An Ontology Framework for Generating Discrete-Event Stochastic Models

Ken Keefe[1], Brett Feddersen[1], Michael Rausch[1],
Ronald Wright[1], William H. Sanders[1]

Information Trust Institute,
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{kjkeefe, bfeddrsn, mjrausc2, wright53, whs}@illinois.edu

Abstract. Discrete-event stochastic models are a widely used approach for studying the behavior of systems that have not been implemented or that it would be too costly to examine directly. Valuable analysis depends on carefully constructed, well-founded models, which are very difficult for humans to create. To address this problem, we propose a framework for generating detailed, low-level models from high-level, block-diagram-style graphical models. Our approach uses extensible, collaborative ontology libraries that contain information about the types of components in a system, the types of relationships that connect those components, and fragments of low-level models that can be constructed together based on the definition of a high-level system model. This framework has been implemented and used in several case studies. We describe the framework and how model generation works by examining its use to generate complex ADversary VIew Security Evaluation (ADVISE) models.

Keywords: Ontology · Model Generation · Executable Models · Discrete-Event Simulation

1 Introduction

Modeling is a critical element of system design and analysis. Through a formal, mathematical description of how the world works, a model provides a representation of a system that can be explored, evaluated, and tested in a scientific, repeatable way. Discrete-event stochastic models provide especially useful views of real-world system designs [1].

Creating accurate, complete stochastic models is very challenging. Typically, human modelers must make decisions about what details and behaviors are important to include in each system model they create. Tools that automatically explore an existing system to construct a complete model can help with the decision-making process, but function only on systems that have already been implemented [2, 3].

This paper's contribution are (1) a formal description of an ontology framework that automatically generates detailed, discrete-event, stochastic models from high-level system design primitives and (2) an implementation of the framework that generates ADVISE security models in the Möbius tool. Figure 1 shows how the pieces of our framework interact. Fine-grained models are generated from (1) an ontology of component and relationship types, as well as model fragment generation rules, and (2) a simple, formal description of a system that

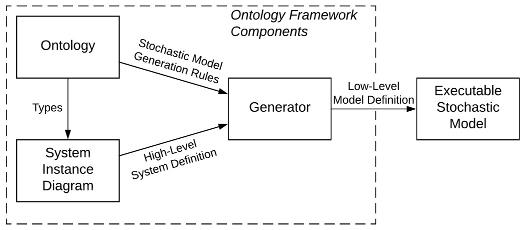
Ontology Framework Components
Ontology
Stochastic Model Generation Rules
Types
Generator
Low-Level Model Definition
Executable Stochastic Model
System Instance Diagram
High-Level System Definition

Fig. 1: An overview of the three parts of the ontology framework and how they work together to automatically generate executable, stochastic models.

uses instances of the types from the ontology. We define the notion of a *system instance diagram (SID)* to be a graphical model of component instances and relationship instance arcs that connect component instances. From the ontology and SID, we use a custom semantic reasoner to automatically construct the low-level model.

We argue that our approach to model generation is inherently more useful, less prone to error, and more complete than conventional approaches. Once an ontology has been defined for certain classes of systems, it is relatively quick and easy to define a SID and generate rich and complex stochastic models. We can now create in hours a low-level model that previously would have taken weeks of painstaking effort. By separating the ontology and SID, ontologies can be refined over time by a community of experts and reused over many system studies. Instead of relying on a human modeler's diligence to be sure that all aspects of a system have been thoroughly and correctly modeled at a detailed level, one can count on the generation process to apply every stochastic model construction rule consistently and accurately. Further, making significant changes to a system design or comparing multiple designs is much faster at the SID level than it is at the traditional detailed stochastic model level.

Our ontology framework has been implemented and tested on a cybersecurity modeling formalism called ADversary VIew Security Evaluation (ADVISE) [4]. The generated ADVISE models are used by the Möbius tool [5] to evaluate custom quantitative metrics through discrete-event simulation [6].

This paper is organized as follows. Section 2 describes our ontology framework. Section 3 explains how the ontology framework is used to create ADVISE models. In Section 4, we provide an overview of two case studies that use our implementation of the ontology framework. In Section 5, we describe prior work that has similarities to our work. We conclude in Section 6.

2 Ontology Framework

An *ontology* is a formal specification of types, attributes of instances of those types, and relationships that can connect instances of those types [7]. The types, attributes, and relationships all have semantic interpretations that the ontology seeks to organize and formalize. A *knowledge base* is associated with an ontology and contains instances of the types from the ontology, values for the attributes associated with each instance's type, and relationships that connect two instances.

We define a *Möbius ontology* to contain a set of component, relationship, and model fragment classes. Each component class may have one or more attributes that can be any single basic data type. Relationship classes in a Möbius ontology define domain and range component class restrictions and provide a type for instances of semantic relationships that connect two component instances. Component and relationship classes have a disjoint inheritance structure that indicates that one type is a more specialized type of another. Model fragment classes are formal definitions of a piece of a detailed, low-level model. A model fragment is tied to a component class by the special *dependent* relationship.

The knowledge base in our framework is called a *system instance diagram (SID)*. When a component class is instantiated in the SID, each model fragment class that is a dependent of the instance's class is also instantiated. A model fragment class includes instructions for connecting the instances to other model fragment instances.

2.1 Formal Definition

Definitions 1 and 2 provide a formal definition of a Möbius ontology and a SID. A detailed discussion of these definitions is provided in subsequent sections.

Definition 1. *A* Möbius ontology *can be defined as the following tuple:*

$$< C, A, R, AF, SF, S, U, Type, Inherits, Attrib, Domain, Range, Dependents >,$$

where

- *C is the set of component classes.*
- *A is the set of attributes.*
- *R is the set of relationship classes.*
- *AF is the set of action model fragment classes.*
- *SF is the set of state variable model fragment classes.*
- *$S \subseteq R$ is the set of relationship classes that are symmetric.*
- *$U \subseteq (AF \cup SF)$ is the set that contains all universal fragment classes.*
- *Type: $A \cup SF \rightarrow T$ provides a type for every attribute or state variable model fragment. T is defined in Section 3.1.1 of [8].*
- *Inherits: $C \cup R \rightarrow \mathcal{P}(C) \cup \mathcal{P}(R)$ identifies the set of classes from which the given component class or relationship class inherits.*
- *Attrib: $C \rightarrow \mathcal{P}(A)$ identifies the set of attributes of a component class.*
- *Domain: $R \rightarrow \mathcal{P}(C)$ identifies the set of source types for the relationship.*
- *Range: $R \rightarrow \mathcal{P}(C)$ identifies the set of target types for the relationship.*
- *Dependents: $C \rightarrow \mathcal{P}(AF \cup SF)$ identifies the set of model fragments that are dependents of the given component type.*

Definition 2. *A* system instance diagram (SID) *can be defined as the following tuple:*

$$< o, CI, RI, AFI, SFI, Class, Value, Arc, Dependents >,$$

where

- *o is the SID's ontology.*
- *CI is the set of component instances.*

- *RI is the set of relationship instances.*
- *AFI is the set of action model fragment instances.*
- *SFI is the set of state variable model fragment instances.*
- *FI is the set of model fragment instances ($AFI \cup SFI$).*
- *Class: $CI \cup RI \cup AFI \cup SFI \rightarrow C \in o \cup R \in o \cup AF \in o \cup SF \in o$ provides the class of the instance.*
- *Value: $CI \times A \rightarrow V$ provides the value of a component instance's attribute, and V is defined such that*

$$\forall ci \in CI, \forall a \in Attrib(Class(ci)), \forall v \in V,$$
$$Value(ci, a) = v \Rightarrow type(v) = Type(a)$$

  where type *and* V *are defined in Section III.A.2 of [8].*
- *Arc: $RI \rightarrow CI \times CI$ provides the source and target component instances of a relationship instance. $Arc_s(ri)$ and $Arc_t(ri)$ denote the source and target component instance of the given relationship instance, respectively.*
- *DependentsI: $CI \rightarrow \mathcal{P}(FI)$ provides the set of model fragment instances that are dependents of the given component instance.*

2.2 Components, Attributes, and Relationships

Our ontology framework defines the notions of components, attributes, and relationships. *Components* are atomic pieces of a system, either physical or conceptual. For example, a component can be a network, a firewall, a human user, a password policy, or a collection of data in a database.

A *relationship* is a semantic connection between two component instances. Class constraints are applied to the *domain* and *range* of the relationship in order to limit instance types that are allowable as the source or target, respectively. For example, a *connectedTo* relationship could connect a firewall component to a network component and capture the information that the source component has an Ethernet connection to the target component. Ontology relationships are directed unless the relationship is in S, indicating that it is symmetric.

An *attribute* is a semantic reference between a component instance and a basic data type. Basic data types are like those described in Section III.A.1 of [8]. They are used to describe qualities of a component, such as the operational state of an IDS service on a network component, or whether malware has infected a network component.

2.3 Class Inheritance

In our ontology framework, classes may inherit features from other classes. Definition 3 defines class inheritance and its consequences. Inheritance plays a pivotal role in the definition of components, relationships, and attributes. Components and relationships have directed, acyclical hierarchies that define the inheritance structure among these classes. The inheritance structures for components and relationships are necessarily disjoint.

Definition 3. *A* ***child*** *class* ***inherits*** *from its* ***parent*** *class(es) and is a specialized subtype of its parent(s). The following properties hold.*

1. $\forall a, b \in C, b \in Inherits(a) \Rightarrow Attrib(b) \subseteq Attrib(a)$

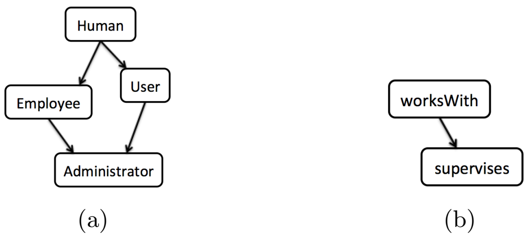

Fig. 2: Example component (a) and relationship (b) class inheritance hierarchies.

2. $\forall a, b \in C, \forall r \in R, b \in \mathit{Inherits}(a) \wedge b \in \mathit{Domain}(r) \Rightarrow a \in \mathit{Domain}(r)$
3. $\forall a, b \in C, \forall r \in R, b \in \mathit{Inherits}(a) \wedge b \in \mathit{Range}(r) \Rightarrow a \in \mathit{Range}(r)$
4. $\forall q, r \in R, \forall a \in C, r \in \mathit{Inherits}(q) \wedge b \in \mathit{Domain}(r) \Rightarrow a \in \mathit{Domain}(q)$
5. $\forall q, r \in R, \forall a \in C, r \in \mathit{Inherits}(q) \wedge b \in \mathit{Range}(r) \Rightarrow a \in \mathit{Range}(q)$

When a child component class inherits from a parent component class, every attribute ascribed to the parent will also be ascribed to the child (Def. 3.1). For example, in Figure 2a, `Human` could have the attribute `Name`, and `User` could have the attribute `Username`. Given the rules of inheritance, an instance of type `User` would have two attributes, `Name` and `Username`.

A relationship that defines its range class constraints as the set {`User`} really has a range class constraints set of {`User`, `Administrator`} (Def. 3.2–3.3). Relationship classes can also have an inheritance structure. For example, in Figure 2b, two relationship classes are defined in the example ontology; both have domain and range class constraints sets that are equal to {`Employee`}. `worksWith` means that two people work for the same employer, while `supervises` means that the source person oversees the work of the target person. It makes sense for inheritance to be defined between these two relationship classes, because if a person supervises another person at work, then both people work for the same employer (Def. 3.4–3.5).

2.4 Model Fragments

The primary objective of our framework is to automatically create detailed low-level models from high-level system instance diagrams (SIDs). Components, attributes, and relationships are the building blocks of the high-level model, and model fragments describe how the lower-level models will be constructed. The connection between the two levels is done through an explicit mapping that uses the *Dependents* (Def. 1) function.

Model fragments are classified into two types: state variable fragments and action fragments. *State variable fragments* describe variables that store a part of the low-level model's state, and *action fragments* describe low-level model actions that transition the model's state. Fragments in the ontology are not themselves state variables and actions, but rather are information about how to create sets of state variables and actions that will be included in the generated low-level model. More precisely, model fragments are classes in the ontology, and the state variables and actions that they create are instances in the knowledge base (SID). Unlike component and relationship instances, model fragment instances do not appear visually in a SID.

Universal Fragments If a model fragment is in U, then it is considered to be an element that should exist in every low-level model, regardless of the composition of components and relationship instances. Since these model fragment instances are always in the SID, they are not dependent on any component instance. For example, it may be desirable for a stochastic model dealing with real-world scheduling of people to track the day of the week. To enable that, one could add a `DayOfTheWeek` universal state variable fragment with the type `char`. Regardless of how many users and which jobs are added to the SID, there will always be a single `DayOfTheWeek` state variable to track the day of the week in the generated stochastic model.

Variable Macros and Path Expressions A variable macro describes an attachment between an action model fragment and a state variable model fragment. Our generator uses these definitions to connect together the generated model fragment instances. Depending on the semantics of the low-level model formalism, these variable macros can impact the model in a variety of ways. For example, in the ADVISE formalism, variable macros cause AEG arcs to be defined between state variable elements and attack steps. A variable macro includes a SID path expression tree to identify which state variables should be included in the variable macro's set.

Path expressions are used to explore the SID and collect a set of model fragment instances. Path expressions are a hierarchy of path constraints, which are often parameterized with a class constraint that is used to filter path steps according to the component's, relationship's, or model fragment's class. There are several kinds of individual path constraint nodes:

- $Dependent(t)$ explores the dependent model fragment instances with class t of the current component instance.
- $Component_{source}(t)$ explores the source component with the class t of the current relationship instance.
- $Component_{target}(t)$ explores the target component with the class t of the current relationship instance.
- $Relationship_{source}(t)$ explores the relationship instances with class t that have the current component instance as the source component.
- $Relationship_{target}(t)$ explores the relationship instances with class t that have the current component instance as the target component.
- *And* evaluates all child path constraints and returns the intersection.
- *Or* evaluates all child path constraints and returns the union.
- $Universal(t)$ returns the universal model fragments with class t.

Algorithm 1 defines how variable macro path expression trees are evaluated. The cases that explore $Component_{target}(t)$ and $Relationship_{target}(t)$ are omitted here to save space, but they are very similar to their *source* counterparts.

2.5 Model Generation Algorithm

The low-level stochastic model is generated in two phases. The first phase happens synchronously as components are added to or removed from the system instance diagram. Any state variable or action model fragments that are dependents of a component are created and added when a component is added to the

Algorithm 1 Path Expression Evaluation

```
 1: Given path expression tree $pet$, and component or relationship instance $cur$
 2: function FSV($pet, cur$)
 3:     $ret \leftarrow \emptyset$
 4:     $pc \leftarrow rootOf(pet)$
 5:     switch $pc$ do
 6:         case $Dependent(t)$
 7:             for all $d \in Dependents(cur)$ do
 8:                 if $Class(d) = t$ then
 9:                     $ret \leftarrow ret \cup \{d\}$
10:         case $Component_{source}(t)$
11:             if $Class(Arc_s(cur)) = t$ then
12:                 $ret \leftarrow ret \cup \mathrm{FSV}(childOf(pc, 0), Arc_s(cur))$
13:         case $Relationship_{source}(t)$
14:             for all $r \in RI$ do
15:                 if $Arc_s(r) = cur$ & $Class(r) = t$ then
16:                     $ret \leftarrow ret \cup \mathrm{FSV}(childOf(pc, 0), r)$
17:         case $And$
18:             $ret \leftarrow \mathbb{U}$
19:             for $i \leftarrow 0 \ldots n$ do
20:                 $ret \leftarrow ret \cap \mathrm{FSV}(childOf(pc, i), cur)$
21:         case $Or$
22:             for $i \leftarrow 0 \ldots n$ do
23:                 $ret \leftarrow ret \cup \mathrm{FSV}(childOf(pc, i), cur)$
24:         case $Universal(t)$
25:             for all $u \in U$ do
26:                 if Class(u) = t then
27:                     $ret \leftarrow ret \cup \{u\}$
28:     return ret
```

SID. Likewise, the model fragment instances are removed from the SID when the component instance on which they depend is removed.

The second phase is performed once the SID has been defined. Algorithm 2 describes the execution of this phase. Every model fragment instance in the SID ($AFI \cup SFI$) is looped through. The component instance ci is stored for each model fragment instance fi. For each variable macro associated with the model fragment instance, the following two steps are performed: (1) find the set of variables by exploring the path expression defined in the variable macro by using the component instance as the starting point, (2) make formalism-specific changes to the model. Once Algorithm 2 has been executed, a well-formed low-level model has been generated.

Complexity The factors that impact the time and space complexity of the first phase of the model generation algorithm include the number of model fragments that are dependents of each component class (d_c) in the ontology and the number of component instances of each component class (i_c) in the SID, giving a time and space complexity of $\mathcal{O}(d_c i_c)$. However, phase 1 can be done synchronously during the creation of a SID, so, in practice, it is $\mathcal{O}(1)$.

Algorithm 2 Model Generation Algorithm

```
1: for all $fi \in AFI \cup SFI$ do
2:     $ci \leftarrow c \in CI \mid fi \in Dependents(c)$
3:     for all $vm \in VariableMacros(fi)$ do
4:         $vs \leftarrow FSV(PathExp(vm), ci)$
5:         $MakeFormalismSpecificChanges(vm, vs, fi)$
```

For the second phase of the generation algorithm, the time complexity is $\mathcal{O}(vp)$, where v is the number of variable macros that must be evaluated across the entire SID and p is the maximum number of nodes in the variable macros' path expressions. The space complexity is entirely dependent on the formalism-specific $MakeFormalismSpecificChanges()$ function. It is typical to expect a formalism to add a single arc for each variable macro replacement, so the complexity would be $\mathcal{O}(v)$.

Scalability To describe the scalability of our approach in more concrete terms, we will mention an example ontology and system that are pending publication and are the largest (that we know of) to date. The system domain is power distribution systems. The ontology contains about 100 component types, 20 relationship types, and approximately 200 model fragment classes. The system instance diagram contains about 50 component instances and 500 relationship instances. On a modern laptop, this system takes approximately 15 minutes to generate the 500 state variables, 200 actions (with 2,000 variable macro resolutions), and 1,400 arcs.

3 Generating ADVISE Security Models

To test the concepts described in Section 2, we have implemented our ontology framework and are able to generate Möbius ADVISE atomic models from SIDs. Möbius provides an interface through which multiple modeling formalisms can be used to define executable, discrete-event systems that can be combined together and evaluated by an array of solution methods implemented in the Möbius tool.

In the remainder of Section 3, we will give a brief overview of the ADVISE formalism, discuss the extensions to the generator that were implemented to accommodate ADVISE model generation, and go through a detailed example that should illuminate the formal definitions given in Section 2.

3.1 The ADVISE Atomic Model Formalism

ADversary VIew Security Evaluation (ADVISE) [4] [9] [10] [11] is a security modeling formalism that models a system from the perspective of an adversary attempting to compromise the system and achieve custom-defined goals. An ADVISE model consists of an executable version of an attack tree called the *attack execution graph (AEG)* and an adversary profile that describes the initial assets of the adversary.

Figure 3 shows a small example fragment of an attack execution graph. The yellow rectangles are the attack steps that change the model state, and every

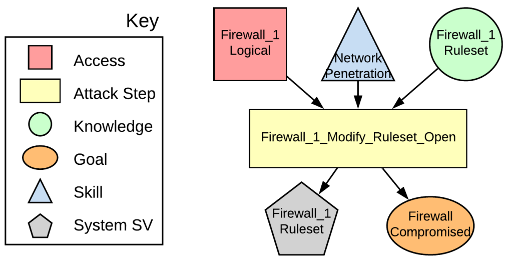

Fig. 3: An example AEG fragment.

other shape is a state variable of some kind. Arcs connecting state variables to or from attack steps indicate that the state variable is used in the execution of the attack step.

Several case studies have used ADVISE to model potential attack scenarios against critical infrastructure systems [11] [12] [13]. However, these models were created through extensive manual effort of human modelers.

3.2 Implementing the ADVISE Generator Extensions

Access, goals, knowledge, skills, and system state variables (SSV) are all state variable elements of an ADVISE model. We defined five model fragment classes to serve as base classes for any ADVISE state variable model fragments defined in the ontology.

When formalism-specific extensions are added to the Möbius ontology framework, model fragments that store the component instance attribute values must be defined. In the ADVISE extension, component instance attributes are generated as system state variables, and the type of the attribute matches the type of the generated system state variable.

ADVISE models have only one kind of Möbius action, the attack step-outcome pair. These pairs are grouped together by the attack step to form Möbius groups (see [14]). The action fragments in the ontology framework work in a similar way, so that many actions can be joined together in a group. When a human ontology developer defines an attack step and its outcomes, he or she does so by creating child classes of the base attack step-outcome class. The same code expressions necessary to define attack steps and outcomes in ADVISE atomic models are also required in the definition of the attack step-outcome model fragment classes.

3.3 The Two Nets Example

We now present an example ontology, system instance diagram, and generated ADVISE model to illuminate the discussion presented so far. Consider a very simple ontology that is used to describe networks and the firewalls that bridge them. C, A, and R are defined:

- **Firewall:** Component that bridges traffic between connected networks.
- **Network:** Component that has a collection of hosts connected to it.

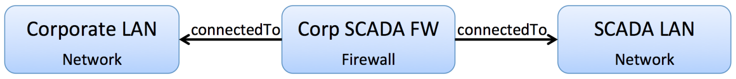

Fig. 4: The system instance diagram for the Two Nets example.

  - **isIDSOperational:** Attribute that stores the operational state of a network's intrusion detection service.
- **connectedTo:** Indicates that a firewall is connected to a network.
  - **Domain: {Firewall}**. **Range: {Network}**.

This ontology can define any organization of networks and firewalls. Figure 4 shows the Two Nets example's SID. It has two instances of `Network` (*Corporate LAN* and *SCADA LAN*) and one instance of `Firewall` (*Corp SCADA FW*).

In order to generate an ADVISE model, ADVISE model fragment classes must be defined in the ontology. The model fragment classes used in the Two Nets example are outlined below. Fragments below a component class are dependents of the component class.

- **Firewall**
  - **Defeat Firewall Attack Step:** This attack step defeats a firewall by using a brute-force approach. Upon successful completion, access is gained to any network connected to the firewall.
- **Network**
  - **IDS Operational SSV:** This SSV stores whether the network's IDS is currently operational. This model fragment's value is inferred from the *isIDSOperational* component attribute value.
  - **Install Malware on Network Attack Step:** This attack step installs malware on the network (on one of the network's hosts).
  - **Malware Installed SSV:** This SSV stores whether malware is installed.
  - **Network Access:** This access stores whether the adversary has some kind of access to the network.
- ***Universal***
  - **Brute Force Skill:** This skill represents the ability of an adversary to effectively perform brute-force attacks.

In Figure 5, the system instance diagram is presented in the left column. The center column shows the state variable model fragment instances that are generated for each of the associated component instances. The center column also shows the universal state variable model fragment instance that is generated for all ADVISE models that use this ontology. The right column shows the action model fragment instances that are generated for each component instance. The center and right columns combined show what the attack execution graph generation looks like after phase one of the generation process. In phase two, variable macro path expressions are evaluated, and AEG arcs are created. Precondition elements will create AEG arcs that target attack steps. Affected elements will create AEG arcs that are sourced from attack steps.

The outline below shows the structure of the variable macro path expression for the *Defeat Firewall Attack Step*. The *Install Malware on Network Attack Step* is omitted here because of space limitations. However, the *Defeat Firewall Attack Step* definition is complex enough to demonstrate the relevant concepts.

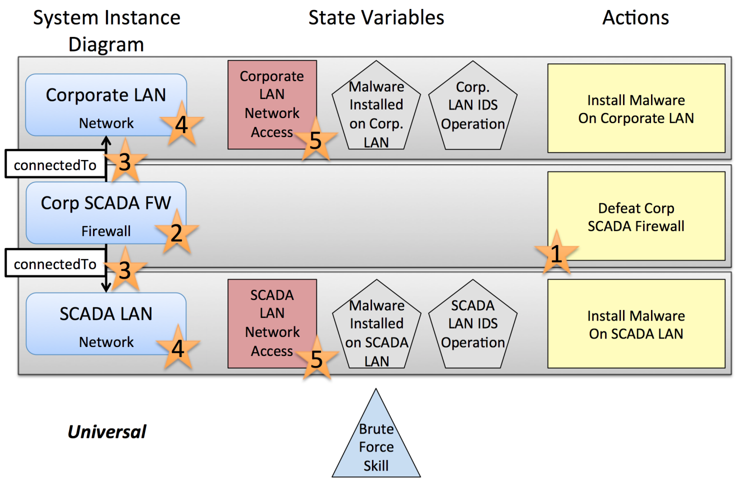

Fig. 5: The generated state variable and action model fragment instances.

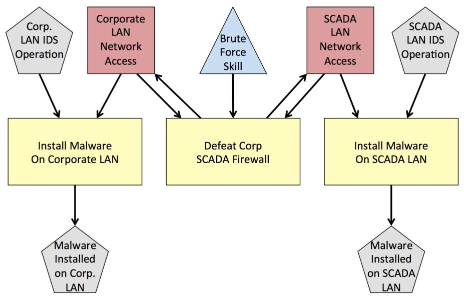
Corp. LAN IDS Operation
Corporate LAN Network Access
Brute Force Skill
SCADA LAN Network Access
SCADA LAN IDS Operation
Install Malware On Corporate LAN
Defeat Corp SCADA Firewall
Install Malware On SCADA LAN
Malware Installed on Corp. LAN
Malware Installed on SCADA LAN

Fig. 6: The generated ADVISE attack execution graph for the Two Nets example.

- **Defeat Firewall Attack Step**
  - **bfs:** Precondition element.
    - $Universal$(`BruteForceSkill`)
  - **netAccesses:** Precondition (and affected) element.
    * $Relationship_{source}$(`connectedTo`)
      * $Component$(`Network`)
        * $Dependent$(`Network Access`)

The path for finding the universal brute-force skill is trivial, but the path of **netAccesses**, which finds all `Network Access` state variables of components connected to the firewall, requires some explanation. Using the orange stars in Figure 5, we will step through the macro path exploration for the *netAccesses* variable macros. The origin of every path is always the component instance of which the macro's parent is a dependent. In this case, the macro's parent is the *Defeat Corp SCADA Firewall* attack step instance (star marked "1"). The component instance on which the attack step is dependent is shown as a star marked "2" and is the origin of the path constraints. The first step in the path expression is to find outgoing relationship instances of the `connectedTo` type, which are shown with stars marked "3." The next step in the path expression looks for components targeted by the discovered arcs with a type of `Network`. Those components are shown as stars marked "4." The final step in the path expression will look at the dependents of the "4" star components that have a class of `Network Access`. These network access instances are shown as stars marked "5." The end result is that AEG arcs will be drawn to and from the *Defeat Corp SCADA Firewall* attack step and will connect every `Network Access` element of a network component that is attached to the firewall component that this attack step is attacking.

Figure 6 shows the final generated attack execution graph that is created by the example ontology and the system instance diagram shown in Figure 4. The central attack step and the arcs that connect it to the graph were created by the variable macros described earlier.

4 Case Studies

The Möbius ontology framework and its implementation have been used by dozens of test users in order to evaluate and refine the overall approach. Two in-depth case studies have been presented at peer-reviewed conferences.

In [15], the authors studied several designs of intrusion detection system deployments on a multi-tiered advanced metering infrastructure in power distribution systems. They calculated metrics that provided the expected cost to the system owner and the probability that the adversary would be detected. In [16], the authors used the Möbius ontology framework and ADVISE to construct a set of physical attack models for a railway station. From these models, the authors determined the likely sequence of attack steps for each class of attacker and identified which kinds of devices should receive additional monitoring resources to maximize the mitigation of attacks.

5 Related Work

We have not found work that attempts to automatically generate detailed, discrete-event stochastic models from an ontological definition of system components and a high-level specification of a general system based on those components. However, we have found many efforts that relate to our approach.

In [17], the authors automatically translate feature-based system models to PRISM models for the purpose of probabilistic model checking. In [18], software code models are translated to Petri-net-based performance models. If one considers the detailed model generation of our approach as a translation from a source model to a target discrete-event model, our source model is comprised of a high-level system description and a reusable, auditable, and extensible ontology of translation rules. We believe that this design difference is critical.

[19] generates attack trees from a π-calculus process language by using static analysis. [20] generates attack trees from a general system model by using a rigid set of rules that expand attack tree nodes based on locations, assets, actors, and processes. These approaches have some similarities in their generation parts, but their generated products are significantly different. They generate attack trees for static analysis, whereas we are generating executable, discrete-event, state-based models that can be evaluated through simulation or analytical solution. Furthermore, their input rules are not extensible.

In [21], the authors discuss an approach for generating code from high-level Petri nets and how it can be extended to generate code from general UML models with additional semantic information. That second part has a clear connection to our approach of coupling a high-level UML-like model with formal semantic definitions in the ontology to construct executable code. However, we generate executable graphical models as an end result, and we take a more formal approach in defining the ontology and system model, thereby enabling a much larger class of potential input models than the one on which [21] focuses.

[22] generates an executable, generalized stochastic Petri net model from a specialized class of UML models, sequence diagrams, and statecharts. [23] generates an OSAN from another restricted class of UML models with Petri net annotations. We believe that an ontology in the Möbius ontology framework can be defined that would generate executable stochastic models equivalent to those described by those authors; it might be an interesting topic for future work.

[24] and [25] construct ontologies that describe discrete-event simulation and use them to generate executable code to evaluate discrete-event models encoded in a knowledge base of the ontology. This approach has several connections to our work, but does not abstract the complexity of discrete-event models of real systems by allowing the user to think about and define the system at the level of custom component building blocks. In essence, the authors of [24] and [25] replicate the existing Möbius framework, but using an ontology to describe the atomic models instead of the variety of formalisms that Möbius provides.

There is a large body of work in the area of model-driven software engineering [26], including many well-known frameworks, such as the Eclipse Modeling Framework (EMF) [27]. These efforts often use formal meta-models to describe the architecture of a software application, and then use the meta-models to automatically generate significant portions of the source code necessary to define the application. The software meta-models can be thought of as a kind of knowledge base on a source code ontology. While it is clear that the Möbius ontology framework addresses a very different kind of problem, we pulled many lessons from the model-driven software engineering work as we developed our framework.

6 Conclusion

Building formal, mathematical models of real-world systems is a challenging endeavor for any human modeler. A vast array of design details must be encoded in complex modeling formalism primitives, often with many parameters on each primitive. Once complete, these low-level models are difficult for others to interpret and difficult for the human modeler to alter or vary to allow for experimental analysis of multiple designs. The development process of low-level models is a subjective effort, and different human modelers will produce different models.

To address those issues, we have developed an ontology framework that enables the formalization of the model development process in the form of an ontology. The low-level model generator uses an ontology and a SID to automatically construct the low-level model. Instead of working with low-level, complex, discrete-event modeling primitives, users create high-level, intuitive block diagram primitives by using familiar component and relationship notions. The ontologies developed in our framework can be shared, audited, improved over time, and reused. The task of constructing an effective ontology is still as challenging as that of creating low-level models in the original approach. However, the process can be collaborative, and once it is done, the ontology can be used for many system models and by many different people. Ontologies can be connected together through the use of the inheritance relationship among types in order to reuse previous ontology developments and to add greater detail.

We have implemented our framework in the Möbius tool and extended the framework's generator to construct Möbius ADVISE models in order to evaluate the efficacy of our approach. From simple system instance diagrams, rich and thorough attack execution graphs are generated. The implementation and underlying concepts of the Möbius ontology framework have been evaluated in two peer-reviewed case studies [15], [16] and shown to be useful and effective for defining complex models and easily evaluating multiple design alternatives.

References

1. D. M. Nicol, W. H. Sanders, and K. S. Trivedi, "Model-based evaluation: From dependability to sec." *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 48–65, 2004.
2. S. Jajodia, S. Noel, and B. O'Berry, "Topological analysis of network attack vulnerability," in *Managing Cyber Threats: Issues, Approaches, and Challenges*, V. Kumar, J. Srivastava, and A. Lazarevic, Eds. Boston, MA: Springer US, 2005, pp. 247–266.
3. R. Ortalo, Y. Deswarte, and M. Kaâniche, "Experimenting with quantitative evaluation tools for monitoring operational security," *IEEE Trans. on Software Engineering*, vol. 25, no. 5, pp. 633–650, 1999.
4. E. LeMay, "Adversary-driven state-based system security evaluation," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2011.
5. G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Möbius modeling tool," in *Proc. of the 9th Int. Workshop on Petri Nets and Perf. Models*, Sept. 2001, pp. 241–250.
6. A. L. Williamson, "Discrete Event Simulation in the Möbius Modeling Framework," Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1998.
7. T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.
8. D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *IEEE Trans. on Software Engineering*, vol. 28, no. 10, pp. 956–969, Oct. 2002.
9. E. LeMay, W. Unkenholz, D. Parks, C. Muehrcke, K. Keefe, and W. H. Sanders, "Adversary-driven state-based system sec. evaluation," in *Proc. of the 6th Int. Workshop on Sec. Measurements and Metrics (MetriSec 2010)*, Bolzano-Bozen, Italy, Sept. 15, 2010.
10. M. D. Ford, K. Keefe, E. LeMay, W. H. Sanders, and C. Muehrcke, "Implementing the ADVISE sec. modeling formalism in Möbius," in *Proc. of the 43rd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2013)*, June 2013, pp. 1–8.
11. E. LeMay, M. D. Ford, K. Keefe, W. H. Sanders, and C. Muehrcke, "Model-based sec. metrics using ADversary VIew Security Evaluation (ADVISE)," in *Proc. of the 8th Int. Conf. on Quantitative Evaluation of SysTems (QEST 2011)*, Aachen, Germany, Sept. 5–8, 2011, pp. 191–200.
12. M. Rausch, B. Feddersen, K. Keefe, and W. H. Sanders, "A comparison of different intrusion detection approaches in an advanced metering infrastructure network using ADVISE," in *Proc. of the 13th Int. Conf. on Quantitative Evaluation of SysTems (QEST 2016)*, 2016, pp. 279–294.

13. R. Wright, K. Keefe, B. Feddersen, and W. H. Sanders, "A case study assessing the effects of cyber attacks on a river zonal dispatcher," in *Proc. of the 11th Int. Conf. on Critical Information Infrastructures Sec. (CRITIS)*, Paris, France, Oct. 10–12, 2016, pp. 252–264.
14. D. Deavours and W. H. Sanders, "Möbius: Framework and atomic models," in *Proc. of the 10th Int. Workshop on Petri Nets and Performance Models*, Sept. 2001, pp. 251–260.
15. M. Rausch, K. Keefe, B. Feddersen, and W. H. Sanders, "Automatically generating sec. models from system models to aid in the evaluation of AMI deployment options," in *Proc. of the 12th Int. Conf. on Critical Information Infrastructures Sec. (CRITIS)*, Lucca, Italy, Oct. 8–13, 2017.
16. C. Cheh, K. Keefe, B. Feddersen, B. Chen, W. G. Temple, and W. Sanders, "Developing models for physical attacks in cyber-physical systems," in *Proc. of the Cyber-Physical Systems Sec. and PrivaCy (CPS-SPC) Workshop*, Dallas, Texas, USA, Nov. 3, 2017, pp. 49–55.
17. P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier, "Family-based modeling and analysis for probabilistic systems – featuring ProFeat," in *Fundamental Approaches to Software Engineering.* Berlin, Heidelberg: Springer, 2016, pp. 287–304.
18. M. Woodside, D. C. Petriu, J. Merseguer, D. B. Petriu, and M. Alhaj, "Transformation challenges: from software models to performance models," *Software & Systems Modeling*, vol. 13, no. 4, pp. 1529–1552, Oct 2014.
19. R. Vigo, F. Nielson, and H. R. Nielson, "Automated generation of attack trees," in *Proc. of the IEEE 27th Comp. Sec. Foundations Symp.*, July 2014, pp. 337–350.
20. M. G. Ivanova, C. W. Probst, R. R. Hansen, and F. Kammüller, "Transforming graphical system models to graphical attack models," in *Graphical Models for Sec.: Second Int. Workshop, GraMSec 2015, Verona, Italy, July 13, 2015, Revised Selected Papers*, S. Mauw, B. Kordy, and S. Jajodia, Eds., vol. LNCS 9390. Springer Int. Publishing, 2016, pp. 82–96.
21. S. Philippi, "Automatic code generation from high-level Petri-nets for model driven systems engineering," *Journal of Systems and Software*, vol. 79, no. 10, pp. 1444–1455, 2006.
22. S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable Petri net models," in *Proc. of the 3rd Int. Workshop on Software and Performance.* ACM, 2002, pp. 35–45.
23. A. Kamandi, M. A. Azgomi, and A. Movaghar, "Transformation of UML models into analyzable OSAN models," *Electronic Notes in Theoretical Comp. Science*, vol. 159, pp. 3–22, 2006.
24. Y. Gheraibia and A. Bourouis, "Ontology and automatic code generation on modeling and simulation," in *6th Int. Conf. on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT).* IEEE, 2012, pp. 69–73.
25. L. Lacy, "Interchanging discrete event simulation process interaction models using PIMODES and SRML," in *Proc. of the Fall 2006 Simulation Interoperability Workshop*, 2006.
26. M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven Software Engineering in Practice.* Morgan & Claypool Publishers, 2012.
27. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.