Refinement and Generalization of Reliability Models Based on Component States

Natasha Jarus[a], Sahra Sedigh Sarvestani[a,*], Ali R. Hurson[a]

[a] *Department of Electrical and Computer Engineering, Missouri University of Science and Technology, 301 W. 16th St., Rolla, MO, 65409 USA*

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.

Keywords: Markov Imbedded Structure Models, Refinement, Generalization

1. 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

*Corresponding author

Email addresses: jarus@mst.edu (Natasha Jarus), sedighs@mst.edu (Sahra Sedigh Sarvestani), hurson@mst.edu (Ali R. Hurson)

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.

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 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 [1]. 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 2 provides a summary of the theory behind our approach. System constraints, generalization, and refinement are defined in Section 3. These operations are connected to MIS models in Section 4. In Section 5, we extend this analysis to MIS models containing superstates and draw a connection to non-deterministic choice. Finally, related work is surveyed in Section 6 and Section 7 presents our conclusions.

2. Background

The central theory that underlies the work in this paper has been articulated in our previous work [1]. 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 $\mathrm{L} \triangleq (\mathbf{L}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$. 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 $\mathbf{Prop} \triangleq (\mathbf{Prop}, \Rightarrow, \bigvee, \bigwedge, \bot_P, \top_P)$ (see Sec. 3) and a complete MIS model lattice $\mathbf{MIS} \triangleq (\mathbf{MIS}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot_M, \top_M)$ (see Sec. 4). 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 2.1. A *Galois connection* $(\mathrm{P}, \alpha, \gamma, \mathrm{M})$ between complete lattices P and M is a pair of functions $\alpha : \mathrm{P} \to \mathrm{M}$ and $\gamma : \mathrm{M} \to \mathrm{P}$ such that

(i) $\forall p \in \mathrm{P}, p \sqsubseteq (\gamma \circ \alpha)(p)$ and

(ii) $\forall m \in \mathrm{M}, (\alpha \circ \gamma)(m) \sqsubseteq m$.

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

Given a Galois connection $(\mathbf{Prop}, \alpha, \gamma, \mathrm{M}\mathbf{IS})$, what do properties 2.1.(i) and 2.1.(ii) mean in terms of system properties and MIS models? Property 2.1.(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 2.1.(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 2.1.(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 2.2. A relation $R_{\mathrm{L}} : \mathbf{Sys} \to \mathrm{L}$ between systems and elements of a lattice L is a *soundness relation* if

(i) if $\mathcal{S}\, R_{\mathrm{L}}\, l_1$ and $l_1 \sqsubseteq l_2$, then $\mathcal{S}\, R_{\mathrm{L}}\, l_2$ and

(ii) if $\mathbf{L}' \subseteq \mathrm{L}$ and $\forall l \in \mathbf{L}', \mathcal{S}\, R_{\mathrm{L}} l$, then $\mathcal{S}\, R_{\mathrm{L}} \bigsqcap \mathbf{L}'$.

We suppose that we have a soundness relation $R_{\mathrm{P}} : \mathbf{Sys} \to \mathrm{P}\mathbf{rop}$ such that $\mathcal{S}\, R_{\mathrm{P}}\, p$ if and only if the properties in p describe $\mathcal{S}$. Every generalization of a correct collection of properties is sound by property 2.2.(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 2.2.(ii) states that they can be refined to a single sound property that implies all known sound properties.

Given the soundness relation R_{P}, we can induce a soundness relation $R_{\mathrm{M}} : \mathbf{Sys} \to \mathrm{M}\mathbf{IS}$ by $\mathcal{S}\, R_{\mathrm{M}}\, m \iff \mathcal{S}\, R_{\mathrm{P}}\, \gamma(m)$. 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 P**rop**; the soundness of our MIS models follows.

3. 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 P**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 $p \in \mathrm{P}\mathbf{rop}$ 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 ($\mathbf{C} = \{c_1, c_2\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = 0.9, \mathbf{D} = \{\langle c_1 \rightsquigarrow c_2, \mathcal{S}\rangle, \langle c_2 \rightsquigarrow c_1, \mathcal{S}\rangle\}$).

3.1. Dependencies

Component dependencies (elements of **Deps**) are represented by the relation $\langle \_ \rightsquigarrow \_ \rangle : \mathcal{P}(\mathbf{C}) \to \mathcal{P}(\mathbf{C} \cup \{\mathcal{S}\})$.[1] 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: $\boxed{111}$ corresponds to the system state where all components are functional, $\boxed{101}$ corresponds to the state where c_2 has failed, etc. A dependency $\langle c_1 \rightsquigarrow \emptyset \rangle$ corresponds to a transition from $\boxed{111}$ to $\boxed{011}$ 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 $\boxed{101}$ to $\boxed{000}$ when c_1 fails and from $\boxed{011}$ to $\boxed{000}$ when c_2 fails; furthermore, in state $\boxed{000}$ the system is considered failed. Sec. 4 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*.

3.1.1. 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:[2]

$$\langle c \cdots_1 \rightsquigarrow c \cdots_2 \rangle \equiv \langle c \cdots_1 \rightsquigarrow \cdots_2 \rangle. \tag{Tautology}$$

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:

$$\left\{ \begin{matrix} \langle \cdots_1 \rightsquigarrow \cdots_2 \rangle \\ \langle \cdots_1 \rightsquigarrow \cdots_3 \rangle \end{matrix} \right\} \equiv \{ \langle \cdots_1 \rightsquigarrow \cdots_2 \cdots_3 \rangle \}. \tag{Union}$$

Finally, a dependency with no causes cannot occur:

$$\{\langle \emptyset \rightsquigarrow \cdots \rangle\} \equiv \emptyset. \tag{Inaction}$$

3.1.2. 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 $\boxed{1 \cdots 1}$ state:

$$\forall c \in \mathbf{C}, \exists \langle c \rightsquigarrow \cdots \rangle \in \mathbf{D}. \tag{Initiality}$$

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

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

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):

$$\exists\langle \cdots_1 \rightsquigarrow \mathcal{S} \cdots_2 \rangle \in \mathbf{D}. \tag{Termination}$$

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.

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

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.

3.1.3. 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, $\textcircled{11}$ and $\textcircled{00}$; 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 $\textcircled{11}$ and the failed superstate $\textcircled{01}\vert\textcircled{10}$. Once the system has failed, we are no longer interested in its behavior; thus, for this system, we consider $\textcircled{00}$ unreachable.

3.2. Generalization

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

3.2.1. 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

$$\begin{aligned} &\mathsf{relax\_rel}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_,\_] : \mathbf{C} \to [0,1] \to \mathbf{Prop} \\ &\mathsf{relax\_rel}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[c,r] \triangleq (\mathbf{C}, \mathsf{R}', \mathbf{D}) \end{aligned} \tag{1}$$

where

$$\mathsf{R}'(c') \triangleq \begin{cases} r & \text{if } c = c' \\ \mathsf{R}(c') & \text{otherwise.} \end{cases} \tag{1a}$$

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 P**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:

$$\begin{aligned}&\mathsf{merge}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_, \_ \to \_] : \mathbf{C} \to \mathbf{C} \to \mathbf{Comps} \to \mathrm{P}\mathbf{rop}\\&\mathsf{merge}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[c_1, c_2 \to c_m] \triangleq (\mathbf{C}', \mathsf{R}', \mathbf{D}')\end{aligned} \tag{2}$$

where

$$\mathbf{C}' \triangleq \{c_m\} \cup \mathbf{C} \setminus \{c_1, c_2\} \tag{2a}$$

$$\mathsf{R}'(c) \triangleq \begin{cases}\min(\mathsf{R}(c_1), \mathsf{R}(c_2)) & \text{if } c = c_m,\\ \mathsf{R}(c) & \text{otherwise.}\end{cases} \tag{2b}$$

$$\mathbf{D}' \triangleq \{\langle m(\mathbf{c}) \rightsquigarrow m(\mathbf{e})\rangle \mid \langle \mathbf{c} \rightsquigarrow \mathbf{e}\rangle \in \mathbf{D}\} \tag{2c}$$

$$m(\mathbf{c}) \triangleq \begin{cases}\{c_m\} \cup \mathbf{c} \setminus \{c_1, c_2\} & \text{if } c_1 \in \mathbf{c} \vee c_2 \in \mathbf{c},\\ \mathbf{c} & \text{otherwise.}\end{cases} \tag{2d}$$

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 dependencies that merge generates are the result of applying the following rules until a fixed point is reached (i.e., no more dependencies match the left-hand side):

$$\begin{aligned}\langle c_1 \cdots_1 \rightsquigarrow \cdots_2\rangle &\mapsto \langle c_m \cdots_1 \rightsquigarrow \cdots_2\rangle\\ \langle c_2 \cdots_1 \rightsquigarrow \cdots_2\rangle &\mapsto \langle c_m \cdots_1 \rightsquigarrow \cdots_2\rangle\\ \langle \cdots_1 \rightsquigarrow c_1 \cdots_2\rangle &\mapsto \langle \cdots_1 \rightsquigarrow c_m \cdots_2\rangle\\ \langle \cdots_1 \rightsquigarrow c_2 \cdots_2\rangle &\mapsto \langle \cdots_1 \rightsquigarrow c_m \cdots_2\rangle\end{aligned}$$

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:

$$\begin{aligned}&\mathsf{add\_dep}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_ \rightsquigarrow \_] : \mathcal{P}(\mathbf{C}) \to \mathbf{C} \to \mathrm{P}\mathbf{rop}\\&\mathsf{add\_dep}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\mathbf{c} \rightsquigarrow e] \triangleq (\mathbf{C}, \mathsf{R}, \mathbf{D}')\end{aligned} \tag{3}$$

where

$$\begin{aligned}\mathbf{D}' \triangleq \{a(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle) \mid \langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle \in \mathbf{D}\}\\ \cup \{\langle \mathbf{c} \rightsquigarrow \mathbf{u} \cup \{e\}\rangle\}\end{aligned} \tag{3a}$$

$$a(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle) \triangleq \begin{cases}\langle \mathbf{c}' \setminus \{e\} \rightsquigarrow \mathbf{e}' \cup \{e\}\rangle & \text{if } \mathbf{c} \subseteq \mathbf{c}',\\ \langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle & \text{otherwise.}\end{cases} \tag{3b}$$

$$\mathbf{u} \triangleq \bigcup \{\mathbf{e}' \mid \langle \mathbf{c}' \rightsquigarrow \mathbf{e}'\rangle \in \mathbf{D} \text{ where } \mathbf{c}' \subseteq \mathbf{c}\} \tag{3c}$$

Again, we can view the dependencies in $\mathbf{D}'$ after adding the dependency $\langle \cdots_1 \rightsquigarrow e\rangle$ as the result of rewriting matching dependencies in $\mathbf{D}$:

$$\begin{aligned}\langle e \cdots_1 \cdots_2 \rightsquigarrow \cdots_3\rangle &\mapsto \langle \cdots_1 \cdots_2 \rightsquigarrow e \cdots_3\rangle\\ \langle \cdots_1 \cdots_2 \rightsquigarrow \cdots_3\rangle &\mapsto \langle \cdots_1 \cdots_2 \rightsquigarrow e \cdots_3\rangle\end{aligned}$$

and adding the dependency $\langle \cdots_1 \rightsquigarrow e \cdots_2 \rangle$ where the components $\cdots_2$ are the effects of failures of components in $\cdots_1$ as required by Monotonicity.

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

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

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

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

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 (3b). Both rules reduce to one via the Union property.

Continuing the example, merging c_2 and c_3 into c_4 gives us the system

$$\begin{aligned} p'' &= \mathsf{merge}_{p'}[c_2, c_3 \to c_4] \\ &= (\mathbf{C}'' = \{c_1, c_4\}, \mathsf{R}''(\_) = 0.9, \mathbf{D}'' = \{ \\ &\quad \langle c_1 \rightsquigarrow \emptyset \rangle, \\ &\quad \langle c_4 \rightsquigarrow \emptyset \rangle, \\ &\quad \langle c_1, c_4 \rightsquigarrow c_4, \mathcal{S} \rangle \equiv \langle c_1, c_4 \rightsquigarrow \mathcal{S} \rangle \\ &\quad \}) \end{aligned}$$

where both components are independent and the failure of both leads to system failure.

3.2.2. 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, $[\![\_]\!](\_) : \mathbf{G}^* \to \mathbf{Prop} \to \mathbf{Prop}$, by

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

With the ability to apply a sequence of generalizations, we now turn to the task of ordering elements of $\mathbf{Prop}$. First, we prove some monotonicity properties of any $q \in \mathbf{Prop}$ generalized from some $p \in \mathbf{Prop}$.

Theorem 3.1. *For all $p = (\mathbf{C}, \mathsf{R}, \mathbf{D}) \in \mathbf{Prop}$ and for all $g \in \mathbf{G}$ where $(\mathbf{C}', \mathsf{R}', \mathbf{D}') = [\![g]\!](p)$,*

(i) $|\mathbf{C}'| \leq |\mathbf{C}|$,

(ii) $\forall c \in \mathbf{C} \cap \mathbf{C}', \mathsf{R}'(c) \leq \mathsf{R}(c)$, *and*

(iii) if $\mathbf{C} = \mathbf{C}'$, $\forall \langle \mathbf{c} \rightsquigarrow \mathbf{e}' \rangle \in \mathbf{D}'$, *if* $\langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle \in \mathbf{D}$, $\mathbf{e} \subseteq \mathbf{e}'$.

Proof. Proceed by case analysis on g.
Case $g = \mathsf{relax\_rel}[c, r]$:

(i) $\mathbf{C}' = \mathbf{C}$. (ii) For $c' \in \mathbf{C}$, $\begin{cases} \mathsf{R}'(c') < \mathsf{R}(c') & \text{if } c = c' \\ \mathsf{R}'(c') = \mathsf{R}(c') & \text{otherwise} \end{cases}$. (iii) $\mathbf{D}' = \mathbf{D}$.

Case $g = \mathsf{merge}[c_1, c_2 \to c_m]$:

(i) $|\mathbf{C}'| = |\mathbf{C}| - 1 \leq |\mathbf{C}|$. (ii) For $c \in \mathbf{C}$, $\begin{cases} \mathsf{R}'(c) \leq \mathsf{R}(c) & \text{if } c = c_m \\ \mathsf{R}'(c) = \mathsf{R}(c) & \text{otherwise} \end{cases}$. (iii) $\mathbf{C} \neq \mathbf{C}'$.

Case $g = \mathsf{add\_dep}[\mathbf{c} \rightsquigarrow e]$:

(i) $\mathbf{C}' = \mathbf{C}$. (ii) $\mathsf{R}' = \mathsf{R}$. (iii) Consider $\langle \mathbf{c}'' \rightsquigarrow \mathbf{e}'' \rangle \in \mathbf{D}'$. If $\mathbf{c} = \mathbf{c}''$, then (by Union) $\langle \mathbf{c}'' \rightsquigarrow \mathbf{e}'' \rangle \equiv \left\{ \begin{array}{c} a(\langle \mathbf{c}'' \rightsquigarrow \mathbf{e_1} \rangle) \\ a(\langle \mathbf{c}'' \cup \{e\} \rightsquigarrow \mathbf{e_2} \rangle) \\ \langle \mathbf{c}'' \rightsquigarrow \mathbf{u} \cup \{e\} \rangle \end{array} \right\}$
so $\mathbf{e}'' = \mathbf{e_1} \cup \mathbf{e_2} \cup \mathbf{u} \cup \{e\}$, where $\mathbf{u}$ is defined as in Equation 3c. If $\langle \mathbf{c}'' \rightsquigarrow \mathbf{e_1} \rangle \in \mathbf{D}$, then $\mathbf{e_1} \subseteq \mathbf{e}''$. Otherwise, $\langle \mathbf{c}'' \rightsquigarrow \mathbf{e}'' \rangle \equiv \left\{ \begin{array}{c} a(\langle \mathbf{c}'' \rightsquigarrow \mathbf{e_1} \rangle) \\ a(\langle \mathbf{c}'' \cup \{e\} \rightsquigarrow \mathbf{e_2} \rangle) \end{array} \right\}$ and $\mathbf{e_1} \subseteq \mathbf{e}'' = \mathbf{e_1} \cup \mathbf{e_2}$ if $\langle \mathbf{c}'' \rightsquigarrow \mathbf{e_1} \rangle \in \mathbf{D}$. □

3.2.3. Generalization as a partial order

To form a partial order on **Prop** using these generalization operations, we say that if p_g generalizes p_r, there exists some sequence of generalizations that witnesses that fact:

Definition 3.1. $p_g \in \mathbf{Prop}$ *generalizes* $p_r \in \mathbf{Prop}$, written $p_r \sqsubseteq p_g$, if $\exists g \in \mathbf{G}^*, [\![g]\!](p_r) = p_g$.

Theorem 3.2. $\sqsubseteq$ *forms a partial order on* **Prop**.

Proof. Reflexivity: $\forall p \in \mathbf{Prop}, [\![()]\!](p) = p \implies \forall p \in \mathbf{Prop}, p \sqsubseteq p$.

Antisymmetry: Take $p, q \in \mathbf{Prop}$ such that $p \sqsubseteq q$ and $q \sqsubseteq p$. Then there exist $g_p, g_q \in \mathbf{G}^*$ such that $[\![g_p]\!](p) = q$ and $[\![g_q]\!](q) = p$. Proceed by case analysis on g_p.

If $g_p = ()$, then $q = [\![g_p]\!](p) = [\![()]\!](p) = p$.

Otherwise $g_p = (g, gs)$; we desire to show, using Theorem 3.1, that g cannot be "undone" by any generalization and thus $q \not\sqsubseteq p$. Let $(\mathbf{C}_p, \mathsf{R}_p, \mathbf{D}_p) = p$, $(\mathbf{C}', \mathsf{R}', \mathbf{D}') = [\![g]\!](p)$, and $(\mathbf{C}_q, \mathsf{R}_q, \mathbf{D}_q) = q$.

If $g = \mathsf{merge}_p[c_1, c_2 \to c_m]$, then $|\mathbf{C}_q| < |\mathbf{C}_p|$.

If $g = \mathsf{relax\_rel}_p[c, r]$, then $\mathsf{R}_q(c) \leq \mathsf{R}'(c) < \mathsf{R}_p(c)$ or $c \notin \mathbf{C}_q$.

If $g = \mathsf{add\_dep}_p[\mathbf{c} \rightsquigarrow e]$, then either $\langle \mathbf{c} \rightsquigarrow \mathbf{e}_p \rangle \notin \mathbf{D}_p$ or $\mathbf{e}_p \subset \mathbf{e}_q$.

Thus, by Theorem 3.1, $q \not\sqsubseteq p$.

Transitivity: Take $p, q, r \in \mathbf{Prop}$ such that $p \sqsubseteq q$ and $q \sqsubseteq r$. Then there exist $g_p, g_q \in \mathbf{G}^*$ such that $[\![g_p]\!](p) = q$ and $[\![g_q]\!](q) = r$. The composition of these generalizations is equal to the concatenation of their sequences: $[\![g_q]\!]([\![g_p]\!](p)) = [\![g_p; g_q]\!](p)$. Furthermore, the concatenation of two finite sequences is a finite sequence, so $g_p; g_q \in \mathbf{G}^*$. As $[\![g_p; g_q]\!](p) = r$, $p \sqsubseteq r$. □

3.3. 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.

3.3.1. One-step Refinements

Corresponding to $\mathsf{relax\_rel}$ we have $\mathsf{tighten\_rel}$ which raises the bound on the reliability of component c to a higher constraint $r > \mathsf{R}(c)$:

$$\begin{aligned} &\mathsf{tighten\_rel}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_, \_] : \mathbf{C} \to [0, 1] \to \mathbf{Prop} \\ &\mathsf{tighten\_rel}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[c, r] \triangleq (\mathbf{C}, \mathsf{R}', \mathbf{D}) \end{aligned} \tag{5}$$

where

$$\mathsf{R}'(c') \triangleq \begin{cases} r & \text{if } c = c' \\ \mathsf{R}(c') & \text{otherwise} \end{cases} \tag{5a}$$

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 $\mathsf{split}_p[c_m \to c_1, c_2]$ is the maximal element of the set $\{q \in \mathrm{P}\mathbf{rop} \mid p = \mathsf{merge}_q[c_1, c_2 \to c_m]\}$.

$$\begin{aligned} &\mathsf{split}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_ \to \_, \_] : \mathbf{C} \to \mathbf{Comps} \to \mathbf{Comps} \to \mathrm{P}\mathbf{rop} \\ &\mathsf{split}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[c_m \to c_1, c_2] \triangleq (\mathbf{C}', \mathsf{R}', \mathbf{D}') \end{aligned} \tag{6}$$

where

$$\mathbf{C}' \triangleq \{c_1, c_2\} \cup \mathbf{C} \setminus \{c_m\} \tag{6a}$$

$$\mathsf{R}'(c) \triangleq \begin{cases} \mathsf{R}(c_m) & \text{if } c = c_1 \vee c = c_2, \\ \mathsf{R}(c) & \text{otherwise.} \end{cases} \tag{6b}$$

$$\mathbf{D}' \triangleq \bigcup \{s(\langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle) \mid \langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle \in \mathbf{D}\} \tag{6c}$$

$$s(\langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle) \triangleq \begin{cases} \left\{ \begin{array}{l} \langle \{c_1, c_2\} \cup \mathbf{c}' \rightsquigarrow \mathbf{e} \rangle \\ \langle \{c_1\} \cup \mathbf{c}' \rightsquigarrow \mathbf{e} \cup \{c_2\} \rangle \\ \langle \{c_2\} \cup \mathbf{c}' \rightsquigarrow \mathbf{e} \cup \{c_1\} \rangle \end{array} \right\} & \text{if } c_m \in \mathbf{c} \\ \left\{ \begin{array}{l} \langle \mathbf{c} \rightsquigarrow \mathbf{e}' \cup \{c_1, c_2\} \rangle \\ \langle \mathbf{c} \rightsquigarrow \mathbf{e}' \cup \{c_1\} \rangle \\ \langle \mathbf{c} \rightsquigarrow \mathbf{e}' \cup \{c_2\} \rangle \end{array} \right\} & \text{if } c_m \in \mathbf{e} \\ \{\langle \mathbf{c} \rightsquigarrow \mathbf{e} \rangle\} & \text{otherwise.} \end{cases} \tag{6d}$$

$$\mathbf{c}' \triangleq \mathbf{c} \setminus \{c_m\} \tag{6e}$$

$$\mathbf{e}' \triangleq \mathbf{e} \setminus \{c_m\} \tag{6f}$$

The resulting dependencies can be understood as the result of applying the following rewrite rules to dependencies in $\mathbf{D}$:

$$\langle \cdots_1 c_m \rightsquigarrow \cdots_2 \rangle \mapsto \begin{cases} \langle \cdots_1 c_1 c_2 \rightsquigarrow \cdots_2 \rangle \\ \langle \cdots_1 c_1 \rightsquigarrow \cdots_2 c_2 \rangle \\ \langle \cdots_1 c_2 \rightsquigarrow \cdots_2 c_1 \rangle \end{cases}$$

$$\langle \cdots_1 \rightsquigarrow \cdots_2 c_m \rangle \mapsto \begin{cases} \langle \cdots_1 \rightsquigarrow \cdots_2 c_1 c_2 \rangle \\ \langle \cdots_1 \rightsquigarrow \cdots_2 c_1 \rangle \\ \langle \cdots_1 \rightsquigarrow \cdots_2 c_2 \rangle \end{cases}$$

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.

$$\begin{aligned} &\mathsf{remove\_dep}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_ \rightsquigarrow \_] : \mathcal{P}(\mathbf{C}) \to \mathbf{C} \to \mathrm{P}\mathbf{rop} \\ &\mathsf{remove\_dep}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\mathbf{c} \rightsquigarrow e] \triangleq (\mathbf{C}, \mathsf{R}, \mathbf{D}') \end{aligned} \tag{7}$$

where

$$\mathbf{D}' \triangleq \{r(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle) \mid \langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle \in \mathbf{D}\} \tag{7a}$$

$$r(\langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle) \triangleq \begin{cases} \langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \setminus \{e\} \rangle & \text{if } \mathbf{c}' \subseteq \mathbf{c}, \\ \langle \mathbf{c}' \rightsquigarrow \mathbf{e}' \rangle & \text{otherwise.} \end{cases} \tag{7b}$$

The dependencies resulting from removing the dependency $\langle \cdots_1 \rightsquigarrow e \rangle$ follow from the rewrite rule:

$$\langle \cdots_2 \rightsquigarrow \cdots_3 e \rangle \mapsto \langle \cdots_2 \rightsquigarrow \cdots_3 \rangle \text{ if } \cdots_2 \subseteq \cdots_1$$

3.3.2. 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.

3.3.3. 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 P**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 3.3. $\forall p_r, p_g \in$ P**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, $p_r \sqsubseteq p_g$.

3.4. 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 P**rop** as a lattice. As such, we need to define top and bottom elements of P**rop**, least upper bounds (or *joins*), and greatest lower bounds (*meets*).

The top element of P**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 $p \in$ P**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 P**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 $\bot \in$ P**rop** such that $\forall p, \bot \sqsubseteq p$.

The join of two elements $(\mathbf{C_1}, \mathsf{R}_1, \mathbf{D_1})$ and $(\mathbf{C_2}, \mathsf{R}_2, \mathbf{D_2})$ is equal, up to renaming of components, to $(\mathbf{C_1} \cap \mathbf{C_2}, \min\{\mathsf{R}_1, \mathsf{R}_2\}, \mathbf{D_1} \cup \mathbf{D_2})$. Joins can be computed by repeatedly applying merge to combine components, then applying add_dep to add dependencies as needed, then applying relax_rel to reduce reliabilities if necessary. The meet, likewise, is equal up to renaming of components to $(\mathbf{C_1} \cup \mathbf{C_2}, \max\{\mathsf{R}_1, \mathsf{R}_2\}, \mathbf{D_1} \cap \mathbf{D_2})$. It can be shown that the meet can be generalized to either element by appropriate application of merge, add_dep, and relax_rel.

4. 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 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., (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

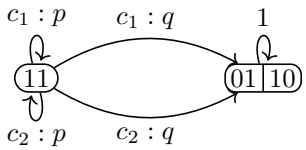


Figure 1: Two-Component Series System Markov Chain

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:

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

As an example, consider the system with two components in series where $\mathsf{R}(c_1) = \mathsf{R}(c_2) = p = 1 - q$. 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

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

The Markov chain this system follows is illustrated in Fig 1 where the transitions are labeled by the component taking them and the probability of being taken.

4.1. Abstraction and Concretization

To apply our formalization of refinement and generalization to MIS models, we need to connect our properties domain **Prop** 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 $(\mathbf{C}, \mathsf{R}, \mathbf{D}) \in \mathbf{Prop}$, for each $c_i \in \mathbf{C}$ let $p_i = 1 - q_i = \mathsf{R}(c_i)$ 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 $\boxed{1 \cdots 1}$. For each dependency $\langle c_i \rightsquigarrow \mathbf{e} \rangle \in \mathbf{D}$, insert a transition from $\boxed{1 \cdots 1}$ to $\boxed{1 \cdots 1}$ with probability p_i in T_i and a transition from $\boxed{1 \cdots 1}$ 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 $\boxed{\mathbf{s}}$ to $\boxed{\mathbf{s}}$ with probability p_i and from $\boxed{\mathbf{s}}$ to $\boxed{\mathbf{s} \setminus \mathbf{e}}$ with probability q_i into T_i. For each component $c_i \in \mathbf{f}$, insert a transition from $\boxed{\mathbf{s}}$ to $\boxed{\mathbf{s}}$ with probability 1 into T_i. Repeat this step until there are no more non-failed states to consider.

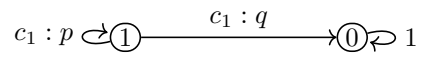
Concretizing properties from an MIS model proceeds in an analogous fashion. For each T_i create a component c_i and set $\mathsf{R}(c_i) = p_i$. For each c_i, first let $\mathbf{s}'$ be the set of components functional after c_i fails from the initial $\boxed{1 \cdots 1}$ state and add a dependency $\langle c_i \rightsquigarrow \mathbf{C} \setminus \mathbf{s}' \rangle$ to $\mathbf{D}$. Then consider all transitions in T_i from state $\boxed{\mathbf{s}}$ to state $\boxed{\mathbf{s}'}$ 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 $\langle \mathbf{C} \setminus \mathbf{s} \setminus \{c_i\} \rightsquigarrow \mathbf{f} \rangle$.

4.2. Examples

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

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

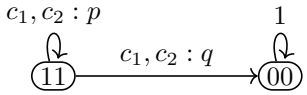
If we refine c_1's reliability to p by $s_1 = \mathsf{tighten\_rel}_{\top}[c_1, p]$, the resulting system has reliability $\mathsf{R}(\mathcal{S}) = p$.



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

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

which abstracts to the Markov chain:

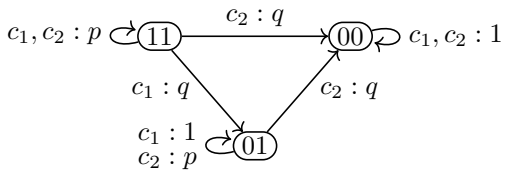


This gives $\mathsf{R}(\mathcal{S}) = p^2$ as we now take two steps through the Markov chain.

We can avoid adding excessive dependencies later by removing two, making c_1 independent: $s_3 = \mathsf{remove\_dep}_{s_2}[c_1 \rightsquigarrow c_2, \mathcal{S}]$.[3]

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

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



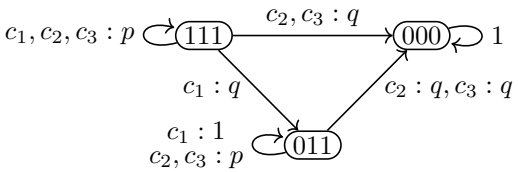
This gives $\mathsf{R}(\mathcal{S}) = p^2 + pq$—either both components remain functional, or c_1 fails and c_2 remains functional.

Next, we introduce c_3 by $s_4 = \mathsf{split}_{s_3}[c_2 \to c_2, c_3]$.

$$\begin{aligned} s_4 = (\{c_1, c_2, c_3\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = \mathsf{R}(c_3) = p, \{ \\ \langle c_1 \rightsquigarrow \emptyset\rangle, \langle c_2 \rightsquigarrow c_1, c_3, \mathcal{S}\rangle, \langle c_3 \rightsquigarrow c_1, c_2, \mathcal{S}\rangle \\ \langle c_1, c_2 \rightsquigarrow \mathcal{S}\rangle, \langle c_1, c_3 \rightsquigarrow c_2, \mathcal{S}\rangle, \langle c_2, c_3 \rightsquigarrow c_1, \mathcal{S}\rangle \\ \}) \end{aligned}$$

The Markov chain is similar to the one abstracted from s_3, but c_3 adds its own transition probabilities.

[3]This is equivalent to performing two remove_dep operations, one for the dependency on c_2 and one for $\mathcal{S}$.

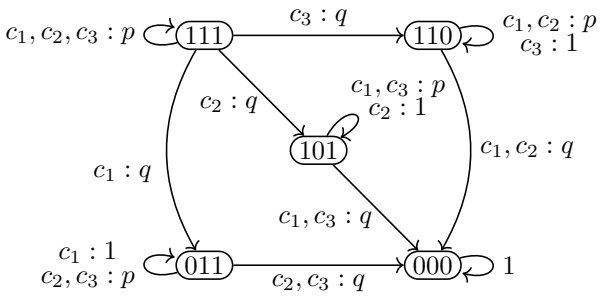


This gives $\mathsf{R}(\mathcal{S}) = p^3 + p^2q$—either all components remain functional, or c_1 fails and c_2 and c_3 remain functional.

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 $s_6 = \mathsf{remove\_dep}_{s_5}[c_3 \rightsquigarrow c_1, c_2, \mathcal{S}]$.

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

The abstracted Markov chain has two new states:

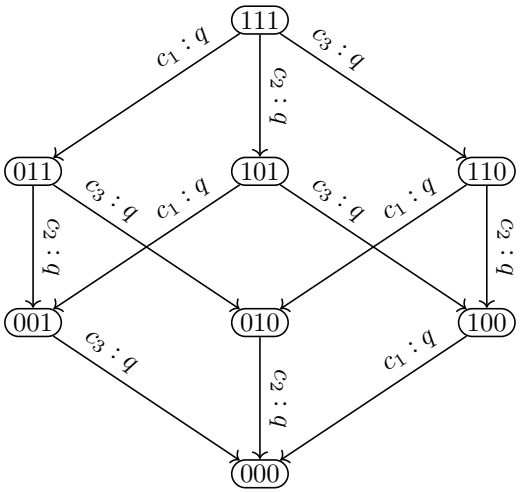


This gives $\mathsf{R}(\mathcal{S}) = p^3 + 3p^2q$—either all components remain functional, or only one fails.

5. Superstates and Non-deterministic Choice

One problem MIS models face is state space explosion: in a naïve model where every component is independent of the others, adding a new component doubles the number of states in the model. A solution to this problem, which does not require numerical approximation methods or otherwise reduce the accuracy of the computed result, is to allow one state to represent more than one configuration of "up" components. In the literature, these states are referred to as *superstates*; we have already seen examples of these in various failed states, such as the superstate $\boxed{01 \mid 10}$ in Figure 1. Thus far, we have modeled the "system failed" superstate in an ad-hoc fashion; we now turn to the issue of modeling arbitrary superstates in P**rop**.

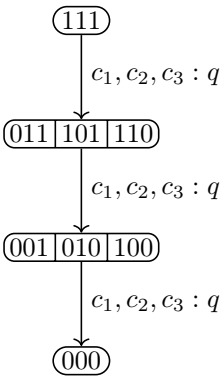
To motivate the following developments, we introduce an example of where superstates become useful. Suppose we have a system with three independent components in parallel. The system is functional as long as any single component remains functional. This reliability structure can be represented by the following Markov chain (for readability, we elide the self-loops for each state):



The corresponding element of P**rop** is:

$$
\begin{aligned}
s_{\text{state}} = (\{c_1, c_2, c_3\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = \mathsf{R}(c_3) = p, \{ \\
&\langle c_1 \rightsquigarrow \emptyset \rangle, \langle c_2 \rightsquigarrow \emptyset \rangle, \langle c_3 \rightsquigarrow \emptyset \rangle \\
&\langle c_1, c_2 \rightsquigarrow \emptyset \rangle, \langle c_1, c_3 \rightsquigarrow \emptyset \rangle, \langle c_2, c_3 \rightsquigarrow \emptyset \rangle \\
&\langle c_1, c_2, c_3 \rightsquigarrow \mathcal{S} \rangle \\
\})
\end{aligned}
$$

This model has $\mathsf{R}(\mathcal{S}) = p^3 + 3p^2q + 3pq^2$. This is the worst-case situation of 3 components leading to 2^3 states. However, the identity of any failed component is irrelevant to the computation; knowledge of the number of failed components suffices. We can reduce the state space of this model by creating superstates $\boxed{011|101|110}$ and $\boxed{001|010|110}$, representing one and two failed components respectively:



In this model, we still have $\mathsf{R}(\mathcal{S}) = p^3 + 3p^2q + 3pq^2$, but significantly fewer states and correspondingly smaller matrices. It is also worth noting that using superstates does not require that all components have the same reliability. Components are still considered individually when computing reliability; however, when defining transitions between superstates, we "forget" the specific state of a superstate which characterizes the system. If the system is in a superstate, we merely know that it is in one of the states of that superstate.

This notion of "forgetting" maps cleanly onto the concept of *non-deterministic choice*, denoted here with the $\oplus$ operator. Given two sets $\mathbf{C_1}, \mathbf{C_2}$, the value of $\mathbf{C_1} \oplus \mathbf{C_2}$ is one of the two specified sets, but it is unknown which set is chosen. By extending the notation of **Deps** to allow non-deterministic set choices to appear in the cause of a dependency, we can represent the Markov chain:

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

In the dependency $\langle c_1 \oplus c_2 \oplus c_3 \rightsquigarrow \emptyset \rangle$, we know only that one of c_1, c_2, or c_3 have failed, not which one. This allows us to capture the notion of "one component" having failed. With this example in hand, we can proceed to formally define and explore non-deterministic choice in **Prop**.

5.1. Non-deterministic choice of causes and effects

Non-deterministic choice plays a key role in abstraction and refinement for both software and systems [2]. In the process of deriving programs from specifications, it encodes the notion that one may pick arbitrarily among the programs which meet a particular specification; any aspect left unspecified by the specification is thus ambiguous. For example, given the specification "$f(x) = y$ such that $y * y = x$", we may implement $f(x)$ by either `sqrt(x)` or `-sqrt(x)`. Using the notation of non-deterministic choice, this specification can be represented by $f(x) =$ `sqrt(x)` $\oplus$ `-sqrt(x)`. When proving properties of $f(x)$, we use the *demonic choice principle*, which states that any statements true of a non-deterministic choice must hold regardless of which case of the choice is taken. Thus, we cannot argue that $f(x) \geq 0$, even though that property holds for one of the cases of the choice. Effectively, the demonic choice principle requires us to consider every case.

In this work, we will use non-deterministic choices to distinguish between sets, rather than specifications:

Definition 5.1. Given a set of sets $\mathcal{C} = \{\mathbf{C_1}, \mathbf{C_2}, \cdots\}$, the set $\bigoplus \mathcal{C}$ is one of the sets $\mathbf{C_1}$, $\mathbf{C_2}$, $\cdots$. The specific set is chosen arbitrarily. Denote $\bigoplus\{\mathbf{C_1}, \mathbf{C_2}\}$ by $\mathbf{C_1} \oplus \mathbf{C_2}$. $\bigoplus\{\mathbf{C_1}\} = \mathbf{C_1}$. $\bigoplus \emptyset$ has no value.

Note that non-deterministic choices can be "flattened"; that is,

$$\bigoplus \left\{ \mathbf{C_1}, \bigoplus \mathcal{C} \right\} = \bigoplus \{\mathbf{C_1}\} \cup \mathcal{C}.$$

It follows that $\oplus$ is commutative and associative by commutativity and associativity of $\cup$. Furthermore, $\mathbf{C} \oplus \mathbf{C} = \mathbf{C}$.

For convenience, we define a function to apply a set-valued function to each case of a non-deterministic choice:

$$\mathsf{map}_{\oplus}(f) \left(\bigoplus \mathcal{C} \right) = \bigoplus \{ f(\mathbf{C}) \mid \mathbf{C} \in \mathcal{C} \} \tag{10}$$

When introducing non-deterministic choice into the causes and effects of dependencies, we must be careful to observe the demonic choice principle. Namely, once we generalize two dependencies $\langle c_1 \rightsquigarrow \cdots \rangle$ and $\langle c_2 \rightsquigarrow \cdots \rangle$ into $\langle c_1 \oplus c_2 \rightsquigarrow \cdots \rangle$, erasing the distinction between the failure of c_1 and c_2, we must ensure that all other dependencies involving c_1 and c_2 also "forget" which of the two has failed. This may entail the definition of additional superstates. An analogous situation occurs when generalizing $\langle \cdots \rightsquigarrow c_1 \rangle$ and $\langle \cdots \rightsquigarrow c_2 \rangle$ into $\langle \cdots \rightsquigarrow c_1 \oplus c_2 \rangle$.

As an example, consider what happens in the previous example if we create the superstate (011|101|110) but leave the states (001), (010), and (100) as-is. This would correspond to the following invalid properties:

$$s_{\text{superstate}} = (\{c_1, c_2, c_3\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = \mathsf{R}(c_3) = p, \{ \\ \langle c_1 \oplus c_2 \oplus c_3 \rightsquigarrow \emptyset \rangle, \\ \langle c_1, c_2 \rightsquigarrow \emptyset \rangle, \langle c_1, c_3 \rightsquigarrow \emptyset \rangle, \langle c_2, c_3 \rightsquigarrow \emptyset \rangle \\ \langle c_1, c_2, c_3 \rightsquigarrow \mathcal{S} \rangle \\ \}).$$

We attempt to define three transitions out of $\boxed{011}\boxed{101}\boxed{110}$, one for each component. In the case that we consider the failure of c_1, it is unclear whether the dependency $\langle c_1, c_2 \rightsquigarrow \emptyset\rangle$ or $\langle c_1, c_3 \rightsquigarrow \emptyset\rangle$ applies, since we cannot determine whether it is c_2 or c_3 that has failed so far. The failure of c_1 in superstate $\boxed{011}\boxed{101}\boxed{110}$ cannot cause two transitions, so we must instead propagate forward the erasure of which component has failed and define a transition from $\boxed{011}\boxed{101}\boxed{110}$ to $\boxed{001}\boxed{010}$ when c_1 fails. Continuing this reasoning, considering the failure of c_2, we would define a transition from $\boxed{011}\boxed{101}\boxed{110}$ to $\boxed{010}\boxed{100}$. However, $\boxed{010}$ is already a member of the superstate $\boxed{001}\boxed{010}$, so we must instead expand this superstate to $\boxed{001}\boxed{010}\boxed{101}$. Therefore we arrive at the result of the previous example, with a (super)state for zero, one, two, and three failed components.

As another example, consider a four-component system with the dependency $\langle c_1 \rightsquigarrow c_2 \oplus c_3\rangle$, where it is ambiguous which component c_1 causes to fail. This corresponds to a transition from $\boxed{1111}$ to $\boxed{0011}\boxed{0101}$ caused by the failure of c_1. An invalid dependency for this system would be $\langle c_1, c_2, c_4 \rightsquigarrow \emptyset\rangle$, as this would only cover one case of the $\boxed{0011}\boxed{0101}$ superstate. Instead, we could write $\langle c_1, c_2, c_4 \oplus c_1, c_3, c_4 \rightsquigarrow \emptyset\rangle$, which would define a transition from $\boxed{0011}\boxed{0101}$ to $\boxed{0010}\boxed{0100}$ when c_4 fails.

5.2. Well-formedness properties with non-deterministic choice

We incorporate the requirements of the demonic choice principle into the well-formedness (WF) properties defined in Section 3.1.2. Since we have $\bigoplus\{\mathbf{C_1}\} = \mathbf{C_1}$, we can state all the WF properties in terms of non-deterministic choice over a set of causes or set of effects, generalizing the properties defined earlier.

For initiality, we allow the component to be part of a non-deterministic choice:

$$\forall c \in \mathbf{C}, \exists\langle\bigoplus\mathcal{C}' \rightsquigarrow \cdots\rangle \in \mathbf{D} \text{ where } \{c\} \in \mathcal{C}'. \tag{NDC-Initiality}$$

Termination comes with two requirements: not only must the system fail, but in any non-deterministic choice over failures, it must fail in every one:

$$\begin{aligned}&\exists\langle\cdots \rightsquigarrow \bigoplus\mathcal{E}\rangle \in \mathbf{D} \text{ where } \exists\mathbf{E} \in \mathcal{E}, \mathcal{S} \in \mathbf{E} \text{ and}\\ &\forall\langle\cdots \rightsquigarrow \bigoplus\mathcal{E}\rangle \in \mathbf{D} \text{ where } \exists\mathbf{E} \in \mathcal{E}, \mathcal{S} \in \mathbf{E}, \text{ then } \forall\mathbf{E} \in \mathcal{E}, \mathcal{S} \in \mathbf{E}.\end{aligned} \tag{NDC-Termination}$$

Monotonicity must hold for some particular cause and all resulting effects:

$$\begin{aligned}&\forall\langle\bigoplus\{\mathbf{C_1}, \mathbf{C_2}, \cdots\} \rightsquigarrow \bigoplus\{\mathbf{E_1}, \mathbf{E_2}, \cdots\}\rangle \in \mathbf{D}\\ &\forall\langle\bigoplus\{\mathbf{C_1} \cup \mathbf{C_3}, \mathbf{C_2} \cup \mathbf{C_4}, \cdots\} \rightsquigarrow \bigoplus\{\mathbf{E_3}, \mathbf{E_4}, \cdots\}\rangle \in \mathbf{D}\\ &\forall\mathbf{E} \in \{\mathbf{E_1}, \mathbf{E_2}, \cdots\}, \forall\mathbf{E}' \in \{\mathbf{E_3}, \mathbf{E_4}, \cdots\},\\ &\exists\mathbf{C}' \in \{\mathbf{C_3}, \mathbf{C_4}, \cdots\}, \mathbf{E} \subseteq \mathbf{E}' \cup \mathbf{C}'\end{aligned} \tag{NDC-Monotonicity}$$

Finally, we introduce a new property to ensure that the "forgetfulness" or "erasure" of demonic choice is preserved. Before we can state this property, we introduce a notation, overloading the combinatorial "binomial choice" operator to work on sets. Given a set of sets $\mathcal{C} = \{\mathbf{C_1}, \mathbf{C_2}, \cdots, \mathbf{C_n}\}$ and an integer $0 \leq x \leq n$, define $\binom{\mathcal{C}}{x}$ to be the set of unions of combinations of $\mathbf{C_1}, \cdots, \mathbf{C_n}$. Thus, for example, $\binom{\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}}{2} = \{\mathbf{C_1} \cup \mathbf{C_2}, \mathbf{C_1} \cup \mathbf{C_3}, \mathbf{C_2} \cup \mathbf{C_3}\}$. Also, $\binom{\mathcal{C}}{0} = \emptyset$ and $\binom{\mathcal{C}}{n} = \bigcup\mathcal{C}$.

$$\begin{aligned}&\forall\langle\mathbf{C_1} \oplus \cdots \oplus \mathbf{C_n} \rightsquigarrow \mathbf{E_1} \oplus \cdots \oplus \mathbf{E_m}\rangle \in \mathbf{D}\\ &\text{if } \exists\langle\bigoplus\mathcal{C}' \rightsquigarrow \cdots\rangle \in \mathbf{D} \text{ where } \mathbf{C_1} \subseteq \mathbf{C}'' \in \mathcal{C}',\\ &\text{then } \mathcal{C}' = \binom{\mathbf{C_1}, \cdots, \mathbf{C_n}}{x} \times \binom{\mathbf{E_1}, \cdots, \mathbf{E_m}}{y}\\ &\text{for some } 1 \leq x \leq n, 0 \leq y \leq m.\end{aligned} \tag{NDC-erasure}$$

This rule forbids, for instance, the existence of two dependencies $\langle c_1 \oplus c_2 \rightsquigarrow \cdots\rangle$ and $\langle c_1 \rightsquigarrow \cdots\rangle$ and likewise given the dependency $\langle c_1 \oplus c_2 \rightsquigarrow c_3 \oplus c_4\rangle$, any dependency with c_1, c_3 in the causes must be of the form $\langle c_1, c_3 \oplus c_1, c_4 \oplus c_2, c_3 \oplus c_2, c_4 \rightsquigarrow \cdots\rangle$. The associativity and commutativity of $\oplus$ means this rule applies to any set of causes and effects in a non-deterministic choice.

5.3. Generalizations and refinements for non-deterministic choice

Introducing a non-deterministic choice constitutes a loss of information about the model, so it is therefore a generalization. Likewise, removing a non-deterministic choice is a refinement.

Introducing a non-deterministic choice implicitly adds dependencies as needed to meet the WF properties. Erasing the distinction between failure of a set of components C_1 and C_2 is done by replacing each instance of either with a non-deterministic choice, then adding sufficient other terms to meet the combinatorial requirements of NDC-erasure:

$$\begin{aligned}&\mathsf{unify}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_,\_] : \mathcal{P}(\mathbf{C}) \to \mathcal{P}(\mathbf{C}) \to \mathbf{Prop}\\ &\mathsf{unify}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\mathbf{C_1},\mathbf{C_2}] \triangleq (\mathbf{C},\mathsf{R},\mathbf{D}')\end{aligned} \tag{11}$$

where

$$\mathbf{D}' \triangleq \left\{\langle \bigoplus U(\mathcal{C}') \rightsquigarrow \bigoplus U(\mathcal{E}')\rangle \;\middle|\; \langle \bigoplus \mathcal{C}' \rightsquigarrow \bigoplus \mathcal{E}'\rangle \in \mathbf{D}\right\} \tag{11a}$$

$$U(\mathcal{C}) \triangleq \begin{cases} A(\mathcal{C}) \times \binom{N(\mathbf{C_1}) \cup N(\mathbf{C_2})}{x(\mathcal{C})} & \text{if } \mathbf{C_1} \subseteq \mathbf{C}' \in \mathcal{C} \text{ or } \mathbf{C_2} \subseteq \mathbf{C}' \in \mathcal{C} \\ \mathcal{C} & \text{otherwise.}\end{cases} \tag{11b}$$

$$A(\mathcal{C}) \triangleq \{\mathbf{C}' \setminus N(\mathbf{C_1}) \setminus N(\mathbf{C_2}) \mid \mathbf{C}' \in \mathcal{C}\} \tag{11c}$$

$$N(\mathbf{C_1}) \triangleq \mathcal{C}' \text{ where } \langle \bigoplus \mathcal{C}' \rightsquigarrow \cdots\rangle \in \mathbf{D} \text{ and } \mathbf{C_1} \in \mathcal{C}' \tag{11d}$$

$$x(\mathcal{C}) \triangleq \max\left\{x' \;\middle|\; \exists \mathbf{B} \in \binom{N(\mathbf{C_1}) \cup N(\mathbf{C_2})}{x'}, \mathbf{B} \subseteq \mathbf{C}' \in \mathbf{C}\right\} \tag{11e}$$

Refining by splitting a non-deterministic choice splits the cases into separate dependencies as needed, first for the causes and then for the effects:

$$\begin{aligned}&\mathsf{separate}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\_ \oplus \_] : \mathcal{P}(\mathbf{C}) \to \mathcal{P}(\mathbf{C}) \to \mathbf{Prop}\\ &\mathsf{separate}_{(\mathbf{C},\mathsf{R},\mathbf{D})}[\mathbf{C_1} \oplus \mathbf{C_2}] \triangleq (\mathbf{C},\mathsf{R},\mathbf{D}'')\end{aligned} \tag{12}$$

where

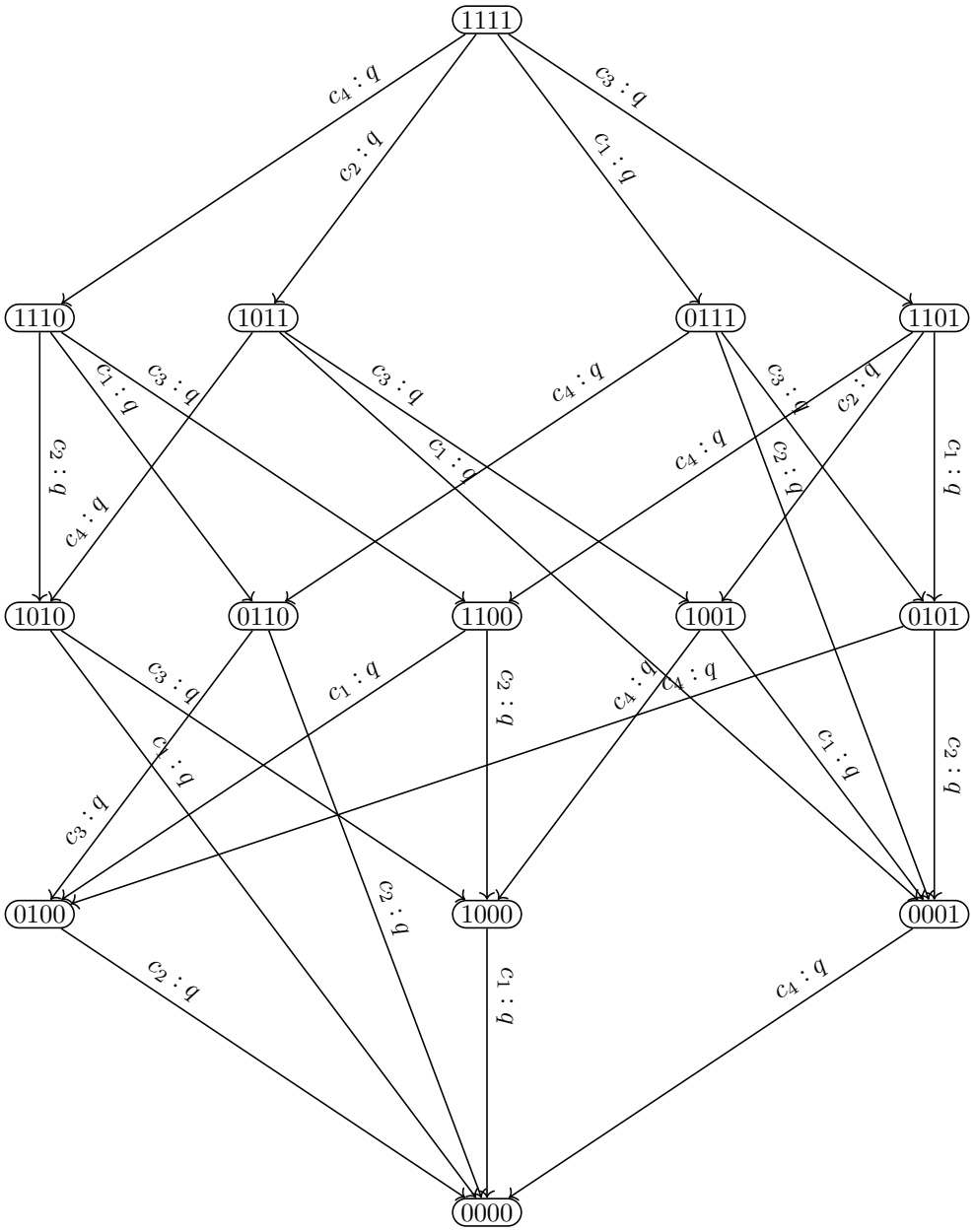
$$\mathbf{D}' \triangleq \left\{SC\left(\langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle\right) \;\middle|\; \langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle \in \mathbf{D}\right\} \tag{12a}$$

$$SC\left(\langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle\right) \triangleq \begin{cases} \left\{\begin{array}{l}\langle \bigoplus \{\mathbf{C}' \in \mathcal{C} \mid \mathbf{C_1} \subseteq \mathbf{C}' \text{ or } \mathbf{C_2} \nsubseteq \mathbf{C}'\} \rightsquigarrow \bigoplus \mathcal{E}\rangle \\ \langle \bigoplus \{\mathbf{C}' \in \mathcal{C} \mid \mathbf{C_2} \subseteq \mathbf{C}' \text{ or } \mathbf{C_1} \nsubseteq \mathbf{C}'\} \rightsquigarrow \bigoplus \mathcal{E}\rangle\end{array}\right\} & \begin{array}{l}\text{if } \mathbf{C_1} \subseteq \mathbf{C}' \in \mathcal{C} \\ \text{or } \mathbf{C_2} \subseteq \mathbf{C}' \in \mathcal{C}\end{array} \\ \langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle & \text{otherwise.}\end{cases} \tag{12b}$$

$$\mathbf{D}'' \triangleq \left\{SE\left(\langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle\right) \;\middle|\; \langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle \in \mathbf{D}'\right\} \tag{12c}$$

$$SE\left(\langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle\right) \triangleq \begin{cases} \left\{\begin{array}{l}\langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \{\mathbf{E}' \in \mathcal{E} \mid \mathbf{C_1} \subseteq \mathbf{E}' \text{ or } \mathbf{C_2} \nsubseteq \mathbf{E}'\}\rangle \\ \langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \{\mathbf{E}' \in \mathcal{E} \mid \mathbf{C_2} \subseteq \mathbf{E}' \text{ or } \mathbf{C_1} \nsubseteq \mathbf{E}'\}\rangle\end{array}\right\} & \begin{array}{l}\text{if } \mathbf{C_1} \subseteq \mathbf{E}' \in \mathcal{E} \\ \text{or } \mathbf{C_2} \subseteq \mathbf{E}' \in \mathcal{E}\end{array} \\ \langle \bigoplus \mathcal{C} \rightsquigarrow \bigoplus \mathcal{E}\rangle & \text{otherwise.}\end{cases} \tag{12d}$$

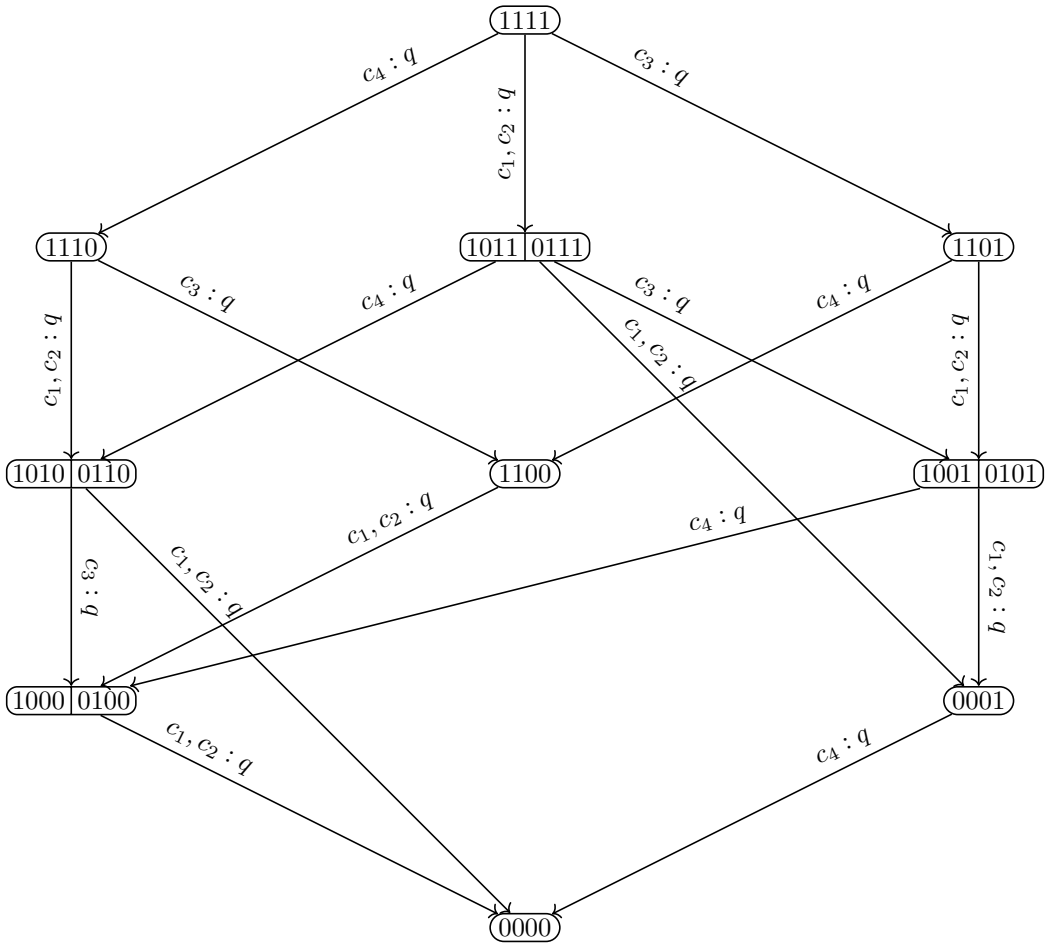
Consider the following example of generalization by introduction of non-deterministic choices. We begin with a system with all components in parallel and independent except that the failure of c_1 and c_2 causes the failure of c_3. The MIS model for this system contains the following Markov chain (again, with self-loops elided for readability):



With the corresponding abridged properties:

$$
\begin{aligned}
s_1 = (\{c_1, c_2, c_3, c_4\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = \mathsf{R}(c_3) = \mathsf{R}(c_4) = p, \{ \\
\langle c_1 \rightsquigarrow \emptyset \rangle, \langle c_2 \rightsquigarrow \emptyset \rangle, \langle c_3 \rightsquigarrow \emptyset \rangle, \langle c_4 \rightsquigarrow \emptyset \rangle, \\
\langle c_1, c_2 \rightsquigarrow c_3 \rangle, \langle c_1, c_2, c_3, c_4 \rightsquigarrow \mathcal{S} \rangle \\
\}).
\end{aligned}
$$

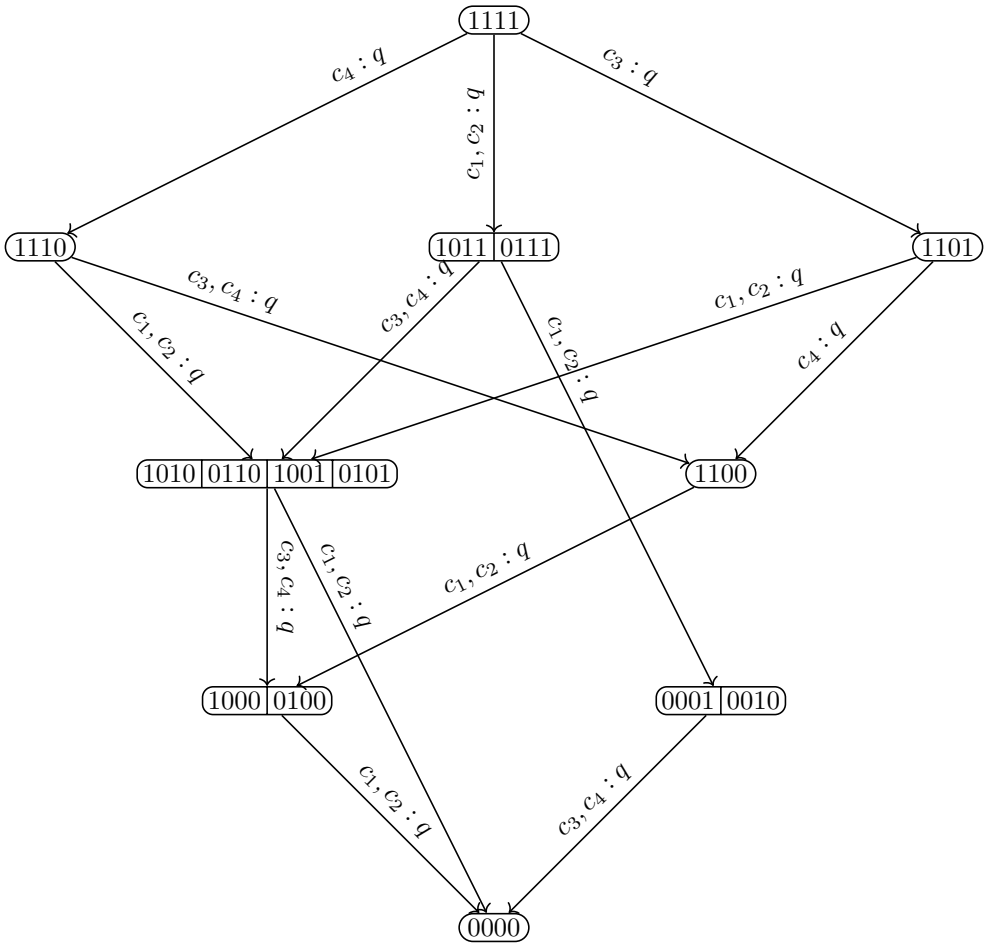
Suppose we generalize the system via $s_2 = \mathsf{unify}_{s_1}[c_1, c_2]$. Transitions caused by c_1 and c_2 consequently align:



With the corresponding abridged properties:

$$
\begin{aligned}
s_2 = (\{c_1, c_2, c_3, c_4\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = \mathsf{R}(c_3) = \mathsf{R}(c_4) = p, \{ \\
&\langle c_1 \oplus c_2 \rightsquigarrow \emptyset \rangle, \langle c_3 \rightsquigarrow \emptyset \rangle, \langle c_4 \rightsquigarrow \emptyset \rangle, \\
&\langle c_1, c_2 \rightsquigarrow c_3 \rangle, \langle c_1, c_2, c_3, c_4 \rightsquigarrow \mathcal{S} \rangle \\
\}).
\end{aligned}
$$

We can see that states with components c_1 and c_2 are combined, and c_1 and c_2 share all transitions. However, the effect of unify is more complex than merge; consider a further generalization of $s_3 = \mathsf{unify}_{s_2}[c_1, c_3, c_1, c_4]$ followed by $s_5 = \mathsf{unify}_{s_4}[c_2, c_3, c_2, c_4]$:



With the corresponding abridged properties:

$$
\begin{aligned}
s_5 = (\{c_1, c_2, c_3, c_4\}, \mathsf{R}(c_1) = \mathsf{R}(c_2) = \mathsf{R}(c_3) = \mathsf{R}(c_4) = p, \{ \\
&\langle c_1 \oplus c_2 \rightsquigarrow \emptyset \rangle, \langle c_3 \rightsquigarrow \emptyset \rangle, \langle c_4 \rightsquigarrow \emptyset \rangle, \\
&\langle c_1, c_2 \rightsquigarrow c_3 \oplus c_4 \rangle, \langle c_1, c_2, c_3 \oplus c_1, c_2, c_4 \rightsquigarrow \emptyset \rangle, \\
&\langle c_1, c_3 \oplus c_2, c_3 \oplus c_1, c_4 \oplus c_2, c_4 \rightsquigarrow \emptyset \rangle, \\
&\langle c_1, c_2, c_3, c_4 \rightsquigarrow \mathcal{S} \rangle \\
\}).
\end{aligned}
$$

Of note here: c_3 and c_4 stay independent (states (1101) and (1110), respectively) as long as both c_1 and c_2 are functional. However, once one of c_1 or c_2 fails, states containing a failed c_3 or c_4 are merged into superstates. Thus, the behavior of this system is beyond something expressible with merge alone. Finally, a note on refinement: $\mathsf{separate}_{s_5}[c_3 \oplus c_4]$ would result in both dependencies $\langle c_1, c_2 \rightsquigarrow c_3 \rangle$ and $\langle c_1, c_2 \rightsquigarrow c_4 \rangle$, as the unify generalization erased the knowledge of which of c_3 or c_4 fails due to the failure of c_1 and c_2. The initial behavior can be refined from this model via calls to remove_dep.

6. 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 [2, 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. Specifications describe "what" a program should do; as a specification is refined, it begins to also describe "how" a program meets that specification. For example, $\sqrt{x} * \sqrt{x} = x$ is a specification for a square root function; this specification can be refined further until the programmer arrives at an implementation of various root-finding techniques. Thus, programs are specifications that are also 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.

7. 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.

References

[1] N. Jarus, S. Sedigh Sarvestani, A. R. Hurson, Formalizing cyber–physical system model transformation via abstract interpretation, in: 19th IEEE International Symposium on High Assurance Systems Engineering (HASE), 2019, pp. 107–114. doi:10.1109/HASE.2019.00025.

[2] A. McIver, C. Morgan, Abstraction, Refinement, and Proof for Probabilistic Systems, Springer monographs in computer science, Springer Science + Business Media Inc., 2005.

[3] M. N. Albasrawi, N. Jarus, K. A. Joshi, S. Sedigh Sarvestani, Analysis of reliability and resilience for smart grids, in: 38th Annual IEEE Computer Software and Applications Conference, 2014, pp. 529–534. doi:10.1109/COMPSAC.2014.75.

[4] K. Marashi, S. Sedigh Sarvestani, A. R. Hurson, Consideration of cyber-physical interdependencies in reliability modeling of smart grids, IEEE Trans. on Sustainable Computing 3 (2) (2018) 73–83. doi:10.1109/TSUSC.2017.2757911.

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

[6] S. Gulwani, O. Polozov, R. Singh, Program synthesis, Foundations and Trends in Programming Languages 4 (1-2) (2017) 1–119. doi:10.1561/2500000010.
URL https://www.nowpublishers.com/article/Details/PGL-010

[7] B. Caillaud, B. Delahaye, K. G. Larsen, A. Legay, M. L. Pedersen, A. Wąsowski, Compositional design methodology with constraint Markov chains, in: 7th International Conference on the Quantitative Evaluation of Systems, 2010, pp. 123–132. doi:10.1109/QEST.2010.23.

[8] B. Delahaye, K. G. Larsen, A. Legay, M. L. Pedersen, A. Wąsowski, Consistency and refinement for interval Markov chains, Journal of Logic and Algebraic Programming 81 (3) (2012) 209–226. doi:10.1016/j.jlap.2011.10.003.
URL https://www.sciencedirect.com/science/article/pii/S1567832611000956

[9] R. Chadha, M. Viswanathan, A counterexample-guided abstraction-refinement framework for Markov decision processes, ACM Trans. on Computational Logic 12 (1) (2010) 1:1–1:49. doi:10.1145/1838552.1838553.
URL http://doi.acm.org/10.1145/1838552.1838553

[10] M. Kattenbelt, M. Kwiatkowska, G. Norman, D. Parker, A game-based abstraction-refinement framework for Markov decision processes, Formal Methods in System Design 36 (3) (2010) 246–280. doi:10.1007/s10703-010-0097-6.
URL http://link.springer.com/10.1007/s10703-010-0097-6