Synthesis and Quantitative Verification of Tradeoff Spaces for Families of Software Systems

Javier Cámara, David Garlan, and Bradley Schmerl

Carnegie Mellon University, Pittsburgh PA 15213, USA,
{jcmoreno, garlan, schmerl}@cs.cmu.edu

Abstract. Designing software subject to *uncertainty* in a way that provides guarantees about its run-time behavior while achieving an acceptable balance between multiple extra-functional properties is still an open problem. Tools and techniques to inform engineers about poorly-understood design spaces in the presence of uncertainty are needed. To tackle this problem, we propose an approach that combines synthesis of spaces of system design alternatives from formal specifications of architectural styles with probabilistic formal verification. The main contribution of this paper is a formal framework for specification-driven synthesis and analysis of design spaces that provides formal guarantees about the correctness of system behaviors and satisfies quantitative properties (e.g., defined over system qualities) subject to uncertainty, which is factored as a first-class entity.

Keywords: Tradeoff analysis, uncertainty, architectural style, architecture synthesis, formal guarantees, quantitative verification, probabilistic model checking

1 Introduction

Engineering modern software-intensive systems requires engineers to explore design spaces that are often poorly understood due to their complexity and different kinds of *uncertainty* about the behavior of their constituent components [8] (e.g., faults, network delays). Achieving a good design with behavioral guarantees and a balance between extra-functional concerns is challenging – especially when the context that the system will run in contains unknown attributes that are hard to predict. Designing for this context is as often a matter of luck as it is principled engineering.

Design decisions frequently involve the selection and composition of loosely-coupled, pre-existing components or services with different levels of quality (e.g., of reliability, performance) that may be offered by independent providers. For instance, modern robotic software systems consist of a set of processes running in components, potentially on a number of different hosts, connected at run time in a peer-to-peer topology [22]. Different implementations of these components (e.g., for navigation, planning) offer different levels of energy consumption, reliability, or accuracy. Similarly, service-based systems are built by composing third-party services with different levels of availability, performance, and cost [18]. Quality attributes of constituent components in such systems are often subject to *uncertainties* introduced by nondeterministic behaviors of individual components (e.g., derived from the lack of control over system components in the cloud, humans-in-the-loop, or physical interactions in cyber-physical

systems) that can be captured in the form of probability distributions (e.g., over the response time of a Web service, fault occurrence). For a designer, *it is difficult to envisage how these uncertainties will affect overall system behavior and qualities, despite the fact that they can sometimes have a remarkable impact on them.*

Often, design spaces are also constrained by the need to design systems within certain patterns or constraints that comprise an architectural style. Architectural styles [23] characterize the design space of families of software systems in terms of patterns of structural organization, defining a *vocabulary* of component and connector types, as well as a set of *constraints* on how they can be combined. Styles help designers constrain design space exploration to within a set of legal structures that the system must conform to. However, while the structure of a system may be constrained by some style, there is still considerable design flexibility left for exploring the tradeoffs on many of the qualities that a system must achieve.

Formal characterization of architectural styles combined with formal methods like Alloy [11] have proved to be a valuable tool to aid designers in exploring rich solution spaces, by synthesizing possible system configurations that satisfy the constraints imposed by a given architectural style [3, 7, 19]. However, these solutions tend to focus on structural properties, and when available, analysis of system behaviors and qualities are performed separately. So, these approaches are limited in their ability to consider interactions between behavioral properties and qualities (e.g., impact of failure in serving a request and a subsequent retry on overall system performance). Moreover, the approaches that explore non-structural properties tend to be based either on dynamic analysis or simulations. Such approaches *cannot exhaustively explore the state space of design alternatives or provide formal guarantees* that encompass both their behavior and qualities (both in general, and in particular, in the presence of *uncertainties*).

Architects need tools and techniques that can help them explore this complex design space and guide them to good designs. Providing such tool support demands investigating questions such as: (i) how to integrate formal descriptions of structural, behavioral, and quality aspects of design alternatives to enable integrated reasoning about all these aspects, and (ii) how to effectively streamline the exploration of the solution space while providing formal guarantees about solutions in the presence of uncertainty (e.g., with respect to correctness of behaviors, or quantitative and structural constraints).

This paper explores these questions by introducing a formal framework that enables the: (i) exhaustive *exploration of a rich space of design alternatives* by automatically synthesizing architecture configurations that satisfy the constraints imposed by an architectural style, and (ii) provision of *formal guarantees with respect to the functional behaviors and qualities* (i.e., extra-functional properties) of configurations by analyzing exhaustively the state space of each configuration's behavior. Our framework explicitly considers *interactions between functional behaviors and extra-functional properties* while factoring in *uncertainty* as a first-class entity.

The framework is grounded on two related formalisms: (i) predicate logic and sets capture the structural aspects of system configurations, and (ii) probabilistic automata and formal quantitative verification (e.g., probabilistic model checking [15]) capture behavior and qualities.

The key novelty of our approach *is that it is the first, to the best of our knowledge, that combines automatic synthesis of design alternatives with quantitative formal verification that factors in uncertainty as a first-class entity*. This combination is enabled by the seamless integration of different types of models by means of common abstractions that enable reasoning about different types of properties in a combined manner. More specifically: (i) interaction points (e.g., ports) on the component-and-connector view of configurations correspond to synchronization points of component and connector behaviors, (ii) uncertainties are captured as probabilities in the behavior models of components and connectors, and (iii) reward structures built on behaviors enable reasoning about quantitative aspects of system behaviors (e.g., qualities). We implemented our approach in a prototype tool that uses a back-end based on Alloy and the PRISM probabilistic model checker [16]. We illustrate the approach on a Tele Assistance System (TAS) [25] for the validation of service compositions.

The rest of this paper is organized as follows: Section 2 provides an overview of our approach. Section 3 describes the TAS exemplar. Next, Section 4 describes the formalization of models employed by our approach. Section 5 details our approach, Section 6 presents results, and Section 7 overviews related work. Finally, Section 8 presents some conclusions and future work.

2 Overview of the Approach

Finding system configurations in an architectural style that satisfy a set of formal guarantees with respect to their behavior and qualities requires appropriate models and mechanisms to: (i) systematically generate configurations in the style, and (ii) formally verify their behavior and qualities. To achieve this goal, we propose a formalization of architectural style extended with behavioral types that specify the abstract behavior of components and connectors, as well as quantitative aspects via reward structures built on their behavioral descriptions (described in Section 4).

Based on our formalization, our approach for design space exploration consists of three stages (Figure 1):

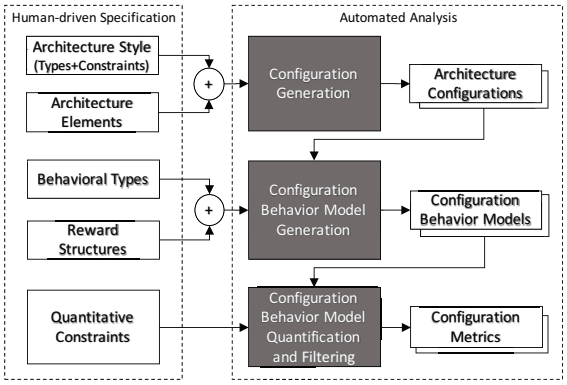


Fig. 1. Overview of the approach.

Configuration generation (Section 5.1), during which a set of configurations that satisfy a set of structural constraints is generated. This process takes as input the description of an architectural style formalized as a set of constraints in predicate logic defined over abstract types (e.g., those imposed by the style, such as *a component of type X can only be connected to a component of type Y*) and a set of concrete architectural element definitions (i.e., the different instances of candidate components and

connectors that can be employed to realize the architecture). The output is the collection of system configurations that satisfy the style constraints.

Configuration behavior model generation (Section 5.2), during which a set of behavioral models that refine the configurations obtained in (1) is generated. This process takes as input: (i) the set of concrete architecture element definitions, (ii) the configurations generated in (1), and (iii) the set of *behavioral types*[1] that capture the behavior of each abstract type in the architectural style. For every configuration, the behavior of each concrete component and connector is instantiated using the behavioral types of their corresponding abstract types. To realize the binding among components and connectors in the behavioral model (via synchronization actions), we employ the topological information of the graph from the system configuration. Note that, while the behavioral type is shared among all component (or connector) instances of the same type, their actual behavior can differ due to the specific attributes of the instance that parameterize its behavior (e.g., response time for a service, or number of retries after a failed service invocation). The behavioral model of a configuration is constructed as the parallel composition of the behavior of all the instances in the configuration.

Quantification, filtering and ranking (Section 5.3), during which behavioral and quantitative properties are checked on the configuration behavioral models. This step filters out configurations that do not meet a set of properties and constraints imposed by designers, which may include: (i) behavioral properties (e.g., safety, liveness), and (ii) quantitative constraints (e.g., on quality attributes). This stage also allows factoring probabilistic aspects into the analysis of behavioral and quantitative properties, as well as solution selection that optimizes quantitative properties.

3 Motivating Scenario

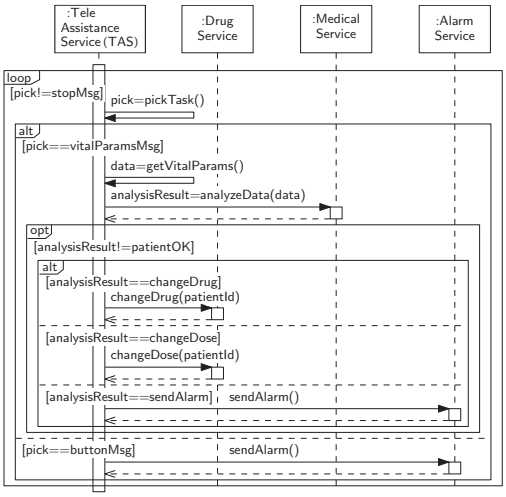
We illustrate our approach the TAS exemplar system [25], whose goal is tracking a patient's vital parameters to adapt drug type or dose when needed, and taking actions in case of emergency. The system combines three service types in a workflow (Figure 2).

When TAS receives a request that includes the vital parameters of a patient, its *Medical Service* analyzes the data and replies with instructions to: (i) change the patient's drug type, (ii) change the drug dose, or (iii) trigger an alarm for first responders in case of emergency. When changing the drug type or dose, TAS notifies a local pharmacy using a *Drug Service*, whereas first responders are notified via an *Alarm Service*.

The functionality of each service type can be implemented by a number of providers that offer the service with different levels of performance, reliability, and cost (Figure 2.a). The metrics employed for the different quality attributes in TAS are the percentage of service failures for reliability, and service response time for performance.

In this context, finding an adequate design for the system entails understanding the tradeoff space by finding the set of system configurations that satisfy: (i) structural constraints imposed by the style (e.g., the *Drug Service* should not be connected to an *Alarm Service*), (ii) behavioral correctness properties (e.g., the system is eventually

[1] Although the notion of behavioral type is more general [21], we employ the term to refer to an abstract state machine specification capturing the behavior of an architectural abstract type.



Service	Name	Fail. rate (%)	Resp. time (ms.)	Cost (usd)
S1	Med. Service 1	0.06	22	9.8
S2	Med. Service 2	0.1	27	8.9
S3	Med. Service 3	0.15	31	9.3
S4	Med. Service 4	0.25	29	7.3
S5	Med. Service 5	0.05	20	11.9
AS1	Alarm Service 1	0.3	11	4.1
AS2	Alarm Service 2	0.4	9	2.5
AS3	Alarm Service 3	0.08	3	6.8
D1	Drug Service	0.12	1	0.1

(a) Properties of TAS service providers.

Name	Description
R1	The average failure rate should not exceed 0.03%.
R2	The average response time should not exceed 26 ms.
R3	Subject to R1 and R2, the cost should be minimized.

(b) Example of quality requirements.

Fig. 2. Tele assistance service workflow, service provider properties, and quality requirements.

going to provide a response – either by dispatching an ambulance or notifying the pharmacy about a change), and (iii) quality requirements, which can be formulated as a combination of quantitative constraints and optimizations (Figure 2.b).

Generalizing from this scenario, the problem to solve is: "*Given an architectural style* A*, a set of concrete architecture elements* E*, a specification of correct behaviors* B*, and a set of quality requirements* Q*, find the set of system configurations combining elements of* E *that: (i) conform to style* A *(i.e., satisfy its structural constraints), (ii) satisfy the specification of correct behaviors* B *(i.e., safety and liveness properties), and (iii) maintain the desired level and/or optimize a set of quality goals specified by* Q*.*"

Exploring the design space to find the best possible configurations that conform to the style goes beyond the mere instantiation of architectural types, and entails flexibility when envisaging design alternatives that may not always be obvious to a human designer. An example in the context of TAS is allowing invocation of multiple alarm services concurrently. This may of course increase the cost of operating the system, but can also potentially reduce the response time and increase the reliability of the system (the combined probability of multiple alarm services failing is much smaller than the probability of failure of each individual alarm service).

In the next section we describe our formal model, and then detail our approach for design space exploration in Section 5.

4 Formalizing Structure, Behavior, and Qualities

4.1 Architectural Style, Configurations, and States

We characterize the possible structures of a family of systems that are related by shared structural and semantic properties employing an *architectural style* [23].

Definition 1 (Architectural Style). *Formally, we characterize an architectural style as a tuple* $(\Sigma, \mathcal{C}_S)$*, where:*

– $\Sigma = (CompT, ConnT, \Pi, \Lambda)$ *is an architectural signature, such that:*
  - $CompT$ *and* $ConnT$ *are disjoint sets of component and connector types.*
  - $\Pi : (CompT \cup ConnT) \rightarrow 2^{\mathcal{D}}$ *is a function that assigns sets of symbols typed by datatypes in a fixed set* $\mathcal{D}$ *to architectural types* $\kappa \in CompT \cup ConnT$. $\Pi(\kappa)$ *represents the properties associated with type* κ. *To refer to a property* $p \in \Pi(\kappa)$, *we simply write* $\kappa.p$. *To denote its datatype, we write* $dtype(\kappa.p)$.
  - $\Lambda : CompT \cup ConnT \rightarrow 2^{\mathcal{P}} \cup 2^{\mathcal{R}}$ *is a function that assigns a set of symbols typed by a fixed set* $\mathcal{P}$ *to components* $\kappa \in CompT$. *This function also assigns a set of symbols in a fixed set* $\mathcal{R}$ *to connectors* $\kappa \in ConnT$. $\Lambda(\kappa)$ *represents the ports of a component (conversely, the roles if* κ *is a connector), which define logical points of interaction with* κ*'s environment. To denote a port/role* $q \in \Lambda(\kappa)$, *we write* $\kappa :: q$.
– $\mathcal{C}_S$ *is a set of structural constraints expressed in a constraint language based on first-order predicate logic in the style of Acme [9] or OCL [24] constraints (e.g.,* $\forall$ t:AssistanceServiceT •$\exists$ a:AlarmServiceT • connected(t,a) *– "every tele assistance service must be connected at least to one alarm service").*

For the remainder of this section, we assume a fixed universe $\mathcal{A}_\Sigma$ of architectural elements, i.e., a finite set of components and connectors for Σ typed by $ConnT \cup CompT$. For a given architectural element $c \in \mathcal{A}_\Sigma$, we denote its type as $type(c)$.

A *configuration* is a graph that captures the topology of a feasible structure of the system in the style.

Definition 2 (Configuration). *A configuration in an architectural style* $(\Sigma, \mathcal{C}_S)$, *given a fixed universe of architectural elements* $\mathcal{A}_\Sigma$, *is a graph* $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ *satisfying the constraints imposed by* $\mathcal{C}_S$, *where:* $\mathcal{N}$ *is a set of nodes, such that* $\mathcal{N} \subseteq \mathcal{A}_\Sigma$, *and* $\mathcal{E}$ *is a set of pairs typed by* $\mathcal{P} \times \mathcal{R}$ *that represent* attachments *between ports and roles.*

A *system state* is the combination of a system configuration, along with an assignment of values for the properties of the nodes in the configuration graph.

Definition 3 (Σ-system State). *A* Σ*-system state* s *is a pair* $(\mathcal{G}, \lambda)$, *where* $\mathcal{G}$ *is a system configuration, and* λ *is a function that assigns a value* $[\![c.p]\!]^s$ *in the domain of* $dtype(\kappa.p)$ *to every pair* $c.p$, *such that* c *is a node of* $\mathcal{G}$, $\kappa = type(c)$, *and* $p \in \Pi(\kappa)$. *The set of all* Σ*-system states is denoted by* $\mathcal{S}_\Sigma$.

Example 1. We can characterize the family of TAS systems by a style with the following architectural signature:

CompT = {MedicalServiceT, DrugServiceT, AlarmServiceT, AssistanceServiceT}
ConnT = {HttpConnT}
Π = {(MedicalServiceT, {FailRate, RespTime, Cost}), ...}
Λ = { (MedicalServiceT, {analyzeDataPS}), (HttpConnT, {CallerR, CalleeR}),
(AssistanceServiceT, {changeDrugPTS, changeDosePTS, sendAlarmPTS, analyzeDataPTS}),
(DrugServiceT, {changeDrugPD, changeDosePD}), (AlarmServiceT, {sendAlarmPAS}) }

Employing the elements of that signature, we can specify a set of structural constraints that the style imposes on valid configurations (c.f. Listing 1.1).

Figure 3 depicts a sample TAS configuration with service instances TAS1, S1, D1, and AS2 (c.f. Fig. 2.a). The connectors are instances of the http connector type (HttpConnT) for each of the operations that are invoked by the assistance service TAS1 to change drug type or dose in D1, invoke an alarm in AS2, and analyze patient data on S1, connecting the corresponding ports on the component instances.

4.2 Behavior

To extend our formalization of architectural style with behaviors, we introduce the notion of *behavioral type*, characterized as a state machine that captures the abstract behavior of an architectural type in a given style.

Our instantiation of behavioral type is inspired by discrete-time Markov chains (DTMC), although it can be easily adapted to other formalisms like Markov decision processes (MDP) or probabilistic timed automata (PTA) to capture aspects such as fully nondeterministic choices or continuous time.

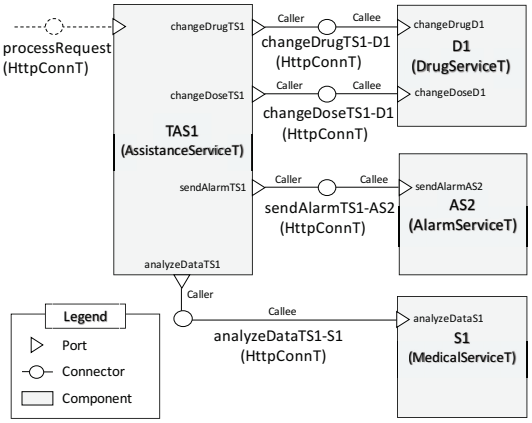


Fig. 3. Sample TAS configuration.

Definition 4 (Behavioral Type). *The behavioral type of an architectural type $\kappa \in CompT \cup ConnT$ is a tuple $(S_\kappa, s_i, P_\Lambda)$, where S_κ is κ's state space, characterized by the set of all possible value assignments for properties $\Pi(\kappa)$, $s_i \in S_\kappa$ is an initial state, and $P_\Lambda : S_\kappa \times S_\kappa \to [0,1] \times (\Lambda(\kappa) \cup \{\bot\})$ is a transition probability matrix extended with ports (if κ is a component) or roles (when κ is a connector).*

In the definition above, each element $P_\Lambda(s, s')$ yields: (i) the probability of making a transition from state s to state s', and (ii) the port/role (if any) on which the architectural element typed by κ interacts with its environment when the transition between s and s' occurs. From a behavioral standpoint, ports and roles define potential synchronization points for the interaction of different architectural elements in a configuration. We denote the behavioral type of an architecture element $c \in \mathcal{A}_\Sigma$ as $btype(c)$.

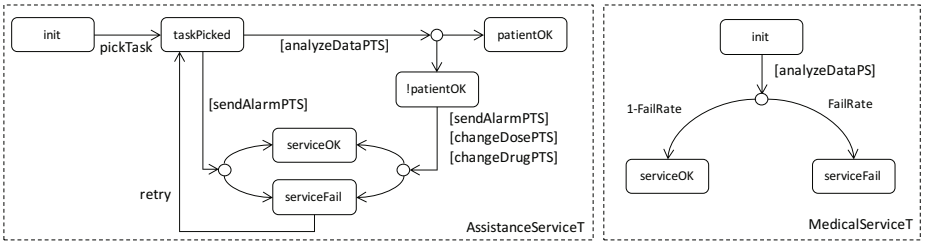


Fig. 4. AssistanceServiceT and MedicalServiceT behavioral types.

Example 2. Figure 4 depicts the abstract behavior specification of the AssistanceServiceT and MedicalServiceT architectural types. Transition labels represent internal actions, which can be internal to the component (e.g., pickTask after the initial state in the assistance service), whereas transition labels between brackets denote potential interactions with the environment. Branching transitions (denoted by a circle) indicate a probabilistic choice, where each branch is labeled by a probability (e.g., the medical service captures the probability of the service invocation failing with a branching

transition parameterized by the value of property MedicalServiceT.FailRate and its complementary). Unlabeled branching transitions implicitly specify a uniform probability distribution. Non-branching transitions indicate probability 1.

The behavior model of a configuration is obtained by instantiating the behavioral type all architecture elements in the configuration (c.f. Section 5.2), and performing the parallel composition (with synchronization on shared actions) of the resulting processes.

Definition 5 (Configuration Behavior Model). *Given an architecture configuration $\mathcal{G} = (\mathcal{N} = \{n_1, \ldots, n_n\}, \mathcal{E})$, we define its behavior model as the parallel composition $(bn_1 || \ldots || bn_n)$, where $bn_{i \in \{1..|\mathcal{N}|\}}$ is an instance of the behavioral type $btype(n_i)$.*

4.3 Qualities

In addition to structure and behavior, we also need to capture quantitative aspects of systems to enable the analysis of their qualities. To achieve this goal, we employ *reward structures* to quantify information that emerges from the combined behavior of the different elements in the system and is not explicitly captured by properties in architectural elements. Two examples are the overall number of lost requests, and average end-to-end response time of a system, which could be employed to analyze run-time quality attributes such as reliability and performance, respectively.

Definition 6 (Reward Structure). *A reward structure for a system with architectural signature Σ is a pair (ρ, ι), where $\rho : \mathcal{S}_\Sigma \to \mathbb{R}_{\geq 0}$ is a function that assigns rewards to system state, and $\iota : \mathcal{S}_\Sigma \times \mathcal{S}_\Sigma \to \mathbb{R}_{\geq 0}$ is a function assigning rewards to transitions.*

State reward $\rho(s)$ is acquired in state $s \in \mathcal{S}_\Sigma$ per time step, i.e., each time that the system spends one time step in s, the reward accrues $\rho(s)$. In contrast, $\iota(s, s')$ is the reward acquired every time that a transition between s and s' occurs. Our approach is agnostic with respect to the way in which reward structures are defined. However, in this paper we assume that rewards over states are defined as sets of pairs (pd, r), where pd is a predicate over states $\mathcal{S}_\Sigma$, and $r \in \mathbb{R}_{\geq 0}$ is the accrued reward when $s \in \mathcal{S}_\Sigma \models pd$. We consider transition rewards as sets of pairs (p, r), in which $p \in \mathcal{P}$ is a port type, and reward $r \in \mathbb{R}_{\geq 0}$ is accrued when an interaction over a port of type p occurs.

Example 3. To compute the cost of operating a TAS configuration, we define a reward structure that accrues the cost of invoking each of the services in a configuration as: (ρ, ι)=($\emptyset$, {(DrugServiceT::changeDrugPD, DrugServiceT.Cost), (DrugServiceT::changeDosePD, DrugServiceT.Cost), (AlarmServiceT::sendAlarmPAS, AlarmServiceT.Cost), (MedicalServiceT::analyzeDataPS, MedicalServiceT.Cost)}).

5 Exploring the Design Space

5.1 Configuration Generation

Generating structurally correct configurations entails: (i) formalizing a set of structural style constraints that all configurations must respect, (ii) instantiating the constraints for

a specific set of architecture entities into a concrete relational model, and (iii) synthesizing the configurations that satisfy the constraints in the relational model.

Formalizing Structural Constraints This is a manual process that can be carried out by producing a specification in an ADL like Acme, and then translated automatically to an Alloy specification [14], or directly producing a specification in the latter. Listing 1.1 shows an excerpt of the encoding of the TAS architectural style in Alloy. Lines 1-4 encode the definitions of abstract architectural elements that belong to the architectural signature like components or connectors, whereas lines 6-8 show a part of the encoding of general constraints of the architecture (e.g., a component cannot be connected to itself). The service types in TAS are encoded as signatures that extend the base signature Component defined in line 1. For instance, the AssistanceServiceT component type definition (lines 16-20) includes constraints indicating that it must contain at least one port for invoking every possible operation type on other services (lines 17-18), and that those invocation port types can only belong to that type of component (lines 19-20).

```
1  abstract sig Component {ports: set Port} // Component and Connector abstract definition
2  abstract sig Connector {roles: set Role}
3  sig Port {component: Component}
4  sig Role {connector: Connector, attachment: one Port}
5  // General constraints of the architecture
6  fact { all p:Port | one r:Role | p in r.attachment } // A port is connected to only one role
7  pred conn[c: Component, c':Component] { some r,r':Role | r!=r' and r.attachment.component=c and
       r'.attachment.component=c' and r.connector=r'.connector } // Two components are connected
8  fact { all c,c':Component | c=c' => not conn[c,c'] } // A component must not be connected to itself
9  ... // TAS−specific definitions
10 pred invokes[p:Port, p':Port] { one r:Caller,r':Callee | r.attachment=p and r'.attachment=p' and
       r.connector=r'.connector } // A port (p) carries out invocations on another one (p')
11 pred invokesOnly[p:Port, p':Port] { invokes[p,p'] and all p'':Port−p' | not invokes[p,p''] } // A port carries
       out invocations ∗only∗ on another specific port
12 abstract sig HttpConnT extends Connector {} // ∗∗∗ HTTP Connector ∗∗∗
13 abstract sig Caller, Callee extends Role{} // An http connector has a caller and a callee role
14 fact { all c:HttpConnT | one r:Caller, r':Callee | r in c.roles and r' in c.roles }
15 fact { all c:HttpConnT | #c.roles=2 } // Every http connector has ∗exactly∗ two roles
16 one abstract sig AssistanceServiceT extends Component{} // ∗∗∗ Tele Assistance Service ∗∗∗
17  { changeDrugPTS & ports != none and changeDosePTS & ports != none and sendAlarmPTS & ports !=
       none and analyzeDataPTS & ports != none} // A TAS has one port for every possible operation
18 abstract sig changeDrugPTS, changeDosePTS, sendAlarmPTS, analyzeDataPTS extends Port{}
19 fact { all p:changeDrugPTS+changeDosePTS+sendAlarmPTS+analyzeDataPTS | p.component in
       AssistanceServiceT }
20 fact { all c:AssistanceServiceT | c.ports in
       changeDrugPTS+changeDosePTS+sendAlarmPTS+analyzeDataPTS }
21 abstract sig DrugServiceT extends Component{ } // ∗∗∗ Drug Service ∗∗∗
22 { changeDrugPD & ports != none and changeDosePD & ports != none and #ports=2 }
23 abstract sig changeDrugPD, changeDosePD extends Port{}
24 fact { all p:changeDrugPD+changeDosePD | p.component in DrugServiceT }
25 fact { all c:DrugServiceT | c.ports in changeDrugPD+changeDosePD }
26 ... // General structure (allowed invocations among ports in different components)
27 fact { all pt:analyzeDataPTS | one ps:analyzeDataPS | invokesOnly[pt,ps] }
28 fact { all pt:changeDrugPTS | one pd:changeDrugPD | invokesOnly[pt,pd] }
29 ...
30 fact { all t:AssistanceServiceT | one d:DrugServiceT | conn[t,d] } // A TAS connects to ∗only one∗ DS
```

Listing 1.1. TAS architecture style constraint specification in Alloy (excerpt).

Instantiating Constraints Once the set of structural constraints of the style is formalized, we can instantiate a full relational model that will enable us to apply these constraints to a set of concrete instances that realize concrete configurations. Listing 1.2 presents an excerpt of concrete components in TAS that correspond to alternative implementations of services available from various providers. This specification includes

the name of the concrete service implementation, along with its type, which matches one of the abstract types in the specification of structural constraints in Listing 1.1, and information related to its quality attributes (Fig 2.a).

Entity definitions are employed to automatically extend the constraints into a full relational model that includes concrete instances of the different entities in the system. Listing 1.3 shows the Alloy code generated to complement the specification in Listing 1.1. Every instance is encoded into a signature that extends its corresponding abstract type. The definition of every signature is preceded by a lone quantifier, indicating that the presence of a specific instance in a valid system configuration is optional. Quality attribute information is not used to analyze structural aspects of the system, and hence is abstracted in the Alloy specification. These are used later for behavioral configuration model generation (Section 5.2).

```
S1 [type: MedicalServiceT, failureRate: 0.06, responseTime: 22, cost: 9.8];
AS1 [type: AlarmServiceT, failureRate: 0.3, responseTime: 11, cost: 4.1];
```

Listing 1.2. Concrete service implementation definitions for TAS (excerpt).

```
lone sig D1 extends DrugServiceT{}
lone sig S1, S2, S3, S4, S5 extends MedicalServiceT{}
lone sig AS1, AS2, AS3 extends AlarmServiceT{}
lone sig TAS1 extends AssistanceServiceT{}
```

Listing 1.3. Concrete service implementation definitions for TAS in Alloy.

Configuration Synthesis Once a model instantiating the style constraints is available, we use the Alloy analyzer to find all relational models that describe configurations satisfying the constraints imposed by the style and employ a set of concrete architecture elements (e.g., TAS service implementations).

To do that, we invoke the run command and impose a constraint on the cardinality of the different sets of entities (determined by the maximum available number of components of each type) using an additional predicate (Listing 1.4). As an example, we run the predicate TAS for a maximum number of 10 instances of each signature in the model, and impose a restriction of one implementation per type of service, except for AlarmServiceT, for which we impose a maximum of 2 instances.

```
pred TAS {#DrugServiceT=1 and #AlarmServiceT=2 and #MedicalServiceT=1}
run TAS for 10
```

Listing 1.4. Synthesizing TAS configurations in Alloy.

Figure 5 shows two TAS configurations, generated from the Alloy model described in this section. The structure on the left is analogous to the one depicted in Figure 3, in which TAS is able to invoke a service of each type. However, the structure on the right describes a configuration in which TAS can invoke alarm services AS2 and AS3, potentially increasing reliability and performance when an alarm is raised, but probably at the expense of a higher cost. This second configuration results from the flexibility in the cardinality constraints imposed by Listing 1.4, line 1, which allows more than one alarm service to be employed in a configuration.

At this point, we can generate alternative configurations for a system in a given style, employing a set of concrete elements as building blocks for the configuration. However, if we want to be able to determine which configurations satisfy some criteria defined over the behavior or the qualities of the solution, we need to include additional

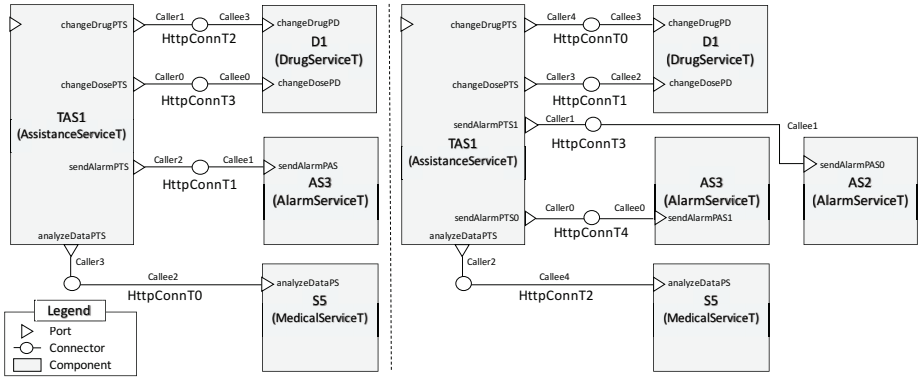


Fig. 5. Graphical representation for two TAS configurations synthesized using Alloy.

specifications that go beyond structure. In the next section, we describe how to expand structures into behavioral models that are amenable to analysis that takes into consideration behavioral and quantitative aspects of system configurations.

5.2 Configuration Behavior Model Generation

The behavior model of a configuration can be obtained by instantiating the behavioral type of each of the architecture elements in the configuration, and performing the parallel composition of the resulting processes. Algorithm 5.2 receives as input the configuration of the system $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and the set of behavioral types for the different architecture elements β, and returns the configuration behavior model for $\mathcal{G}$.

Algorithm 1 Configuration behavior model generation

1: $B := \emptyset$
2: **for all** $n \in \mathcal{N}$ **do**
3: $\quad P_\nu := \emptyset$
4: $\quad P^*_\Lambda := \{t \in P_\Lambda \mid btype(n) = (S_\kappa, s_i, P_\Lambda) \wedge ip(t) \neq \bot\}$
5: $\quad$ **for all** $t \in P^*_\Lambda$ **do**
6: $\quad\quad A_t := \{(p, r) \in \mathcal{E} \mid (parent(p) = n \vee parent(r) = n) \wedge iptype(p) = ip(t)\}$
7: $\quad\quad$ **for all** $a_t \in A_t$ **do**
8: $\quad\quad\quad P_\nu := P_\nu \cup \{states(t) \mapsto (prob(t)/|A_t|, label(a_t))\}$
9: $\quad\quad$ **end for**
10: $\quad$ **end for**
11: $\quad B := B \cup \{(S_\kappa, s_i, (P_\Lambda \backslash P^*_\Lambda) \cup P_\nu)\}$
12: **end for**
13: **return** $(b_1 || \ldots || b_n) \bullet b_{i \in \{1..|\mathcal{N}|\}} \in B$

The algorithm starts with an empty set of behaviors B (line 1), and incrementally adds the behavior of each node in the configuration graph, which is instanced by: **(1)** Determining the set of transitions P^*_Λ of the behavioral type that interact with the environment (line 4). Function ip returns the interaction point (port or role type) associated with every element of P_Λ in the behavioral type. $btype$ returns the behavioral type of an architecture element. **(2)** For each transition identified in (1), creating an instance of the transition for every other node to which the current one is attached in the configuration (lines 6-9). In line 6, the set of attachments in the configuration graph for the current node is identified. Here, interaction point type function $iptype$ identifies the type of a port or role, whereas $parent$ returns the node that a port or role belongs to. Line 8 adds new transition instances, adjusting the probability contribution of the transition according to the number of instances created for a given transition in P^*_Λ.[2] Function $states$ return the pair of source and target state for a transi-

[2] The semantics of behavioral types are inspired by discrete-time Markov chains, so the original probability of the transition $prob(t)$ is divided equally among transition instances.

tion, whereas $prob$ returns its associated probability. Function $label$ generates a unique label for an attachment, defined as a pair port-role. **(3)** Creating a new behavior instance incorporating the original elements of $btype(n)$ (line 11). This process describes the behavior of graph node n, in which transitions identified in (1) are substituted by the new set of transition instances P_ν identified in (2).
The algorithm finishes returning the parallel composition of the processes in B.

5.3 Quantification, Filtering and Ranking

After obtaining the behavioral models for the possible configurations of the system, we can assess behavioral, as well as quantitative constraints and properties on them. This analysis might also include probabilistic aspects in the behavioral and quantitative properties (e.g., reliability of services on which TAS relies), so we propose to employ probabilistic temporal logics to capture them. We illustrate formalization using PCTL [15], although these specifications can be adapted to other types of probabilistic temporal logic for behavioral descriptions inspired by other formalisms (e.g., continuous-time Markov chains, probabilistic timed automata).

This step identifies configurations that do not meet a set of properties and constraints imposed by designers, which may include: (i) behavioral properties (e.g., safety, liveness), and (ii) quantitative constraints (e.g., on quality attributes).

Example 4. We want to assess the overall response time, reliability, and cost of configurations in TAS. We define $\mathsf{serviceOK} \triangleq \mathsf{changeDoseOK} \vee \mathsf{changeDrugOK} \vee \mathsf{sendAlarmOK}$ as a predicate indicating that TAS provided some of the possible service types correctly. Moreover, we assume the predicate $\mathsf{timeout}$ captures failed service invocation.

Based on these predicates, we define properties $\mathsf{R}_{\mathsf{rt}=?}[\mathsf{F}\ (\mathsf{serviceOK} \vee \mathsf{timeout})]$ and $\mathsf{R}_{\mathsf{cost}=?}[\mathsf{F}\ (\mathsf{serviceOK} \vee \mathsf{timeout})]$ that employ the reward quantifier of PCTL to quantify the expected response time and cost of a configuration by accruing the response time and cost rewards rt and cost, respectively. Property $1 - \mathsf{P}_{=?}[\mathsf{F}\ \mathsf{serviceOK}]$ quantifies the overall reliability of a configuration (i.e., that the system will fail to provide correct service) by employing the probabilistic quantifier of PCTL.

6 Results

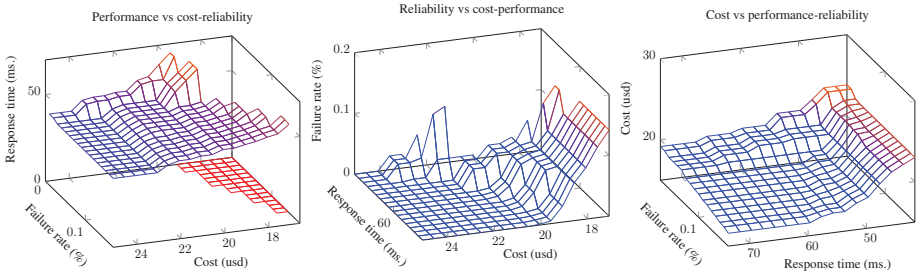


Fig. 6. TAS configurations constrained by: (a) cost and reliability (left), (b) cost and performance (center), and (c) performance and reliability (right).

We present in this section our experimental results. To test our proposal, we ran a prototype implementation of our approach that employed Alloy 4.2 for synthesizing

configurations and PRISM 4.3.1 for behavioral and quantitative analysis. The experiment was run on an Intel Core i7 2.8GHz with 16 GB RAM. We ran our analysis to compute the set of feasible solutions for TAS that meet the set of structural constraints described in Listing 1.1, using the set of service implementations described in Fig. 2.a.

Space size and computation time. Table 1 shows that the overall computation time for generating and analyzing the solution space was approximately 9 seconds, out of which 15% was used to generate 90 configurations (Alloy) and 270 behavioral configuration models (90 x 3 possible values for the parameter that specifies number of retries after a failed service invocation). Checking deadlock freeness and the three quantitative properties defined in Example 4 took approximately 85% of the time.

Analysis results. The plot on the left of Figure 6 shows the best response time that can be achieved in a system configuration when the cost and failure rate are constrained to the thresholds on the horizontal axes. As expected, we observe that lower response times and failure rates incur higher cost. This is consistent with the properties of service providers (better response times and reliability are more expensive), and the fact that having the flexibity to add redundant services (e.g., alarm service) to increase reliability and reduce response time increases cost. Our technique enables us to identify the thresholds in cost and failure rate for which there are no system configurations satisfying style constraints (in the range $\leq$ 19 usd – the red squares on the bottom plane).

# Configurations	90
# Configuration behavioral models	270
Configuration behavior model generation time	1.361 s. (15.1 %)
Configuration behavioral model checking time (PRISM)	7.66 s. (84.9 %)
Total computation time	**9.021 s.**

Table 1. Problem instance size and computation time.

The plot in the center shows how failure rate of configurations increases noticeably with lower costs, whereas with high cost, it is fairly stable and does not vary much with overall response time, except for very low values.

Finally, the plot on the right shows the overall cost of configurations for different levels of response time and reliability. As expected, we can observe how higher response times and failure rates correspond to lower costs, whereas peaks in cost are reached with lowest failure rates and response times.

An architect can take these results and make informed design decisions based, for instance, on the available budget for the project and legal constraints on the level of reliability and timeliness demanded of systems for first-aid response.

7 Related Work

Work related to our proposal can be categorized into: (i) formalization of architectural styles, and (ii) architecture-based quantitative analysis and optimization.

(1) Formalization of architectural styles: Formalization of styles has been explored to define formal semantics of modeling languages. Kim and Garlan [14] propose an automatic translation from Acme into Alloy relational models on which they verify properties implied by the style. Wong et al. [26] also employ Alloy to check the consistency of rules among multiple styles that might be combined in complex systems. In addition to property verification, other approaches also explore constraint solving for

synthesizing architectures [3, 19]. Bagheri and Sullivan [3] employ architecture synthesis for generating architectural models from architecture-independent application models, emphasizing the separation of style choices from application description. In contrast, Maoz et al. [19] propose an approach that employs synthesis to merge different partial component-and-connector views. All the aforementioned approaches focus on structural properties and differ from ours in that they do not consider behavioral, quantitative, or probabilistic aspects of system descriptions, being unable to systematically analyze nondeterministic system behaviors and their effects on quality attributes.

(2) Architecture-based quantitative analysis and optimization: Other approaches focus on analyzing and optimizing quantitative aspects of architectures using mechanisms that include stochastic search and/or Pareto analysis [1, 5, 20]. *PerOpteryx* [20] takes as input an architectural model described using the Palladio component model and tries to automatically improve it by searching for pareto-optimal solutions employing a genetic algorithm. *ArcheOpterix* [1] uses an evolutionary algorithm for optimizing the architecture of embedded systems. *DeepCompass* [5] is a framework that analyzes different architectural alternatives along the dimensions of performance and cost to find pareto-optimal solutions. While these and other approaches in systems engineering (e.g., [17]) can give estimates and optimize quantitative aspects of designs, they do not support synthesis of configurations (which have to be manually specified), and do not provide any formal guarantees concerning the behavior or quantitative properties of the variants.

Other approaches [4, 7] have recently combined architecture synthesis with simulation and dynamic analysis to provide estimates of quantitative properties of system variants. *TradeMaker* [4] synthesizes design spaces for object-relational database mappings, in which individual designs are subject to static and dynamic analysis to extract performance metrics. Dwivedi et al. [7] propose using architectural models coupled with automated design space generation for making fidelity and timeliness tradeoffs. These approaches share with ours the idea of synthesizing a solution space from a set of constraints and analyzing individual solutions independently. However, they do not explore exhaustively the state space of individual solutions and hence are unable to provide guarantees about solution behaviors or their interaction with system qualities.

8 Conclusions and Future Work

We have presented an approach to help architects explore the design space of families of software systems, giving them a tool to make informed design decisions by providing insight into the formal guarantees of solutions and tradeoffs among their qualities. Our approach enables the analysis of behavioral (i.e., safety, liveness) and quality properties (e.g., quantitative constraints, optimality) of solutions, considering interactions among them, as well as uncertainties captured via probabilities in models.

Concerning generality, the current embodiment of the approach is inspired by a specific model of formal architectural description (Acme) and behavioral formalism (DTMC). However, most constructs employed to formalize the architectural style are fairly standard and the approach for synthesis of configurations is adaptable to other languages and underlying models. In terms of behavior descriptions, DTMCs constrain the analysis to a discrete time model and average case of probabilities/rewards, although

straightforward adaptations can be carried out to adapt behavioral analysis to other probabilistic behavior descriptions such as MDPs (for worst-case scenario analysis) or PTAs for finer-grained time analysis. We will explore these areas in future work.

Moreover, although in this paper we have focused on spaces in which design decisions are dominated by the selection and composition of pre-existing components, design spaces in which a non-trivial part of the system components have to be built from scratch have been left out of scope. We plan on extending our approach for such systems by exploring probabilistic parametric model checking techniques [10] to automatically find the ranges for quality attribute values that components to be implemented would have to provide to satisfy global system constraints on qualities.

A third direction for future work concerns scalability. The degree of formal assurance on configurations provided by the approach is computationally expensive, and entails risks on the computation cost of configuration synthesis (derived from the cost of finding instances of configurations in a rich configuration space) and configuration behavior analysis (derived from exploring potentially large state spaces of individual configuration behavior). These risks can be mitigated by exploiting the hierarchical structure and relations that are naturally present in complex architectures in which components interact in a structured way. Hence, synthesis of different subsystems with local constraints can be done independently and then composed, reducing the cost of configuration synthesis. This approach has been successfully used in other works that exploit mappings between specifications defined at different levels of abstraction [13], or incremental analysis techniques [2]. This mitigation also allows exploiting parallelism in the analysis, during which the behavior of configurations of subsystems can be independently analyzed using assume-guarantee compositional quantitative verification [12]. In this case, the computation time for the analysis would be dominated by the largest subsystem that can be independently analyzed (prior experience with PRISM suggest times under 10s for configurations of 250+ components, including probabilistic behavior [6]).

Acknowledgments. This material is based on research sponsored by AFRL and DARPA under agreement number FA8750-16-2-0042. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, DARPA or the U.S. Government.

References

1. Aleti, A., Bjornander, S., Grunske, L., Meedeniya, I.: Archeopterix: An extendable tool for architecture optimization of aadl models. In: Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES '09. ICSE Workshop on (2009)
2. Bagheri, H., Malek, S.: Titanium: efficient analysis of evolving alloy specifications. In: Proc. of the 24th Symposium on Foundations of Software Engineering, FSE 2016 (2016)
3. Bagheri, H., Sullivan, K.J.: Model-driven synthesis of formally precise, stylized software architectures. Formal Asp. Comput. 28(3) (2016)
4. Bagheri, H., Tang, C., Sullivan, K.J.: Trademaker: automated dynamic analysis of synthesized tradespaces. In: 36th Int. Conf. on Software Engineering. ACM (2014)
5. Bondarev, E., Chaudron, M.R.V., de Kock, E.A.: Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In: 6th WS on Software and Performance. WOSP, ACM (2007)

6. Cámara, J., Garlan, D., Schmerl, B., Pandey, A.: Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In: 30th ACM Symposium on Applied Computing (SAC) (2015)
7. Dwivedi, V., Garlan, D., Pfeffer, J., Schmerl, B.: Model-based assistance for making time/fidelity trade-offs in component compositions. In: 11th International Conference on Information Technology: New Generations, ITNG 2014. IEEE CS (2014)
8. Garlan, D.: Software engineering in an uncertain world. In: Proc. of the Workshop on Future of Software Engineering Research, FoSER (2010)
9. Garlan, D., Monroe, R.T., Wile, D.: Acme: an architecture description interchange language. In: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada. IBM (1997)
10. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Param: A model checker for parametric markov models. In: Computer Aided Verification. Springer (2010)
11. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11(2) (Apr 2002)
12. Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering. CBSE '13, ACM (2013)
13. Kang, E., Milicevic, A., Jackson, D.: Multi-representational security analysis. In: Proc. of the 24th Symposium on Foundations of Software Engineering, FSE (2016)
14. Kim, J., Garlan, D.: Analyzing architectural styles. J Syst Software 83(7) (2010)
15. Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: FM for Performance Evaluation, 7th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems. LNCS, vol. 4486. Springer (2007)
16. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification. LNCS, vol. 6806. Springer (2011)
17. MacCalman, A.D., Beery, P.T., Paulo, E.P.: A systems design exploration approach that illuminates tradespaces using statistical experimental designs. Syst. Eng. 19(5) (2016)
18. Mahdavi-Hezavehi, S., Galster, M., Avgeriou, P.: Variability in quality attributes of service-based software systems: A systematic literature review. Inf Softw Technol 55(2) (2013)
19. Maoz, S., Ringert, J.O., Rumpe, B.: Synthesis of component and connector models from crosscutting structural views. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'13. ACM (2013)
20. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: Int. Conf. on Performance Engineering. WOSP/SIPEW, ACM (2010)
21. Maydl, W., Grunske, L.: Behavioral types for embedded software – a survey. In: Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends. Springer (2005)
22. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA WS on Open Source Software (2009)
23. Shaw, M., Garlan, D.: Software architecture - perspectives on an emerging discipline. Prentice Hall (1996)
24. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley (2003)
25. Weyns, D., Calinescu, R.: Tele assistance: A self-adaptive service-based system exemplar. In: 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015. IEEE Computer Society (2015)
26. Wong, S., Sun, J., Warren, I., Sun, J.: A scalable approach to multi-style architectural modeling and verification. In: 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008) (2008)