Synthesizing Tradeoff Spaces with Quantitative Guarantees for Families of Software Systems

Javier Cámara[1], David Garlan[2], Bradley Schmerl[2]

Abstract

Designing software in a way that guarantees run-time behavior while achieving an acceptable balance among multiple quality attributes is an open problem. Providing *guarantees* about the satisfaction of the same requirements under uncertain environments is even more challenging. Tools and techniques to inform engineers about poorly-understood design spaces in the presence of *uncertainty* are needed, so that engineers can explore the design space, especially when tradeoffs are crucial. To tackle this problem, we describe 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 treated as a first-class entity. We illustrate our approach in two case studies: a service-based adaptive system and a mobile robotics architecture. Our results show how the framework can provide useful insights into how *average case probabilistic guarantees* can differ from *worst case guarantees*, emphasizing the relevance of combining quantitative formal verification methods with structural synthesis, in contrast with techniques based on simulation and dynamic analysis that can only provide *estimates* about average case probabilistic properties.

Keywords: Tradeoff analysis, uncertainty, architectural style, architecture synthesis, quantitative guarantees, 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 [1] (e.g., faults, network delays). Achieving a good design with behavioral guarantees and a balance be-

Email addresses: javier.camaramoreno@york.ac.uk (Javier Cámara), garlan@cs.cmu.edu (David Garlan), schmerl@cs.cmu.edu (Bradley Schmerl)
[1]University of York, Heslington, York YO10 5GH, UK.
[2]Carnegie Mellon University, Pittsburgh PA 15213, USA.

tween 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, or performance) that may be offered by independent providers. For instance, many current 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 [2]. 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 [3]. 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 [4] 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 to which the system must conform. 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 [5] 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 [6, 7, 8]. However, these approaches tend to focus on structural properties; when available, analysis of aspects like system behaviors and quality are performed separately. So, approaches that reason separately about these aspects 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:

- (RQ1) How can we integrate formal descriptions of structural, behavioral, and

quality aspects of design alternatives to enable integrated reasoning about all these aspects?

- (RQ2) How can we 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)?
- (RQ3) If such an approach is feasible, what would be its applicability in different contexts (e.g., application domains, objectives, sources of uncertainty)?
- (RQ4) What is the added value of analyzing tradeoffs in quantitative guarantees (as opposed to other methods based e.g., on simulation or dynamic analysis)?

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 exhaustively analyzing the state space of each configuration's behavior. Our framework explicitly considers *interactions between functional behaviors and extra-functional properties* while factoring in *uncertainty* [3] as a first-class concern.

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 [9]) 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 concern*. This combination is not trivial because reasoning about all these aspects in a unified way entails synthesizing structurally complex behavioral models (where structure is given by the topology of the communication among processes) from structural and behavioral models (in particular, those used for quantitative verification), which are fundamentally heterogeneous in representation and semantics. Bridging this heterogeneity gap goes beyond mere pipelining of structural synthesis and probabilistic model checking, and demands the alignment of model abstractions, as well as specialized algorithms for the synthesis of structurally complex probabilistic behavioral models, which are the key contribution of our approach.

With respect to the alignment of models: (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

[3] Note that uncertainty can be *epistemic* in nature (i.e., due to the lack of, incomplete, or inaccurate information), or *aleatoric* or due to randomness. Our approach focuses on analyzing quantitative guarantees in systems where uncertainties are mostly aleatoric and can be captured in terms of probabilistic or fully-nondeterministic choices. In the remainder of this article, the word uncertainty refers to aleatoric uncertainty, unless explicitly stated otherwise.

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 [10].

In [11] we introduced a preliminary version of this work that enabled quantitative analysis of average probabilities and rewards based on discrete-time Markov chains (DTMC), and illustrated the approach on a Tele Assistance System (TAS) [12] for the validation of service compositions originally proposed by Baresi et al. in [13]. In this paper, we extend our formal framework to provide worst and best case scenario guarantees by enabling analysis of maximum/minimum probabilistic and reward guarantees based on Markov decision processes (MDP). Moreover, we illustrate our approach in a new robotics scenario based on a ROS [2] pub-sub architecture, and extend evaluation with the robotics case study and additional results in TAS.

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 describes the ROS robotics case study. Section 7 presents evaluation, and Section 9 overviews related work. Finally, Section 10 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):

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 architecture 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 architecture configurations that satisfy the style constraints.

Configuration behavior model generation (Section 5.2), during which a set of configuration behavioral models that refine the architecture configurations obtained in (1) is generated. This process takes as input: (i) the set of concrete architecture element

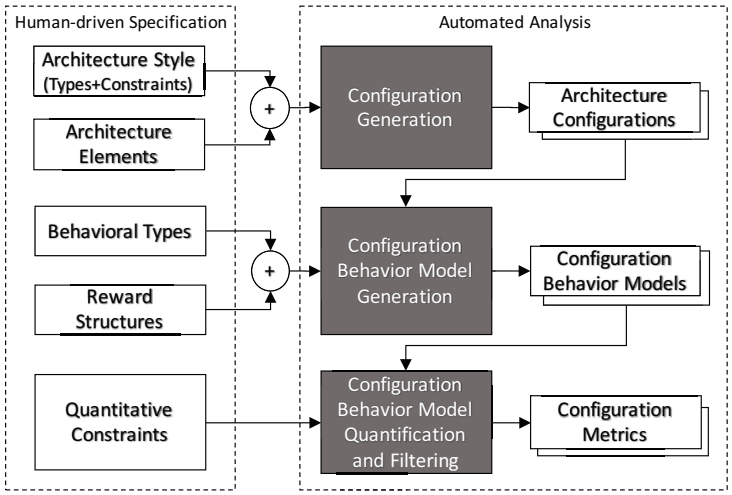

Figure 1: Overview of the approach.

definitions, (ii) the configurations generated in (1), and (iii) the set of *behavioral types*[4] 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 architecture 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.

[4]Although the notion of behavioral type is more general [14], we employ the term to refer to an abstract state machine specification capturing the behavior of an architectural abstract type.

3. Motivating Scenario

We illustrate our approach on the TAS exemplar system [12], 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).

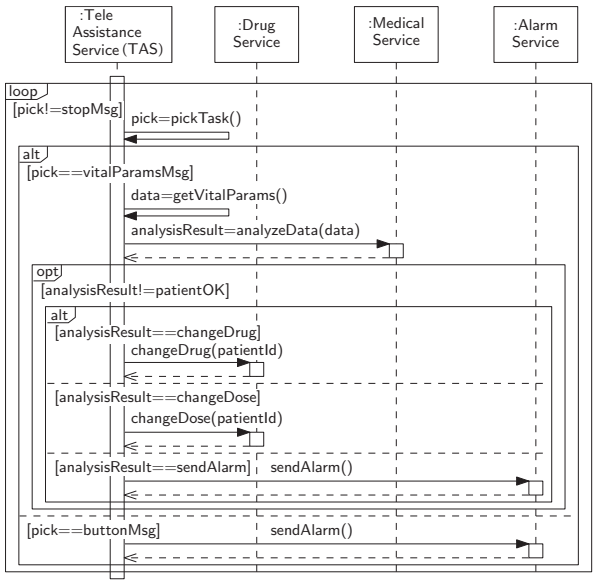

Figure 2: Tele assistance service workflow (adapted from [12])

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*, and 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. The metrics employed for the different quality attributes in TAS are the percentage of service failures for reliability, and service response time for performance. We consider that five service providers offer the Medical Service, three offer the Alarm Service, and only one offers the Drug Service (Table 1).

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 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 (Table 2).

Service	Name	Fail. rate (%)	Resp. time (ms.)	Cost (usd)
S1	Medical Service 1	0.06	22	9.8
S2	Medical Service 2	0.1	27	8.9
S3	Medical Service 3	0.15	31	9.3
S4	Medical Service 4	0.25	29	7.3
S5	Medical 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

Table 1: 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.

Table 2: Example of quality requirements for TAS.

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* [4].

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 [15] or OCL [16] constraints (e.g.,* $\forall$ t:AssistanceServiceT $\bullet \exists$ a:AlarmServiceT $\bullet$ connected(t,a) *– "every tele assistance service must be connected to at least 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).

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.*

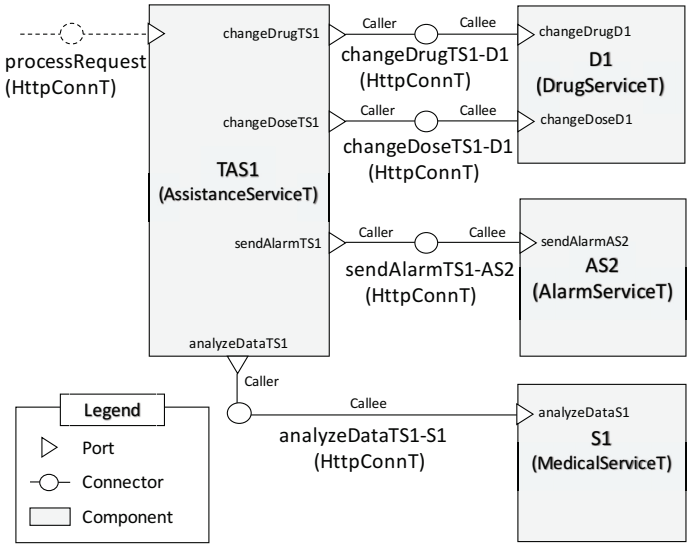

Figure 3: Sample TAS configuration.

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.

We instantiate our notion of behavioral type both for discrete-time Markov chains (DTMC) and Markov decision processes (MDP). The latter instantiation allows capturing aspects such as fully nondeterministic choices that enable maximum/minimum probability and reward analysis, useful to provide, e.g., worst case scenario probabilistic guarantees that can be obtained only via exhaustive state-space exploration techniques like model checking (unlike average case probabilistic guarantees on DTMC, which can be *approximated* via statistical model checking and other Monte Carlo-style simulation techniques).

Definition 4 (Behavioral Type - DTMC). *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 $P_\Lambda : S_\kappa \times S_\kappa \rightarrow [0,1] \times (\Lambda(\kappa) \cup \{\perp\})$ *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 $btype(c)$.

Definition 5. *[Behavioral Type - MDP] The behavioral type of an architectural type* $\kappa \in CompT \cup ConnT$ *is a tuple* $(S_\kappa, s_i, E, \Delta)$*, 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,* $E = \Lambda(\kappa) \cup \{\tau\}$ *is a set of events that correspond to the ports in the component type (respectively, roles in the connector type) extended with the internal action* τ*, and* $\Delta : S_\kappa \times E \rightarrow \mathcal{D}(S_\kappa)$ *is a (partial) probabilistic transition function.* $\mathcal{D}(X)$ *denotes the set of discrete probability distributions over finite set* X*.*

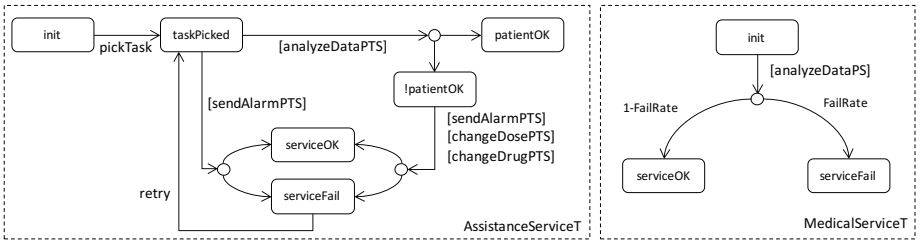

Figure 4: AssistanceServiceT and MedicalServiceT behavioral types.

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).*

Under DTMC semantics, unlabeled branching transitions implicitly specify a uniform probability distribution, and non-branching transitions indicate probability 1. In contrast, for MDP semantics, both unlabeled branching transitions and multiple outgoing transitions from a given state introduce local nondeterministic choices.

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 6 (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 $btype(n_i)$. Note that all the behavioral types that participate in this parallel composition must have the same semantics (all of them must be DTMC, or all of them must be MDP).*

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 7 (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::change-DosePD, 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 [17], or directly producing a specification in the latter. Listing 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: 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 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, 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 3 shows the Alloy code generated to complement the specification in Listing 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 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 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 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 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 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 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

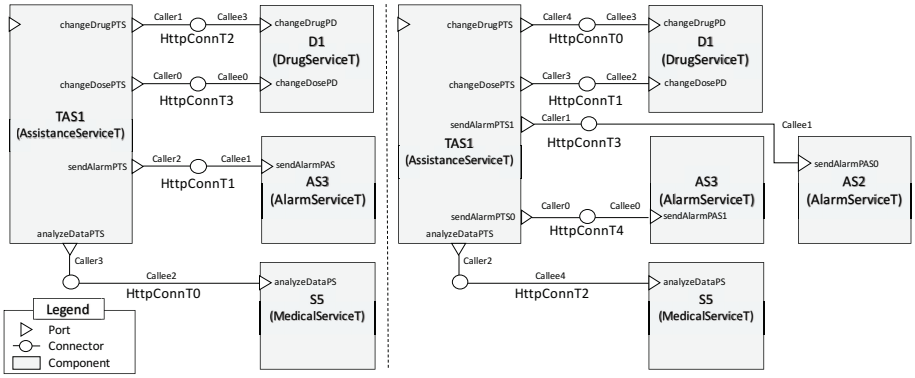

Figure 5: Graphical representation for two TAS configurations synthesized using Alloy.

parallel composition of the resulting processes. Algorithms 1 and 2 receive 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 return the configuration behavior model for $\mathcal{G}$ (for the DTMC-based and MDP-based semantics, respectively).

Algorithm 1 Configuration behavior model generation (DTMC)

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** $(b_1 || \ldots || b_n) \bullet b_{i \in \{1..|\mathcal{N}|\}} \in B$

5.2.1. DTMC-based Behavior

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. Since this version of the semantics of behavioral types is inspired by discrete-time Markov chains, the original probability of the transition $prob(t)$ is divided equally among transition instances. Function $states$ returns the pair of source and target state for a transition, 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.

5.2.2. MDP-based Behavior

Similarly to Algorithm 1, Algorithm 2 starts with an empty set of behaviors B (line 1), and incrementally adds the behavior of each node in the configuration graph. However, in this case the instantiation process changes due to the differences in the underlying semantics of MDP. Specifically, the instantiation process consists of: **(1)** Determining the set of tuples in Δ that correspond to internal actions (Δ^*, line 5). We recall that each tuple in Δ encodes the mapping of a state and an event (including the special event τ that designates internal actions) to a probability distribution. **(2)** For each tuple of Δ corresponding to an action that interacts with the environment (i.e., not in Δ^*), creating an instance of the tuple for every other node to which the current one is attached in the configuration (lines 6-11), which are stored in Δ_ν. Similarly to the DTMC-based semantics case, the set of attachments in the configuration graph for the current node is identified in line 7. For each tuple in Δ_ν, the original event is substituted by a unique label used for synchronization generated using the function $label$ presented earlier (line 9). **(3)** The new behavior instance is put together by replacing the original set of events in the behavioral type by the aggregated set of labels generated for synchronization in E_ν, and the original Δ by the union of the set of tuples that correspond to internal events Δ^* with the set of generated tuples for synchronization Δ_ν (line 13). The algorithm returns the parallel composition of the processes in B.

Algorithm 2 Configuration behavior model generation (MDP)

1: $B := \emptyset$
2: $E_\nu := \emptyset$
3: **for all** $n \in \mathcal{N}$ **do**
4: $\quad \Delta_\nu := \emptyset$
5: $\quad \Delta^* := \{((s,e),\mathcal{D}) \in \Delta \mid e = \tau\}$
6: $\quad$ **for all** $((s,e),\mathcal{D}) \in \Delta \backslash \Delta^*$ **do**
7: $\quad\quad A_t := \{(p,r) \in \mathcal{E} \mid (parent(p) = n \vee parent(r) = n) \wedge iptype(p) = e\}$
8: $\quad\quad$ **for all** $a_t \in A_t$ **do**
9: $\quad\quad\quad \Delta_\nu := \Delta_\nu \cup \{(s, label(a_t)) \mapsto \mathcal{D}\}$
10: $\quad\quad\quad E_\nu := E_\nu \cup \{label(a_t)\}$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad B := B \cup \{(S_\kappa, s_i, E_\nu, \Delta^* \cup \Delta_\nu)\}$
14: **end for**
15: **return** $(b_1 || \ldots || b_n) \bullet b_{i \in \{1..|\mathcal{N}|\}} \in B$

PCTL Property (avg. case)	PCTL Property (worst case)	Description
$\mathsf{R}^{\mathsf{rt}}_{=?}[\mathsf{F}\ (\mathsf{serviceOK} \vee \mathsf{timeout})]$	$\mathsf{R}^{\mathsf{rt}}_{\mathsf{max}=?}[\mathsf{F}\ (\mathsf{serviceOK} \vee \mathsf{timeout})]$	Performance - response time accrued until transaction is finished or there is a timeout.
$\mathsf{R}^{\mathsf{cost}}_{=?}[\mathsf{F}\ (\mathsf{serviceOK} \vee \mathsf{timeout})]$	$\mathsf{R}^{\mathsf{cost}}_{\mathsf{max}=?}[\mathsf{F}\ (\mathsf{serviceOK} \vee \mathsf{timeout})]$	Cost accrued from using services in a transaction.
$1 - \mathsf{P}_{=?}[\mathsf{F}\ \mathsf{serviceOK}]$	$1 - \mathsf{P}_{\mathsf{min}=?}[\mathsf{F}\ \mathsf{serviceOK}]$	Reliability - probability of success in completing a transaction.

Table 3: PCTL properties for TAS.

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 [9], 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 in Table 3 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. The third 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.*

The first column of properties in the table quantify the probabilistic aggregate value across all possible executions of a DTMC. However, we can formulate analogous properties to analyze worst case scenario guarantees based on the probabilistic quantifiers $\mathsf{P}_{\mathsf{min}}/\mathsf{P}_{\mathsf{max}}$ *and* $\mathsf{R}_{\mathsf{min}}/\mathsf{R}_{\mathsf{max}}$ *over MDP of PCTL [18]. Hence, for TAS, the properties for worst case scenario analysis in the second column of the table capture minimization of the probability of success for reliability, as well as maximization of response time and cost for the other two qualities.*

6. Case study: Mobile Robotics Software Architecture

We apply our approach to a second case study in the domain of mobile service robotics. These systems are used to perform tasks like fetching mail, or escorting a visitor to an office within a building. To achieve their mission, these robots must carry out actions like navigating from one location to another while avoiding obstacles that might dynamically appear, and where batteries may require recharging. These systems are also limited in what they can sense, and this creates uncertainty in their location and the resources they may have left to complete a plan. Despite these uncertainties, they must attempt to ensure safe operation and effective use of resources (such as battery).

We focus on a subset of the software architecture of a mobile service robot that deals with sensing and robot localization.

6.1. Objectives

The non-functional requirements of this sub-architecture are maximizing operation safety, and minimizing energy consumption (Table 4). Concretely, requirements R1 and R2 are related to safety because the localization service should provide accurate localization information (R1), in a timely manner (R2). Accurate information not provided in a timely fashion is effectively rendered inaccurate because it does not reflect anymore the state of the world and increases the chances of undesirable incidents (e.g., collisions). Finally, requirement R3 concerns energy consumption minimization, subject to safety requirements.

Name	Description
R1	The accuracy of localization information should not be below `accuracy_thr`%.
R2	The response time of localization should not exceed `timeliness_thr` ms.
R3	Subject to R1 and R2, energy consumption should be minimized.

Table 4: Example of quality requirements for the robotics architecture

6.2. Architectural Style

The underlying implementation of this architecture is based on the robotics operating system ROS [2], which follows a pub-sub style [5] in which different components (*nodes* and *nodelets* [6]) publish information to topics, which is consumed by other subscribing components. In particular, configurations in this architecture contain three component categories (Table 5): (a) *sensing*, which manages physical sensors that capture information from the environment, (b) *localization*, which produces information about the position of the robot in the environment, based on lower-level information (either directly captured by a sensor, or processed by another component), and (c) *auxiliary*, which perform different functions like transforming data, so that it can be consumed by a particular type of (localization) component.

[5] The formal Alloy specification of this style can be found in Appendix A.

[6] A *nodelet* is a particular component type in ROS designed not to run as a separate process (as in nodes), but as an algorithm that can be loaded within a single process along with other nodelets, thus reducing communication and memory copying overhead.

Category	Name	Energy Consumption (mwhr)	Accuracy	Requires
Sensing	lidar	30	N/A	-
	kinect	10	N/A	laserscanNodelet
	camera	12	N/A	markerRecognizer headlamp*
Localization	amcl	12	0.98	-
	mrpt	5	0.90	-
	markerLocalization	0.1	0.99	-
Auxiliary	laserscanNodelet	0	N/A	-
	markerRecognizer	0	N/A	-
	headlamp	2	N/A	-

Table 5: Mobile robotics architecture: component types and properties

Name	Delay (ms)	Reliability	Publishers	Subscribers
laserScanTopic	1.2	0.97	lidar laserscanNodelet	amcl mrpt
sensorMsgsImageTopic	2.25	0.87	kinect camera	laserscanNodelet markerRecognizer
markerPoseTopic	0.8	0.94	markerRecognizer	markerLocalization

Table 6: Mobile robotics architecture: topics and properties

Every component in the architecture has a given energy consumption rate,[7] and might require additional auxiliary components to interoperate with the rest of the system. For instance, Table 5 shows that the kinect component requires an additional laserscanNodelet component, whereas the lidar component does not. This is explained because the lidar publishes sensed information from the laser scanner in a format that can be directly consumed by localization components like amcl and mrpt, whereas the information published by the kinect has to be preprocessed by the laserscanNodelet before it can be consumed by the localization components. Similarly, the camera component requires an additional markerRecognizer component that can take the images published by the camera and generate visual marker information that can be consumed by the markerLocalization component, which generates robot pose information based on the recognition of visual markers placed in the robot's environment. Precisely because this type of localization relies on visual information, the camera may require the additional headlamp component under low light conditions.

Communication among components is done via topics (Table 6), which introduce different delays due to the processing of messages in queues, and also have different levels of reliability derived from the fact that when queues are full, messages are dropped with the probability given in the table. The allowed publishers and subscribers to each of the topics are also displayed in the table. For instance, both amcl and mrpt localization can consume laser scan information published to the laserScanTopic (ei-

[7]Quantitative values of the properties in Table 5 are used for illustration purposes and do not necessarily reflect values on the actual robot.

ther published by the lidar directly, or by the laserscanNodelet, based on kinect sensed information). In terms of reliability and delay, the sensorMsgsImageTopic to which both the camera and the kinect publish, has the lowest reliability and highest delay, given that publishing of images requires more bandwidth than laser scan information, and the processing of messages is more intensive.

6.3. Behavioral Types

Figure 6 shows a graphical representation of the behavioral types for the robotics architecture. Topics (left) receive a message from publishers via a pub port, and then execute the internal action process, which can result with probability DROP_RATE in the dropping of the message (the value of the reliability for topics displayed in Table 6 corresponds to 1-DROP_RATE). If the message is not dropped, then it is sent to subscribers via sub ports.

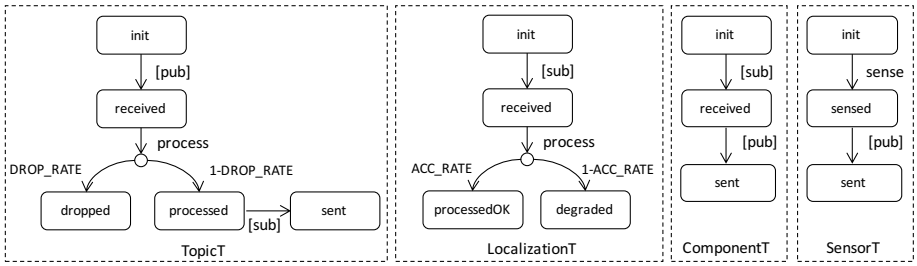

Figure 6: Mobile robotics architecture behavioral types.

Focusing on components, their general behavior is receiving a message via the sub port, and then publish it via the pub port. In contrast, sensing components do not subscribe to any topics. Instead, they first capture information via the internal action sense, and then publish the information to a topic via pub. Localization components only subscribe to a topic, and can successfully process a message with probability ACC_RATE (which corresponds to the values in column accuracy on Table 5), or generate degraded information instead with probability 1-ACC_RATE. This probabilistic choice models the inherent level of accuracy of different localization mechanisms.

6.3.1. Rewards

This robotics scenario incorporates two reward structures that encode energy consumption and time. Concerning time, only topics introduce delays (constant DELAY in ms., according to the values shown in column delay of Table 6) in the processing of messages, which is accumulated in a time reward whenever the internal action process is executed. With respect to energy, every component and topic accumulates an amount of energy consumed per time unit (ERATE, column energy consumption in Tables 5 and 6) in a energy reward.

6.4. Quantifying Satisfaction of Objectives

Overall, we want to assess the accuracy of the localization information, delay, and energy consumption for every possible configuration of this architecture. To do so, we

PCTL property	Description
$P_{=?}[\mathsf{F}\ \mathsf{processedOK}]$	Probability that the localization component in a configuration will eventually process the message and produce localization information without degrading.
$R^{\mathsf{energy}}_{=?}[\mathsf{F}\ (\mathsf{processedOK} \vee \mathsf{degraded} \vee \mathsf{dropped})]$	Energy reward accrued until the message is either dropped, or processed by the localization component (either correctly or degraded).
$R^{\mathsf{time}}_{=?}[\mathsf{F}\ (\mathsf{processedOK} \vee \mathsf{degraded} \vee \mathsf{dropped})]$	Time reward accrued until the message is either dropped, or processed by the localization component (either correctly or degraded).

Table 7: PCTL properties for quantification and ranking of objectives in the robotics scenario

employ the PCTL formulas shown in Table 7.

The first formula is used to determine the satisfaction of the accuracy requirement R1 in Table 4. By quantifying the probability that the localization component present in the configuration will eventually process correctly the message and produce accurate localization information (encoded in processedOK), this property factors in both the reliability of all the topics through which the message has to be communicated from sensor to localization component (i.e., the probability that the message will be dropped along the way), as well as the inherent accuracy of each localization component.

The second and third formulas are analogous, and they both compute the accrued reward for every possible type of execution, which may end up dropping a message in any of the topics (captured by dropped), or processing the message either in degraded mode or accurately (processedOK).

7. Evaluation

We present in this section our evaluation, which aims at answering the four research questions posed in the introduction.

To evaluate the approach, we ran a prototype implementation of our proposal both on the TAS exemplar and the robotics scenario described earlier in the paper. The prototype that employs Alloy 4.2 for synthesizing configurations and PRISM 4.3.1 for behavioral and quantitative analysis. The experiments were run on an Intel Core i7 2.8GHz with 16 GB RAM.

7.1. Tele Assistance System (TAS)

We ran our analysis to compute the set of feasible solutions for TAS that meet the set of structural constraints described in Listing 1, using the set of service implementations described in Fig. 1.

7.1.1. Space Size and Computation Time

Table 8 shows that the overall computation time for generating and analyzing the solution space was approximately 20 seconds, out of which 8% was used to generate 90 configurations (Alloy) and 270 behavioral configuration DTMC models (90 x 3

possible values for the parameter that specifies number of retries after a failed service invocation), and the corresponding 270 MDP models for worst case analysis. Checking deadlock freeness and the six quantitative properties defined in Example 4 took approximately 92% of the time.

# Configurations	90
# Configuration behavioral models (DTMC)	270
# Configuration behavioral models (MDP)	270
Configuration behavior model generation time	1.661 s. (8.18 %)
Configuration behavioral model checking time (PRISM - DTMC)	7.66 s. (37.73 %)
Configuration behavioral model checking time (PRISM - MDP)	10.98 s. (54.08 %)
Total computation time	**20.301 s.**

Table 8: Problem instance size and computation time

7.1.2. Analysis Results

The plots on Figure 7 show the best cost that can be achieved in a system configuration when the response time and failure rate are constrained to the thresholds on the horizontal axes. The plots on the left show average case probabilities and rewards based on the analysis of a DTMC version of the model, whereas the right-hand side of the figure shows plots that correspond to worst case scenario analysis based on a MDP version of the model.

Moreover, results on the top of the figure correspond to analysis of all the possible behaviors of the configurations, whereas the plots in the bottom focus only on the behaviors that involve the selection of an alarm as the task to carry out (i.e., it does not include the behaviors that correspond to changing the drug dose or type). The reason why we analyze this additional case is showcasing that our analysis technique can be flexibly tailored to target specific behaviors of the system that may be of particular interest to architects. In this case, the rationale is that reliability of configurations is not so important when changing drug type or dose, but it is critical when raising an alarm, and we want to discover how the satisfaction of extra-functional requirements compares to the general case.

We start by focusing on the general-average case (Figure 7 top left). The plot shows that lower response times and higher reliability levels incur higher cost. This is aligned with expectations, and 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.

Moving on to the general-worst case analysis (top right), we can observe that the general observations made for the average case still hold, but the actual levels of reliability, timeliness, and cost minimization that can be guaranteed by configurations noticeably differ from the average case. The plot shows much more conservative values, with reliabilty values close to 0.96 for the cheapest configurations (compared to 0.98 for the average case), and much higher costs in general for analogous levels of timing and reliability.

The bottom half of the figure shows the plots that restrict analysis to the behaviors in which the selected task is raising the alarm. Hence, instead of using all the PCTL

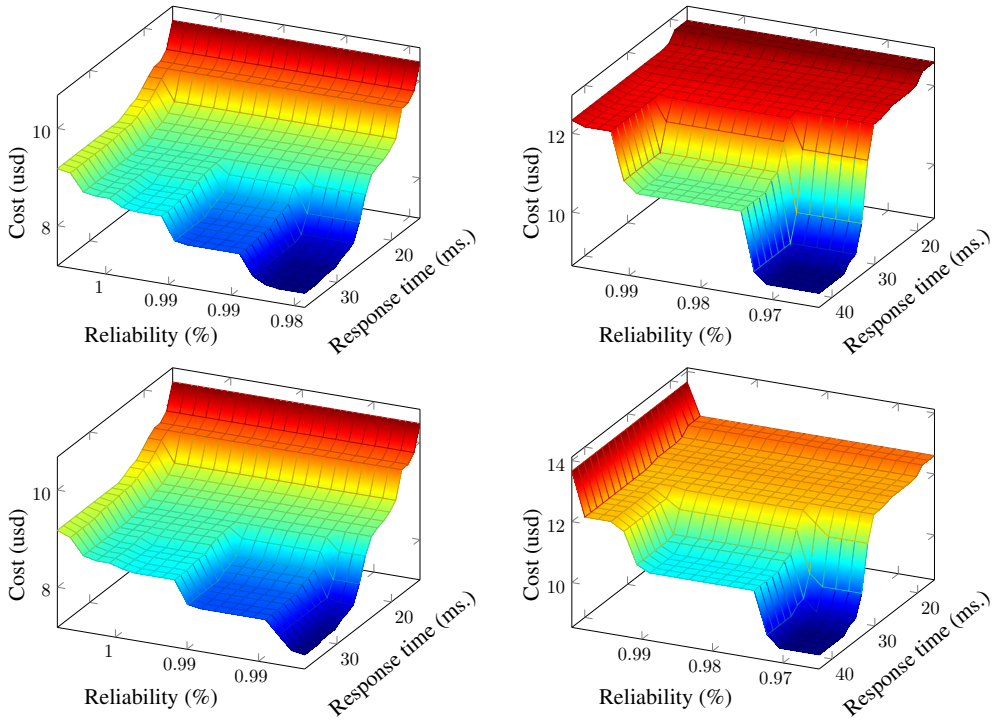

Figure 7: TAS analysis results: average case probabilistic guarantees (left) and worst case scenario analysis (right).

properties employed for the general case described in Example 4, we substitute the reliability property $(1 - \mathsf{P}_{=?}[\mathsf{F}\ \mathsf{serviceOK}])$ used to evaluate R1 by:

$$1 - \mathsf{P}_{=?}[(\mathsf{F}\ \mathsf{sendAlarm}) \Rightarrow (\mathsf{F}\ \mathsf{serviceOK})]$$

Hence, only execution paths in which there is an occurrence of an alarm invocation are taken into consideration to determine the probability of successfully completing the service (sendAlarm encodes an invocation of the alarm service). For worst-case scenario analysis, we substitute the probabilistic quantifier P by $\mathsf{P}_{\mathsf{min}}$.

In this case, average case results (bottom left) show similar results to the general case, with slightly higher reliability levels. However, we can see that there is a noticeable difference in worst case analysis (bottom right), which yields results that clearly differ from the general case. In particular, we can observe that the highest levels of reliability are preserved, but at a higher expense. This is consistent with the results shown in the configuration maps [8] shown in Figure 8 (which correspond to the four

[8] Configuration maps depict which configurations are optimal for the quantitative guarantees explored in every point of the state space. Every color in the map represents a configuration, which is optimal in the enclosed area of that color. It is useful to imagine the maps in Figure 8 as a zenithal planar projection of the plots in Figure 7.

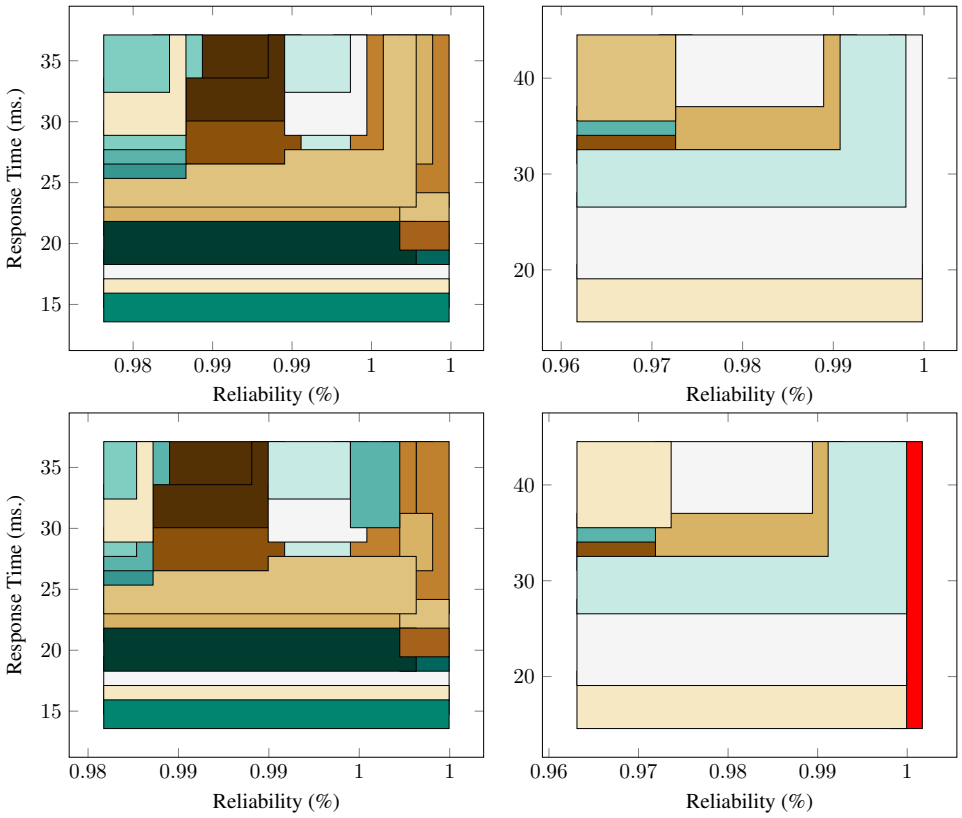

Figure 8: TAS analysis results: configuration maps

plots shown in Figure 7). The map in the bottom left shows a different configuration (marked in red) for the highest reliability level, compared to the map on the top left. Indeed, this configuration increases the overall reliability of the system by incorporating an additional alarm service (c.f. Figure 5), which in turn increases the cost of the configuration.

An additional observation obtained from the configuration maps in Figure 8 is that, while both the general case and the constrained send alarm case show different configurations (particularly, when approaching higher levels of reliability), worst case analysis shows that there is a much more reduced set of optimal configurations for guaranteeing different levels of reliability and timeliness, compared to the average case, which is much more fragmented. This can provide interesting insights to architects, who can consider the suitability of supporting certain variants of system configurations, which can provide benefits only in a limited number of cases.

7.2. Mobile Robotic System

We analyze the satisfaction of the requirements for the different configurations in the robotics architecture, according to the properties described in Table 7, and the com-

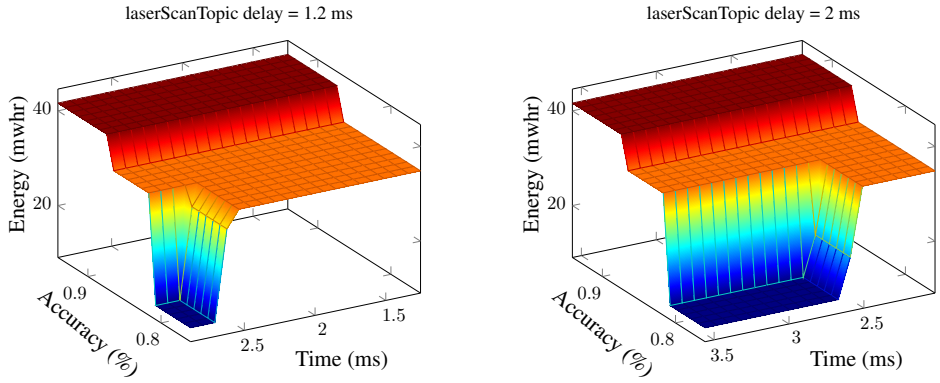

Figure 9: ROS TurtleBot configuration analysis results

ponent and topic properties described in Tables 5 and 6.

7.2.1. Space Size and Computation Time

Table 9 shows that the overall computation time for generating and analyzing the solution space was approximately 1.6 seconds, out of which 30% was used to generate six configurations (Alloy) and 12 behavioral configuration DTMC models (6 x 2 possible values for the parameter that specifies latency of the laserScanTopic). Checking deadlock freeness and the three quantitative properties defined in Table 7 took approximately 69% of the time.

# Configurations	6
# Configuration behavioral models (DTMC)	12
Configuration behavior model generation time	0.504 s. (30.93 %)
Configuration behavioral model checking time (PRISM - DTMC)	1.125 s. (69.07 %)
Total computation time	**1.629 s.**

Table 9: Problem instance size and computation time

7.2.2. Analysis Results

Figure 9 shows the results of analyzing the energy cost, accuracy, and timeliness tradeoffs of different configurations for two different execution contexts: in the first one (left), the system is operating in normal conditions and the delay for the laserScanTopic is 1.2 ms, whereas in the other one (right), we assume an operating context under which the topic is saturated and its latency is 2 ms.

Results show clear difference in extra-functional requirement satisfaction in the two cases. While the low latency case shows that less accurate configurations are energy efficient and timely (maximum processing time is around 2.7 ms), the high latency case shows that timeliness decreases noticeably (maximum 3.5 ms) for that same level of accuracy.

These results are consistent with our expectations, and are backed by the configuration maps shown in Figure 10, which show an increased coverage of Configuration

2 (camera + markerLocalization) in the high latency case, in detriment of Configuration 0 (lidar+mrpt), which is dominated in that region of the space, since it provides similar or lower levels of timeliness and energy efficiency in this case.

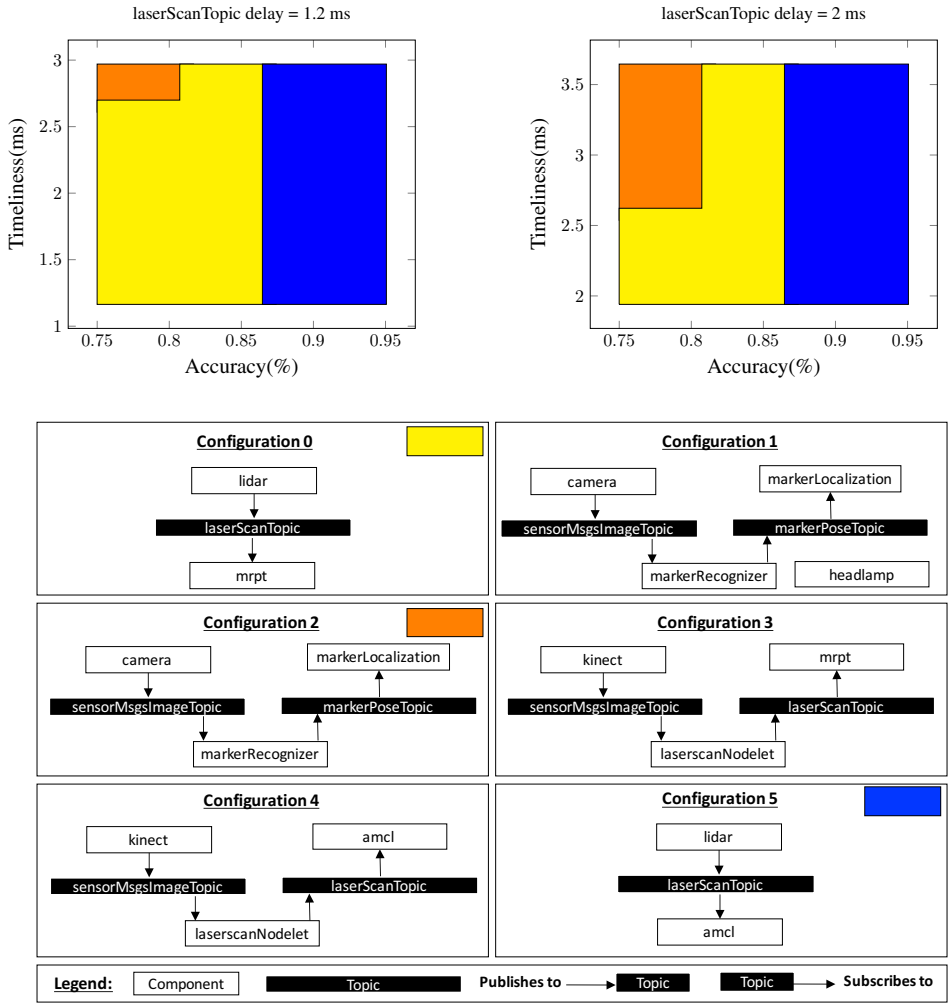

Figure 10: ROS TurtleBot configuration maps

7.3. Discussion

Our evaluation has shown that our approach is able to bridge the gap between structural synthesis and quantitative verification, providing the mechanisms required to generate and analyze quantitative guarantee tradeoff spaces for families of software systems (RQ1 and RQ2).

An architect can take the results provided by our approach 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, or energy efficiency and reliability of software components in robotics systems.

Instantiating the approach in two different domains, with different architectural styles, quality objectives, sources of uncertainty, and types of analysis (average and worst-case scenario) provides a promising indicator of generality (RQ3), even if the current embodiment of the approach is inspired by a specific model of formal architectural description (Acme), and probabilistic formalisms (DTMC and MDP). 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 (e.g., OCL). In terms of behavior descriptions, DTMC and MDP constrain the analysis to a discrete time model and average case of probabilities/rewards, although adaptations can be carried out to adapt behavioral analysis to other probabilistic behavior descriptions such as PTA ot CTMC for finer-grained time analysis.

An additional aspect that our evaluation has underlined is that there can be important differences between disparate classes of quantitative guarantees (e.g., average vs. worst case), which point to the relevance of methods that integrate formal quantitative verification with structural synthesis, as opposed to methods based only on dynamic analysis or simulation, which can provide only probabilistic average approximations with relative precision (RQ4). Moreover, evaluation has shown that the configurations that provide good guarantees for a certain set of objectives do not necessarily coincide across different types of analysis, further emphasizing the importance of providing architects with tools and techniques to enable the exploration of quantitative guarantees across architectural design spaces.

8. Threats to validity

The approach is based on a specific style of structural description (Alloy) and probabilistic behavioral formalisms (DTMC and MDP). However, the constructs employed to formalize structures are fairly standard and synthesis of configurations is adaptable to other languages/models (e.g., OCL). Concerning behavior descriptions, the fact that the approach was successfully instantiated for different probabilistic formalisms and analyses hints at feasibility of adapting the approach to other formalisms such as continuous-time Markov chains (CTMC) or probabilistic timed automata (PTA) for finer-grained time analysis.

An additional assumption that our approach relies on is that there is an existing architectural specification of the components and connectors in the system, as well as behavioral specifications of their interfaces. Although such specifications might not be readily available in all systems, architecture discovery (with formal underpinnings that rely on architectural styles) from running systems has already been covered in previous research [19]. Learning of probabilistic behavioral specifications from observed system behavior has also been covered in the literature [20]. The latter approach enables extraction of probabilistic behavior descriptions for component and connectors in the form of probabilistic automata (DTMC and MDP among them) that are compatible with probabilistic model checking and our approach.

Focusing on internal validity, the degree of formal assurance on configurations provided by the approach is computationally expensive, and entails risks derived from the cost both of configuration synthesis and behavior analysis (derived from exploring potentially large state spaces of individual configuration behavior). These risks can be mitigated by exploiting the hierarchical relations that are naturally present in software designs, in which components interact in a structured way [21]. Hence, synthesis of different subsystems with local constraints can be done independently and then composed, reducing the cost of configuration synthesis.

9. Related Work

Uncertainty in software architecture is a subject that has been broadly explored in recent years and includes approaches that have tackled uncertainties due to epistemic uncertainties that have to do with the imprecision and incompleteness of information [22, 23]. Due to space constraints, we cannot provide an exhaustive compilation of related works, and focus instead on the subset of works akin to our proposal. These works can be categorized into: (i) formalization of architectural styles, and (ii) architecture-based quantitative analysis and optimization techniques that deal with aleatoric uncertainties.

(1) Formalization of architectural styles: Formalization of styles has been explored to define formal semantics of modeling languages. Kim and Garlan [17] propose an automatic translation from Acme into Alloy relational models on which they verify properties implied by the style. Wong et al. [24] 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 [6, 8]. Bagheri and Sullivan [6] 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. [8] 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 [25, 26, 27]. *PerOpteryx* [26] 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* [25] uses an evolutionary algorithm for optimizing the architecture of embedded systems. *DeepCompass* [27] 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., [28]) 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.

A different category of approach based on formal quantitative verification is based on product line reliability analysis [29, 30, 31, 32]. Such approaches can analyze collections of system designs encoded in a feature model individually or collectively. A recent approach to continuous-time probabilistic design synthesis [33] uses a template-based solution to generate and analyze alternative system designs, but does not emphasize high level modeling, assuming an existing encoding of design options in a set of discrete variables and leaving out of scope any systematic enforcement of structural variants in the designs. Compared with these product line and template-based approaches, our proposal focuses not only on variability, but also on structure, being able to synthesize design alternatives that satisfy complex structural constraints from much more abstract specifications that do not require detailed description of variability points.

Other approaches [7, 34] have recently combined architecture synthesis with simulation and dynamic analysis to provide estimates of quantitative properties of system variants. *TradeMaker* [34] 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. This shortcoming is particularly evident in the case of worst case scenarios, which our approach is able to analyze with as shown in Section 7, in contrast with the other approaches described above.

10. 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.

Our evaluation shows feasibility (RQ1 and RQ2) and indicates potential for generality derived from the application of our approach to case studies in different domains and types of probabilistic analysis (RQ3). However, one of the most interesting findings stems from the fact that our results show that configurations that are optimal in providing average case guarantees do not coincide in general with configurations that optimize worst case guarantees (only accessible via exhaustive state space exploration techniques like probabilistic model checking, in contrast with related work that are based on simulation or dynamic analysis). This underlines the importance of incorporating the class of approach we propose to the array of tools available to software architects (RQ4).

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 [35] 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 second 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 [21], or incremental analysis techniques [36]. 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 [37]. 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 [38]).

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] D. Garlan, Software engineering in an uncertain world, in: Proc. of the Workshop on Future of Software Engineering Research, FoSER, 2010.

[2] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, Ros: an open-source robot operating system, in: ICRA WS on Open Source Software, 2009.

[3] S. Mahdavi-Hezavehi, M. Galster, P. Avgeriou, Variability in quality attributes of service-based software systems: A systematic literature review, Inf Softw Technol 55 (2). doi:http://dx.doi.org/10.1016/j.infsof.2012.08.010.

[4] M. Shaw, D. Garlan, Software architecture - perspectives on an emerging discipline, Prentice Hall, 1996.

[5] D. Jackson, Alloy: A lightweight object modelling notation, ACM Trans. Softw. Eng. Methodol. 11 (2). doi:10.1145/505145.505149.

[6] H. Bagheri, K. J. Sullivan, Model-driven synthesis of formally precise, stylized software architectures, Formal Asp. Comput. 28 (3). doi:10.1007/s00165-016-0360-8.

[7] V. Dwivedi, D. Garlan, J. Pfeffer, B. Schmerl, 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. doi:10.1109/ITNG.2014.107.

[8] S. Maoz, J. O. Ringert, B. Rumpe, 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. doi:10.1145/2491411.2491414.

[9] M. Z. Kwiatkowska, G. Norman, D. Parker, Stochastic model checking, in: FM for Performance Evaluation, 7th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, Vol. 4486 of LNCS, Springer, 2007.

[10] M. Z. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: Computer Aided Verification, Vol. 6806 of LNCS, Springer, 2011.

[11] J. Cámara, D. Garlan, B. R. Schmerl, Synthesis and quantitative verification of tradeoff spaces for families of software systems, in: A. Lopes, R. de Lemos (Eds.), Software Architecture - 11th European Conference, ECSA 2017, Canterbury, UK, September 11-15, 2017, Proceedings, Vol. 10475 of Lecture Notes in Computer Science, Springer, 2017, pp. 3–21.

[12] D. Weyns, R. Calinescu, 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. doi:10.1109/SEAMS.2015.27.

[13] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, P. Spoletini, Validation of web service compositions, IET Software 1 (6). doi:10.1049/iet-sen:20070027.

[14] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, G. Zavattaro, Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (1) (2016) 3:1–3:36.

[15] D. Garlan, R. T. Monroe, D. Wile, 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. doi:10.1145/782010.782017.

[16] J. Warmer, A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, Addison-Wesley, 2003.

[17] J. Kim, D. Garlan, Analyzing architectural styles, J Syst Software 83 (7). doi:10.1016/j.jss.2010.01.049.

[18] M. Z. Kwiatkowska, D. Parker, Automated verification and strategy synthesis for probabilistic systems, in: D. V. Hung, M. Ogawa (Eds.), Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings, Vol. 8172 of Lecture Notes in Computer Science, Springer, 2013, pp. 5–22.

[19] H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich, R. Kazman, Discotect: A system for discovering architectures from running systems, in: A. Finkelstein, J. Estublier, D. S. Rosenblum (Eds.), 26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom, IEEE Computer Society, 2004, pp. 470–479. doi:10.1109/ICSE.2004.1317469.
URL https://doi.org/10.1109/ICSE.2004.1317469

[20] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, B. Nielsen, Learning deterministic probabilistic automata from a model checking perspective, Machine Learning 105 (2) (2016) 255–299.

[21] E. Kang, A. Milicevic, D. Jackson, Multi-representational security analysis, in: Proc. of the 24th Symposium on Foundations of Software Engineering, FSE, 2016.

[22] N. Esfahani, K. Razavi, S. Malek, Dealing with uncertainty in early software architecture, in: W. Tracz, M. P. Robillard, T. Bultan (Eds.), 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012, ACM, 2012, p. 21. doi:10.1145/2393596.2393621.
URL https://doi.org/10.1145/2393596.2393621

[23] L. Cheung, L. Golubchik, N. Medvidovic, G. S. Sukhatme, Identifying and addressing uncertainty in architecture-level software reliability modeling, in: 21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA, IEEE, 2007, pp. 1–6. doi:10.1109/IPDPS.2007.370524.
URL https://doi.org/10.1109/IPDPS.2007.370524

[24] S. Wong, J. Sun, I. Warren, J. Sun, A scalable approach to multi-style architectural modeling and verification, in: 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008), 2008.

[25] A. Aleti, S. Bjornander, L. Grunske, I. Meedeniya, 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. doi:10.1109/MOMPES.2009.5069138.

[26] A. Martens, H. Koziolek, S. Becker, R. Reussner, Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms, in: Int. Conf. on Performance Engineering, WOSP/SIPEW, ACM, 2010.

[27] E. Bondarev, M. R. V. Chaudron, E. A. de Kock, Exploring performance tradeoffs of a jpeg decoder using the deepcompass framework, in: 6th WS on Software and Performance, WOSP, ACM, 2007.

[28] A. D. MacCalman, P. T. Beery, E. P. Paulo, A systems design exploration approach that illuminates tradespaces using statistical experimental designs, Syst. Eng. 19 (5).

[29] C. Ghezzi, A. M. Sharifloo, Model-based verification of quantitative non-functional properties for software product lines, Information & Software Technology 55 (3) (2013) 508–524.

[30] P. Chrszon, C. Dubslaff, S. Klüppelholz, C. Baier, Profeat: feature-oriented engineering for family-based probabilistic model checking, Formal Aspects of Computing.

[31] T. Castro, A. Lanna, V. Alves, L. Teixeira, S. Apel, P. Schobbens, All roads lead to rome: Commuting strategies for product-line reliability analysis, Sci. Comput. Program. 152 (2018) 116–160.

[32] A. Lanna, T. Castro, V. Alves, G. N. Rodrigues, P. Schobbens, S. Apel, Feature-family-based reliability analysis of software product lines, Information & Software Technology 94 (2018) 59–81.

[33] R. Calinescu, M. Ceska, S. Gerasimou, M. Kwiatkowska, N. Paoletti, Designing robust software systems through parametric markov chain synthesis, in: 2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017, IEEE, 2017, pp. 131–140.

[34] H. Bagheri, C. Tang, K. J. Sullivan, Trademaker: automated dynamic analysis of synthesized tradespaces, in: 36th Int. Conf. on Software Engineering, ACM, 2014.

[35] E. M. Hahn, H. Hermanns, B. Wachter, L. Zhang, Param: A model checker for parametric markov models, in: Computer Aided Verification, Springer, 2010.

[36] H. Bagheri, S. Malek, Titanium: efficient analysis of evolving Alloy specifications, in: Proc. of the 24th Symposium on Foundations of Software Engineering, FSE 2016, 2016.

[37] K. Johnson, R. Calinescu, S. Kikuchi, 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. doi:10.1145/2465449.2465456.

[38] J. Cámara, D. Garlan, B. Schmerl, A. Pandey, Optimal planning for architecture-based self-adaptation via model checking of stochastic games, in: 30th ACM Symposium on Applied Computing (SAC), 2015.

Appendix A: Robotics Architectural Style Alloy Model

```
1
2  // Components and connectors
3  abstract sig component {pub: set topic, sub: set topic}
4  abstract sig topic {pub: set component, sub: set component}
5
6  fact {all c:component,t:topic | (c in t.sub <=> t in c.sub) and (c in t.pub <=> t in c.pub) }
7
8  // Components
9
10 // Sensing
11 abstract sig sensing extends component {}
12 lone sig kinect extends sensing {}
13 lone sig lidar extends sensing {}
14 lone sig camera extends sensing {}
15
16 // Localization
17 abstract sig localization extends component {}
18 lone sig amcl extends localization {}
19 lone sig mrpt extends localization {}
20 lone sig markerLocalization extends localization{}
21 lone sig laserscanNodelet extends component {}
22
23 // Auxiliary
24 lone sig markerRecognizer extends component {}
25 lone sig headlamp extends component {}
26
27 // Connectors (Topics)
28 lone sig laserScanTopic extends topic{}
29 lone sig sensorMsgsImageTopic extends topic{}
30 lone sig markerPoseTopic extends topic{}
31
32 // Constraints - PUB-SUB
33 pred publishesTo[c:component, t:topic] { t in c.pub }
34 pred onlyPublishesTo[c:component, t:topic] { publishesTo[c,t] and all t':topic-t | not publishesTo[c,t'] }
35
36 pred subscribesTo[c:component, t:topic] { t in c.sub }
37 pred onlySubscribesTo[c:component, t:topic] { subscribesTo[c,t] and all t':topic-t | not subscribesTo[c,t']
       }
38
39 pred doesNotSubscribe[c:component] { all t:topic | not subscribesTo[c,t] }
40 pred doesNotPublish[c:component] { all t:topic | not publishesTo[c,t] }
41
42 fact { all t:topic | t in component.pub+component.sub }
43
44 // Constraints - ROS-TurtleBot
45
46 // Sensing constraints
47 fact { all c:lidar | onlyPublishesTo[c,laserScanTopic] }
48 fact { all c:kinect | onlyPublishesTo[c,sensorMsgsImageTopic] }
49 fact { all c:camera | onlyPublishesTo[c,sensorMsgsImageTopic] }
50 fact { all c:sensing | doesNotSubscribe[c] }
51
52 // Aux component constraints
53 fact { all c:laserscanNodelet | onlyPublishesTo[c,laserScanTopic] }
54 fact { all c:laserscanNodelet | onlySubscribesTo[c,sensorMsgsImageTopic] }
55
56 fact { all c:markerRecognizer | onlySubscribesTo[c,sensorMsgsImageTopic] }
```

```
57 fact { all c:markerRecognizer | onlyPublishesTo[c,markerPoseTopic] }
58
59 fact { doesNotSubscribe[headlamp] and doesNotPublish[headlamp] }
60
61 // Localization constraints
62 fact { all c:localization-markerLocalization | onlySubscribesTo[c,laserScanTopic] }
63 fact { all c:markerLocalization | onlySubscribesTo[c,markerPoseTopic] }
64 fact { all c:localization | doesNotPublish[c] }
65
66 pred config{
67   (some camera or some kinect) <=> some sensorMsgsImageTopic
68   (some laserscanNodelet or some lidar) <=> some laserScanTopic
69   some markerRecognizer <=> some markerPoseTopic
70   some kinect <=> some laserscanNodelet
71   some camera <=> some markerLocalization
72   some camera <=> some markerRecognizer
73   some headlamp => some camera
74   one sensing
75   one localization
76 }
```

Listing 5: Robotics architecture style specification in Alloy.