HaiQ: Synthesis of Software Design Spaces with Structural and Probabilistic Guarantees

Javier Cámara
Department of Computer Science, University of York
javier.camaramoreno@york.ac.uk

ABSTRACT

Formal methods used to validate software designs, like Alloy, OCL, and B, are powerful tools to analyze complex structures (e.g., architectures, object-relational mappings) captured as sets of relational constraints. However, their applicability is limited when software is subject to *uncertainty* (derived, e.g., from lack of control over third-party components, interaction with physical elements). In contrast, quantitative verification has emerged as a powerful way of providing *quantitative guarantees* about the performance, cost, and reliability of systems operating under uncertainty. However, quantitative verification methods do not retain the flexibility of relational modeling in describing structures, forcing engineers to trade structural exploration for analytic capabilities that concern probabilistic and other quantitative *guarantees*. This paper contributes a method (HaiQ) that enhances structural modeling/synthesis with quantitative guarantees in the style provided by quantitative verification. It includes a language for describing structure and (stochastic) behavior of systems, and a temporal logic that allows checking probability and reward-based properties over sets of feasible design alternatives implicitly described by the relational constraints in a HaiQ model. We report the results of applying a prototype tool in two domains, on which we show the feasibility of synthesizing structural designs that optimize probabilistic and other quantitative guarantees.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; *Software design tradeoffs*; • **Theory of computation → Verification by model checking**.

KEYWORDS

Uncertainty, guarantees, quantitative verification, relational modeling, probabilistic model checking, Alloy, PRISM, HaiQ, M-PCTL

ACM Reference Format:
Javier Cámara. 2020. HaiQ: Synthesis of Software Design Spaces with Structural and Probabilistic Guarantees. In *8th International Conference on Formal Methods in Software Engineering (FormaliSE '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3372020.3391562

FormaliSE '20, October 7–8, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7071-4/20/05.
https://doi.org/10.1145/3372020.3391562

1 INTRODUCTION

Modern software-intensive systems are commonly affected by *uncertainties* derived, for instance, from the lack of control over third-party components (e.g., residing in the cloud), humans in the loop, and complex interactions between software and physical elements in cyber-physical systems [25]. Designing such systems in a way that provides *guarantees* that concern the satisfaction of functional requirements while achieving an acceptable balance between multiple extra-functional properties is still an open problem because designers typically rely on intuition to navigate these design spaces. However, getting these wrong can lead to systems that fail to guarantee the qualities required by users.

This is a difficult problem because system structure and stochastic behavior, as well as *their interplay* can distinctly impact functional and extra-functional requirement satisfaction when systems are subject to *uncertainties* [11, 22, 49]. Currently, joint systematic analysis of structural and quantitative/probabilistic *guarantees* of systems is limited in existing methods. On the one hand, formal methods used to validate software designs like Alloy [31], Z [50], B [2], VDM [9] and OCL [51], share a common conceptual foundation that enables designers to implicitly describe collections of alternative structural designs (e.g., software architectures [43, 53], database object-relational mappings [6], network and security models [5, 54]) as sets of relational constraints and exploring them systematically, but they are not equipped for analyzing stochastic behavior. On the other hand, *quantitative verification* or *probabilistic model checking* [13, 23, 38] can analyze quantitative guarantees of systems (e.g., on performance, reliability) under uncertainty, but use notations (e.g., PRISM [39], PEPA [27], cpGCL [34]) that do not retain the flexibility in describing structures of relational methods. Recent product line reliability analysis approaches [16, 17, 26, 41] improve on flexibility by introducing systematic treatment of variability, but are not equipped to synthesize or explore complex structures, and lack languages tailored to check sophisticated properties across design spaces (i.e., temporal logics employed like PCTL [38] can capture properties only about a single model, not collections of individual variants). Other works in quantitative optimization of architectures [3, 6, 8, 10, 11, 21, 22, 29, 44, 45] can in some cases synthesize complex structures [6, 21], but are not compatible with formal verification and can only provide *estimates* of probabilistic properties that could differ from actual guarantees (e.g., worst case) available only via exhaustive state-space exploration.

This situation forces designers to trade systematic exploration of alternative structural designs for analytic capabilities related to probabilistic and other quantitative *guarantees*.

To improve on this situation, we investigate in this paper the following research questions:

(RQ1) How can we automate analysis of quantitative/probabilistic *guarantees* across sets of structural variants of system designs?
(RQ2) If feasible, is such an approach general enough to be applicable to different domains and probabilistic formalisms/analyses?
(RQ3) What are the trade-offs of employing this class of approach with respect to existing probabilistic modeling/analysis techniques?

This paper explores these questions by contributing an approach that integrates structural exploration with the analytic capabilities of probabilistic model checking. The contribution is twofold. First, we introduce HaiQ, a language to describe both structure and behavior of systems, including probabilistic and other quantitative attributes (e.g., time, cost). Second, we introduce *Manifold Probabilistic Computation-Tree Logic* (M-PCTL), an extension of the probabilistic logic PCTL that allows quantifying probability and reward-based properties over the set of alternative structural designs implicitly described by a HaiQ model. The approach is embodied in a tool that implements analysis both of average probabilities/rewards (based on discrete-time Markov chains or DTMC) and best/worst case probabilities/rewards (based on Markov decision processes or MDP). We applied the approach to problems in service-based systems and distributed adaptive systems.

2 RELATED WORK

Related approaches can be categorized into: (i) relational modeling and structural verification, (ii) probabilistic/quantitative verification, and (iii) structural-quantitative optimization.

Relational Modeling and Structural Verification (RMSV). Pioneering formal methods like Z [50] have a very expressive notation that limits automated theorem proving, which often requires guidance from an experienced user. Alloy [31] can be considered a first-order subset of Z with finite models, which makes it automatically analyzable. Other techniques and languages like VDM [9] and OCL [51] are also based on first-order logic and include tools that can support design-time analysis that allows exhaustive search over a finite space of cases, similarly to Alloy. All the aforementioned methods have a strong focus on structure, and are limited in terms of analyzing complex concurrent behaviors. In contrast, methods like DynAlloy [24] and Event-B [1] include more sophisticated constructs to efficiently analyze behaviors on top of structures. Despite being able to analyze concurrent behaviors, these methods are not equipped to capture probabilistic aspects of system behavior.

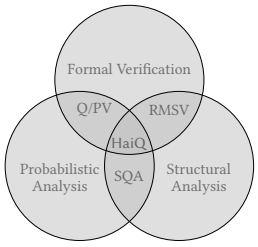

Figure 1: Relation between HaiQ and other works

Quantitative/probabilistic analysis and verification (Q/PV). Modern probabilistic model checkers like PRISM [39] and STORM [20] capture models using textual languages like PRISM, PEPA or low-level matrix-based descriptions. Stochastic variants of UPPAAL [19] employ graphical specification languages. Although these tools efficiently analyze probabilistic system behaviors, their specification languages and analysis mechanisms do not separate process instance attributes from process types, hampering reusability. Möbius [18] overcomes this limitation by exploiting domain-specific models. However, none of the tools described are equipped for exploring collections of system variants. In contrast, product line reliability analysis approaches [16, 17, 26, 41] can analyze collections of system designs encoded in feature models individually or collectively. A recent approach to continuous-time probabilistic design synthesis [12] uses a template-based solution to analyze alternative designs, but assumes an existing encoding of design options in a set of discrete variables and does not systematically enforce any structural constraints in the designs. Compared with these approaches, HaiQ's focus is not only on variability, but also on structure, being able to synthesize design alternatives that satisfy complex structural constraints. Moreover, HaiQ includes a temporal logic (M-PCTL) that specifically targets analysis across collections of alternatives and could also complement analysis in feature model-based approaches, streamlining specification of complex properties to identify interesting regions of the solution space.

Structural quantitative analysis and optimization (SQA). There is extensive related work on model-based performance prediction [7] and optimization of quantitative aspects of structures (e.g., architectures) [29] that typically use mechanisms like stochastic search and Pareto analysis [3, 8, 10, 11, 22, 44, 45]. These and other approaches in systems engineering (e.g., [42]) can optimize quantitative aspects of designs, but do not support structural synthesis. Other approaches [6, 21] combine structural synthesis with simulation and dynamic analysis to provide estimates of quantitative properties of design variants. These approaches share with ours the idea of synthesizing a solution space from a set of constraints and analyzing individual solutions independently. However, they are not compatible with formal verification, and hence are not equipped to provide quantitative guarantees, which rely on checking sophisticated properties (typically encoded as temporal logic formulas) via numerical methods and exhaustive state space exploration.

In summary, our approach supports streamlined analysis of probabilistic guarantees in the presence of complex structural variability, going beyond existing approaches in which limited structural variability is captured in templates and feature models (Figure 1).

3 OVERVIEW OF THE APPROACH

The complementary strengths of relational and quantitative verification approaches can be appreciated in the client-server example of Figure 2. The top part shows an Alloy specification in which two types or *signatures* (which represent groups of objects in Alloy) `c` (clients) and `s` (servers), extend an abstract signature `comp` (component), which includes a relation `l` (i.e., a component is *linked* to others). A predicate `clientserver` includes the constraints that describe how instances of `c` and `s` relate (in this case, clients can only be linked to servers, and the relation is symmetric). This specification implicitly describes the set of structures shown on the right of the figure for a scope of maximum one client and two servers. Although this is a trivial example, real system designs include large sets of components and complex structural constraints that result

```
abstract sig comp {l: some comp}
some sig c, s extends comp{}
pred clientserver {
 disj[c.l,c] //only
 disj[s.l,s] //c<->s
 l = ~l } //symmetric
run clientserver for 1 c, 2 s
--------------------------------
dtmc
module c0
 x: bool init false;
 [r] (!x) -> 1:(x'=true);
endmodule
module s0
 y: bool init false;
 [r] (!y) -> 0.4:(y'=false)
           + 0.6:(y'=true);
endmodule
```

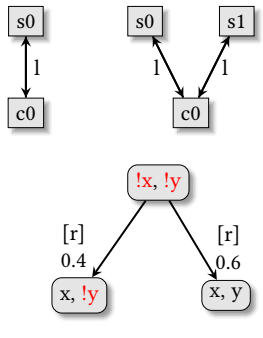

Figure 2: Client-server Alloy and PRISM specifications

in vast collections of possible system structures that can be automatically synthesized. Despite their structural flexibility, these approaches are not equipped to analyze probabilistic behaviors.

In contrast, the bottom of the figure shows a PRISM DTMC [38] model with two processes or *modules* (client `c0` and server `s0`) that synchronize on a shared request action `[r]`. Boolean variables `x` and `y` encode that the request has been correctly issued and received, respectively. The request is always correctly sent by the client (probability 1), whereas in the server, the action has two possible outcomes specified with different probabilities (0.4/0.6). Probabilistic model checkers are equipped with mechanisms to compose in parallel the behavior of such stochastic processes (Figure 2 bottom right) and analyze *probabilistic and other quantitative guarantees* captured in temporal logics like PCTL and CSL [38]. Despite this analytical advantage, in this kind of specification, *modules* denote processes among which synchronization is "hardwired" via shared action names (i.e., relations among processes are fixed explicitly in models). This specification style results in rigid structures, compared to the ones that we can specify in Alloy, meaning that if a designer wants to explore quantitative guarantees in different combinations of a collection of processes arranged in different topologies, she has to *generate and analyze* different alternatives manually, or use ad-hoc solutions that do not provide any structural guarantees about the resulting models. These ad-hoc approaches are labor-intensive, error-prone, and simply infeasible when process behaviors and structural system constraints are non-trivial.

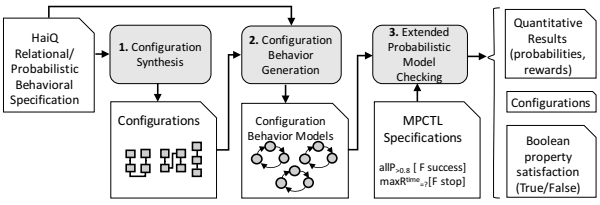

Figure 3: Overview of the approach

To investigate automated joint analysis of structural guarantees, (probabilistic) quantitative guarantees, and their interaction, our approach must: (i) provide simple language constructs, yet expressive enough to capture structure, stochastic behavior, and their dependencies in models, (ii) allow systematic checking of sophisticated properties on such models, and (iii) be general enough to be applied to different domains and probabilistic analyses.

To address these requirements, we propose an approach that includes: (i) a high-level language (HaiQ) for modeling collections of probabilistic models as *relational/probabilistic behavioral specifications*, and (ii) a language for specifying (and associated mechanisms for checking) quantitative temporal properties (probability- and reward-based) on collections of probabilistic models (*manifold probabilistic computation tree logic* or M-PCTL).

We embody these languages and analysis mechanisms in a relational probabilistic model checker that combines synthesis of structural designs from relational constraints with model checking of individual designs obtained from the structural specification of synthesized design variants (Figure 3). Our proposal includes three stages: **(i) Configuration Synthesis**, during which topological descriptions of configurations are synthesized from the relational constraints of a HaiQ model, **(ii) Configuration Behavioral Model Generation**, which uses configuration descriptions synthesized in the previous stage as a blueprint to generate a probabilistic behavioral model for every configuration (supported model types are DTMC and MDP), **(iii) Extended Probabilistic Model Checking**, which uses as input properties specified in M-PCTL to provide quantitative results about the set of configuration behavioral models generated. Supported analyses include average probability-based and reward-based guarantees for DTMC and worst/best case probability-based and reward-based guarantees for MDP.

The HaiQ **Modeling Language.** HaiQ specifications juxtapose static (or structural) relational descriptions with blocks of probabilistic behavioral specification that can exploit the information about system structure encoded in relations.

To illustrate the main ideas behind the language, we focus on a simple HaiQ specification that builds on our introductory example (Figure 4). Here, the possible structures of the system are similar to the ones captured by the Alloy specification in Figure 2. In contrast with Alloy, HaiQ signatures include blocks of behavior specifications (between "`</`" and "`/>`") that enclose guarded probabilistic commands. This is because *every signature in* HaiQ *is also a process type*. Let us emphasize that, in contrast with the languages employed by probabilistic model checkers, *behavioral descriptions in* HaiQ *are at a higher level of abstraction and denote process types, not process instances*. This means that *a* HaiQ *model implicitly describes a process type hierarchy associated with signatures in which behaviors can be inherited and extended*. In the example, this can be observed in the command included in signature `comp`, which is defined in a generic manner and inherited by signatures `c` and `s`. In particular, the synchronization action `r` in the command is prefixed by the relation `l` defined in the static part of the specification, indicating that the *synchronization matches only processes that correspond to signature instances in the relation (independently of the topology under which they are instantiated), rather than being "hard-wired"* as we saw in the PRISM example in Figure 2. The top-right of Figure 4 shows the two possible structure instantiations of the specification, in which process instances `s0` and `s1` synchronize with `c0` on actions generated based on the instantiation of relation `l`. The corresponding probabilistic state machines that result from the parallel composition of the behaviors of the `c-s` instances are shown in the right-bottom.

The details of the modeling language are described in Section 4.

```
abstract sig comp {l: some comp}
</ var x: bool init false;
   formula p;
   [l:r] (!x) -> p : (x'=false)
            + 1-p : (x'=true); />
some sig c extends comp {}
</ formula p=0; />
some sig s extends comp {}
</ formula p=0.4; />
pred clientserver {
 disj[c.l,c] //only
 disj[s.l,s] //c<:->s
 l = ~l } //symmetric relation
run clientserver for 1 c, 2 s
```

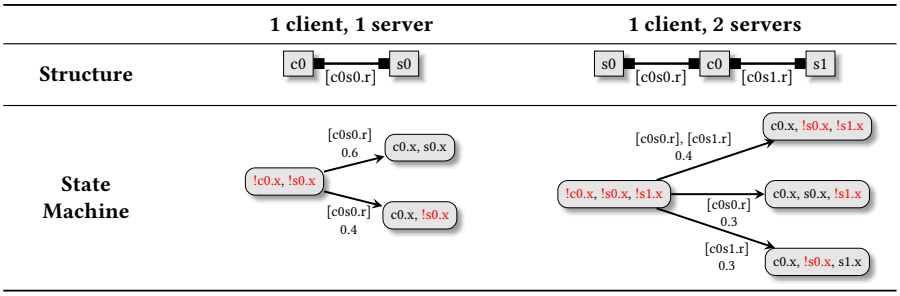

Figure 4: Client-server HaiQ specification (left), implicit structure collection and corresponding state machines (right)

Manifold Probabilistic Computation Tree Logic (M-PCTL). Ultimately, a HaiQ model defines a set of probabilistic state machines (DTMC or MDP), where each one results from the parallel composition of processes instantiated in accordance with each topology that satisfies the model's structural constraints. Formulas in existing probabilistic temporal logics like PCTL capture properties of a single probabilistic state machine. Hence, our work extends existing languages to *check properties across collections of probabilistic state machines*, rather than individual instances. M-PCTL is an extension of PCTL including new operators and quantifiers that enable designers to check properties like "across all configurations, the minimum probability that a service will never time out is greater than 0.95", and "there exists a robot configuration that requires less than 3.2 whr to achieve its goal."

In our client-server example, allP$_{\geq 0.6}$[F all c.x ∧ some s.x] is a property that can be interpreted as "for all configurations, the probability that all requests will be correctly sent by the client and at least one of them will be correctly received by a server is at least 0.6" In this case, the property check returns true, because for the two possible configurations (with one and two servers), the probabily that [F all c.x ∧ some s.x] is satisfied is exactly 0.6 in both cases. Moreover, *M-PCTL properties can also return (sets of) structures that optimize or satisfy some quantitative property*. For instance, the property SmaxP [F all c.x ∧ some s.x] returns a set including a description of the one-server and the two-server configurations, since both of them have the same probability of satisfying [F all c.x ∧ some s.x] and hence maximize the value of the property in this case because there are no other feasible solutions.

The syntax and semantics of M-PCTL are described in Section 5.

4 THE HAIQ MODELING LANGUAGE

In this section, we formalize the modeling language and illustrate its main features by modeling a simple network security scenario introduced by Kwiatkowska et al [37]. The scenario models the progression of a virus infecting a network formed by a grid of N×N nodes. The virus remains at a node that is infected and repeatedly tries to infect any uninfected neighbors by first attacking the neighbor's firewall and, if successful, trying to infect the node.

In the network there are 'low' and 'high' nodes divided by 'barrier' nodes that scan the traffic between them. Initially, only one corner 'low' node is infected. A graphical representation of the network when N=3 is given in Figure 5.

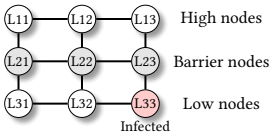

Figure 5: Network virus infection scenario

4.1 Language Features

4.1.1 Signatures as Structural and Behavioral Building Blocks. The modeling language is tailored for the incremental construction of structured probabilistic models. The basic unit of specification is the *signature*, which contains structural and behavioral code fragments. The structural part of the language is a subset of the static part of Alloy (c.f. [32], 2.1), in which every signature field is a set that defines implicitly a binary relation between the instances of signatures in which the field is declared and the elements in the set. For example, the field conn declared in the first line of the node definiton listing encodes a relation capturing the fact that a node can be connected to one or more neighboring nodes (i.e., conn is implicitly encoding the arcs in the network graph shown in Figure 5). Constraining fields of signatures to this type of relation suffices to capture sophisticated topologies and limits the complexity of specifications. Note that the declaration states that signature node is *abstract*, so it can be extended by other signatures:

```
abstract sig node {conn: some node}
</ enum modes:{uninfected, breached, infected};
   var s:[modes] init uninfected;
   formula detect=node_detect;
   [conn:attack] (s=uninfected) -> 1-detect : (s'=breached)
    /* firewall attacked */        + detect : (s'=s);
   [] (s=breached) ->  1-infect : (s'=uninfected)
                     + infect : (s'=infected); //firewall breached
   [conn:attack] (s=infected) -> true; //attack from infected node />
```

The second part of the signature encodes its behavior and is surrounded by the process block delimiters "</" and "/>". Process blocks include a set of local variables that can be bounded integers (including enumerated types, which are internally treated as sets of integer constants) or booleans, as well as formulas that are used as shorthand for complex expressions to avoid code duplications. In the example, variable s specifies the state of the node (uninfected, breached, infected), and formula detect encodes the probability that the virus is detected by the firewall in the node. Constants node_detect and infect are global constants that encode a default probability value for firewall detection and node infection.

Finally, the process block includes a set of probabilistic guarded commands that describe the behavior of the signature. Each command describes the possible transitions of the process in states that

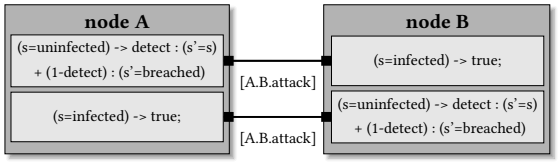

Figure 6: Sample synchronization scheme between nodes

satisfy its guard (specified before "->"), and a set of updates that specify each one of the possible outcomes as sets of new values for (primed or post-) state variables (e.g., `s'=infected`). Each update is associated with a probability, and unspecified state variables in an update indicate no change in value. In the first command of the example, the guard specifies that when the node is not infected, it can be attacked with two possible outcomes. With probability `detect`, the firewall will remain unbreached, and with probability `1-detect`, the firewall will be successfully breached by the attack.

Note that the prefixing of the action by `conn` indicates that the behavior specified by this command is replicated for each of the `node` instances in the relation. Hence, for an example system structure with a node instance `A` in which `A.conn={B,C}`, each command in the process block labeled as `[conn:attack]` is denoting two alternative actions `[A.B.attack]` and `[A.C.attack]`[1] that can synchronize with actions of the same name in nodes `B` and `C`.

4.1.2 Synchronization. Different signature processes in a specification synchronize on shared action names. The third command of the `node` behavior definition contains an action labeled `attack` prefixed by `conn`, just like in the first command that we saw earlier. In this case, the command models the transition in which the infected node attacks a neighboring node with probability 1 (unspecified probability in a command defaults to 1). Consider an example of two neighbor node instances `A` and `B`, in which `A.conn={B}` and `B.conn={A}` (Figure 6). In the scheme, nodes synchronize on the same action name `[A.B.attack]` both when the node attacks and receives an attack from a neighboring node. However, the commands synchronize pairwise because the guards in commands on the same module are mutually exclusive. Hence, whenever node `A` attacks, node `B` receives the attack, and vice versa.

4.1.3 Subtyping and Extension. Signatures can be extended to incorporate new structural and behavioral elements. Similarly to Alloy, a signature that extends another one is considered its subtype. However, in this case subtyping also implies inheritance not only of the structural elements of the extended signature, but also of its behavioral definition. The following listing encodes the subtypes of node for the network scenario:

```
sig barrierNode extends node {} </formula detect=barrier_detect;/>
sig highNode, lowNode extends node
one sig infectedNode extends lowNode </var s:[modes] init infected;/>
```

First, signature `barrierNode` extends signature `node` to create a special type of node in which the attack detection probability is different to the rest of the nodes (encoded in constant `barrier_detect`). This is achieved by overriding the original definition of the formula `detect` in the abstract signature `node`.

[1]The syntax of these labels is used only for illustration purposes. Internally, the HaiQ compiler generates a unique identifier using a commutative function (the identifier for inputs (A,B) is the same as for (B,A)), resulting in a label that can be employed on both endpoints of process synchronization (c.f. Section 4.2.2).

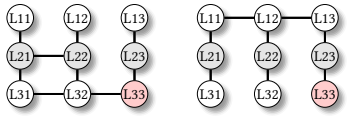

Figure 7: Alternative legal network configurations

In contrast, `highNode` and `lowNode` are subtypes of `node` that preserve all its structural and behavioral elements intact because we are only interested in formally distinguishing them as special locations, but do not require to incorporate any special behaviors.

Finally, we incorporate `infectedNode` as a subtype of `lowNode` to include an infected node in the scenario. This type is constrained to a singleton (keyword `one`) and overrides the declaration of the node state variable, which is now initialized as `infected`.

4.1.4 Topology. The possible topologies of design alternatives that can be generated by a HaiQ model are defined implicitly by a set of structural constraints expressed in relational logic sentences.

```
pred virus{ all n:node | node in n.*conn //network is connected
  (no iden & conn) and (conn = ~conn) //no self-loops, symmetric conn
  disj[lowNode.conn,highNode] } //disjoint low-high node connections
```

In the example, the first constraint imposes that all nodes must be reachable in the network from any other node. The second constraint specifies that connections are symmetric and that no node must be connected to itself, whereas the third one imposes that low nodes and high nodes must never be directly connected. Figure 7 shows two of the many legal configurations that can be synthesized from the virus infection scenario model (apart from the one shown in Figure 5) for a scope of `N=3` by running: `run virus for exactly 3 lowNode, exactly 3 barrierNode, exactly 3 highNode.`

4.1.5 Rewards. HaiQ can be used to reason, not just about the probability that a collection of behaviors specified by a model behaves in a certain fashion, but about a wider range of quantitative measures relating to behaviour. This includes properties such as "expected time" or "expected power consumption."

This is achieved by augmenting signatures with rewards, which are real values associated with certain model states or transitions. We include in our example a reward that enables us to quantify the number of attack attempts carried out by the different nodes:

```
abstract sig node {conn: some node}
</ ... reward attacks [conn:attack]  true : 1; />
```

Every time a transition in which the action `attack` is executed from a state in which the guard before the colon is satisfied (in this case, the guard `true` is satisfied in all states), a reward of one unit will be accrued for `attacks`. The prefixing of the action by relation `conn` results in reward cumulation for any of the instances of the action executed among any arbitrary pair of nodes.

4.2 Formal Model

A HaiQ specification is formally characterized as a tuple (Σ, C, δ), where Σ is a set of signatures that implicitly define a hierarchy of types (including their behavior), C is a set of structural constraints expressed in first-order predicate logic that determines the allowed topologies when instantiating the hierarchy defined by Σ, and the total function $\delta : \Sigma \rightarrow \mathbb{N}$ is a scope that determines the maximum allowable number of instances of every signature type in Σ.

Definition 4.1 (Signature). A signature σ is a tuple $(\sigma^{\uparrow}, \mathcal{R}, \beta)$:
• $\sigma^{\uparrow} \in \Sigma \cup \{\bot\}$ references the supertype signature that the current signature extends (if any).
• $\mathcal{R}$ is a set of tuples (id, σ') typed by $sVarIds \times \Sigma$, where $sVarIds$ is the set of identifiers that form a static variable namespace. Each tuple in the set implicitly declares a relation between every instance of σ and a set of the instances of σ' (possibly constrained by C).
• β is a behavior specification.

Definition 4.2 (Behavior specification). A behavior specification is a tuple $(\mathcal{V}, \mathcal{K}, \mathcal{L})$, where $\mathcal{V}$ is a set of local state variables, $\mathcal{K}$ is a set of commands that describe the behavior of the signature, and $\mathcal{L}$ is a set of reward functions for states and transitions.

We define the extension function for a signature set element $X \in \{\mathcal{R}, \beta.\mathcal{V}, \beta.\mathcal{K}, \beta.\mathcal{L}\}$ as $ext(\sigma, X) \triangleq ext(\sigma.\sigma^{\uparrow}.X) \uplus \sigma.X$ if $\sigma.\sigma^{\uparrow} \neq \bot$, and $ext(\sigma, X) \triangleq \sigma.X$ otherwise. We obtain a "flat" set of expanded signature types (denoted Σ_f) by applying the function $flat(\sigma) \triangleq (\bot, ext(\sigma, \mathcal{R}), (ext(\sigma, \beta.\mathcal{V}), ext(\sigma, \beta.\mathcal{K}), ext(\sigma, \beta.\mathcal{L})))$ to all signatures in Σ. For convenience, we use the shorthand σ_f for $flat(\sigma)$.

Definition 4.3 (Command). A command is a tuple $(\alpha, r, g, \mathcal{U})$, s.t.:
• $\alpha \in actIds \cup \{\bot\}$ is a label drawn from an action namespace. If $\alpha = \bot$, the command encodes an internal action.
• $r \in \sigma_f.\mathcal{R} \cup \{\bot\}$ is a relation that constrains synchronization on action α to instances in r. If $r = \bot$, the scope of the command labeled by α is global and the action can therefore synchronize with any other process that includes a command with the same label.
• g is a guard built over variables in $\sigma_f.\mathcal{V}$. • $\mathcal{U}$ is a set of probabilistic updates. Each update is a pair (λ, u), where $\lambda \in [0, 1]$ is a probability and u is a set of assignments of fresh values to local state variables (v, v) ($v \in \sigma_f.\mathcal{V}$, and v is a value typed by v's datatype).

All commands $[r : \alpha]\ g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$ must satisfy the invariants $\sum_{i:\{1\ldots n\}} \lambda_i = 1$, and $\alpha = \bot \Rightarrow r = \bot$.

4.2.1 Configurations. The specification (Σ, C, δ) implicitly defines a set of *configurations* or structures. Each configuration is composed by a set of isignature instances, connected in a topology that satisfies C. A signature instance σ is just a labeled process with fresh variables that corresponds to the behavior definition of σ_f.

Definition 4.4 (Instance). An instance of a signature σ is a pair (l, β), where $l \in procIds$ is a label drawn from a process namespace, and $\beta = \sigma_f.\beta_v$ is a process specification with fresh variables. We denote the type of an instance n of signature σ by $type(n) \triangleq \sigma_f$.

The set of nodes in a configuration take values in the possible Σ-signature instances, denoted by $\mathcal{I}_\Sigma$.

Definition 4.5 (Configuration). A configuration is a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ satisfying the constraints imposed by C and δ, where $\mathcal{N} \subseteq \mathcal{I}_\Sigma$ is a set of nodes, and $\mathcal{E}$ is a set of labeled arcs typed by $sVarIds \times \mathcal{I}_\Sigma \times \mathcal{I}_\Sigma$ that capture relations between instances.

4.2.2 From Configurations to Stochastic Behavior Models. A HaiQ specification implicitly describes a collection of probabilstic models $\mathcal{M}$. Each model in $\mathcal{M}$ is obtained as the parallel composition of the signature behaviors in one possible instantiation of Σ that satisfies the constraints imposed by C and δ.

Algorithm 1 describes the generation of a probabilistic state machine from a configuration c. The algorithm starts by extending every process n in the configuration to produce the wiring for communication with the rest of the processes. The new set of commands for each process $\mathcal{K}_v$ is initialized in line 3 with the commands that do not contain any references to relations (i.e., both commands for internal actions and global events). Then, the algorithm substitutes every command k that refers to a relation ($k.r \neq \bot$) by a set of commands $\mathcal{K}^*$, where every command contains the resolved reference to every element of the relation (lines 4-8). Resolution of references is done via function *uid* in line 7, which generates a unique label id for a pair of labels. The operation is commutative, so that synchronization on the generated id is possible on both ends of the communication (n and n'). Finally, the algorithm returns the probabilistic state machine that corresponds to the standard parallel composition [47] of the extended processes for the instances in the configuration.

Algorithm 1 Configuration Stochastic State Machine Generation

```
1:  N_v := ∅
2:  for all n : c.N do
3:      K_v := {k : type(n).β.K | k.r = ⊥}
4:      for all k : type(n).β.K\K_v do
5:          K* := ∅
6:          for all {n' : c.N | ∃(x, n, n') ∈ c.E, x ∈ sVarIds} do
7:              K* := K* ∪ {(uid(n.l, n'.l), ⊥, k.g, k.U)}
8:          end for
9:          K_v := K_v ∪ K*
10:     end for
11:     N_v := N_v ∪ {(n.l, (n.V, K_v, n.L))}
12: end for
13: return ||_{n:N_v} n.β
```

5 M-PCTL

Probabilistic Computation Tree Logic (PCTL) [30] is used to quantify properties related to probabilities and rewards in *single system specifications described as a probabilistic state machine* (e.g., DTMC, MDP, probabilistic timed automata or PTA). This section introduces Manifold PCTL, which in contrast targets *quantification across collections of design alternatives* that correspond, in this case, to the state machines generated from the set of structures that satisfy the constraints of a HaiQ specification (c.f. Section 4.2).

In the remainder of this section, we first overview a version of PCTL extended with a reward quantifier targeted at checking properties over DTMC and MDP extended with reward structures [4], and then we build on it to introduce M-PCTL.

5.0.1 PCTL. In the syntax definition below, Φ and ϕ are respectively, formulas interpreted over states and paths of a probabilistic state machine M extended with rewards, i.e., (M, ρ). Properties in PCTL are specified exclusively as state formulas. Path formulas have an auxiliary role on probability and reward quantifiers P/R:
$\Phi ::= \texttt{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{P}_{\sim pb}[\phi] \mid \mathsf{R}^{r}_{\sim rb}[\phi] \quad \phi ::= \mathsf{X}\Phi \mid \Phi\ \mathsf{U}\ \Phi$,
where a is an atomic proposition, $\sim \in \{<, \leq, \geq, >\}, pb \in [0, 1], rb \in \mathbb{R}^{+}_{0}$, and $r \in \rho$.

Intuitively, $\mathsf{P}_{\sim pb}[\phi]$ is satisfied in a state s of M if the probability of choosing a path starting in s that satisfies ϕ (denoted as $Pr_s(\phi)$ [2]) is within the range determined by $\sim pb$, where pb is a probability bound. Quantification of properties based on $\mathsf{R}^{r}_{\sim rb}$ works analogously, but considering rewards, instead of probabilities. The

[2] See [38] for details. In the following, we write $Pr_s(\phi)$ as $Pr(\phi)$ for simplicity.

intuitive meaning of path operators X and U is analogous to the ones in other standard temporal logics. Additional boolean and temporal operators are derived in the standard way (e.g., $\mathsf{F}\Phi \equiv \mathsf{true}\ \mathsf{U}\ \Phi$).

5.0.2 M-PCTL. The main idea behind M-PCTL is to extend checking of probability- and reward-based properties to collections of models. Hence, quantification occurs over a pair $(\mathcal{M}, \rho)$, where $\mathcal{M}$ is a set of models, and ρ is a set of reward functions.

M-PCTL includes three types of formula. Similarly to PCTL, it includes path (ϕ) formulas (which are the same as in PCTL) and state (Φ) formulas, but also an additional type of set formula (Ψ) that returns the collection of models that satisfy a particular quantitative constraint. The syntax of M-PCTL is:

$$\Phi ::= \mathsf{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid$$
$$\mathsf{someP}_{\sim pb}[\phi] \mid \mathsf{allP}_{\sim pb}[\phi] \mid \mathsf{maxP}_{\sim pb}[\phi] \mid \mathsf{minP}_{\sim pb}[\phi] \mid$$
$$\mathsf{someR}^{r}_{\sim rb}[\phi] \mid \mathsf{allR}^{r}_{\sim rb}[\phi] \mid \mathsf{maxR}^{r}_{\sim rb}[\phi] \mid \mathsf{minR}^{r}_{\sim pb}[\phi]$$
$$\Psi ::= U \mid \Psi \bigcup \Psi \mid \Psi \bigcap \Psi \mid \Psi^{C} \mid \mathsf{SallP}_{\sim pb}[\phi] \mid$$
$$\mathsf{SmaxP}[\phi] \mid \mathsf{SminP}[\phi] \mid \mathsf{SallR}^{r}_{\sim rb}[\phi] \mid \mathsf{SmaxR}^{r}[\phi] \mid \mathsf{SminR}^{r}[\phi]$$

Concerning state formula quantifiers, allP and someP determine if the evaluation of $Pr(\phi)$ on all or some model in $\mathcal{M}$ satisfies $\sim pb$, whereas maxP determines if the maximum probability evaluated across elements of $\mathcal{M}$ satisfies $\sim pb$. We define their semantics as:

$$[\![\mathsf{someP}_{\sim pb}[\phi]]\!] \equiv \exists M \in \mathcal{M} : Pr_M(\phi) \sim pb$$
$$[\![\mathsf{allP}_{\sim pb}[\phi]]\!] \equiv \forall M \in \mathcal{M} : Pr_M(\phi) \sim pb$$
$$[\![\mathsf{maxP}_{\sim pb}[\phi]]\!] \equiv \max_{M \in \mathcal{M}} Pr_M(\phi) \sim pb,$$

where $Pr_M(\phi)$ denotes the evaluation of the probability $Pr(\phi)$ on model M. The analogous reward-based quantifiers someR^r, allR^r, maxR^r, and minR^r, are defined over the expected reward measure of PCTL, instead of the probabilistic one Pr (c.f. [38]). The use of maxP/minP and maxR/minR quantifiers without a bound implies the quantification of the actual maximum/minimum probability or reward for the path formula ϕ, e.g.:

$$[\![\mathsf{maxP}[\phi]]\!] \equiv \max_{M \in \mathcal{M}} Pr_M(\phi).$$

In set formulas, U denotes the universe of models in $\mathcal{M}$, whereas Ψ^C is the standard complement operator of set algebra. Set subtraction is derived as $\Psi_1 \backslash \Psi_2 \equiv \Psi_1 \cap \Psi_2^C$. The semantics of the main quantifiers in set formulas is:

$$[\![\mathsf{SallP}_{\sim pb}[\phi]]\!] \equiv \{M : \mathcal{M} \mid Pr_M(\phi) \sim pb\}$$
$$[\![\mathsf{SmaxP}[\phi]]\!] \equiv \arg\max_{M \in \mathcal{M}} Pr_M(\phi)$$

In the virus scenario example, we can write for instance a property to check which network structures have a probability below 0.4 of having all high nodes infected after 50 time units:

$$\mathsf{resilient} \equiv \mathsf{SallP}_{\leq 0.4}\ [\mathsf{F}^{<=50}\ \mathsf{all\ highNode.s = infected}]$$

The checking of formulas that employ probability and reward quantifiers in M-PCTL can also be constrained to subsets of $\mathcal{M}$ by employing a scope operator $\langle\Psi\rangle$ as a prefix for formulas, e.g.:

$$[\![\langle\Psi\rangle\ \mathsf{someP}_{\sim pb}[\phi]]\!] \equiv \exists M \in [\![\Psi]\!] : Pr_M(\phi) \sim pb.$$

Hence, the use of a quantifier without the scope operator in a state formula Φ is equivalent to $\langle U\rangle\ \Phi$.

The use of the scope operator enables composition of formulas to check complex properties on the space of design alternatives, e.g., $\langle\mathsf{resilient}\rangle\ \mathsf{maxR}^{\mathsf{attacks}}[\mathsf{F\ all\ highNode.s = infected}]$.

The property above determines the expected maximum number of attacks required by the virus to infect all high nodes across all possible network structures that satisfy the resilient property.

Best/Worst case scenario. The quantifiers P/R described above are based on the average probabilistic and reward measures of PCTL [38]. However, PCTL can also be used to analyze maximum/minimum probabilities/rewards over probabilistic formalisms that feature nondeterminism like MDP or PTA using the quantifiers $\mathsf{P}_{\mathsf{max}}/\mathsf{P}_{\mathsf{min}}$ and $\mathsf{R}^{\mathsf{r}}_{\mathsf{max}}/\mathsf{R}^{\mathsf{r}}_{\mathsf{min}}$[40]. We define analogous versions of all probability and reward-based quantifiers for M-PCTL to enable best and worst case scenario analyses. For instance, property $\mathsf{SminP}_{\mathsf{max}}[\mathsf{F}^{\leq \mathsf{t}}\ \mathsf{all\ highNode.s = infected}]$ can be considered as the best configuration in terms of worst case scenario in the network virus infection example, i.e., the configuration that minimizes the maximum probability of high node infection across all feasible configurations.

6 EVALUATION

In this section, we evaluate our approach in case studies from different domains to determine its applicability. We embodied our approach in the HaiQ analyzer [14], a prototype tool that implements the generation of design collections from HaiQ specifications as described in Section 4.2, as well as checking of M-PCTL properties on them. The tool is implemented in Java, and its backend uses Alloy and PRISM's APIs for synthesizing configurations and model checking properties on their behavior models, respectively.

We describe the application of the tool to our running example, a service-based system and a distributed self-protecting system. The scenarios were chosen because they instantiate different types of structure (architecture, network) and sources of uncertainty. We finish this section with a discussion of our results.

6.1 Case Studies

6.1.1 Virus Network Infection. Figure 8 shows both worst case and average case scenario probabilistic guarantees (resulting from MDP-based and DTMC-based analysis, respectively) of the best and worst legal network configurations in our running example (in black and red, respectively). Dashed lines represent average case, and solid lines represent worst-case scenarios.

The plot on the left shows the probability over time that all high nodes in the network will become infected. The average case DTMC analysis is much more optimistic than the actual worst case, which gives a more realistic approximation of the minimum infection probability that the system can guarantee. Note that probabilistic estimates in average case analysis can be approximated with some degree of precision using statistical model checking and monte carlo trace-based simulation methods, whereas worst case analysis requires a technique like probabilistic model checking that performs exhaustive state space exploration.

The plot on the right shows a different property in which the probability analyzed is that of having some high node infected, instead of all of them. If we focus for instance on the best configurations in terms of worst case scenario guarantees, the property $\mathsf{minP}_{\mathsf{max}}[\mathsf{F}^{\leq \mathsf{t}}\ \mathsf{some\ highNode.s = infected}]$ (solid black) minimizes across all legal configurations, the maximum probability within the configuration of having some high node infected after t seconds (we assume a time discretization parameter of one second).

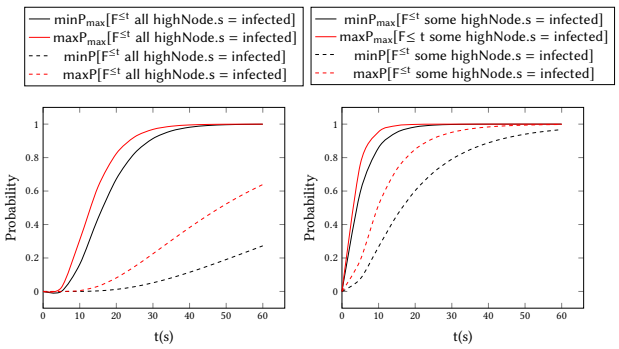

Figure 8: Network virus infection results

6.1.2 Tele-Assistance System (TAS). The goal of the TAS exemplar system [52] is tracking a patient's vital parameters to adapt drug type or dose when needed, and taking actions in case of emergency. TAS combines three service types in a workflow. When TAS receives a request that includes the vital parameters of a patient, its *Medical Service* analyzes the data and replies with instructions to: (i) change the patient's drug type, (ii) change the drug dose, or (iii) trigger an alarm for first responders in case of emergency. When changing the drug type or dose, TAS notifies a local pharmacy using a *Drug Service*, whereas first responders are notified via an *Alarm Service*.

The following excerpt shows the TAS workflow modeled as a signature in which static fields correspond to service bindings, and its behavior specification defines a set of local variables that keep track of the workflow status:

```
one sig TASWorkflow {MSBindings: some MedicalAnalysisService, ...}
</ enum tasks:{notSelected, getVitalParams, buttonMsg};
   enum analysisResultTypes:{none, patientOK, ..., sendAlarm};
   var task:[tasks] init notSelected; ...
   var MSInvoked, ..., workflowOK, workflowDone : bool init false;
   [pickTask] (task=notSelected) -> //PickTask
      0.5: (task'=getVitalParams) + 0.5: (task'=buttonMsg);
   [] (task=buttonMsg) & (!MSInvoked) ->
      (MSInvoked'=true) & (analysisResult'=sendAlarm);
   [MSBindings:analyzeData] (task=getVitalParams) & (!MSInvoked) ->
      (MSInvoked'=true); //PickTask selected getVitalParams
   [MSBindings:analysisResultPatientOK]  (MSInvoked) ->
      (analysisResult'=patientOK) & (workflowOK'=true)
    & (workflowDone'=true); ...
   [MSBindings:analysisResultSendAlarm]  (MSInvoked) ->
      (analysisResult'=sendAlarm); ...
   [MSBindings:timeout] (timeouts=0) & (MSInvoked) ->
      (workflowDone'=true); ... />
```

Calls to service operations are prefixed by the service binding relation (e.g., `analyzeData` is prefixed by `MSBinding`), so that the actual binding between the workflow and the services will be automatically created by the tool when configurations are generated.

The functionality of each service type in TAS is provided by third parties with different levels of performance, reliability, and cost. The metrics employed for the quality attributes are the percentage of service failures for reliability, and service response time for performance. Service providers can be created as abstract signatures that encode these attributes as formulas, and include a constraint to include a binding on the service side to the workflow:

```
abstract sig ServiceProvider {WorkflowBinding: one TASWorkflow}
fact {all sp:ServiceProvider, w:TASWorkflow |
   sp in w.ServiceBindings <=> w=sp.WorkflowBinding}
</ formula failure_rate, response_time, cost; />
```

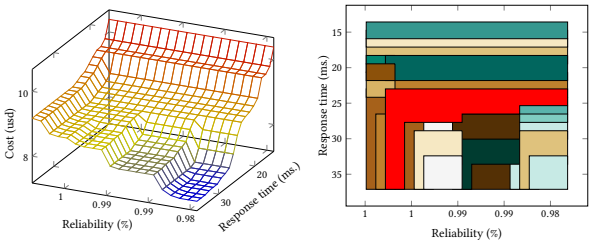

Figure 9: TAS analysis results

Service providers are subtyped by the types of service involved in the composition. Service invocation includes two probabilistic outcomes that model the possibility of service failure:

```
abstract sig MedicalAnalysisService extends ServiceProvider {}
</ var serviceOK: bool init false;
   var ready : bool init true;
   [WorkflowBinding:analyzeData] (ready) ->
        failure_rate: (serviceOK'=false) & (ready'=false)
    + 1-failure_rate: (serviceOK'=true) & (ready'=false);
   [WorkflowBinding:analysisResultPatientOK](!ready) & (serviceOK) ->
      (serviceOK'=false) & (ready'=true); ...
   reward costRew [WorkflowBinding:analyzeData] true : cost;
   reward timeRew [WorkflowBinding:analysisResultPatientOK]
      true : response_time; />
```

Every concrete service extends a service type encoded with a set of attribute values. The use of the quantifier `lone` indicates that the use of every instance is optional, giving flexibility to use alternative services of the same type in the composition:

```
lone sig MS1 extends MedicalAnalysisService{}
</ formula failure_rate=0.06, response_time=22, cost=9.8; />
```

Finding an adequate design for the system entails understanding the tradeoff space by finding the set of system configurations that satisfy: (i) structural constraints, e.g., the *Drug Service* must 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 a change to the pharmacy), and (iii) quality requirements, which can be formulated as a combination of quantitative constraints and optimizations, e.g.: (R1) The average failure rate must not exceed `fr` %, (R2) The average response time must not exceed `rt` ms, and (R3) Subject to R1 and R2, the cost should be minimized.

We can automatically search the design space to find the best legal configurations with respect to these requirements by checking the composite M-PCTL property constrained_mincost:

$$\text{reliable} \equiv \text{SallP}_{\leq \text{fr}}[\text{F some TASWorkflow.workFlowOK}]$$

$$\text{performant} \equiv \text{SallR}^{\text{timeRew}}_{\leq \text{rt}}[\text{F some TASWorkflow.workFlowDone}]$$

$$\text{mincost} \equiv \text{minR}^{\text{costRew}}[\text{F some TASWorkflow.workFlowDone}]$$

$$\text{constrained\_mincost} \equiv \langle \text{reliable} \cap \text{performant} \rangle \text{ mincost}$$

The formulas labeled as reliable and performant obtain the set of configurations that satisfy the reliability and performance requirements R1 and R2, respectively. Then, we can quantify the minimum cost entailed by these joint requirements by scoping the quantification of the third property mincost to the subset of designs that satisfy the first two properties. For obtaining the configuration(s) that minimize cost for the specified performance and reliability levels, we substitute the quantifier in mincost by SminR.

Figure 9 shows analysis results. The plot on the left shows the minimized cost of configurations for different levels of constraints on response time and reliability. It was computed by checking property constrained_mincost in the region of the state space in

which `fr` $\in [0.98, 1]$ and `rt` $\in [15, 35]$. As expected, higher response times and lower reliability correspond to lower costs, whereas peaks in cost are reached with lowest failure rates and response times.

The plot on the right is a map that shows which configurations best satisfy design criteria. Out of the set of 90 configurations that can be generated for TAS, only 24 satisfy the criteria in some subregion of the state space. If we consider that designers are interested e.g., in systems with response times ≤26ms, and with a reliability of ≥99%, we can determine which are the configurations that best satisfy constraints by checking the version of constrained_mincost in which the quantifier for mincost is SminR, with `rt` = 26 and `fr` = 0.01 (highlighted in red in the figure).

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

6.1.3 Distributed Self-Protecting System. Some large-scale systems are composed of federated entities that use self-adaptation to improve their behavior with respect to defined quality standards. For example, Netflix's software infrastructure includes deployments in multiple regions controlled by a local manager (Scryer) to provision the resources required for maintaining resilience against changing customer traffic [33, 46].

We apply our approach on a model of a Collective Adaptive System (CAS) with similar cooperative systems that defend against an external attack (e.g., DoS). We analyze the resilience that results from the selection of different communication topologies to disseminate security information for preemptive adaptation, quantified as the probability that all members of the CAS survive the attack.

In the scenario [28], an external attacker uses a defined amount of available resources to attempt to breach members of the CAS (e.g., by placing a high number of requests). Each CAS member has the ability to detect the attack, defend itself against it by employing a fixed set of defense resources, and has the ability to notify other members of the CAS of the attack. Once a CAS member is notified, it will adapt and become invulnerable to the attempted breach.

```
 some sig sam {conn: some sam, attackVector: one attacker}
</ enum modes:{normal, attackDetected, compromised, adapted};
   var status:[modes] init normal; ... formula detect;
   [attackVector:attack] (resources>0) & (status=normal) -> //attacked
          detect: (status'=attackDetected) & (resources'=deplete);
      + 1-detect: (status'=normal) & (resources'=deplete);
   [conn:alert] (status=attackDetected) -> true; //notify attack
   [] (resources>0) & (status=attackDetected) ->
      (status'=adapted); //adapt if attack detected (or notified)
   [conn:alert] (resources>0) & (status=normal) -> //receive alert
         CHANNEL_RELIABILITY: (status'=attackDetected)
     + 1-CHANNEL_RELIABILITY: (status'=normal);
   [] (resources=0) -> (status'=compromised); />
```

The excerpt above shows the basic encoding of a local self-adaptive manager (signature `sam`). Environmental and system conditions, such as the reliability of communication channels and the sensitivity of detection mechanisms, can play an important role in the emergent behavior of the CAS and are explicitly captured by parameters that affect the outcome of certain actions (e.g., constant `CHANNEL_RELIABILITY` limits the ability of a local manager to be notified of an attack by other managers, whereas formula `detect` encodes the probability that a `sam` will detect the attack).

We can create models with different communication topologies by encoding a basic set of constraints that impose a connected network without self-loops and with symmetric relations:

```
(all s:sam | sam in s.*conn) and (no iden & conn) and (conn = ~conn)
```

And then add extra constraints like the ones shown on the right of Figure 10 for each one of the topologies to encode.

We can reason about the resilience of the CAS by encoding a property that determines the topologies with the highest chance of attack survival: resilience ≡ maxP[F all sam.status = adapted].

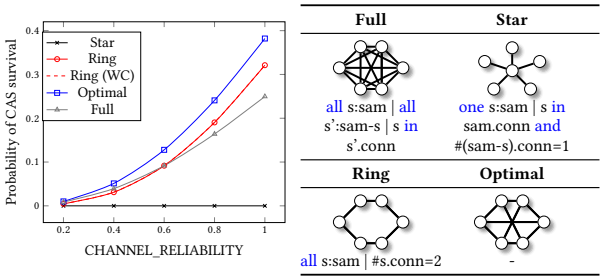

Figure 10: CAS topologies and resilience results

The left-hand side of Figure 10 shows the resilience of the different topologies for values of channel reliability in the set {0.2, 0.4, 0.6, 0.8, 1}. As expected, all topologies present increased chances of survival with higher channel reliability, saving the star topology, for which resilience is always zero for the amount of `sam` and attacker resources employed for the experiments (the central node is a weak link that hampers inter-node communication if compromised). To obtain the optimal resilience (and its corresponding topology) for our scenario, we check the M-PCTL resilience property, and an analogous property that employs the SmaxP quantifier on a model that does not include any extra constraints.

6.2 Discussion

Basing on our results, in this section we discuss the research questions posed in the introduction and threats to validity.

6.2.1 (RQ1) Feasibility. We have shown that *raising the level of abstraction at which probabilistic models are described and queried can effectively enable joint analysis of structural and quantitative guarantees of design spaces.* In terms of modeling, this is achieved by incorporating language constructs that allow structural relations to be referenced from elements of (probabilistic) behavioral specifications. In terms of analysis, incorporating novel quantifiers to check properties on collections of models in probabilistic temporal logics enables streamlined specification of sophisticated properties to study guarantees across the solution space.

6.2.2 (RQ2) Generality. Our results show that *our approach is applicable to systems in different domains, in which uncertainty is introduced by disparate sources.* Our approach is *particularly suited to problems in which structure/topology is relevant and multiple instances of similar, but different, components that exhibit probabilistic behavior (i.e., with different parameters or slight variations in behavior) interact in complex ways.* Moreover, we have shown that the approach can be applied to different analyses (average and best/-worst case probabilities and rewards) and probabilistic formalisms (MDP and DTMC). This is effectively enabled by the fact that HaiQ

operates one level higher in the abstraction ladder, with respect to existing description languages in probabilistic model checking, and that the target language (PRISM) in which configuration behavioral models are generated (Step 2 in Figure 3, implementing Algorithm 1) supports alternative underlying semantics for MDP and DTMC (c.f. [47]). Similarly, M-PCTL extends PCTL, which can be naturally used to check properties both in DTMC and MDP.

6.2.3 (RQ3) Tradeoffs. With respect to alternative approaches to analyze probabilistic behavior, HaiQ comes with tradeoffs in terms of effort, reusability, computational cost, and analytical capabilities:

Effort. In HaiQ, structure and probabilistic behavior are expressed in a compact manner, and their combined analysis is streamlined. This contrasts with ad-hoc solutions that demand developing specific infrastructure and are error-prone. The analysis of a CAS scenario similar to the one in Section 6.1.3 [28] required combining the PRISM preprocessor [48] with scripts that demanded topologies to be encoded as matrixes in separate text files, leading to multiple trial and error rounds (due to errors in matrix encodings, script tuning). For TAS, the problem has also been solved employing a custom template engine and a python script that generates probabilistic models based on analysis of Alloy specifications [15]. These solutions *required weeks of work, contrasting with a single-day specification effort (at most) required to solve the problem with* HaiQ *in both cases.* Other approaches that do not include structural synthesis (e.g., direct specification in a model checker) require encoding explicitly all elements of every design alternative (e.g., topology), making it more error-prone and orders of magnitude more demanding than HaiQ in terms of specification effort for non-trivial problems.

Reusability. Compared to the ad-hoc solutions developed for the case studies presented, which required specific infrastructure, HaiQ *provides an infrastructure that can be reused across a range of different domains.* Furthermore, *specifications in* HaiQ *are also more reusable than those of existing probabilistic model checkers*, in which behavior types and communication topology are intertwined with the specific instances of processes. In contrast, behavioral type hierarchies can be reused as "libraries" across the same problem class in HaiQ, since the specifics of instances are encoded separately.

Computational cost and analytical capabilities. Prototype analysis performance behaves differently, depending on the problem type. In TAS, checking the compositional property constrained_mincost entailed exploring 90 configurations for an overall time of 6s (≃2% was used for configuration synthesis).[3] Checking the resilience property for a fixed topology in CAS took ≤40s in the worst case. However, checking for the optimal configuration took exploring 254 configurations (≃1hr, ≃0.01% used for synthesis). These numbers are explained by the low complexity of synthesis (few structural constraints to consider) relative to the possibly large number of configuration behaviors that must be checked individually. In any case, HaiQ itself introduces a negligible overhead in analysis, and *is as efficient as the underlying verification technology that it employs* (backend engines capable of PCTL checking on PRISM models like STORM [20] work out of the box).

6.2.4 Threats to Validity. The approach is inspired by a specific style of relational description (Alloy) and 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) for finer-grained time analysis. Focusing on *internal validity*, the degree of formal assurance on configurations provided by the approach is computationally expensive, and entails risks on 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 [36]. Hence, synthesis of different subsystems with local constraints can be done independently and then composed, reducing the cost of configuration synthesis. This mitigation also allows parallelism in the analysis to be exploited, in which the behavior of configurations of subsystems can be independently analyzed in parallel [35]. Another risk derived from structural synthesis is that Alloy can generate additional isomorphic configurations that can add unnecessary computation time in some situations, although this does not affect the soundness of the results.

[3] Experiments were run on MacOS10.1.5, Java 1.8.0_111, 2.8GHz Core i7, 16GB RAM.

7 CONCLUSIONS AND FUTURE WORK

This paper introduces what is, to the best of our knowledge, the first approach that combines the advantages of relational modeling, structural synthesis, and quantitative verification. Our experience applying it in different domains shows that: (RQ1) raising the level of abstraction by *(i)* incorporating modeling constructs that allow structural relations to be referenced from elements of (probabilistic) behavioral specifications and *(ii)* incorporating novel quantifiers to check properties across collections of models in probabilistic temporal logics, enables automated joint reasoning about structural and (probabilistic) quantitative guarantees across spaces of alternative system designs, (RQ2) our approach is general enough to be applied to different probabilistic formalisms (DTMC and MDP), types of analyses (average and best/worst probability- and reward-based analysis), and domains where uncertainty is introduced by disparate sources, and (RQ3) the approach brings new analytical capabilities to the validation of designs of software systems that operate under uncertainty, compared to existing quantitative verification approaches (e.g., automated identification of structural variants that optimize probability/reward-based guarantees). With respect to (RQ3), our experience indicates reduction of specification effort and improved reusability with respect to existing probabilistic model checking techniques (c.f. Section 6.2.3), although we still have to conduct an extensive study before we can consolidate our claims about these added benefits of the approach. We will perform this study as one of our next steps.

Our approach is currently limited to analyzing behaviors within specific configurations, but is not able to reason about behaviors that imply structural changes. Future work will extend our approach to reason about such behaviors. Moreover, we will explore hierarchical specification [36] and compositional verification [35] techniques (c.f., Section 6.2.4) to improve modularity and scalability.

REFERENCES

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press.

[2] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. 1991. The B-Method. In *VDM '91 - Formal Software Development (LNCS)*, Vol. 552. Springer, 398–405.

[3] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya. 2009. 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.* 61–71.

[4] Suzana Andova, Holger Hermanns, and Joost-Pieter Katoen. 2003. Discrete-Time Rewards Model-Checked. In *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS (LNCS)*, Vol. 2791. Springer, 88–104.

[5] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Trans. Software Eng.* 41, 9 (2015), 866–886.

[6] Hamid Bagheri, Chong Tang, and Kevin J. Sullivan. 2014. TradeMaker: automated dynamic analysis of synthesized tradespaces. In *36th Int. Conf. on Software Engineering*. ACM, 106–116.

[7] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. 2004. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. Software Eng.* 30, 5 (2004), 295–310.

[8] Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. 2009. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82, 1 (2009), 3–22.

[9] Dines Bjørner. 1978. The Vienna development method (VDM): Software specification & program synthesis. In *Mathematical Studies of Information Processing, Proceedings of the International Conference (LNCS)*, Vol. 75. Springer, 326–359.

[10] Egor Bondarev, Michel R. V. Chaudron, and Erwin A. de Kock. 2007. Exploring Performance Trade-offs of a JPEG Decoder Using the Deepcompass Framework. In *6th WS on Software and Performance (WOSP)*. ACM, 153–163.

[11] Franz Brosch, Heiko Koziolek, Barbora Buhnova, and Ralf H. Reussner. 2012. Architecture-Based Reliability Prediction with the Palladio Component Model. *IEEE Trans. Software Eng.* 38, 6 (2012), 1319–1339.

[12] Radu Calinescu, Milan Ceska, Simos Gerasimou, Marta Kwiatkowska, and Nicola Paoletti. 2017. 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, 131–140.

[13] Radu Calinescu, Carlo Ghezzi, Marta Z. Kwiatkowska, and Raffaela Mirandola. 2012. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM* 55, 9 (2012), 69–77.

[14] Javier Cámara. 2020. HaiQ website. (2020). Accessed Apr. 1st, 2020 from http://www.haiqmodelchecker.org.

[15] Javier Cámara, David Garlan, and Bradley R. Schmerl. 2017. Synthesis and Quantitative Verification of Tradeoff Spaces for Families of Software Systems. In *Software Architecture - 11th European Conference, ECSA (LNCS)*, Vol. 10475. Springer, 3–21.

[16] Thiago Castro, André Lanna, Vander Alves, Leopoldo Teixeira, Sven Apel, and Pierre-Yves Schobbens. 2018. All roads lead to Rome: Commuting strategies for product-line reliability analysis. *Sci. Comput. Program.* 152 (2018), 116–160.

[17] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2017. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing* (2017).

[18] Tod Courtney, Shravan Gaonkar, Ken Keefe, Eric Rozier, and William H. Sanders. 2009. Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. IEEE CS, 353–358.

[19] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. 2015. Uppaal Stratego. In *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, Vol. 9035. Springer Berlin Heidelberg, 206–211.

[20] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is Coming: A Modern Probabilistic Model Checker. *CoRR* abs/1702.04311 (2017). arXiv:1702.04311 http://arxiv.org/abs/1702.04311

[21] Vishal Dwivedi, David Garlan, Jürgen Pfeffer, and Bradley Schmerl. 2014. 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.

[22] Naeem Esfahani, Sam Malek, and Kaveh Razavi. 2013. GuideArch: guiding the exploration of architectural solution space under uncertainty. In *35th International Conference on Software Engineering, ICSE*. IEEE CS, 43–52.

[23] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2011. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE*. ACM, 341–350.

[24] Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, and Nazareno Aguirre. 2005. DynAlloy: upgrading alloy with actions. In *27th International Conference on Software Engineering (ICSE)*. ACM, 442–451.

[25] David Garlan. 2010. Software engineering in an uncertain world. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 125–128.

[26] Carlo Ghezzi and Amir Molzam Sharifloo. 2013. Model-based verification of quantitative non-functional properties for software product lines. *Information & Software Technology* 55, 3 (2013), 508–524.

[27] Stephen Gilmore and Jane Hillston. 1994. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Computer Performance Evaluation, Modeling Techniques and Tools, 7th International Conference (LNCS)*, Vol. 794. Springer, 353–368.

[28] Thomas J. Glazier, Javier Cámara, Bradley R. Schmerl, and David Garlan. 2015. Analyzing Resilience Properties of Different Topologies of Collective Adaptive Systems. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops*. IEEE CS, 55–60.

[29] Lars Grunske and Aldeida Aleti. 2013. Quality optimisation of software architectures and design specifications. *Journal of Systems and Software* 86, 10 (2013), 2465–2466.

[30] Hans Hansson and Bengt Jonsson. 1994. A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6, 5 (1994), 512–535.

[31] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

[32] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press.

[33] Daniel Jacobson, Danny Yuan, and Neeraj Joshi. 2013. Scryer: Netflix's Predictive Auto Scaling Engine. http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html. (Dec. 2013). [Online; accessed 8-2017].

[34] He Jifeng, K. Seidel, and A. McIver. 1997. Probabilistic models for the guarded command language. *Science of Computer Programming* 28, 2 (1997), 171 – 192. Formal Specifications: Foundations, Methods, Tools and Applications.

[35] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. 2013. 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.

[36] Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. 2016. Multi-representational security analysis. In *Proc. of the 24th Symposium on Foundations of Software Engineering, FSE*.

[37] M. Kwiatkowska, G. Norman, D. Parker, and M.G. Vigliotti. 2009. Probabilistic Mobile Ambients. *Theoretical Computer Science* 410, 12–13 (2009), 1272–1303.

[38] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2007. Stochastic Model Checking. In *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM (LNCS)*, Vol. 4486. Springer, 220–270.

[39] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification - 23rd International Conference, CAV*, Vol. 6806. Springer, 585–591.

[40] Marta Z. Kwiatkowska and David Parker. 2013. Automated Verification and Strategy Synthesis for Probabilistic Systems. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings (Lecture Notes in Computer Science)*, Dang Van Hung and Mizuhito Ogawa (Eds.), Vol. 8172. Springer, 5–22.

[41] André Lanna, Thiago Castro, Vander Alves, Genaína Nunes Rodrigues, Pierre-Yves Schobbens, and Sven Apel. 2018. Feature-family-based reliability analysis of software product lines. *Information & Software Technology* 94 (2018), 59–81.

[42] Alex D. MacCalman, Paul T. Beery, and Eugene P. Paulo. 2016. A Systems Design Exploration Approach That Illuminates Tradespaces Using Statistical Experimental Designs. *Syst. Eng.* 19, 5 (2016), 409–421.

[43] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2013. 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, 444–454.

[44] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. 2010. Automatically Improve Software Architecture Models for Performance, Reliability, and Cost Using Evolutionary Algorithms. In *Int. Conf. on Performance Engineering (WOSP/SIPEW)*. ACM, 105–116.

[45] Indika Meedeniya, Irene Moser, Aldeida Aleti, and Lars Grunske. 2011. Architecture-based reliability evaluation under uncertainty. In *7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS*. ACM, 85–94.

[46] Ruslan Meshenberg, Naresh Gopalani, and Luke Kosewski. 2013. Active-Active for Multi-Regional Resiliency. http://techblog.netflix.com/2013/12/active-active-for-multi-regional.html. (Dec. 2013). [Online; accessed 8-2017].

[47] Dave Parker. 2002. The PRISM Language - Semantics. (2002). Accessed on Jan. 22nd, 2020 from www.prismmodelchecker.org/doc/semantics.pdf.

[48] Dave Parker. 2002. The PRISM Preprocessor. (2002). Accessed Jan. 22nd, 2020 from http://www.prismmodelchecker.org/prismpp/.

[49] Diego Perez-Palacin and Raffaela Mirandola. 2014. Uncertainties in the Modeling of Self-adaptive Systems: A Taxonomy and an Example of Availability Evaluation.

In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE '14)*. ACM, 3–14.
[50] J. Michael Spivey. 1992. *Z Notation - a reference manual (2. ed.).* Prentice Hall.
[51] Jos Warmer and Anneke Kleppe. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley.
[52] Danny Weyns and Radu Calinescu. 2015. Tele Assistance: A Self-Adaptive Service-Based System Exemplar. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*. IEEE CS, 88–92.
[53] S. Wong, J. Sun, I. Warren, and J. Sun. 2008. A Scalable Approach to Multi-style Architectural Modeling and Verification. In *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*. 25–34.
[54] Pamela Zave. 2005. A Formal Model of Addressing for Interoperating Networks. In *FM 2005: Formal Methods, International Symposium of Formal Methods Europe (LNCS)*, Vol. 3582. Springer, 318–333.