Towards Refinement and Generalization of Reliability Models Based on Component States

Natasha Jarus, Sahra Sedigh Sarvestani, and Ali R. Hurson
Department of Electrical and Computer Engineering
Missouri University of Science and Technology
Rolla, MO 65409, USA
Email: {jarus, sedighs, hurson}@mst.edu

Abstract—Complex system design often proceeds in an iterative fashion, starting from a high-level model and adding detail as the design matures. This process can be assisted by metamodeling techniques that automate some model manipulations and check for or eliminate modeling mistakes. Our work focuses on metamodeling reliability models: we describe generalization and refinement operations for these models. Generalization relaxes constraints that may be infeasible or costly to evaluate; refinement adds further detail to produce a model that more closely describes the desired system. We define these operations in terms of operations on system constraints. To illustrate the proposed method, we relate these constraints to a common Markov chain-based reliability modeling formalism.

I. Introduction

Designers of critical complex systems—such as autonomous vehicles, power grids, or water distribution networks—must ensure their systems can dependably meet performance requirements. Dependability encompasses a variety of system metrics that describe the ability of a system to continue to provide service as its components degrade. Among the most common of these metrics is *reliability*: the probability that a system remains functional up to time t. Reliability takes a binary view of system function: components, and the system, are either functional or failed. Reliability models based on component states compute a system's reliability as a function of the reliabilities of its components. This function is determined by the structure of the system—how its components are connected. For example, a power grid consisting of two transmission lines in parallel is more reliable than the system with the same lines connected in series.

Complex systems are often designed iteratively. Requirements are gathered and an initial design is prepared, modeled, and analyzed. Based on the results, the design is modified to better fit the requirements (or the requirements are modified so the design can better fit them) and the process repeats. Initial designs and models may be quite general; but they become more detailed as the design progresses. As the design process can have many iterations, *metamodeling* approaches, which model operations applied to models, are often used to reduce the labor involved, eliminate certain modeling mistakes, and even to help explore the design space.

When modifying a model, we typically want to either add more detail—a new component, a stronger constraint on how

that component behaves—or we want to remove a constraint that is unrealistic or would render the design infeasible. The first action we call *refinement* and the second *generalization*. Refinement can be used to fill out detail in a high-level model that meets design requirements; generalization can be used to "back out" of a design choice that isn't working. Both can be used together to explore the design space—refinement asks "what is the smallest detail that could be added to this model?"; generalization asks "what happens if this detail is removed?" It is our goal to make these actions explicit and exact, enabling further analysis and software automation.

In this work, we propose a method for generalization and refinement of Markov Imbeddable Structure (MIS) reliability models where system-level states are identified based on component-level states. The initial state is one where every component is functional; the terminal state is one where enough components have failed to cause system failure, and intermediate states correspond to the system remaining functional despite some the failure of some of its components. These models describe a system composed of n components as a Markov chain, encoding each component's reliability and the effect of its failure on other components. The reliability of the system is then the probability that the system remains functional after taking n steps through the Markov chain. Our work focuses on MIS models where the states of the Markov chain are defined by component status (e.g., "component 3 failed" or "only component 2 functional") and where the component status described by a state remains the same regardless of which component's failure is being considered. This encompasses the vast majority of MIS models, especially as used in practice; however, it does not encompass certain unusual MIS models, such as models of consecutive-k-of-n systems.[1] These we will address in future work.

When formalizing generalization and refinement, we should consider system properties that are preserved by these operations. Roughly speaking, if the model m_r is a refinement of a model m_g, the constraints imposed on the system by m_r should imply the constraints imposed by m_g. For example, if m_r requires a component c to have reliability ≥ 0.9, m_g can require that c have reliability ≥ 0.7—this constraint is strictly

---

[1] In short, the transition probability matrices for consecutive-k-of-n systems are not upper triangular; for more detail, see [1, pp. 344–345].

weaker than the constraint of m_r. However, m_g could not require c to have reliability ≥ 0.99. In other words, a system meeting the requirements of m_r would provide equal or better reliability than a system meeting m_g's requirements alone. If m_r refines m_g, then m_g generalizes m_r, so we can use the same implication relationship to describe both refinement and generalization. We formally abstract system properties and implication to analyze the soundness of our definitions of generalization and refinement.

Another advantage of describing refinement and generalization in this fashion is that it can be used for model-to-model transformations as shown in our previous work [2]. Provided another formalism represents some of the same system properties, we can relate these MIS models to this formalism in a way that lets us soundly convert between the two. Thus, the effort required to develop this formalism enables more than the single application this work discusses.

The rest of this paper is as follows. Section II provides a summary of the theory behind our approach. System constraints, generalization, and refinement are defined in Section III. These operations are connected to MIS models in Section IV. Finally, related work is surveyed in Section V and Section VI presents our conclusions.

II. BACKGROUND

The central theory that underlies the work in this paper has been articulated in our previous work [2]. Here we recap the results in terms of the goals of this paper.

Our goal is to relate two domains—a domain of MIS models and a domain of system properties—so that if a certain set of properties describe a given system, the model generated from those properties also describes the system. Likewise, if a model describes a system, the properties generated from that model also describe the system. We use this relationship to define generalization and refinement on MIS models based on generalization and refinement of properties.

For our approach, the domains must both be complete lattices $\mathbb{L} \triangleq (\mathbf{L}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$. Recall that $\sqsubseteq$ is a partial order relation; for any subset $\mathbf{L}' \subseteq \mathbf{L}$, $\bigsqcup \mathbf{L}'$ is the least upper bound (*join*) and $\bigsqcap \mathbf{L}'$ the greatest lower bound (*meet*) of $\mathbf{L}'$; and $\bot$ and $\top$ are the least and greatest elements of the lattice. For $\mathbf{L}' = \{l_1, l_2\}$, we write $\bigsqcap \mathbf{L}'$ as $l_1 \sqcap l_2$ and $\bigsqcup \mathbf{L}'$ as $l_1 \sqcup l_2$.

Suppose we have a complete system properties lattice $\mathbb{P}\mathbf{rop} \triangleq (\mathbf{Prop}, \Rightarrow, \bigvee, \bigwedge, \bot_P, \top_P)$ (see Sec. III) and a complete MIS model lattice $\mathbb{M}\mathbf{IS} \triangleq (\mathbf{MIS}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot_M, \top_M)$ (see Sec. IV). We order both domains by *specificity*. Intuitively, properties p_1 are more specific than properties p_2 (i.e., $p_1 \sqsubseteq p_2$) if p_1 provides additional information about the system that p_2 does not. Likewise with models: if $m_1 \sqsubseteq m_2$, m_1 may offer more detail about the system; for example, m_1 may divide a component in m_2 into several components with a more complex interrelationship. The meet of two properties $p_1 \sqcap p_2$ is their logical conjunction; the join $p_1 \sqcup p_2$ is their disjunction. We will discuss both of these domains in more detail later in the paper.

We use a Galois connection to soundly relate elements of these two domains. A Galois connection between complete lattices is a pair of functions α and γ with properties similar to, but less strict than, those of an order isomorphism. Informally, Galois connections allow one of the lattices to have "more detail" than the other; they are often used in cases where one lattice is an abstraction of the other.

Definition II.1. A *Galois connection* $(\mathbb{P}, \alpha, \gamma, \mathbb{M})$ between complete lattices $\mathbb{P}$ and $\mathbb{M}$ is a pair of functions $\alpha : \mathbb{P} \to \mathbb{M}$ and $\gamma : \mathbb{M} \to \mathbb{P}$ such that

(i) $\forall p \in \mathbb{P}, p \sqsubseteq (\gamma \circ \alpha)(p)$ and
(ii) $\forall m \in \mathbb{M}, (\alpha \circ \gamma)(m) \sqsubseteq m$.

α is called the *abstraction function* (or *abstraction operator*); γ is called the *concretization function (operator)*.

Given a Galois connection $(\mathbb{P}\mathbf{rop}, \alpha, \gamma, \mathbb{M}\mathbf{IS})$, what do properties II.1.(i) and II.1.(ii) mean in terms of system properties and MIS models? Property II.1.(i) states that for every collection of properties p, $p \Rightarrow (\gamma \circ \alpha)(p)$: if we abstract a model from p, then concretize properties from that model; the result is at worst more general than the properties with which we began. Likewise, property II.1.(ii) states that for every MIS model m, $(\alpha \circ \gamma)(m) \sqsubseteq m$. Thus, concretizing properties from an MIS model, then abstracting a model from those properties, produces at worst a model more specific than the initial model. (It is often the case that the $\sqsubseteq$ in II.1.(ii) is equality.)

What remains is to relate our domains and the Galois connection between them to a notion of *soundness*. Soundness is a relative property; whether a model or a collection of properties is sound or not depends on the system being modeled. Let $\mathcal{S} \in \mathbf{Sys}$ denote the system we are modeling. We encode soundness by a relation:

Definition II.2. A relation $R_\mathbb{L} : \mathbf{Sys} \to \mathbb{L}$ between systems and elements of a lattice $\mathbb{L}$ is a *soundness relation* if

(i) if $\mathcal{S} R_\mathbb{L} l_1$ and $l_1 \sqsubseteq l_2$, then $\mathcal{S} R_\mathbb{L} l_2$ and
(ii) if $\mathbf{L}' \subseteq \mathbb{L}$ and $\forall l \in \mathbf{L}', \mathcal{S} R_\mathbb{L} l$, then $\mathcal{S} R_\mathbb{L} \bigsqcap \mathbf{L}'$.

We suppose that we have a soundness relation $R_\mathbb{P} : \mathbf{Sys} \to \mathbb{P}\mathbf{rop}$ such that $\mathcal{S} R_\mathbb{P} p$ if and only if the properties in p describe $\mathcal{S}$. Every generalization of a correct collection of properties is sound by property II.2.(i). Not every refinement of a collection of properties is necessarily sound—otherwise, every property would be sound for every system. However, if we know several sound properties, property II.2.(ii) states that they can be refined to a single sound property that implies all known sound properties.

Given the soundness relation $R_\mathbb{P}$, we can induce a soundness relation $R_\mathbb{M} : \mathbf{Sys} \to \mathbb{M}\mathbf{IS}$ by $\mathcal{S} R_\mathbb{M} m \iff \mathcal{S} R_\mathbb{P} \gamma(m)$. Therefore, if properties p_r soundly refine p_g, then $\alpha(p_r)$ soundly refines $\alpha(p_g)$. In short, we need only consider the soundness of refinements in $\mathbb{P}\mathbf{rop}$; the soundness of our MIS models follows.

III. PROPERTIES

Before we describe refinement and generalization of MIS models, we formalize the constraints they place on system design. The MIS models we consider in this work place three broad constraints on a system: what components are in the system, how reliable each component is, and which components

depend on others to remain functional. The properties domain $\mathbb{P}\mathbf{rop}$ defines these as a lattice, allowing us to relate these properties to MIS models.

As we will need some way to identify components, let $\mathbf{Comps} \triangleq \{c_1, c_2, \dots\}$ be the set of all possible component names.

Each element $p \in \mathbb{P}\mathbf{rop}$ is a triplet $p = (\mathbf{C}, \mathsf{R}, \mathbf{D})$ where

- $\mathbf{C} \subseteq \mathbf{Comps}$ is the finite set of names of components in the system (e.g., $\{c_1, c_2, c_3\}$);
- $\mathsf{R} : \mathbf{C} \to [0, 1]$ is a function that specifies a lower bound for the reliability of each component: if the reliability of c is p, then $\mathsf{R}(c) \leq p$; and
- $\mathbf{D} \subseteq \mathbf{Deps}$ is the finite set of component dependencies, as described in the next section.

For example, a system consisting of two 90% reliable power lines in parallel where the failure of one causes the other to become overloaded and thus fail as well would be described by the properties $(\mathbf{C} = \{c_1, c_2\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = 0.9, \mathbf{D} = \{\langle c_1 \rightsquigarrow c_2, \mathcal{S}\rangle, \langle c_2 \rightsquigarrow c_1, \mathcal{S}\rangle\})$.

A. Dependencies

Component dependencies (elements of $\mathbf{Deps}$) are represented by the relation $\langle \_ \rightsquigarrow \_ \rangle : \mathcal{P}(\mathbf{C}) \to \mathcal{P}(\mathbf{C} \cup \{\mathcal{S}\})$.[2] The statement $\langle \cdots_1 \rightsquigarrow \cdots_2 \rangle$ means "the failure of the components in the set $\cdots_1$ immediately leads to the failure of the components in $\cdots_2$". Should $\mathcal{S}$ appear in $\cdots_2$, the system also fails as a result of the components of $\cdots_1$ failing. The components on the left side ($\cdots_1$) are referred to as *causes* and the components on the right ($\cdots_2$) as *effects*.

These dependencies correspond to state transitions. Suppose we have a system with components $\mathbf{C} = \{c_1, c_2, c_3\}$. We can represent the state of the components as three-bit strings: $\widetilde{111}$ corresponds to the system state where all components are functional, $\widetilde{101}$ corresponds to the state where c_2 has failed, etc. A dependency $\langle c_1 \rightsquigarrow \emptyset\rangle$ corresponds to a transition from $\widetilde{111}$ to $\widetilde{011}$ when c_1 fails—the failure of c_1 does not influence the functionality of other components in the system. Likewise, a dependency $\langle c_1, c_2 \rightsquigarrow c_3, \mathcal{S}\rangle$ corresponds to transitions from $\widetilde{101}$ to $\widetilde{000}$ when c_1 fails and from $\widetilde{011}$ to $\widetilde{000}$ when c_2 fails; furthermore, in state $\widetilde{000}$ the system is considered failed. Sec. IV formalizes this correspondence.

As there are a number of ways to write dependencies, we place some constraints on them to ensure the constraints on the system are consistent with how components fail and fully cover all cases of system behavior. These constraints are split into *equivalences* and *well-formedness (WF) properties*.

1) Equivalences: The first equivalence rule states that if a component appears on both sides of $\rightsquigarrow$, we can remove it from the right side. The failure of any component trivially causes that component to fail; this rule states that we need not write this fact explicitly:[3]

$$\langle c \cdots_1 \rightsquigarrow c \cdots_2 \rangle \equiv \langle c \cdots_1 \rightsquigarrow \cdots_2 \rangle. \qquad \text{(Tautology)}$$

The remaining two equivalences are between sets of dependencies, rather than between two individual dependencies. If we have two dependencies with the same cause but different effects, we can produce one dependency that represents both by taking the union of their effects:

$$\left\{ \begin{array}{l} \langle \cdots_1 \rightsquigarrow \cdots_2 \rangle \\ \langle \cdots_1 \rightsquigarrow \cdots_3 \rangle \end{array} \right\} \equiv \{\langle \cdots_1 \rightsquigarrow \cdots_2 \cdots_3 \rangle\}. \qquad \text{(Union)}$$

Finally, a dependency with no causes cannot occur:

$$\{\langle \emptyset \rightsquigarrow \cdots \rangle\} \equiv \emptyset. \qquad \text{(Inaction)}$$

2) Well-formedness Properties: The WF properties describe a system-level view of dependencies: what dependencies need to be present in $\mathbf{D}$ to make a consistent set of system constraints. First, every component must have a dependency where it is the sole cause of failure (although the effect may be the empty set). These correspond to transitions from the initial $\widetilde{1 \cdots 1}$ state:

$$\forall c \in \mathbf{C}, \exists \langle c \rightsquigarrow \cdots \rangle \in \mathbf{D}. \qquad \text{(Initiality)}$$

In addition, at least one sequence of failures must lead to the system failing (otherwise, the system's reliability would be 1 and there would be nothing to model):

$$\exists \langle \cdots_1 \rightsquigarrow \mathcal{S} \cdots_2 \rangle \in \mathbf{D}. \qquad \text{(Termination)}$$

Finally, components cannot recover as a result of the failure of other components. Thus, if components $\cdots_1$ cause components $\cdots_2$ to fail, any other dependency where $\cdots_1$ have failed must also have $\cdots_2$ failed.

$$\begin{aligned} &\forall \langle \cdots_1 \rightsquigarrow \cdots_2 \rangle \in \mathbf{D}, \\ &\forall \langle \cdots_1 \cdots_3 \rightsquigarrow \cdots_4 \rangle \in \mathbf{D}, \qquad \text{(Monotonicity)} \\ &\cdots_2 \subseteq \cdots_3 \cup \cdots_4. \end{aligned}$$

For instance, if we have $\langle c_1 \rightsquigarrow c_2 \rangle$, Monotonicity would permit the dependencies $\langle c_1, c_3 \rightsquigarrow c_2 \rangle$ and $\langle c_1, c_2 \rightsquigarrow c_3 \rangle$ but forbid $\langle c_1, c_3 \rightsquigarrow \emptyset \rangle$, as c_2 must always fail when c_1 fails.

3) Examples: Before addressing generalization and refinement of properties, we demonstrate a few examples of how dependencies are used to specify system behavior. First, consider the dependencies in the earlier parallel-component example: $\mathbf{D} = \{\langle c_1 \rightsquigarrow c_2, \mathcal{S}\rangle, \langle c_2 \rightsquigarrow c_1, \mathcal{S}\rangle\}$. In this system, the failure of component c_1 leads to the failure of c_2 and system failure, and vice versa for c_2. This system has two states, $\widetilde{11}$ and $\widetilde{00}$; the failure of either component causes a transition from the first to the second.

By contrast, a parallel-component system where the two components are independent would be specified by $\mathbf{D} = \{\langle c_1 \rightsquigarrow \emptyset\rangle, \langle c_2 \rightsquigarrow \emptyset\rangle, \langle c_1, c_2 \rightsquigarrow \mathcal{S}\rangle\}$. This system has all four possible states and all valid transitions between states.

A system with two components in series produces a more interesting "failed" state. These components are independent, as one failing does not cause the other to fail, but both need to be functional for the system to function: $\mathbf{D} = \{\langle c_1 \rightsquigarrow \mathcal{S}\rangle, \langle c_2 \rightsquigarrow \mathcal{S}\rangle\}$. This system also has two states: the initial state $\widetilde{11}$ and the failed superstate $\widetilde{01|10}$.[4] Once the system has failed, we are no longer interested in its behavior; thus, for this system, we consider $\widetilde{00}$ unreachable.

---

[2] $\mathcal{P}(\mathbf{S})$ denotes the set of subsets ("powerset") of the set $\mathbf{S}$.

[3] A note on notation: $c \cdots_1$ refers to a set containing the component c and the components of the set $\cdots_1$.

[4] MIS modeling requires a single "failed" system (super)state; we leave unification of functional states into superstates for future work.

B. Generalization

Now that we have described the elements of $\mathbb{P}\mathbf{rop}$, we can describe how to generalize them. The goal of generalizing an element of $\mathbb{P}\mathbf{rop}$ is to produce an element of $\mathbb{P}\mathbf{rop}$ that relaxes the constraints of the first element but does not contradict it. Understanding how constraints can be generalized allows us to order $\mathbb{P}\mathbf{rop}$ by generalization.

1) One-step generalizations of dependencies: For a given reliability model, one way to generalize dependencies is to lower the constraint on a component's reliability: a more reliable component can always be substituted for a less reliable one. We can relax the reliability of a component, c, to a lower constraint $r < \mathsf{R}(c)$ by

$$\mathsf{relax\_rel}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_,\_] : \mathbf{C} \to [0,1] \to \mathbb{P}\mathbf{rop}$$
$$\mathsf{relax\_rel}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[c,r] \triangleq (\mathbf{C},\mathsf{R}',\mathbf{D}) \qquad (1)$$

where

$$\mathsf{R}'(c') \triangleq \begin{cases} r & \text{if } c = c' \\ \mathsf{R}(c) & \text{otherwise.} \end{cases} \qquad (1.1)$$

The other means of generalizing system constraints is to generalize component dependencies. We begin by considering the smallest actions we can take that generalize system dependencies while maintaining the WF properties. There are two possible operations: merging two components and adding a new dependency $\langle \cdots \rightsquigarrow c \rangle$ among existing components. Both of these operations take one element of $\mathbb{P}\mathbf{rop}$ and infer another.

Two distinct components c_1 and c_2 can be merged into a single component c_m (where the name c_m does not already appear in $\mathbf{C} \setminus \{c_1, c_2\}$) by replacing every instance of c_1 and c_2 with c_m:

$$\mathsf{merge}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_,\_ \to \_] : \mathbf{C} \to \mathbf{C} \to \mathbf{Comps} \to \mathbb{P}\mathbf{rop}$$
$$\mathsf{merge}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[c_1, c_2 \to c_m] \triangleq (\mathbf{C}',\mathsf{R}',\mathbf{D}') \qquad (2)$$

where

$$\mathbf{C}' \triangleq \{c_m\} \cup \mathbf{C} \setminus \{c_1, c_2\} \qquad (2.1)$$

$$\mathsf{R}'(c) \triangleq \begin{cases} \min(\mathsf{R}(c_1), \mathsf{R}(c_2)) & \text{if } c = c_m, \\ \mathsf{R}(c) & \text{otherwise.} \end{cases} \qquad (2.2)$$

$$\mathbf{D}' \triangleq \{\langle m(\mathbf{c}) \rightsquigarrow m(\mathbf{e}) \rangle \mid \langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle \in \mathbf{D}\} \qquad (2.3)$$

$$m(\mathbf{c}) \triangleq \begin{cases} \{c_m\} \cup \mathbf{c} \setminus \{c_1, c_2\} & \text{if } c_1 \in \mathbf{c} \vee c_2 \in \mathbf{c}, \\ \mathbf{c} & \text{otherwise.} \end{cases} \qquad (2.4)$$

When defining a generalization, we should ensure that it only relaxes constraints. Thus, when choosing the reliability bound $\mathsf{R}'(c_m)$ of the merged component, we must pick the least restrictive choice $\min(\mathsf{R}(c_1), \mathsf{R}(c_2))$. Effectively, this choice performs two generalizations: first, we relax the tighter of the reliability bounds of c_1 and c_2 by setting $\mathsf{R}(c_1) = \mathsf{R}(c_2)$, then we merge c_1 and c_2 into one component.

The other possible generalization is adding a dependency among existing components. This may seem counterintuitive; however, it is a stronger claim to say that a component is independent of another—the fewer dependencies a system has, the more reliable it is. Adding a dependency from a nonempty set of components $\mathbf{c}$ to a component $e \notin \mathbf{c}$ means that whenever the components in $\mathbf{c}$ cause a failure, e is amongst the effects. As all the components in $\mathbf{c}$ and e are in $\mathbf{C}$ already, we need only modify the dependencies:

$$\mathsf{add\_dep}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_ \rightsquigarrow \_] : \mathcal{P}(\mathbf{C}) \to \mathbf{C} \to \mathbb{P}\mathbf{rop}$$
$$\mathsf{add\_dep}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\mathbf{c} \rightsquigarrow e] \triangleq (\mathbf{C},\mathsf{R},\mathbf{D}') \qquad (3)$$

where

$$\mathbf{D}' \triangleq \{a(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle) \mid \langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle \in \mathbf{D}\} \qquad (3.1)$$
$$\cup \{\langle \mathbf{c} \rightsquigarrow \mathbf{u} \cup \{e\} \rangle\}$$

$$a(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle) \triangleq \begin{cases} \langle \mathbf{c}' \setminus \{e\} \rightsquigarrow \mathbf{e}' \cup \{e\} \rangle & \text{if } \mathbf{c} \subseteq \mathbf{c}', \\ \langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle & \text{otherwise.} \end{cases} \qquad (3.2)$$

$$\mathbf{u} \triangleq \bigcup \{\mathbf{e}' \mid \langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle \in \mathbf{D} \text{ where } \mathbf{c}' \subseteq \mathbf{c}\} \qquad (3.3)$$

For an example of the effect of generalization operations on a system, consider a system with three independent components:

$$p = (\mathbf{C} = \{c_1, c_2, c_3\}, \mathsf{R}(\_) = 0.9, \mathbf{D} = \{$$
$$\langle c_1 \rightsquigarrow \emptyset \rangle, \langle c_2 \rightsquigarrow \emptyset \rangle, \langle c_3 \rightsquigarrow \emptyset \rangle,$$
$$\langle c_1, c_2, c_3 \rightsquigarrow \mathcal{S} \rangle$$
$$\})$$

Introducing a dependency $\langle c_1, c_2 \rightsquigarrow c_3 \rangle$ results in the following system:

$$p' = \mathsf{add\_dep}_p[c_1, c_2 \rightsquigarrow c_3]$$
$$= (\mathbf{C}' = \{c_1, c_2, c_3\}, \mathsf{R}'(\_) = 0.9, \mathbf{D}' = \{$$
$$\langle c_1 \rightsquigarrow \emptyset \rangle, \langle c_2 \rightsquigarrow \emptyset \rangle, \langle c_3 \rightsquigarrow \emptyset \rangle,$$
$$\left. \begin{matrix} \langle c_1, c_2 \rightsquigarrow c_3 \rangle^{\dagger}, \\ \langle c_1, c_2 \rightsquigarrow c_3, \mathcal{S} \rangle^{\ddagger} \end{matrix} \right\} \equiv \langle c_1, c_2 \rightsquigarrow c_3, \mathcal{S} \rangle$$
$$\})$$

Of note: the dependency marked † is the new dependency added by $\mathsf{add\_dep}$ and the dependency marked ‡ is the result of the first substitution rule in (3.2). Both rules reduce to one via the Union property.

2) Multi-step generalization of dependencies: The example of the previous section illustrates the process by which successive generalization steps are applied to system properties. To describe this more formally, let $\mathbf{G}$ be the set of all generalization operations and $\mathbf{G}^*$ be the set of finite sequences of elements of $\mathbf{G}$. We define the act of applying a sequence of generalizations to an element of properties, $[\![\_]\!](\_) : \mathbf{G}^* \to \mathbb{P}\mathbf{rop} \to \mathbb{P}\mathbf{rop}$, by

$$[\![g]\!](p) \triangleq \begin{cases} p & \text{if } g = () \\ [\![gs]\!](g'_p) & \text{if } g = (g', gs). \end{cases} \qquad (4)$$

With the ability to apply a sequence of generalizations, we now turn to the task of ordering elements of $\mathbb{P}\mathbf{rop}$.

3) Generalization as a partial order: To form a partial order on $\mathbb{P}\mathbf{rop}$ using these generalization operations, we say that if p_g generalizes p_r, there exists some sequence of generalizations that witnesses that fact:

Definition III.1. $p_g = \in \mathbb{P}\mathbf{rop}$ *generalizes* $p_r \in \mathbb{P}\mathbf{rop}$, written $p_r \sqsubseteq p_g$, if $\exists g \in \mathbf{G}^*, [\![g]\!](p_r) = p_g$.

Theorem III.1. $\sqsubseteq$ *forms a partial order on* $\mathbb{P}\mathbf{rop}$.

C. Refinement

In addition to generalization of constraints, we are interested in refining them: adding new constraints or increasing the strictness of existing ones. Refinements are dual to generalizations, so for each generalization we expect a corresponding refinement.

1) One-step Refinements: Corresponding to relax_rel we have tighten_rel which raises the bound on the reliability of component c to a higher constraint $r > R(c)$:

$$\text{tighten\_rel}_{(\mathbf{C},\mathbf{R},\mathbf{D})}[\_,\_] : \mathbf{C} \to [0,1] \to \mathbb{P}\mathbf{rop}$$

$$\text{tighten\_rel}_{(\mathbf{C},\mathbf{R},\mathbf{D})}[c,r] \triangleq (\mathbf{C}, \mathsf{R}', \mathbf{D}) \tag{5}$$

where

$$\mathsf{R}'(c') \triangleq \begin{cases} r & \text{if } c = c' \\ \mathsf{R}(c') & \text{otherwise} \end{cases} \tag{5.1}$$

To undo a merge, we split one component, c_m, into two, c_1 and c_2 (where $c_1, c_2 \notin \mathbf{C} \setminus \{c\}$). When splitting two components, we make each fully dependent on the other, as that is the most general set of constraints we can generate. In other words, the result of $\text{split}_p[c_m \to c_1, c_2]$ is the maximal element of the set $\{q \in \mathbb{P}\mathbf{rop} \mid p = \text{merge}_q[c_1, c_2 \to c_m]\}$.

$$\text{split}_{(\mathbf{C},\mathbf{R},\mathbf{D})}[\_ \to \_, \_] : \mathbf{C} \to \mathbf{Comps} \to \mathbf{Comps} \to \mathbb{P}\mathbf{rop}$$

$$\text{split}_{(\mathbf{C},\mathbf{R},\mathbf{D})}[c_m \to c_1, c_2] \triangleq (\mathbf{C}', \mathsf{R}', \mathbf{D}') \tag{6}$$

where

$$\mathbf{C}' \triangleq \{c_1, c_2\} \cup \mathbf{C} \setminus \{c_m\} \tag{6.1}$$

$$\mathsf{R}'(c) \triangleq \begin{cases} \mathsf{R}(c_m) & \text{if } c = c_1 \vee c = c_2, \\ \mathsf{R}(c) & \text{otherwise.} \end{cases} \tag{6.2}$$

$$\mathbf{D}' \triangleq \bigcup \{s(\langle \mathbf{c} \rightsquigarrow \mathbf{e}\rangle) \mid \langle \mathbf{c} \rightsquigarrow \mathbf{e}\rangle \in \mathbf{D}\} \tag{6.3}$$

$$s(\langle \mathbf{c} \rightsquigarrow \mathbf{e}\rangle) \triangleq \begin{cases} \left\{ \begin{array}{l} \langle \{c_1, c_2\} \cup \mathbf{c}' \rightsquigarrow \mathbf{e}\rangle \\ \langle \{c_1\} \cup \mathbf{c}' \rightsquigarrow \mathbf{e} \cup \{c_2\}\rangle \\ \langle \{c_2\} \cup \mathbf{c}' \rightsquigarrow \mathbf{e} \cup \{c_1\}\rangle \end{array} \right\} & \text{if } c_m \in \mathbf{c} \\ \left\{ \begin{array}{l} \langle \mathbf{c} \rightsquigarrow \mathbf{e}' \cup \{c_1, c_2\}\rangle \\ \langle \mathbf{c} \rightsquigarrow \mathbf{e}' \cup \{c_1\}\rangle \\ \langle \mathbf{c} \rightsquigarrow \mathbf{e}' \cup \{c_2\}\rangle \end{array} \right\} & \text{if } c_m \in \mathbf{e} \\ \{\langle \mathbf{c} \rightsquigarrow \mathbf{e}\rangle\} & \text{otherwise.} \end{cases} \tag{6.4}$$

$$\mathbf{c}' \triangleq \mathbf{c} \setminus \{c_m\} \tag{6.5}$$

$$\mathbf{e}' \triangleq \mathbf{e} \setminus \{c_m\} \tag{6.6}$$

Finally, remove_dep corresponds to undoing an add_dep operation. Adding a dependency $\langle \cdots_1 \rightsquigarrow e\rangle$ states that e depends on all of $\cdots_1$ and therefore every dependency containing $\cdots_1$ is rewritten to preserve Monotonicity. Removing a dependency $\langle \cdots_1 \rightsquigarrow e\rangle$ states that e is *independent* of all components in $\cdots_1$, so every dependency whose causes are contained in $\cdots_1$ is rewritten.

$$\text{remove\_dep}_{(\mathbf{C},\mathbf{R},\mathbf{D})}[\_ \rightsquigarrow \_] : \mathcal{P}(\mathbf{C}) \to \mathbf{C} \to \mathbb{P}\mathbf{rop}$$

$$\text{remove\_dep}_{(\mathbf{C},\mathbf{R},\mathbf{D})}[\mathbf{c} \rightsquigarrow e] \triangleq (\mathbf{C}, \mathsf{R}, \mathbf{D}') \tag{7}$$

where

$$\mathbf{D}' \triangleq \{r(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle) \mid \langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle \in \mathbf{D}\} \tag{7.1}$$

$$r(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle) \triangleq \begin{cases} \langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \setminus \{e\}\rangle & \text{if } \mathbf{c}' \subseteq \mathbf{c}, \\ \langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle & \text{otherwise.} \end{cases} \tag{7.2}$$

2) Multi-step Refinements: As with generalizations, let $\mathbf{R}$ be the set of all refinement operations and $\mathbf{R}^*$ be the set of all sequences of refinements. We abuse notation slightly to define application of a sequence of refinements using the same notation: for $rs \in \mathbf{R}^*$, $[\![rs]\!](p)$ is the result of applying that sequence of refinements to some system properties p.

3) Refinement as the dual of generalization: Each generalization operation and its corresponding refinement are not necessarily inverses, as most generalization operations map several elements of $\mathbb{P}\mathbf{rop}$ to the same more general system (i.e., they are not injective). Thus, we do not have that $\forall g \in \mathbf{G}$, if $q = [\![g]\!](p)$ then $\exists r \in \mathbf{R}, p = [\![r]\!](q)$. However, we can show the opposite: if $q = [\![r]\!](p)$, then p *covers* q: there is no r such that $q \sqsubset r \sqsubset p$.

Furthermore, the refinement operations form a dual order to the order defined by generalization:

Theorem III.2. $\forall p_r, p_g \in \mathbb{P}\mathbf{rop}$, $p_r \sqsubseteq p_g$ *if and only if* $\exists rs \in \mathbf{R}^*, p_r = [\![rs]\!](p_g)$.

As such, p_r *refines* p_g if $p_g \sqsupseteq p_r$, or, equivalently, $p_r \sqsubseteq p_g$.

D. The Properties Lattice

To be able to use a Galois connection to relate our notions of generalization and refinement to MIS models, we must define $\mathbb{P}\mathbf{rop}$ as a lattice. As such, we need to define top and bottom elements of $\mathbb{P}\mathbf{rop}$, least upper bounds (or *joins*), and greatest lower bounds (*meets*).[5]

The top element of $\mathbb{P}\mathbf{rop}$ is the one-element system with unconstrained component reliability:

$$\top \triangleq (\{c\}, \mathsf{R}(c) = 0, \{\langle c \rightsquigarrow \mathcal{S}\rangle\}). \tag{8}$$

Any other one-element system constrains component reliability and thus can be generalized to $\top$ by relax_rel. Removing the one dependency results in a system that does not meet the WF properties, and no further dependencies can be added without adding another component. Finally, given $p \in \mathbb{P}\mathbf{rop}$, we can show $p \sqsubseteq \top$ by repeatedly merging components in p until the result has one component, then relaxing that component's reliability bound, if necessary.

The bottom element of $\mathbb{P}\mathbf{rop}$ is a special element which corresponds to an "overdetermined" system—one where the constraints are contradictory. We do not concern ourselves with its representation, but simply define it as the element $\bot \in \mathbb{P}\mathbf{rop}$ such that $\forall p, \bot \sqsubseteq p$.

IV. MIS MODELS

Markov Imbeddable Structure models are one approach to deriving a system's reliability from the reliability of its components. These models consist of states and transitions

---

[5]Discussion of meets and joins is omitted for lack of space.

between states caused by the failure of components. The reliability of the system is determined by computing the probability of the system not reaching the "failed" state after considering the effect of each component.

This paper considers MIS models where the states are defined by the components functional in that state; e.g., $\underline{(1101)}$ corresponds to the state of a 4-component system where components 1, 2, and 4 are functional and component 3 has failed. Components cannot repair themselves, so every transition is either from one state to that same state or from one state to a state with more failed components. The *failed* state is absorbing—once the system fails, we are no longer interested in its behavior.

These transitions are usually represented in the form of transition probability matrices (TPMs) T_i, one for each component. As the system always starts in the fully functional state, the initial state probability vector is $\Pi_0 \triangleq [1, 0, \dots]$. Another vector $u \triangleq [1, \dots, 0]$ defines which states are considered functional. The system reliability is given by the product of the initial state probabilities, the TPMs, and the u vector:

$$R(\mathcal{S}) \triangleq \Pi_0^T * T_1 * T_2 * \cdots * T_n * u \qquad (9)$$

As an example, consider the system with two components in series where $R(c_1) = R(c_2) = p = 1 - q$. The TPM for both components is given by

$$T_1 = T_2 = \begin{pmatrix} p & q \\ 0 & 1 \end{pmatrix}$$

and the resulting system reliability is

$$R(\mathcal{S}) = \Pi_0^T * T_1 * T_2 * u = p^2$$

A. Abstraction and Concretization

To apply our formalization of refinement and generalization to MIS models, we need to connect our properties domain $\mathbb{P}\mathbf{rop}$ to MIS models. We achieve this by an *abstraction* operator which converts system constraints to MIS models and a *concretization* operator which derives constraints from MIS models.

To abstract an MIS model from $(\mathbf{C}, R, \mathbf{D}) \in \mathbb{P}\mathbf{rop}$, for each $c_i \in \mathbf{C}$ let $p_i = 1 - q_i = R(c_i)$ be its reliability and let T_i be its TPM. Let $n = |\mathbf{C}|$ be the number of components in the system. Then, begin with the initial fully-functional state $\underline{(1 \cdots 1)}$. For each dependency $\langle c_i \rightsquigarrow \mathbf{e} \rangle \in \mathbf{D}$, insert a transition from $\underline{(1 \cdots 1)}$ to $\underline{(1 \cdots 1)}$ with probability p_i in T_i and a transition from $\underline{(1 \cdots 1)}$ to the state where all components except c_i and those in $\mathbf{e}$ are functional with probability q_i in T_i. If $\mathcal{S} \in \mathbf{e}$, then mark that state as "failed". For each non-"failed" state added in the previous step, let $\mathbf{s}$ be the components functional in that state and let $\mathbf{f} = \mathbf{C} \setminus \mathbf{s}$ be the set of failed components. For each component $c_i \in \mathbf{s}$, select the dependency $\langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle \in \mathbf{D}$ where $c_i \in \mathbf{c}$ and $\mathbf{c}$ is the largest set such that $\mathbf{c} \subset \mathbf{f}$. Insert transitions from $\underline{(s)}$ to $\underline{(s)}$ with probability p_i and from $\underline{(s)}$ to $\underline{(s \setminus e)}$ with probability q_i into T_i. For each component $c_i \in \mathbf{f}$, insert a transition from $\underline{(s)}$ to $\underline{(s)}$ with probability 1 into T_i. Repeat this step until there are no more non-failed states to consider.

Concretizing properties from an MIS model proceeds in an analogous fashion. For each T_i create a component c_i and set $R(c_i) = p_i$. For each c_i, first let $\mathbf{s}'$ be the set of components functional after c_i fails from the initial $\underline{(1 \cdots 1)}$ state and add a dependency $\langle c_i \rightsquigarrow \mathbf{C} \setminus \mathbf{s}' \rangle$ to $\mathbf{D}$. Then consider all transitions in T_i from state $\underline{(s)}$ to state $\underline{(s')}$ where $\mathbf{s}' \subset \mathbf{s}$. Let $\mathbf{f} \triangleq \mathbf{s} \setminus \mathbf{s}' \setminus \{c_i\}$ be the set of components that also fail as a result of the failure of c_i. Take $\langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle \in \mathbf{D}$ where $c_i \in \mathbf{c}$ and $\mathbf{c}$ is the largest set such that $\mathbf{c} \subset (\mathbf{C} \setminus \mathbf{s})$. If $\mathbf{e} \neq \mathbf{f}$, add a dependency $\langle \mathbf{C} \setminus \mathbf{s} \setminus \{c_i\} \rightsquigarrow \mathbf{f} \rangle$.

B. Examples

As an example of the power of this approach, let us refine a 2-of-3 system from $\top$. Our starting system is

$$\top = (\{c_1\}, R(c_1) = 0, \{\langle c_1 \rightsquigarrow \mathcal{S} \rangle\}).$$

If we refine c_1's reliability to p by $s_1 = \mathsf{tighten\_rel}_\top[c_1, p]$, the resulting system has reliability $R(\mathcal{S}) = p$.

First, we create another component via $s_2 = \mathsf{split}_{s_1}[c_1 \rightarrow c_1, c_2]$, we get the following system:

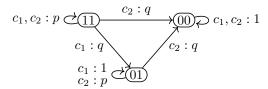$$\begin{aligned} s_2 = (\{c_1, c_2\}, R(c_1) = R(c_2) = p, \{ \\ \langle c_1 \rightsquigarrow c_2, \mathcal{S} \rangle, \langle c_2 \rightsquigarrow c_1, \mathcal{S} \rangle \\ \langle c_1, c_2 \rightsquigarrow \mathcal{S} \rangle \\ \}) \end{aligned}$$

This gives $R(\mathcal{S}) = p^2$ as we now take two steps through the Markov chain.

We can avoid adding excessive dependencies later by removing two, making c_1 independent: $s_3 = \mathsf{remove\_dep}_{s_2}[c_1 \rightsquigarrow c_2, \mathcal{S}]$.

$$\begin{aligned} s_3 = (\{c_1, c_2\}, R(c_1) = R(c_2) = p, \{ \\ \langle c_1 \rightsquigarrow \emptyset \rangle, \langle c_2 \rightsquigarrow c_1, \mathcal{S} \rangle \\ \langle c_1, c_2 \rightsquigarrow \mathcal{S} \rangle \\ \}) \end{aligned}$$

Removing these dependencies adds a new state to the Markov chain:

This gives $R(\mathcal{S}) = p^2 + pq$—either both components remain functional, or c_1 fails and c_2 remains functional.

Next, we introduce c_3 by $s_4 = \mathsf{split}_{s_3}[c_2 \rightarrow c_2, c_3]$.
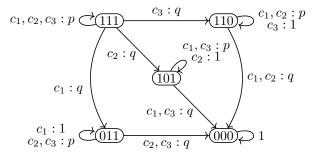
$$\begin{aligned} s_4 = (\{c_1, c_2, c_3\}, R(c_1) = R(c_2) = R(c_3) = p, \{ \\ \langle c_1 \rightsquigarrow \emptyset \rangle, \langle c_2 \rightsquigarrow c_1, c_3, \mathcal{S} \rangle, \langle c_3 \rightsquigarrow c_1, c_2, \mathcal{S} \rangle \\ \langle c_1, c_2 \rightsquigarrow \mathcal{S} \rangle, \langle c_1, c_3 \rightsquigarrow c_2, \mathcal{S} \rangle, \langle c_2, c_3 \rightsquigarrow c_1, \mathcal{S} \rangle \\ \}) \end{aligned}$$

The Markov chain is similar to the one abstracted from s_3, but c_3 adds its own transition probabilities. This gives $R(\mathcal{S}) = p^3 + p^2 q$—either all components remain functional, or c_1 fails and c_2 and c_3 remain functional.

Finally, we arrive at the desired 2-of-3 system by removing unneeded dependencies: $s_5 = \mathsf{remove\_dep}_{s_4}[c_2 \rightsquigarrow c_1, c_3, \mathcal{S}]$ and $s_6 = \mathsf{remove\_dep}_{s_5}[c_3 \rightsquigarrow c_1, c_2, \mathcal{S}]$.

$$s_6 = (\{c_1, c_2, c_3\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = \mathsf{R}(c_3) = p, \{$$
$$\langle c_1 \rightsquigarrow \emptyset \rangle, \langle c_2 \rightsquigarrow \emptyset \rangle, \langle c_3 \rightsquigarrow \emptyset \rangle$$
$$\langle c_1, c_2 \rightsquigarrow \mathcal{S} \rangle, \langle c_1, c_3 \rightsquigarrow c_2, \mathcal{S} \rangle, \langle c_2, c_3 \rightsquigarrow c_1, \mathcal{S} \rangle$$
$$\})$$

The abstracted Markov chain has two new states:

This gives $\mathsf{R}(\mathcal{S}) = p^3 + 3p^2 q$—either all components remain functional, or only one fails.

V. Related Work

Markov chains form the theoretical basis for numerous system reliability analyses. Of particular relevance to this work are two applications of MIS modeling to smart grids—power grids augmented with cyber monitoring and control capabilities to improve their dependability [3], [4]. These studies demonstrate how MIS modeling can be applied to real-world systems to capture system reliability and component interdependencies.

Refinement of specifications for software programs has been studied extensively; see [5] for an introduction and [6] for a recent survey of the literature. The essence of program specification and refinement is augmenting a programming language with a specification language. Thus, programs become specifications that are executable. To derive programs from non-executable specifications, a refinement relation is defined and various refinements of specifications are developed. This allows one to start with a high-level specification of a program's behavior and derive, through repeated refinement, an executable program whose specification refines the initial specification.

Research on refinement of Markov chains has taken two forms. The first focuses on Interval Markov Chains (IMCs) and their extension, Constraint Markov Chains (CMCs) [7], [8]. In these formalisms, transition probabilities are not given exactly, but are bounded within an interval or given by algebraic constraints, respectively. As each IMC or CMC corresponds to a collection of Markov chains that satisfy the requirements given, it is possible to define refinement directly in terms of these formalisms, rather than using a separate "system constraints" formalism, as we do. Each system specification can be written as an IMC or CMC and then refined into a complete system model via refinement and conjunction operations.

The second approach uses counterexample generation to validate Markov chain abstractions used in model checking [9]. Starting with a coarse approximation of the original Markov chain, model checking is performed until a counterexample is found. This counterexample is checked against the original specification; if the counterexample does not hold, the approximate system is refined so the counterexample no longer holds. This process repeats until a genuine counterexample is found (one that holds for the original specification) or the model checking algorithm cannot find a counterexample. A related work [10] bounds the uncertainty introduced by this approach to state-space reduction by separately modeling the uncertainty present in the model and the uncertainty added through abstraction.

VI. Conclusion

In this paper, we have proposed and demonstrated an approach to refinement and generalization of MIS reliability models. Key to this approach is a system constraints domain, which captures the behavior of a system in an abstract, easily manipulated fashion. These constraints describe the components of the system, their reliability, and dependencies that describe how one set of component failures can trigger another. Given these constraints, we create generalization and refinement operators that allow us to relax or add constraints as needed. Thus, we can simplify a system for easier evaluation by generalizing it or we can iteratively develop one through repeated refinement. Finally, we link these constraints to MIS reliability models, enabling us to refine or generalize models of a common modeling formalism.

We plan to extend the refinement framework to other modeling formalisms and to define higher-level modeling operations, e.g., model composition, in terms of refinement. Furthermore, we intend to implement this as a software tool so that it can be easily applied to large-scale systems.

References

[1] W. Kuo and M. J. Zuo, *Optimal reliability modeling: principles and applications*. John Wiley & Sons, 2003.

[2] N. Jarus, S. Sedigh Sarvestani, and A. R. Hurson, "Formalizing cyber–physical system model transformation via abstract interpretation," in *19th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, pp. 107–114, Jan. 2019.

[3] M. N. Albasrawi, N. Jarus, K. A. Joshi, and S. Sedigh Sarvestani, "Analysis of reliability and resilience for smart grids," in *38th Annual IEEE Computer Software and Applications Conference*, pp. 529–534, July 2014.

[4] K. Marashi, S. Sedigh Sarvestani, and A. R. Hurson, "Consideration of cyber-physical interdependencies in reliability modeling of smart grids," *IEEE Trans. on Sustainable Computing*, vol. 3, pp. 73–83, Apr. 2018.

[5] C. Morgan, *Programming from Specifications*. Prentice Hall international series in computer science, Prentice Hall, 1990.

[6] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends in Programming Languages*, vol. 4, pp. 1–119, July 2017.

[7] B. Caillaud, B. Delahaye, K. G. Larsen, A. Legay, M. L. Pedersen, and A. Wąsowski, "Compositional design methodology with constraint Markov chains," in *7th International Conference on the Quantitative Evaluation of Systems*, pp. 123–132, Sept. 2010.

[8] B. Delahaye, K. G. Larsen, A. Legay, M. L. Pedersen, and A. Wąsowski, "Consistency and refinement for interval Markov chains," *Journal of Logic and Algebraic Programming*, vol. 81, pp. 209–226, Apr. 2012.

[9] R. Chadha and M. Viswanathan, "A counterexample-guided abstraction-refinement framework for Markov decision processes," *ACM Trans. on Computational Logic*, vol. 12, pp. 1:1–1:49, Nov. 2010.

[10] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "A game-based abstraction-refinement framework for Markov decision processes," *Formal Methods in System Design*, vol. 36, pp. 246–280, Sept. 2010.