methods for verification and validation is gaining attention particularly in the self-adaptive systems [17] and the software product-lines engineering communities [18], [19], [20]. The novel feature of the work presented herein is the combination of the two, i.e., formal methods in support of architectural design for dynamic software product lines.

An important challenge in self-adaptive systems engineering is to provide assurances that the desired adaptation properties are satisfied. To address this challenge, state of the art in self-adaptive systems suggests use of formal methods. Weyns et al. [17] surveyed use of formal methods in self-adaptive systems. According to the survey results, 60.0% of the studies use formal methods for reasoning purpose, and majority of the authors use a formal specification to reason about the design of a self-adaptive system. However, there were only few studies which employ formal methods to actually provide evidence for the self-adaptive properties of interest. Moreover, most of the existing approaches, including [21], [15], [22], [23], do not provide detailed guidance to transform requirements specifications into formal models. Due to the

mathematical origin, formal methods are considered difficult to understand and work with [8] particularly when there are no explicit guidelines on how to transform informal specifications and models into formal ones. This study contributes with guidelines for the three activities that transform behavioral requirement specifications into formal models.

In software product-line engineering, formal methods are applied to express semantics of feature models [18], [19] and to verify the modeled semantics [20]. Schobbens et al. [18] proposed a generic formalization approach that provides basis for more rigorous investigations of feature models. Czarnecki et al. [19] used context-free grammar to define semantics of feature models. An important issue in product-line engineering is scalability, i.e., support for potential growth in the number of products and required features. Classen et al. [20] contributed a model checking technique which support scalable modeling and efficient verification. We have not yet investigated this aspect of product-line engineering. It is, however, part of our future work.

VI. Conclusions

In this paper we introduced an analytical framework for rigorous architectural reasoning of self-adaptive systems. We have described the framework activities: identification, modeling, and verification step-by-step and illustrated it with the znn.com exemplar.

Future work includes several steps. First, we are currently planning a controlled experiment that aims at comparing the formal extension of eARF with a competing approach. Second, we plan to provide tool support to better guide architects when applying the formal eARF approach. Such support includes the automation of (parts of) the reasoning process, for example, managing automata fragment specification and fragment composition A particular challenge will be scalability to manage a potentially high number of valid product configurations. Third, we plan to study and develop a more formalized language for expressing the fragments of dQAS, which is particularly relevant for automating parts of the reasoning process. This is two-edged sword, as we want to preserve sufficient expressiveness, while providing machine-readable specifications that can be interpreted automatically. Fourth, as uncertainty is an inherent property of self-adaptive systems, the models and properties verified at design-time may become invalid at run-time. One interesting way to tackle this challenging problem is exploiting "executable formal models at run-time" [24]. In this advanced approach, the verified models of the self-adaptive system together with the formally specified requirements are kept alive at run-time, enabling to continue verification after deployment when new knowledge becomes available to resolve uncertainties. Fifth, we plan to consolidate our expertise with applying the eARF framework in the form of a library of rigorous models of commonly used architectural tactics and patterns to deal with particular quality requirements in product lines of self-adaptive systems.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston, MA, USA: A-W Longman Publishing Co., Inc., 2003.

[2] L. Bass, J. Ivers, M. Klein, P. Merson, and K. Wallnau, "Encapsulating quality attribute knowledge," in *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, ser. WICSA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 193–194.

[3] R. De Lemos, H. Giese, H. A. Müller *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[4] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[5] N. Abbas and J. Andersson, "Harnessing variability in product-lines of self-adaptive software systems," in *Proc. of the 19th International Conference on Software Product Line*. ACM, 2015.

[6] N. Abbas and J. Andersson, "Architectural reasoning support for product-lines of self-adaptive software systems - a case study," in *European Conference on Software Architecture*, ser. LNCS. Springer, 2015, vol. 9278.

[7] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, Dec. 1996.

[8] A. Hall, "Seven myths of formal methods," *Software, IEEE*, vol. 7, no. 5, pp. 11–19, Sept 1990.

[9] A. v. Lamsweerde, "Formal specification: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 147–159.

[10] G. Behrmann and others., "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems*, ser. LNCS. Springer Berlin Heidelberg, 2004, vol. 3185.

[11] N. Abbas, J. Andersson, and D. Weyns, "Modeling variability in product lines using domain quality attribute scenarios," in *Proceedings of the WICSA/ECSA 2012 Companion Volume*. New York, NY, USA: ACM, 2012, pp. 135–142.

[12] N. Abbas, "eARF evaluation," Dept. of Computer Science, Linnaeus university, Tech. Rep., 2016, http://homepage.lnu.se/staff/janmsi/eARFEvaluationTR.pdf.

[13] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *Software Engineering for Adaptive and Self-Managing Systems*, May 2009.

[14] D. G. D. L. Iglesia and D. Weyns, "MAPE-K formal templates to rigorously design behaviors for self-adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 10, no. 3, pp. 15:1–15:31, Sep. 2015.

[15] D. Weyns, S. Malek, and J. Andersson, "Forms: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 1, pp. 8:1–8:61, May 2012.

[16] S. Hallsteinsen *et al.*, "Dynamic software product lines," *IEEE Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[17] D. Weyns *et al.*, "A survey of formal methods in self-adaptive systems," in *Proc. of the Fifth International C* Conference on Computer Science and Software Engineering*. New York, NY, USA: ACM, 2012.

[18] P. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature diagrams: A survey and a formal semantics," in *Requirements Engineering, 14th IEEE International Conference*, Sept 2006, pp. 139–148.

[19] K. Czarnecki *et al.*, "Staged configuration using feature models," in *Software Product Lines*, ser. LNCS. Springer Berlin Heidelberg, 2004, vol. 3154, pp. 266–283.

[20] A. Classen *et al.*, "Model checking lots of systems: Efficient verification of temporal properties in software product lines," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010.

[21] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006.

[22] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," *J. Syst. Softw.*, vol. 74, no. 1, pp. 45–54, Jan. 2005.

[23] A. Filieri, C. Ghezzi, and G. Tamburrelli, "A formal approach to adaptive software: Continuous assurance of non-functional requirements," *Form. Asp. Comput.*, vol. 24, no. 2, pp. 163–186, Mar. 2012.

[24] M. U. Iftikhar and D. Weyns, "ActivFORMS: Active formal models for self-adaptation," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS 2014. New York, NY, USA: ACM, 2014, pp. 125–134.