Exploring the Interaction of Design Variability and Stochastic Operational Uncertainties in Software-Intensive Systems through the Lens of Modeling

Javier Cámara[1*]

[1*]ITIS Software, Universidad de Málaga, Bulevar Luis Pasteur, 35, Málaga, 29071, Spain.

Corresponding author(s). E-mail(s): jcamara@uma.es;

Abstract

In software-intensive systems, navigating the complexities that emerge from the interaction of design variability and stochastic operational uncertainties, presents a daunting challenge. This paper delves into the dynamics between these two dimensions of uncertainty, offering novel insights about how modeling can contribute to the analysis of their combined impact upon system properties. By elevating the abstraction level at which probabilistic models are conceptualized, our approach enables an integrated analysis framework that considers both structural and quantitative dimensions of design spaces. Through the introduction of novel language constructs, our methodology facilitates the direct referencing of structural relationships within probabilistic behavioral specifications. Furthermore, the adoption of novel quantifiers in probabilistic temporal logic enables evaluating complex properties across diverse design variants, thereby streamlining the assessment of guarantees within the solution space. We demonstrate the feasibility of this approach on four case studies, showcasing its potential to offer comprehensive insights into the trade-offs and decision-making processes inherent in managing different types of structural design variability and operational uncertainties in software-intensive systems.

Keywords: design variability, operational uncertainty, uncertainty interaction, quantitative verification

1 Introduction

In the evolving landscape of software engineering, uncertainty plays a critical and often challenging role. Software-intensive systems, fundamental in various domains, are increasingly subject to various uncertainties during their lifecycle, encompassing design, construction, deployment, and evolution. The inherent complexity of these systems, coupled with their essential roles in supporting crucial tasks, makes managing uncertainty a pivotal concern. Examples are abundant, ranging from mission-critical systems used in disaster relief operations to automatic stock management systems, where the cost of erroneous decisions under uncertainty can lead to severe consequences.

The pursuit to effectively manage and mitigate the effects of uncertainty in software-intensive systems is viewed as a promising avenue for engineering systems that are resilient to runtime changes and the various uncertainties that arise from their execution environment, such as resource availability, human interactions, and system-specific issues like faults or the use of machine learning components [1].

So far, the focus has been on developing methodologies to detect, represent, and mitigate uncertainties originating from diverse sources, each impacting the system's functional and non-functional requirements in unique ways. However, a critical challenge that persists is the interaction of uncertainties.

Individual uncertainties, whether pertaining to goals, resources, or other factors that concern a software-intensive system and its environment, are seldom independent. They often compound, subtly and unpredictably affecting the achievement of system goals and other properties. This phenomenon, recently described as the *Uncertainty Interaction Problem* (UIP) [2, 3], highlights the complexity of understanding how different types of uncertainties, arising from various sources, interact with each other and impact the properties of software-intensive systems. Addressing this problem involves representing these combined uncertainties, analyzing their emergent effects on requirement satisfaction, and developing strategies for their mitigation. Despite ongoing research efforts, there is a limited understanding of the specific ways these uncertainties interact and the full extent of their impact on system properties [3].

In this paper, we narrow our focus to a specific class of uncertainty interaction that has a remarkable influence on software design and operation: the interplay between design variability and operational uncertainties. Design variability refers to the spectrum of design choices available during the software development process. Each choice leads to different system behaviors and performance levels across various quality dimensions. This variability, while offering flexibility and customization, introduces uncertainty in predicting system behavior and performance outcomes.

Concurrently, operational uncertainties arise at run time. These are often stochastic in nature, stemming from unpredictable factors such as user interactions, environmental conditions, and system load variations. Operational uncertainties are particularly challenging to model and mitigate, as they are often outside the control of system designers and emerge in some cases, only after the system has been deployed.

The interaction between these two types of uncertainty presents a unique challenge. Design choices made under uncertainty can significantly impact how a system responds to operational uncertainties. For instance, a design optimized for certain operational conditions may perform suboptimally when those conditions change unexpectedly. Conversely, operational uncertainties can reveal limitations or unforeseen behaviors in certain design choices, affecting the system's overall resilience and adaptability.

This specific intersection of design variability and operational uncertainties requires a nuanced approach to modeling and analysis. Traditional models that treat these uncertainties in isolation (e.g., constraint solvers, probabilistic model checkers) are insufficient for capturing the complex dynamics at play when these uncertainties interact. On the one hand, formal methods used to validate software designs like Alloy [4], Z [5], B [6], VDM [7] and OCL [8], share a common conceptual foundation that enables designers to capture design variability by implicitly describing collections of alternative structural designs (e.g., software architectures [9, 10], database object-relational mappings [11], network and security models [12, 13]) as sets of relational constraints and reason about them systematically, but they are not equipped for analyzing stochastic behavior. On the other hand, *quantitative verification* or *probabilistic model checking* [14–16] approaches can reason under uncertainty about quantitative guarantees of systems (e.g., on performance, reliability), but use notations (e.g., PRISM [17], PEPA [18], cpGCL [19]) that do not retain the flexibility in describing design variants of relational methods. Recent product line reliability analysis approaches [20–23] improve on flexibility by introducing systematic treatment of variability, but are not equipped to synthesize or reason about complex design variants (e.g., those that satisfy a non-trivial set of structural constraints of the style that are usually captured by, for instance, OCL or Alloy), and lack languages tailored to check complex properties across design variants (i.e., temporal logics employed like PCTL [14] can capture properties only about a single variant, not collections of variants that describe the design space).

This situation forces designers to explore design variants independently of stochastic factors in the system's operational environment, limiting their ability to systematically assess how their mutual interdependence affects relevant system properties.

Our research aims to address this gap by proposing a modeling and analysis framework that integrates both design variability and stochastic operational uncertainties. This framework is aimed at enabling a more holistic understanding of how design choices influence system behavior under various operational scenarios and vice versa. We posit that this will contribute to the development of more robust, adaptable, and resilient software systems capable of reliable operation in uncertain and changing environments. To further our vision, we investigate the following research questions:

(RQ1): How can we jointly model and reason about (non-trivial) design space variability and stochastic aspects in a system's operational environment?

(RQ2): If feasible, is such an approach general enough to be applicable to different domains and types of analysis?

(RQ3): What are the trade-offs of employing this class of approach with respect to existing techniques for analyzing systems that operate under uncertainty?

This paper explores these questions by contributing what is, to the best of our knowledge, the first approach that integrates into a single framework (HaiQ) the specification of complex design spaces with stochastic behavior modeling and analysis. Such integration enables obtaining quantitative guarantees on the satisfaction of system properties in the presence of the emergent effects of design space variability and stochastic operational uncertainties.

From a modeling perspective, the first core feature of HaiQ is its capability to transform configurations, derived from relational constraints, into detailed stochastic behavior models. This transformation process facilitates a comprehensive exploration of various system configurations under uncertainty, using probabilistic model checking to evaluate both average and extreme-case scenarios.

The second core feature of the framework is Manifold Probabilistic Computation Tree Logic (M-PCTL), an extension of existing probabilistic temporal logics that enables users to evaluate properties across a collection of probabilistic state machines, each of which captures the behavior of a system design variant. This ability is crucial for assessing the resilience of different system configurations to operational uncertainties.

This article is a revised and extended version of the paper presented at the 8th International Conference on Formal Methods in Software Engineering (FormaliSE) in 2020 [24]. In this updated version, the original paper is expanded to deepen the understanding of how modeling within frameworks like HaiQ can address some of the challenges posed by the Uncertainty Interaction Problem, particularly by capturing the interplay of design variability and operational uncertainties faced by software-intensive systems. The paper now opens with a revised introduction that contextualizes this interaction within the UIP. Moreover, it includes a broadened review of related literature, a novel case study on a cyber-physical system that introduces unique variations and uncertainties, and an enriched discussion incorporating these new findings. Additionally, an appendix detailing the HaiQ modeling language is provided, alongside a publicly released repository containing HaiQ's binaries, source code, and the models from the presented case studies.

The remainder of this paper starts by providing a discussion of related work (Section 2), following with an overview of the approach (Section 3). Then, we give further details about HaiQ's modeling language (Section 4) and M-PCTL (Section 5). Next, we illustrate and evaluate the approach on four scenarios in various domains (Section 6), discussing research questions and threats to validity. The paper finishes with conclusions and directions for future work.

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 [5] have a very expressive notation that limits automated theorem proving, which often requires guidance from an experienced user. Alloy [4] can be considered a first-order subset of Z with finite models, which makes it automatically analyzable. Other techniques and languages like VDM [7] and OCL [8] 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 reasoning about complex concurrent behaviors. In contrast, methods like DynAlloy [25], Electrum [26] and Event-B [27] include more sophisticated constructs to reason efficiently about behaviors on top of structures. Despite being able to reason about concurrent behaviors, these methods are not equipped to capture or reason about probabilistic aspects of system behavior.

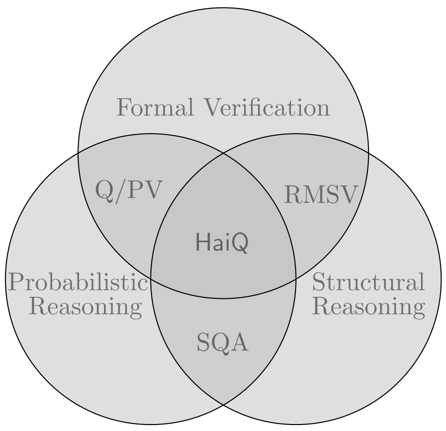


Fig. 1 Relation between HaiQ and other works

Quantitative/probabilistic analysis and verification (Q/PV). Modern probabilistic model checkers like PRISM [17] and STORM [28] capture models using textual languages like PRISM, PEPA or low-level matrix-based descriptions. Stochastic variants of UPPAAL [29] employ graphical specification languages. Although these tools are efficient at analyzing complex probabilistic system behaviors, their specification languages and reasoning mechanisms do not separate process instance attributes from process types, hampering reusability. Möbius [30] overcomes this limitation by exploiting domain-specific models. However, none of the tools described are equipped for reasoning about design variants. In contrast, product line reliability analysis approaches [20–23] can analyze collections of system designs encoded in a feature model individually or collectively. A recent approach to continuous-time probabilistic design synthesis [31] uses a template-based solution to generate and analyze alternative system designs, but does not make any emphasis on 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 constraints in the designs. Compared with the approaches described above, HaiQ's focus is not only on variability, but also on structure, being able to capture and analyze design alternatives that satisfy complex structural constraints. Moreover, HaiQ includes a temporal logic (M-PCTL) that specifically targets analysis across multiple design variants.

Structural quantitative analysis and optimization (SQA). There is extensive related work on model-based performance prediction [32] and optimization of quantitative aspects of structures (e.g., architectures) [33] that typically use mechanisms like stochastic search and/or Pareto analysis [34–40]. These and other approaches in systems engineering (e.g., [41]) can optimize quantitative aspects of designs, but they do not support synthesis of configurations. Other approaches [11, 42] 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 under uncertainty, which rely on checking sophisticated properties (typically encoded as temporal logic formulas) via numerical methods and exhaustive state space exploration techniques. An ad-hoc combination of architectural synthesis from Alloy specifications with a template engine to generate probabilistic models [43] enables analysis of quantitative properties of configurations, but does not include its own modeling nor property languages.

In summary, HaiQ is to the best of our knowledge, the only existing approach that supports the combined modeling and exploration of structurally complex design variants with stochastic operational uncertainties that enables streamlined analysis of structural and quantitative/probabilistic guarantees across design spaces (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 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 discrete-time Markov chain (DTMC) [14] 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 reason about *probabilistic and other quantitative guarantees* captured in temporal logics like PCTL and CSL [14]. 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 and explicitly encoded 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/probabilistic properties 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 in cases in which process behaviors and structural system constraints are non-trivial.

```
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
```

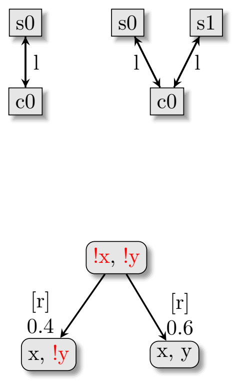
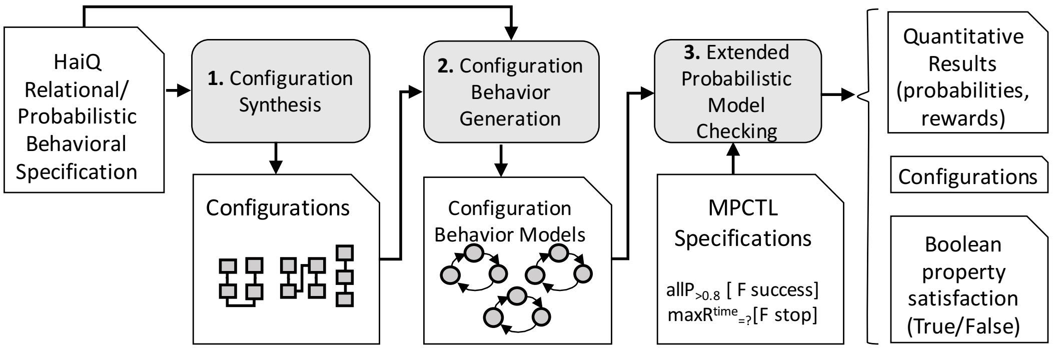


Fig. 2 Client-server Alloy and PRISM specifications

To investigate automated joint modeling and analysis of structural design variants, (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 design variants from relational constraints with probabilistic model checking of such variants (Figure 3). Our proposal includes three stages:



HaiQ Relational/ Probabilistic Behavioral Specification
1. Configuration Synthesis
2. Configuration Behavior Generation
3. Extended Probabilistic Model Checking
Configurations
Configuration Behavior Models
MPCTL Specifications
$allP_{>0.8}$ [F success]
$maxR^{time}_{=?}$[F stop]
Quantitative Results (probabilities, rewards)
Configurations
Boolean property satisfaction (True/False)

Fig. 3 Overview of the approach

```
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 rel.
run clientserver for 1 c, 2 s
```

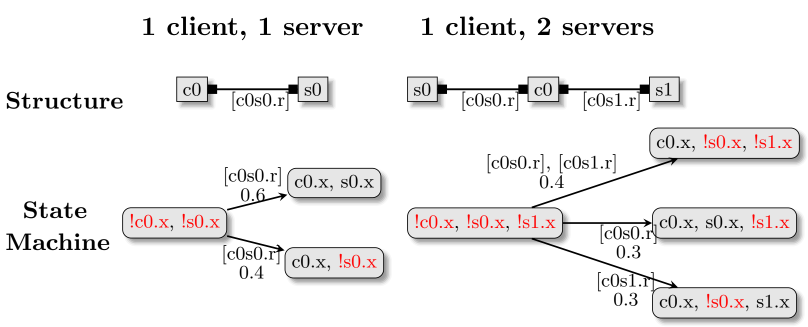


Fig. 4 Client-server HaiQ specification (left), implicit structure collection and corresponding state machines (right)

(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 existing 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 "hardwired"* 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.

Manifold Probabilistic Computation Tree Logic (M-PCTL). Ultimately, a HaiQ model defines a set of probabilistic state machines (DTMC or MDP), where each one of them 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 query HaiQ specifications and *check properties across entire 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, the property $\mathsf{allP}_{\geq 0.6}[\mathsf{F\ all\ c.x} \wedge \mathsf{some\ s.x}]$ 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 probably that $[\mathsf{F\ all\ c.x} \wedge \mathsf{some\ s.x}]$ is satisfied is exactly 0.6 in both cases. Moreover, *M-PCTL properties can also return structures that optimize or satisfy some quantitative property.* For instance, the property $\mathsf{SmaxP}\ [\mathsf{F\ all\ c.x} \wedge \mathsf{some\ s.x}]$ 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 $[\mathsf{F\ all\ c.x} \wedge \mathsf{some\ s.x}]$ and hence maximize the value of the property in this case because there are no other feasible solutions.

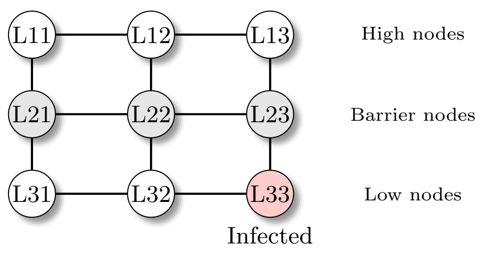


Fig. 5 Network virus infection scenario

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 [44].[1] 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.

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

[1] A PRISM model for this network security scenario can be found in http://www.prismmodelchecker.org/casestudies/virus.php.

code fragments. The structural part of the language is a subset of the static part of Alloy (c.f. [45], 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.[2] 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);
   [conn:attack] (s=infected) -> true; />
```

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[3] 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 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]`[4] 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.

[2] HaiQ's grammar is described in Appendix A.

[3] For convenience, the language also provides enumerated types, which are internally treated as integer variables.

[4] 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).

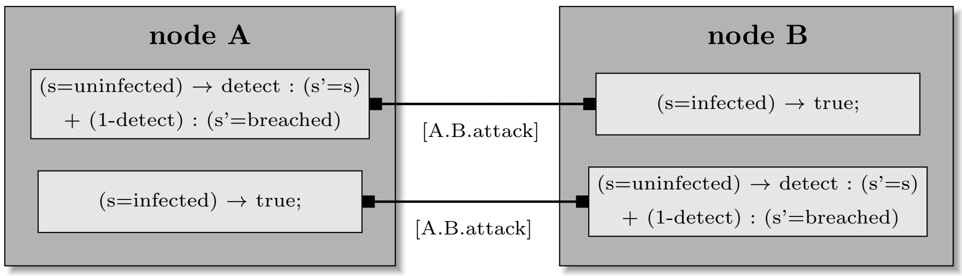

node A
(s=uninfected) → detect : (s'=s)
+ (1-detect) : (s'=breached)
(s=infected) → true;
[A.B.attack]
[A.B.attack]
node B
(s=infected) → true;
(s=uninfected) → detect : (s'=s)
+ (1-detect) : (s'=breached)

Fig. 6 Sample synchronization scheme between nodes

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`.

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.

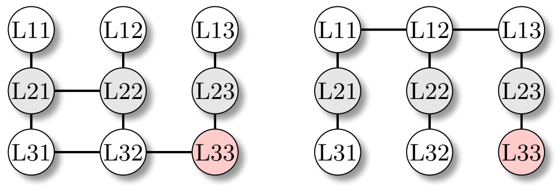


Fig. 7 Alternative legal network configurations

```
pred virus{
all n:node | node in n.*conn //connected
 (no iden & conn) //no self-loops,
 (conn=~conn) //symmetry
 disj[lowNode.conn,highNode]//disj. l-h nodes
}
```

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 no node must be connected to itself. The third one forces connections to be symmetric, whereas the fourth constraint 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 possible 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; />
```

The reward is associated with transitions in which the action `attack` is executed from any state. Every time the action is executed from a state in which the guard before the colon is satisfied (in this case, the guard is `true` and is therefore always satisfied), 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 $(\Sigma, \mathcal{C}, \delta)$, where Σ is a set of signatures that implicitly define a hierarchy of types (including their behavior), $\mathcal{C}$ 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 1 (Signature)**.** *A signature σ is a tuple $(\sigma^{\uparrow}, \mathcal{R}, \beta)$:*

- *$\sigma^{\uparrow} \in \Sigma \cup \{\perp\}$ 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 $\mathcal{C}$).*
- *β is a behavior specification.*

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

Each command in $\mathcal{K}$ is a tuple $(\alpha, r, g, \mathcal{U})$, where:

- $\alpha \in actIds \cup \{\perp\}$ is a label drawn from an action namespace. If $\alpha = \perp$, the command encodes an internal action.
- $r \in \sigma_f.\mathcal{R} \cup \{\perp\}$ is a relation that constrains synchronization on action α to instances in r. If $r = \perp$, the scope of the command labeled by α is global and the action can synchronize with any other process that includes a command with the same label.
- g is a guard built over the variables in $\sigma_f.\mathcal{V}$.
- $\mathcal{U}$ is a set of probabilistic updates. Each update is a couple (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, ν) ($v \in \sigma_f.\mathcal{V}$, and ν 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 $\alpha = \perp \Rightarrow r = \perp$.

4.2.1 Configurations

The specification $(\Sigma, \mathcal{C}, \delta)$ implicitly defines a set of *configurations* or structures. Each configuration is composed by a set of instances of signatures, connected in a topology that satisfies $\mathcal{C}$.

The instance of a signature σ is just a labeled process with fresh variables that corresponds to the behavior definition of σ_f.

Definition 3 (Instance)**.** *An instance of a signature σ is a couple (l, β), where $l \in procIds$ is a label drawn from a process namespace, and $\beta = \sigma_f.\beta_\nu$ 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 (Configuration)**.** *A configuration is a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ satisfying the constraints imposed by $\mathcal{C}$ 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 $\mathcal{C}$ 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 $\mathcal{K}_\nu$ 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 ($k.r \neq \perp$) 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 parallel composition of the extended processes for the instances in the configuration.[5]

Algorithm 1 Configuration Stochastic State Machine Generation

```
1:  $\mathcal{N}_\nu := \emptyset$
2:  for all $n : c.\mathcal{N}$ do
3:      $\mathcal{K}_\nu := \{k : type(n).\beta.\mathcal{K} \mid k.r = \perp\}$
4:      for all $k : type(n).\beta.\mathcal{K} \backslash \mathcal{K}_\nu$ do
5:          $\mathcal{K}^* := \emptyset$
6:          for all $\{n' : c.\mathcal{N} \mid \exists (x, n, n') \in c.\mathcal{E}, x \in sVarIds\}$ do
7:              $\mathcal{K}^* := \mathcal{K}^* \cup \{(uid(n.l, n'.l), \perp, k.g, k.\mathcal{U})\}$
8:          end for
9:          $\mathcal{K}_\nu := \mathcal{K}_\nu \cup \mathcal{K}^*$
10:     end for
11:     $\mathcal{N}_\nu := \mathcal{N}_\nu \cup \{(n.l, (n.\mathcal{V}, \mathcal{K}_\nu, n.\mathcal{L}))\}$
12: end forreturn $\underset{n:\mathcal{N}_\nu}{\parallel} n.\beta$
```

[5]The standard process followed for parallel composition of a set of processes in the form provided by $\mathcal{N}_\nu$ is described [46].

5 Manifold Probabilistic Computation Tree Logic (M-PCTL)

Probabilistic Computation Tree Logic (PCTL) [47] 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).

This section first overviews a version of PCTL extended with a reward quantifier targeted at checking properties over DTMC and MDP extended with reward structures [48], and then builds on it to introduce M-PCTL.

5.1 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 (Φ). Path formulas (ϕ) have an auxiliary role on probability and reward quantifiers P/R:

$$\Phi ::= \texttt{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{P}_{\sim pb}[\phi] \mid \mathsf{R}^r_{\sim rb}[\phi]$$
$$\phi ::= \mathsf{X}\Phi \mid \Phi \,\mathsf{U}\, \Phi$$

In this definition, 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 $Pr_s(\phi)$ [6]) 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 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., $\mathsf{F}\Phi \equiv \mathsf{true}\ \mathsf{U}\ \Phi$).

[6]See [14] for details. In the following, we write $Pr_s(\phi)$ as $Pr(\phi)$ for simplicity.

5.2 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:

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

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. [14]). 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. The semantics of the main quantifiers in set formulas is:

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

Quantifier SsomeP returns a singleton with an element drawn nondeterministically from $\mathsf{SallP}_{\sim pb}[\phi]$ if the set is not empty, and $\emptyset$ otherwise. Set subtraction is derived as $\Psi_1 \backslash \Psi_2 \equiv \Psi_1 \cap \Psi_2^C$.

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:

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

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., $\langle\mathsf{resilient}\rangle\ \mathsf{maxR}^{\mathsf{attacks}}[\mathsf{F\ all\ highNode.s} = \mathsf{infected}]$.

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 [14]. 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 $\mathsf{R}^{\mathsf{r}}_{\mathsf{max}}/\mathsf{R}^{\mathsf{r}}_{\mathsf{min}}$[49]. 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 $\mathsf{SminP}_{\mathsf{max}}[\mathsf{F}^{\leq \mathsf{t}}\ \mathsf{all\ highNode.s} = \mathsf{infected}]$ 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[7], 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 back-end employs Alloy's and PRISM's APIs for synthesizing configurations and model checking properties on their behavior models, respectively.

6.1 Case Studies

We describe the application of our approach to a service-based system, a distributed self-protecting system, and a mobile robotics scenario. The scenarios were chosen because they instantiate different forms of structural variability (architecture, network, physical space) and sources of stochastic operational uncertainties. We finish this section with a discussion of our results.

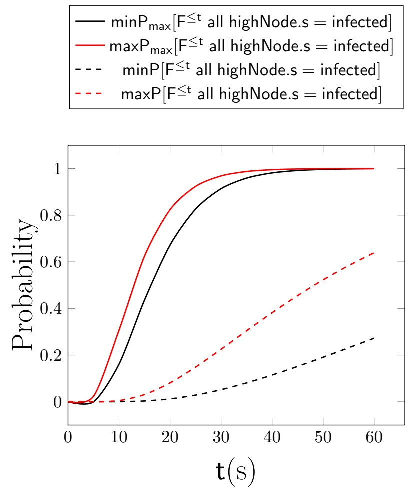
6.1.1 Virus Network Infection

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 top 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 much 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 bottom 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 $\mathsf{minP_{max}[F^{\leq t}\ some\ highNode.s = infected]}$ (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).

[7]The source code for the HaiQ analyzer, example models found in this paper, and OSX binaries for a ready-to-use version of the tool are available at https://github.com/javier-camara/HaiQ



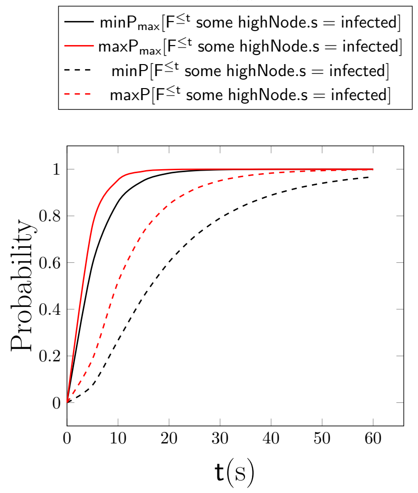


Fig. 8 Network virus infection results

6.1.2 Tele-Assistance System (TAS)

The goal of the TAS exemplar system [50] 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 (Figure 9).

When TAS receives a request that includes the vital parameters of a patient, its *Medical Service* analyzes the data and replies with instructions to:

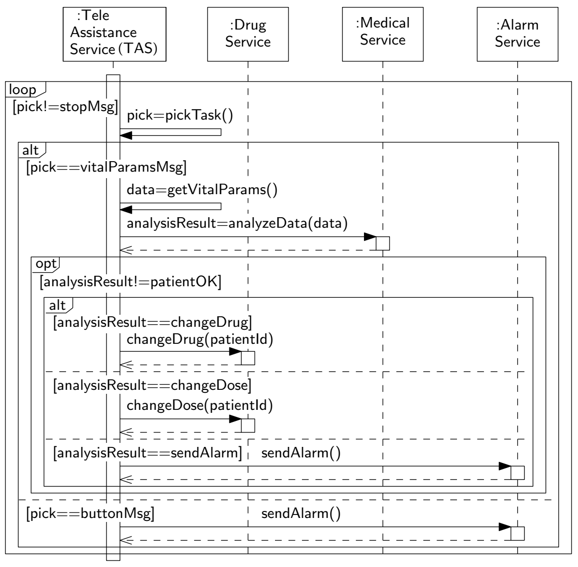


Fig. 9 Tele assistance service workflow

(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, ...
 ASBindings: some AlarmService}
</ 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);
   [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 (Table 1). 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; />
```

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_cost:

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

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

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

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

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 10 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

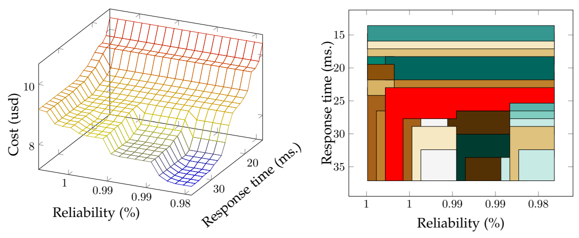


Fig. 10 TAS analysis results

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 $\leq$26ms, and with a reliability of $\geq$99%, we can determine which are the configurations that best satisfy constraints by checking confs_mincost 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 [51, 52].

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 [53], an external attacker uses a defined amount of available resources to attempt

Table 1 Properties of TAS service providers

Service	Name	Fail. rate (%)	Resp.time (ms)	Cost (usd)
MS1	Medical Service 1	0.06	22	9.8
MS2	Medical Service 2	0.1	27	8.9
MS3	Medical Service 3	0.15	31	9.3
MS4	Medical Service 4	0.25	29	7.3
MS5	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

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;
   //attacked
   [attackVector:attack]
      (resources>0) & (status=normal) ->
          detect: (status'=attackDetected)
                & (resources'=deplete);
        + 1-detect: (status'=normal)
                  & (resources'=deplete);
   //notify attack
   [conn:alert]
    (status=attackDetected) -> true;
   //adapt if attack detected (or notified)
   [] (resources>0)
    & (status=attackDetected) ->
      (status'=adapted);
   //receive alert
   [conn:alert]
     (resources>0) & (status=normal) ->
         CHANNEL_RELIABILITY:
           (status'=attackDetected)
     + 1-CHANNEL_RELIABILITY:
           (status'=normal);
   [] (resources=0) ->
        (status'=compromised); />
```

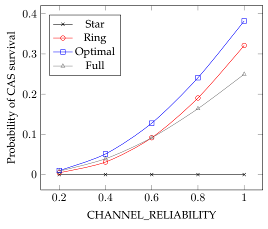
The excerpt above shows the basic encoding of a local self-adaptive manager (signature `sam`). Environmental and system conditions, such as the reliability of communication channels and the sensitivity of detection mechanisms, can play an important role in the emergent behavior of the CAS and are explicitly 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 11 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 $\equiv$ maxP[F all sam.status = adapted].



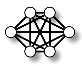
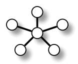
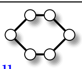
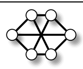
Full	**Star**
 all s:sam — all s′:sam-s — s in s′.conn	 one s:sam — s in sam.conn and #(sam-s).conn=1
Ring	**Optimal**
 all s:sam — #s.conn=2	 -

Fig. 11 CAS topologies and resilience results

The left-hand side of Figure 11 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 Mobile Robotics

Physical interactions in software-intensive systems are also a source of uncertainty that impacts system goals. We illustrate this in a scenario in which a drone navigates autonomously in a building from a starting location A to target location E with limited battery, and without bumping into obstacles (Figure 12). The arcs in the map graph designate drone trajectories of distance `d`, where `pc` captures the probability of drone collision when completing a trajectory. Locations in the map among which the drone can move are captured by signature `loc`:

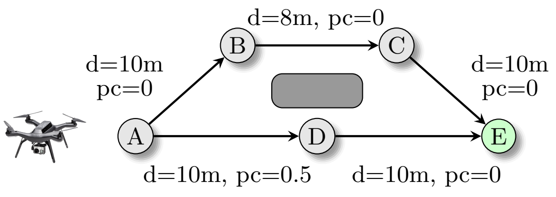


Fig. 12 Simple mobile robotics scenario

```
abstract sig loc {conn: set loc}
</ var  droneIn:  bool init false;
   formula pc, d;
   [conn:moveTo]
     (droneIn) -> (droneIn'=false);
   [conn:moveTo]
     (!droneIn) & (conn.droneIn) ->
        pc: (droneIn'=false)
      + 1-pc: (droneIn'=true);
   reward energyRew [conn:moveTo]
     (!droneIn) & (conn.droneIn):
        (d/SPEED)*BDRATE; />
```

In the static part of the signature, field `conn` encodes a relation capturing connections among neighboring locations. In the behavior block, variable `droneIn` specifies if the drone is positioned in the location, and formula `pc` encodes the probability of drone collision with an obstacle when moving towards the location.

Locations synchronize on action `moveTo` both when the drone leaves and arrives in the location. When the drone leaves location `A` (`A.droneIn'=false`), it arrives in location `B`, and vice versa. Note that the guard to receive the robot refers to the value of the counterpart process variable (`conn.droneIn`), which must be true (i.e., "contain the drone"), to trigger the action. Reward `energyRew` is accrued when the drone arrives to a location and is computed based on the distance traveled by the drone (`d`), `SPEED` constant, and a `BDRATE` constant that encodes the battery discharge rate.

To encode location subtypes, we first extend `loc` in signature `startLoc` to create a starting location in which the original initialization value of variable `droneIn` is overriden and set to `true`:

```
abstract sig startLoc extends loc {}
  </var droneIn: bool init true;/>
abstract sig targetLoc extends loc {}
one sig A extends startLoc {}
  </ formula pc=0, d=0; />
lone sig C extends loc {}
  </ formula pc=0.5, d=10;  /> ...
one sig E extends targetLoc {}
  </ formula pc=0, d=10; />
```

In contrast, `targetLoc` is a subtype of `loc` that preserves all its structural and behavioral elements intact. Both start and target locations are defined as abstract signatures, so we incorporate subtypes of them constrained as singletons to instantiate map locations.

We can consider mission design variability in terms of path alternatives for the drone by writing a set of general relational constraints that guarantee non-cyclic paths in which a target location is reachable from the starting location:

```
//path endpoints
all l:startLoc+targetLoc | #l.conn=1
//path waypoints
all l:loc-(startLoc+targetLoc) | #l.conn=2
(disj[targetLoc, startLoc]) and  (conn=~conn)
(no iden & conn)
all l:loc | loc in l.*conn //connected path
```

To complete the constraints for paths, we add the conjunction of all topology constraints imposed by the map:

```
      disj[A.conn,loc-(B+D)] and ...
              and disj[E.conn,loc-(C+D)]
```

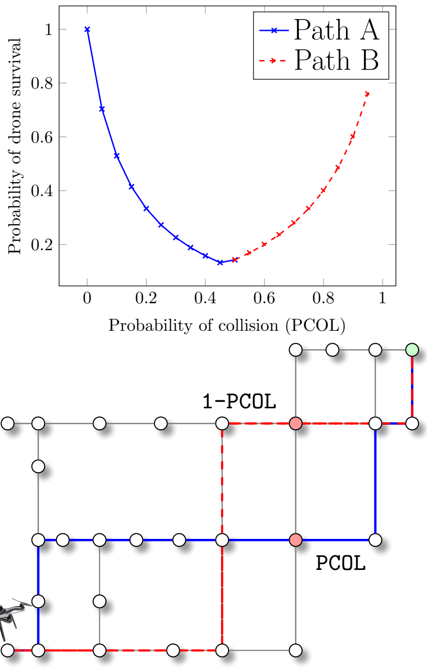


Fig. 13 Mobile robotic scenario analysis results

To check which of these paths maximizes the probability of arrival to a target location within the battery constraint, we can write:

$$\mathsf{feasible} \equiv \mathsf{SallR}^{\mathsf{energyRew}}_{<\mathsf{e}}[\mathsf{F\ some\ targetLoc.droneIn}]$$
$$\mathsf{maxsafety} \equiv \langle\mathsf{feasible}\rangle\mathsf{maxP}[\mathsf{F\ some\ targetLoc.droneIn}]$$

Where feasible paths are those in which energy does not exceed the drone battery capacity e, and maxsafety identifies paths that maximize the probability of reaching a target location.

Figure 13 shows results for safety without battery constraint on a 30-location map in which highlighted locations in red have probability of collision `PCOL` and `1-PCOL`, respectively. As expected, higher values of `PCOL` decrease chances of drone survival, until it hits a minimum of $\simeq$0.14 in `PCOL=0.5`. From that point, the best configuration shifts from path A to path B (highlighted in red).

6.3 Discussion

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

6.3.1 (RQ1) Feasibility

We have shown that *raising the level of abstraction at which probabilistic models are described and queried can effectively enable joint exploration of design space variability and stochastic operational uncertainties by reasoning about 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 logic enables streamlined specification of sophisticated properties to study guarantees across the solution space, including worst and best case analysis that considers both variability in design, as well as operational uncertainties that are stochastic in nature.

6.3.2 (RQ2) Generality

Our results show that *our approach is applicable to systems in different domains, in which design variability, as well as operational uncertainties are 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 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. [46]). Similarly, M-PCTL extends PCTL, which can be naturally used to check properties both in DTMC and MDP.

6.3.3 (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 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 [53] required combining the PRISM preprocessor [54] 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 [43]. We also built a planner that solves the class of problem described in the robotics scenario (Section 6.2) using an algorithm that generates paths and encodes them into PRISM models to analyze solutions in a probabilistic space [55]. All these solutions *required weeks of work, contrasting with a single-day specification effort (at most) required to solve the problem with* HaiQ *in all 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 modeling 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 defined separately. This can be observed e.g., in the specification of service types in TAS, or map and path constraints in the robotics scenario, which are reusable in other service-based and mobile robotics systems, respectively. To the best of our knowledge, *this level of reusability existed in relational modeling approaches, but not in quantitative verification.*

Computational cost and analytical capabilities. Prototype analysis performance behaves differently, depending on the problem type. In TAS, checking the compositional property constrained_mincost entailed exporing 90 configurations for an overall time of 6s ($\simeq$2% was used for configuration synthesis).[8] Checking the resilience property for a fixed topology in CAS took $\leq$40s in the worst case. However, checking for the optimal configuration took exploring 254 configurations ($\simeq$1hr, $\simeq$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 configurations whose behavior has to be checked individually. In contrast, the mobile robotics scenario showed that increased number of map locations shifted the configuration synthesis-model checking ratio to $\simeq$50-50% with 80 locations (overall time 5s). This is explained by the relatively low number of feasible configurations, compared to the cost required to model check their behavior. Although HaiQ solution times are acceptable, the PRISM-based robotics planner outperforms HaiQ for maps with 50+ locations, indicating that for this problem type our approach is valid for prototyping and design-time analysis, but run-time infrastructure requires investing in optimized solutions. 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 checking PCTL on PRISM models like STORM [28] work out of the box). Anyway, it is worth noting that *the most computationally-demanding use cases of the approach correspond to analytical capabilities that did not exist in other approaches*, like obtaining the configurations that best satisfy a set of quantitative guarantees for a given set of assumptions about stochastic uncertainties in the environment.

6.3.4 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

[8]Experiments were run on OSX10.1.5, Java 1.8.0_111, 2.8GHz Core i7, 16GB RAM.

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 [56]. 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 [57, 58]. 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.

7 Conclusions and Future Work

This paper introduces what is, to the best of our knowledge, the first approach that enables the joint modeling and exploration of design variability and stochastic operational uncertainties in a way that is able to provide quantitative and structural guarantees across the solution space.

This is enabled by combining 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-intensive 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 and further works that have employed HaiQ since its inception (e.g., [59]) also point at reduction of specification effort and improved reusability with respect to existing probabilistic model checking techniques (c.f. Section 6.3.3).

In future work, we plan to support to expand the range of formalisms that allow reasoning about continuous-time (e.g., PTA) and game-theoretical quantitative verification (e.g., stochastic, multiplayer games or SMG). Moreover, we plan to extend our approach to support further sources and types of uncertainty, hierarchical specification [56] and compositional verification [57] techniques (c.f., Section 6.3.4) to improve modularity and scalability.

Appendix A Language Syntax

The modeling language is formed by a static part (a subset of the relational logic of Alloy , c.f., http://alloytools.org/download/alloy-language-reference.pdf), and a behavioral part based on a guarded probabilistic command language.

Figure A1 shows the grammar of HaiQ. The definitions in black correspond to the static part of the language, while definitions in blue correspond to the behavioral aspects of the language.

A.0.1 Static/Structural

The static part of HaiQ excludes the following from its syntax, with respect to Alloy:

1. Functions and assertions.
2. Imports.
3. Non-unary relations (i.e., only sets are allowed).

```
1.  ⟨model⟩ ::= [ModelType : ⟨modeltype⟩] ⟨paragraph⟩*
2.  ⟨modeltype⟩ ::= dtmc | mdp
3.  ⟨paragraph⟩ ::= ⟨sigDecl⟩ | ⟨factDecl⟩ | ⟨predDecl⟩ | ⟨cmdDecl⟩
4.  ⟨sigDecl⟩ ::= [abstract] [mult] sig ⟨name⟩ [extends ⟨name⟩] { (⟨decl⟩,)* } [⟨behBlock⟩]
5.  ⟨behBlock⟩ ::= <\ ⟨enumDecl⟩* ⟨formulaDecl⟩* ⟨varDecl⟩* ⟨grdCmd⟩* ⟨rwdDecl⟩* \>
6.  ⟨enumDecl⟩ ::= enum ⟨name⟩ : { (⟨name⟩,)+ } ;
7.  ⟨varDecl⟩ ::= var ⟨name⟩ : ⟨varTypeDecl⟩ [init ⟨name⟩] ;
8.  ⟨varTypeDecl⟩ ::= bool | [⟨name⟩..⟨name⟩] | [⟨name⟩]
9.  ⟨grdCmd⟩ ::= [ [⟨event⟩] ] ⟨behPred⟩ − > (⟨behUpdate⟩+)+ ;
10. ⟨event⟩ ::= ⟨name⟩ | ⟨name⟩ : ⟨name⟩
11. ⟨rwdDecl⟩ ::= [ [⟨event⟩]] ⟨behPred⟩ : ⟨behExpr⟩ ;
12. ⟨behPred⟩ ::= ⟨behExpr⟩ | ⟨behExpr⟩ ⟨behCompareOp⟩ ⟨behExpr⟩ | ( ⟨behPred⟩ )
13. ⟨behExpr⟩ ::= ⟨name⟩ | ⟨number⟩ | ⟨behUnOp⟩ ⟨behExpr⟩ | ⟨behExpr⟩ ⟨behBinOp⟩ ⟨behExpr⟩
              | true | false | ( ⟨behExpr⟩ )
14. ⟨behUpdate⟩ ::= [⟨behExpr⟩ :] (⟨behUpdateExpr⟩&)+
15. ⟨behUpdateExpr⟩ ::= ⟨name⟩ ’ = ⟨behExpr⟩
16. ⟨behCompareOp⟩ ::= = | <= | >= | < | > | ! =
17. ⟨behUnOp⟩ ::= - | !
18. ⟨behBinOp⟩ ::= + | - | * | / | => | <=> | | | &
19. ⟨formulaDecl⟩ ::= formula ⟨name⟩ = [ ⟨behExpr⟩ | ⟨behPred⟩ ]
20. ⟨mult⟩ ::= lone | some | one
21. ⟨decl⟩ ::= [disj] ⟨name⟩+ : [disj] ⟨expr⟩
22. ⟨factDecl⟩ ::= fact [⟨name⟩] ⟨block⟩
23. ⟨predDecl⟩ ::= pred ⟨name⟩ ⟨paraDecls⟩ ⟨block⟩
24. ⟨paraDecls⟩ ::= ( (⟨decl⟩,)* ) | [ (⟨decl⟩,)* ]
25. ⟨cmdDecl⟩ ::= run ⟨name⟩ ⟨scope⟩
26. ⟨scope⟩ ::= for ⟨number⟩ [but ⟨typescope⟩+] | for ⟨typescope⟩+
27. ⟨typescope⟩ ::= [exactly] ⟨number⟩ ⟨name⟩
28. ⟨expr⟩ ::= ⟨const⟩ | ⟨name⟩ | ⟨expr⟩ [⟨expr⟩*] | ⟨unOp⟩ ⟨expr⟩ | ⟨expr⟩ ⟨binOp⟩ ⟨expr⟩
                 | ⟨expr⟩ [! | not] ⟨compareOp⟩ ⟨expr⟩ | ⟨expr⟩ ( => | implies) ⟨expr⟩ else ⟨expr⟩
                 | ⟨quant⟩ ⟨decl⟩+ ⟨blockOrBar⟩ | { ⟨decl⟩+ ⟨blockOrBar⟩ } | ( ⟨expr⟩ ) | ⟨block⟩
29. ⟨const⟩ ::= ⟨number⟩ | none | univ | iden
30. ⟨unOp⟩ ::= ! | not | no | ⟨mult⟩ | set | # |˜ |* |ˆ
31. ⟨binOp⟩ ::= || | or | && | and | <=> | iff | => | implies | & | + | - | ++ | .
32. ⟨compareOp⟩ ::= in | = | <= | >= | < | >
33. ⟨block⟩ ::= { ⟨expr⟩* }
34. ⟨blockOrBar⟩ ::= ⟨block⟩ | | ⟨expr⟩
35. ⟨quant⟩ ::= all | no | sum | ⟨mult⟩
```

Fig. A1 HaiQ’s modeling language grammar.

Every model starts by a declaration of its type (lines 1-2). If model type is not declared, the default is dtmc. Then, the model can include arbitrary sets of signature, facts, predicates, and command declarations (line 3). For fact and predicate declarations, the syntax is the standard one used in Alloy, with the exceptions noted above, which eliminate the arrow operator, as well as the domain and range restriction operators (limitation 3). Command declarations (lines 25-27) are standard.

With respect to signature declarations, inheritance in Alloy can be multiple, and this is not allowed in HaiQ, so *a signature can be the subtype (and therefore extend) only one signature* (line 4).

A.0.2 Behavioral

Every signature declaration includes a behavioral block that defines a probabilistic process type. These probabilistic process types form the behavioral part of the language (highlighted in blue). The behavioral block of a signature (line 5) includes enumerated types, variable, commands, formulas, and reward declarations.

The main body of the behavioral block consists of sets of guarded probabilistic commands (c.f., line 9) that encode process types (either DTMC or MDP) that follow the pattern:

$$[event]\ guard \text{ -> } p_1 : update_1 + \ldots + p_n : update_n$$

Here, *event* (c.f., ⟨event⟩, line 10) can be a simple label that is ’hardwired’ to synchronize with any other event of the same name in a different process (e.g., 'send'), or a label prefixed by a relation (e.g., 'conn:send'), indicating that the event can synchronize with any other event named 'send' in related processes via 'conn'. If no event is specified, the command encodes an internal action ('[ ]').

After the specification of the event, *guard* is a predicate (line 12) over the behavioral model variables (lines 7-8), which determines if the precondition to execute the command is satisfied.

The command also encodes a set of alternative postconditions for the execution of the command, defined as probabilistic updates (lines 14-15). Each probabilistic update consists of a probability (p_i) which can be encoded as a simple rational number or as a complex expression (c.f., line 13), and the variable update itself, update_i, which is specified as a set of primed variable updates (see ⟨behUpdateExpr⟩ in line 15), e.g., '(x'=1) & (y'=true)'. Unspecified probability for an update defaults to probability 1.

In the MDP model type, multiple commands with overlapping guards introduce local nondeterminism, whereas in DTMC the overlapping transitions result in a probabilistic choice in which probabilities of the different branches are normalized (similarly to the standard semantics described for reactive modules-style languages like PRISM).

Acknowledgements. Work partially funded by the Spanish Government (FEDER, Ministerio de Ciencia e Innovación–Agencia Estatal de Investigación) under projects TED2021-130523B-I00 and PID2021-125527NB-I00.

References

[1] Weyns, D., Calinescu, R., Mirandola, R., Tei, K., Acosta, M., Bennaceur, A., Boltz, N., Bures, T., Cámara, J., Diaconescu, A., Engels, G., Gerasimou, S., Gerostathopoulos, I., Yaman, S.G., Grassi, V., Hahner, S., Letier, E., Litoiu, M., Marsso, L., Musil, A., Musil, J., Rodrigues, G.N., Perez-Palacin, D., Quin, F., Scandurra, P., Vallecillo, A., Zisman, A.: Towards a research agenda for understanding and managing uncertainty in self-adaptive systems. ACM SIGSOFT Softw. Eng. Notes **48**(4), 20–36 (2023)

[2] Cámara, J., Troya, J., Vallecillo, A., Bencomo, N., Calinescu, R., Cheng, B.H.C., Garlan, D., Schmerl, B.R.: The uncertainty interaction problem in self-adaptive systems. Softw. Syst. Model. **21**(4), 1277–1294 (2022)

[3] Cámara, J., Calinescu, R., Cheng, B.H.C., Garlan, D., Schmerl, B.R., Troya, J., Vallecillo, A.: Addressing the uncertainty interaction problem in software-intensive systems: challenges and desiderata. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS, pp. 24–30. ACM, ??? (2022)

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

[5] Spivey, J.M.: Z Notation - a Reference Manual (2. Ed.). Prentice Hall International Series in Computer Science. Prentice Hall, ??? (1992)

[6] Abrial, J., Lee, M.K.O., Neilson, D., Scharbach, P.N., Sørensen, I.H.: The b-method. In: VDM '91 - Formal Software Development. LNCS, vol. 552, pp. 398–405. Springer, ??? (1991)

[7] Bjørner, D.: The vienna development method (VDM): software specification & program synthesis. In: Mathematical Studies of Information Processing, Proceedings of the International Conference. LNCS, vol. 75, pp. 326–359. Springer, ??? (1978)

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

[9] Maoz, S., Ringert, J.O., Rumpe, B.: Synthesis of component and connector models from crosscutting structural views. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'13, pp. 444–454. ACM, ??? (2013)

[10] Wong, S., Sun, J., Warren, I., Sun, J.: A scalable approach to multi-style architectural modeling and verification. In: 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008), pp. 25–34 (2008)

[11] Bagheri, H., Tang, C., Sullivan, K.J.: Trademaker: automated dynamic analysis of synthesized tradespaces. In: 36th Int. Conf. on Software Engineering, pp. 106–116. ACM, ??? (2014)

[12] Zave, P.: A formal model of addressing for interoperating networks. In: FM 2005: Formal Methods, International Symposium of Formal Methods Europe. LNCS, vol. 3582, pp. 318–333. Springer, ??? (2005)

[13] Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: COVERT: compositional analysis of android inter-app permission leakage. IEEE Trans. Software Eng. **41**(9), 866–886 (2015)

[14] Kwiatkowska, M.Z., Norman, G., Parker, D.: 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, pp. 220–270. Springer, ??? (2007)

[15] Calinescu, R., Ghezzi, C., Kwiatkowska, M.Z., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Commun. ACM **55**(9), 69–77 (2012)

[16] Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE, pp. 341–350. ACM, ??? (2011)

[17] Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification - 23rd International Conference, CAV, vol. 6806, pp. 585–591. Springer, ??? (2011)

[18] Gilmore, S., Hillston, J.: 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, pp. 353–368. Springer, ??? (1994)

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

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

[21] Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: Profeat: feature-oriented engineering for family-based probabilistic model checking. Formal Aspects of Computing (2017)

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

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

[24] Cámara, J.: HaiQ: Synthesis of software design spaces with structural and probabilistic guarantees. In: FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, pp. 22–33. ACM, ??? (2020)

[25] Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: Dynalloy: upgrading alloy with actions. In: 27th International Conference on Software Engineering (ICSE), pp. 442–451. ACM, ??? (2005)

[26] Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, pp. 373–383. ACM, ??? (2016)

[27] Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, ??? (2010)

[28] Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022)

[29] David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal stratego. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 9035, pp. 206–211. Springer, ??? (2015)

[30] Courtney, T., Gaonkar, S., Keefe, K., Rozier, E., Sanders, W.H.: 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, pp. 353–358. IEEE CS, ??? (2009)

[31] Calinescu, R., Ceska, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: 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, pp. 131–140. IEEE, ??? (2017)

[32] Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Software Eng. **30**(5), 295–310 (2004)

[33] Grunske, L., Aleti, A.: Quality optimisation of software architectures and design specifications. Journal of Systems and Software **86**(10), 2465–2466 (2013)

[34] Esfahani, N., Malek, S., Razavi, K.: Guidearch: guiding the exploration of architectural solution space under uncertainty. In: 35th International Conference on Software Engineering, ICSE, pp. 43–52. IEEE CS, ??? (2013)

[35] Aleti, A., Bjornander, S., Grunske, L., Meedeniya, I.: Archeopterix: An extendable tool for architecture optimization of aadl models. In: Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES '09. ICSE Workshop On, pp. 61–71 (2009)

[36] Meedeniya, I., Moser, I., Aleti, A., Grunske, L.: 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, pp. 85–94. ACM, ??? (2011)

[37] Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: Int. Conf. on Performance Engineering. WOSP/SIPEW, pp. 105–116. ACM, ??? (2010)

[38] Bondarev, E., Chaudron, M.R.V., Kock, E.A.: Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In: 6th WS on Software and Performance. WOSP, pp. 153–163. ACM, ??? (2007)

[39] Becker, S., Koziolek, H., Reussner, R.H.: The palladio component model for model-driven performance prediction. Journal of Systems and Software **82**(1), 3–22 (2009)

[40] Brosch, F., Koziolek, H., Buhnova, B., Reussner, R.H.: Architecture-based reliability prediction with the palladio component model. IEEE Trans. Software Eng. **38**(6), 1319–1339 (2012)

[41] MacCalman, A.D., Beery, P.T., Paulo, E.P.: A systems design exploration approach that illuminates tradespaces using statistical experimental designs. Syst. Eng. **19**(5), 409–421 (2016)

[42] Dwivedi, V., Garlan, D., Pfeffer, J., Schmerl, B.: Model-based assistance for making time/-fidelity trade-offs in component compositions. In: 11th International Conference on Information Technology: New Generations, ITNG 2014. IEEE CS, ??? (2014)

[43] Cámara, J., Garlan, D., Schmerl, B.R.: Synthesis and quantitative verification of tradeoff spaces for families of software systems. In: Software Architecture - 11th European Conference, ECSA. LNCS, vol. 10475, pp. 3–21. Springer, ??? (2017)

[44] Kwiatkowska, M., Norman, G., Parker, D., Vigliotti, M.G.: Probabilistic mobile ambients. Theoretical Computer Science **410**(12–13), 1272–1303 (2009)

[45] Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press, ??? (2006)

[46] The PRISM Language - Semantics. www.prismmodelchecker.org/doc/semantics.pdf. [Online; accessed 2-2024]

[47] Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6**(5), 512–535 (1994)

[48] Andova, S., Hermanns, H., Katoen, J.: Discrete-time rewards model-checked. In: Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS. LNCS, vol. 2791, pp. 88–104. Springer, ??? (2003)

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

[50] Weyns, D., Calinescu, R.: Tele assistance: A self-adaptive service-based system exemplar. In: 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, pp. 88–92. IEEE CS, ??? (2015)

[51] Meshenberg, R., Gopalani, N., Kosewski, L.: Active-Active for Multi-Regional Resiliency. http://techblog.netflix.com/2013/12/active-active-for-multi-regional.html. [Online; accessed 2-2024] (2013)

[52] Jacobson, D., Yuan, D., Joshi, N.: Scryer: Netflix's Predictive Auto Scaling Engine. http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html. [Online; accessed 2-2024] (2013)

[53] Glazier, T.J., Cámara, J., Schmerl, B.R., Garlan, D.: Analyzing resilience properties of different topologies of collective adaptive systems. In: IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops, pp. 55–60. IEEE CS, ??? (2015)

[54] Parker, D.: The PRISM Preprocessor. http://www.prismmodelchecker.org/prismpp/. [Online; accessed 2-2024] (2002)

[55] Cámara, J., Schmerl, B.R., Garlan, D.: Software architecture and task plan co-adaptation for mobile service robots. In: Honiden, S., Nitto, E.D., Calinescu, R. (eds.) SEAMS '20: IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Seoul, Republic of Korea, 29 June - 3 July, 2020, pp. 125–136. ACM, ??? (2020)

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

[57] Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering. CBSE '13. ACM, ??? (2013)

[58] Stevens, C., Bagheri, H.: Parasol: efficient parallel synthesis of large model spaces. In: Roychoudhury, A., Cadar, C., Kim, M. (eds.) Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, pp. 620–632. ACM, ??? (2022)

[59] Skandylas, C., Khakpour, N., Cámara, J.: Security countermeasure selection for component-based software-intensive systems. In: 22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022, pp. 63–72. IEEE, ??? (2022)