A Framework for Incremental Model Construction and Analysis

Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{eskang,aleks,dnj}@mit.edu

Abstract. When a formal analysis procedure suceeds, one may be tempted to jump to the conclusion that the system satisfies the desired property. In practice, the system may suffer from a violation of the property, despite the successful analysis, because the model used in the analysis omits details about the system that could lead to the violation. Constructing a model that adequately captures all relevant aspects of a system, however, is a challeging task that is often carried out in an ad-hoc fashion.
In this paper, we propose a framework to aid a system analyst in incremental construction and analysis of a system model. The key component of our framework is a model-merging technique that incrementally elaborates an initial, partial system model by composing it against a repository of domain-specific models. We apply our framework to an analysis of two well-known web security protocols–[(aleks): should this be an mdash instead of ndash?]OAuth and OpenID.

1 Introduction

In a typical analysis process, the analyst constructs a model that describes the system and its environment, and applies an analysis technique, such as model checking and theorem proving, to determine whether the model satisfies a given property. If the analysis completes without finding a violation, the analyst may be inclined to assume that the actual system satisfies the desired property.

In practice, the system might suffer from a failure, despite the successful analysis, because the model used in the analysis (purposely or inadvertently) omits details about the system that could lead to a violation of the property. The analyst, for instance, may decide to leave out the description of a seemingly unimportant system function, or simply be unaware of certain classes of interactions with the enviroment. For example, an ATM system for a major bank in UK, which relied on a provably correct cryptographic mechanism to protect a PIN number on a card, suffered from security failures because the designer had neglected to consider the possibility that a malicious person could simply modify the account number on his card to that of another customer (which enabled him to withdraw money from the victim's account) [**?**].

Constructing a model that is sufficiently detailed to account for possible failures, however, is a challenging task that requires significant expertise in the

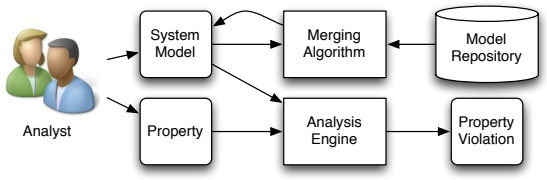


Fig. 1: Overview of our framework on model construction and analysis.

problem domain. The analyst—often the designer of the system—may initially have a rather limited, partial knowledge about the system, or ignore details that appear irrelevant to its core functionality. As the analyst improves her understanding of the system, its operating environment, and potential failures, she may incrementally elaborate the existing system model and perform multiple iterations of analysis, each time discovering new violations that the prior analyses had ruled out. Unfortunately, this process is often carried out in an ad hoc, manual fashion.

In this paper, we propose a framework to aid the analyst in incremental construction and analysis of a system model. Our framework consists of three key components, as shown in Figure 1: (1) a repository of models that encode partial knowledge about the problem domain, (2) an algorithm that merges two independent models of a system, and (3) an analysis engine that checks whether a system model satisfies a given property, and if not, produces a counterexample. In a typical workflow, the analyst begins by constructing a partial model of the system, and runs an initial analysis to check the system against a desired property. The analyst then incrementally elaborates the initial model into a more detailed one by ~~merging it with the existing models in the repository, one-by-one~~selecting a set of existing models from the repository, relevant for the particular category of the system under design, and merging them into the initial one. Each such model from the repository is written by a domain expert and serves as a formal specification of a known feature (e.g., aspect, consideration, security threat, vulnerability, etc.) of an entire category of systems (e.g., the HTTP protocol, web servers, web applications, etc.).[(AM): it's probably not very important to say here that this is done one at a time; your tool could in principal merge many of them]. After every merge step, the analyst re-runs the analysis on the elaborated model to detect any potential violations that were absent from prior analyses.

A technical challenge in composing two independent models is that they may not be amenable to composition due to *abstraction mistmach.*[(AM): would it be worth noting here that the models are also declarative, written in logic, which also makes it difficult?] The two models may describe some aspect of the system at different layers of abstraction, using different sets of vocabulary terms, so standard composition techniques that connect components at their interfaces may fail. To resolve this mismatch, we allow the analyst to specify relationships between various parts of the two models. Our merging algorithm then automatically constructs a new model that is the result of composing two mismatched models according to the specified relationships.

Our goal is not to completely automate the model construction process; we still rely on the analyst to use insights about the system to drive the model elaboration process, and interpret the output of the analysis engine to devise meaningful fixes to the model. Our framework facilitates the model construction and analysis process in the following ways:

- **Knowledge reuse**: A repository of generic domain models, constructed by a domain expert, can be reused for analysis of multiple systems. For example, a repository of models about common attacks on web applications may be used to analyze the security of multiple, independent web protocols.
- **Incrementality**: Instead of having to come up with a complete, monolithic model of a system, the analyst can incrementally explore different aspects of the system, one at a time. For example, the analyst may rank the available attack models in the order of their criticality, and analyze the system for the most pressing issues first.
- **Adaptiveness**: As the repository is updated with new models (e.g., newly discovered attacks or new operating environments), the analyst may repeat the analysis process without having to build the model from the scratch.

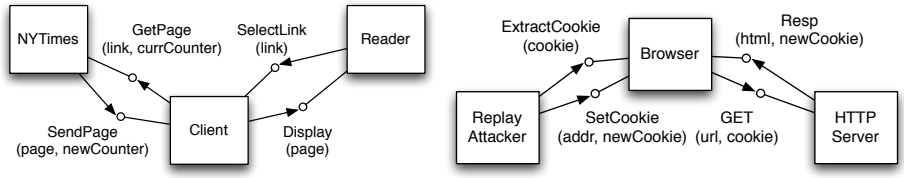
Contributions The contributions of this paper are as follows:[(AM): small first letters in the following list?]

- A framework for incremental construction and analysis of a system model that leverages a domain-specific model repository,
- A technique for merging two potentially mismatched models,
- An implementation of the framework, including a language for specifying system models and an automated analysis engine built on top of a model finder, and
- Application of the framework to the analysis of two well-known web protocols, OAuth [2] and OpenID [**?**].

Outline This paper is structured as follows. Section 2 provides an overview of our framework with an example from an online news paywall system. Section 3 introduces the underlying formalism for modeling systems. Section 4 describes the types of relationships that the analyst can specify between two potentially mismatched models, and the merging algorithm that composes the two based on the given relationships. Section 5 discusses the implementation of the modeling formalism as a domain-specific language as well as an automated analysis of merged models. Section 6 describes the case studies on applying our approach to the OAuth and OpenID protocols. The paper concludes by discussing the related work (Section 7), and the potential benefits and limitations of our approach (Section 8).

2 Overview

In this section, we give an informal overview of how an analyst would use the framework to incrementally construct and analyze a model of a system. We use an example from the web application domain to illustrate our appraoch.



(a) NYTimes Paywall (b) Cookie Replay

Fig. 2: Models of (a) the New York Times paywall system and (b) a cookie replay attack. A box represents a module, and a directed edge from one module A to another module B with label O represents A's invocation of operation O that is exported by B.

Building an Initial Model Let us assume that the *New York Times*, a well-known news organization, wishes to design a *paywall* system for its online web site. A paywall system is used to limit readers to a set number of free articles over a time period, requiring them to signup for a paid subscription once they reach the limit. At the same time, for optimal user experience, new users should not be required to create an account to start reading the news articles. One desirable property of this paywall system is that *readers who have already exceeded the limit should not be able to access an article.*

Alice, the chief web designer of the New York Times, devises a simple design of the paywall system, as shown in Figure 2(a). There are three basic participants in this design: the NYTimes server, which stores and serves articles, Reader, an end-user who wishes to access an article, and Client, which mediates the transfer of articles between NYTimes and Reader. When Reader selects a link to an article, Client forwards the request for the article to the NYTimes server, sending along a counter (currCounter) that represents the number of articles Reader has accessed so far. NYTimes accepts a request GetPage from Client only if currCounter is less than the limit. Once processing the request, NYTimes then sends back a page that contains the requested article, along with an increment of the original counter (newCounter).

Note that the model in Figure 2(a) describes the system interactions in terms of web API operations. This level of abstraction is suitable for the designer to capture the essential functionality of the system; it omits low-level details that are irrelevant to the system workflow, such as what kind of devices various participants are deployed on, how various API operations are implemented, or the type of data structure that is used to represent counters and pages.

To ensure that the design is satisfactory, Alice decides to run the model in Figure 2(a) through an analysis engine that exhaustively explores the behavior of the model. As expected, the engine concludes that the system satisfies the property by preventing the reader from accessing an article beyond the limit.

Elaborating the System Model Being satisfied with the initial design, Alice moves onto exploring lower-level design choices for the system. She decides that NYTimes will run on top of an HTTP server, with GetPage implemeneted as a standard HTTP GET operation. Similarly, she designates Client to run on top

of a standard web browser, and the access counter to be stored as a cookie inside the browser.

Alice decides to explore the potential implications of these design decisions on the security of the system using our framework. ~~Let us assume that the repository is already populated with models that describe different types of attacks that are applicable to web-based systems.~~Therefore, Alice first searches the repository for security threat models defined for the category of web-based system. ~~For example, one type of attack that the paywall system might be vulnerable to is a *cookie replay attack*~~She finds a model of the *cookie replay attack* and decides to check if her system is vulnerable to it.

The cookie replay attack involves a malicious user obtaining a cookie and repeating its transmission to a server. In the corresponding model, shown in Figure 2(b), Browser communicates to HTTPServer by sending GET requests and receiving responses (Resp) in return. In addition, a malicious user (named ReplayAttacker) interacts with Browser by extracting a cookie from it (ExtractCookie), or overwriting an existing cookie at a particualr address (SetCookie); this allows ReplayAttacker to manipulate Browser into sending a request with a fixed cookie value.

A next step for Alice would be to extend the original paywall model with new details from the cookie replay model, and re-run the analysis to see whether the elaborated model still satisfies the desired property. However, the two models, as in their current form, are not readily amenable to composition due to an *abstraction mismatch*. These models describe the system at different layers of abstraction, using two distinct sets of vocabulary terms to describe modules, operations, and data elements. They serve as *partial* descriptions of a system, including only the details of the system that are necessary to illustrate a typical workflow (as in the paywall model) or the characteristics of a particular attack (in the replay model). Thus, additional guidance is needed to identify relationships between parts of the two models before they can be merged.

Our observation is that relationships between a pair of models correspond to the designer's decisions about how different parts of the system are to be realized. For example, based on Alice's decisions about the implementation of the paywall system, we may infer that NYTimes and HTTPServer can be treated as the same component; in particular, NYTimes can be considered a type of HTTPServer that performs a specialized function of serving articles to its client. A similar kind of relationship can be specified between a pair of operations (e.g., GetPage and GET) or data types (e.g, Link and URL). Given the relationships specified by Alice, our framework merges the two models and produces a new model that combines both aspects of the system, as shown in Figure 3.

Re-Analyzing the Model Alice re-runs the analysis engine on the new, merged model to check whethe the elaborated system still satisfies the desired property. This time, the analysis engine discovers a counterexample that demonstrates how the system might allow the reader to access an article beyond the limit. In this scenario, Reader, now acting like[(AM): as?] ReplayAttacker, extracts a cookie that represents the access counter prior to reaching the limit. Then, Reader simply

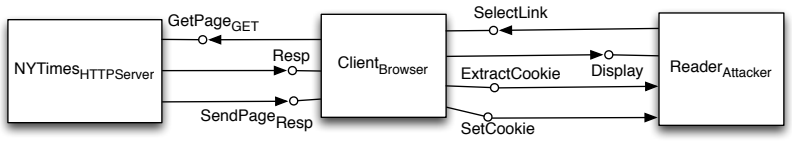


Fig. 3: Result of merging the two models from Figure 2. A subscript Y in X_Y implies that X is a type of Y.

overwrites the existing cookie with the extracted once the limit is exceeded, and NYTimes continues to serve GetPage requests from Client [1]. To prevent this type of attack, Alice modifies the system such that when NYTimes sends back a cookie, it generates and includes a unique token that cannot be guessed by Reader, thereby rendering old cookies invalid.

[(AM): CHECK: can we say the following?]

It is important to note that, to merge her initial model with the cookie replay attack, Alice did not have to explicitly specify that the Reader should be mapped to ReplayAttacker. In contrast, Alice only had to specify her architectural decision to implement the system on top of HTTP, which meant implementing NYTimes::GetPage as HTTPServer::GET and Client::SendPage as Browser::Resp. The information about potential vulnerability of HTTP systems to cookie replay is inherently present in the repository model, and our system is able to take advantage of it.

Alice may repeat the whole process again, elaborating the system model with further details from other types of attack models in the repository, and re-running the analysis engine to discover more potential design flaws.

[(AM): General comments: (1) renaming suggestions: GetPage → GetArticle, SendPage → ReceiveArticle, Client → NYTClient, Resp → ReceiveResp]

3 Formalism

In this section, we first introduce the underlying formalism that our framework uses to specify a system and its behavior. Our modeling approach is based on standard event-based architectural formalisms such as Darwin [**?**].

Structure In our approach, a system consists of a set of modules M that asynchronously communicate with each other through operations O. Each module *exports* zero or more operations at its interface, and *stores* some set of data D ~~that can be transmitted~~. Data can flow from one to another module ~~by *invoking* the latter's operations~~only by means of operation *invocations*; this enables our system to accurately track data flows and automatically check standard information flow properties such as *integrity* and *confidentiality*.[(AM): This data flow business is certainly important, but I'm not sure it belongs here. Maybe move it a few paragraphs

[1] The actual paywall system for the New York Times suffered this type of attack when it was first introduced in 2008 [**?**].

Sorts		**Operation** (O)	
M, O, D, V		$args : N \nrightarrow D$	// arguments
N, F	// Name, Formula		
		Data (D)	
Module (M)		$fields : N \nrightarrow D$	// data fields
$exports : \mathcal{P}(O)$	// exported ops		
$invokes : \mathcal{P}(O)$	// invoked ops	**View** (V)	
$stores : N \nrightarrow D$	// stored data	$mods : \mathcal{P}(M)$	// modules
$cons : O \nrightarrow \mathcal{P}(F)$	// constraints	$data : \mathcal{P}(D)$	// data types

Fig. 4: Basic components of our modeling formalism.

below, where it talks about passing arguments.]. For example, consider the NYTimes server from Figure 2(a). One way to specify this module in our approach is as follows[2]:

$\mathsf{NYTimes}.exports = \mathsf{GetPage}$

$\mathsf{NYTimes}.invokes = \mathsf{SendPage}$

$\mathsf{NYTimes}.stores(\mathsf{articles}) \in \mathsf{Map[Link,Page]}$[$(AM)$: why $\in$ and not =? Map is also undefined, as well as the Map[K,V] syntax]

where $\mathsf{GetPage} \subseteq O$ is the type of operation that corresponds to a request for an article page, and $\mathsf{SendPage} \subseteq O$ is the type of operation for transferring a page to Client. $\mathsf{Map[Link,Page]} \subseteq D$ represents a type of data structure that maps a link to a page. [(AM): No need to explain these operations/datatypes again]

Each operation $o \in O$ ~~is associated with~~has zero or more named *arguments*~~, which the invoker uses to transmit data to its exporter~~; a module invoking an operation transmits data to the module exporting that operation by setting the operation's arguments. For example, every operation o of type GetPage carries two arguments: a link to an article (link), and a counter (currCounter) that represents the number of articles accessed; i.e.,

$$dom(o.args) = \{\mathsf{link}, \mathsf{currCounter}\}$$
$$o.args(\mathsf{link}) \in \mathsf{Link} \wedge o.args(\mathsf{currCounter}) \in \mathsf{Int}$$

[(AM): replace with? (it doesn't require additional syntax, like *dom* and it's more succinct)]

$$\mathsf{GetPage}.args = \{\text{“}link\text{”} \nrightarrow \mathsf{Link}, \text{“}currCounter\text{”} \nrightarrow \mathsf{Int}\}$$

Additionally, each data element may be assigned a named set of *fields*, allowing construction of complex data types from simple ones.

Finally, a system view $v \in V$ is simply defined as a set of modules ($v.mods$) and data types ($v.data$).

Behavior A module may not allow every incoming operation, and may not send out arbitrary messages to other modules. For every type of operation that

[2] For convenience, we use the dot (.) notation to refer a tuple element by name; for example, $m.exports$ refers to $exports$ element of module m.

it exports, the module can impose a constraint on the set of the operations that it is willing to accept; the function *cons* is used to specify such a constraint. For example, NYTimes may only accept an incoming request for an article only if currCounter provided with the request does not exceed the limit. Formally, for each operation o of type GetPage, NYTimes.$cons(o)$ returns a formula f such that

$$f \equiv o.args(\mathsf{currCounter}) \leq \mathsf{LIMIT}$$

where LIMIT is a constant that represents the number of allowed articles.

Similarly, a module may impose a constraint on the set of operations that it can send out. For example, the authorization server may invoke SendPage on the client only if it has already received and accepted a valid GetPage message. In addition, SendPage must include, as arguments, an article that corresponds to the requested link, and an increment of the current counter. Formally, for every $o \in m.invokes \cap \mathsf{SendPage}$, NYTimes.$cons(o)$ returns formula f such that:

$$f \equiv \exists o' \in \mathsf{GetPage} \cdot o.args(\mathsf{page}) = (\mathsf{NYTimes}.stores(\mathsf{articles}))[o'.args(\mathsf{link})] \wedge \\ o.args(\mathsf{newCounter}) = o'.args(\mathsf{currCounter}) + 1$$

In effect, constraints in *cons* are being used to declaratively specify the behavior of a module in terms of input and output operations that it is allowed to perform. The overall behavior of module m is characterized by sets $m.in$ and $m.out$:

$$m.in = \{o \in m.exports \mid \bigwedge m.cons(o)\}$$
$$m.out = \{o \in m.invokes \mid \bigwedge m.cons(o)\}$$

Intuitively, $m.in$ represents the set of incoming operations that m accepts, and $m.out$ represents the set of invocations that m is allowed to make. Note that the domain of $m.cons$ is restricted to the set of operations that m exports or invokes (i.e., $m.exports \cup m.invokes$). We assume that if the analyst does not specify a constraint for an operation that it exports or invokes, then it is assigned truth value $\top$ (i.e., the operation is always allowed).

4 Technique for Merging Models

In this section, we describe a technique for merging two potentially mistmached models that describe different aspects of a system. The merging process is two-fold. First, the analyst specifies the relationship between two views as a mapping between different elements of the views. Then, given this mapping, we apply a merging algorithm to produce a new view that retains the characteristics of the two, with potentially new behaviors that arise from the interaction of the modules from the previously separate views.

```
view Paywall

Paywall.data = {Page, Link}
Paywall.mods = {NYTimes, Client, Reader}

NYTimes.exports = {GetPage}
NYTimes.invokes = {SendPage}
NYTimes.stores  = {"articles" ↠ Map[Link, Page]}
NYTimes.cons    = {??}

// similar for Client and Reader

GetPage.args = {"link" ↠ Link, "currCounter" ↠ Int}
```

Fig. 5: Formal specification of the Paywall example.

4.1 Specifying Relationships

To resolve potential mismatch between two models, the analyst may specify a relationship between a pair of modules, operations, or data elements. Our notion of a relationship between two entities is that of a *subset* relation. Informally, two entities are related if they both describe the same concept in the real world, but one is a generic representation of the other. For example, NYTimes can be regarded as a specialized type of HTTPServer with the role of controlling access to articles that it stores. A similar kind of relationship can be introduced between operations or data elements. Operation GetPage, implemented using a generic HTTP request, can be regarded as a type of GET operation. Similarly, since a counter is stored as a cookie, Counter can be treated as a type of Cookie.

Definition 1 *A relationship $r \in R$ between two views is a tuple (r_M, r_O, r_D) where $r_M \subseteq M \times M$, $r_O \subseteq \mathcal{P}(O) \times \mathcal{P}(O)$, and $r_D \subseteq \mathcal{P}(D) \times \mathcal{P}(D)$ represent declarations of subset relations between a pair of modules, operation types, and data types, respectively.*

For example, to merge the paywall and cookie replay models from Figure 2, Alice may specify the following relationships between the two views:

$$r_M = \{(\mathsf{Client}, \mathsf{Browser}), (\mathsf{NYTimes}, \mathsf{HTTPServer}), (\mathsf{Reader}, \mathsf{ReplayAttacker})\}$$
$$r_O = \{(\mathsf{GetPage}, \mathsf{GET}), (\mathsf{GET}, \mathsf{GetPage}), (\mathsf{SendPage}, \mathsf{Resp})\}$$
$$r_D = \{(\mathsf{Article}, \mathsf{HTML}), (\mathsf{Link}, \mathsf{URL}), (\mathsf{Counter}, \mathsf{Cookie})\}$$

A subset relationship is asymmetrical, and so two entities can be specified as equivalent by including a tuple and its inverse in the same relation; for this example, Alice here indicates that the set of operations of type GetPage is exactly the operations of type GET.

4.2 Merging Algorithm

The algorithm for merging, shown in Algorithm 1, accepts two views, $v1, v2 \in V$, and a relationship $r \in R$ between them, and produces a new view, $v' \in V$. The algorithm is divided into two steps. First, for every pair of entities specified in r, it merges the two entities into a single entity that retains the characteristics of the two original entities (Line 2). Then, the algorithm builds a new view by replacing the original entities in r with the new entities (Line 3).

Step 1: Merging Related Entities For every pair of entities $e1, e2$ specified in r, our goal is to produce a new entity e' that (1) replaces $e1$ in the merged view v', and (2) obtains the characteristics of $e2$. The procedure BuildMapping performs this task in two-fold, first merging each pair of related data sets (Line 7), and then merging the related modules (Line 9). As it performs the merging tasks, BuildMapping incrementally builds a pair of functions (h_D and h_M) that map the original modules and data sets from r to their merged counterparts.

Merging Data For each pair of data sets $(D1, D2)$ in r_D, MergeData produces a new set of data, D', that replaces $D1$ in the merged view v', and can still be treated like instances of $D2$.

For example, given $(\mathsf{Article}, \mathsf{HTML}) \in r_D$, MergedData produces a new data set $\mathsf{Article}_{\mathsf{HTML}}$, which conceptually represents the encodings of the New York Times articles in HTML format. Article and $\mathsf{Article}_{\mathsf{HTML}}$ are isomorphic; for every $a \in \mathsf{Article}$, v' contains a unique $a' \in \mathsf{Article}_{\mathsf{HTML}}$ such that a' represents an HTML encoding of a. Note that v' may contain some HTML documents that do not correspond to any article (i.e., $\mathsf{Article}_{\mathsf{HTML}}$ may be a strict subset of HTML).

Formally, D' represents the result of *embedding* $D1$ into $D2$.

Definition 2 *An embedding from set A to B is an injective function $f : A \to B$.*

The embedding establishes a subset relation between $D1$ and $D2$ by ensuring that every element d of $D1$ has a unique counterpart $f(d)$ that is a member of $D2$. For example, the embedding from Article to HTML is a function f that maps each article to its HTML counterpart. Then, in the merged view v', every $a \in \mathsf{Article}$ can be replaced by $f(a)$ without invalidating existing references to Article in $v1$.

Theorem 1 *For every $D1, D2 \subseteq D$, there always exists a function $f : D1 \to D2$ such that f is an embedding from $D1$ to $D2$.*

[**TODO**: need proof here]

Finally, the set of fields associated with D' is constructed as the union of fields in $D1$ and $D2$.

Merging Modules Given a pair of modules $(m1, m2) \in r_M$, along with pairs of related operations r_O, MergeModule produces a new module m' behaves like $m1$ while also obtaining the characteristics of $m2$.

The behaviors of the two modules may be related through one or more pairs of operations specified in r_O. If $m1$ exports or invokes a set of operations $O1$ that are related to another set of operations $O2$ in $m2$, MergeModule merges the two sets by using the procedure MergeOp (lines 34 and 39).

Merging Operations Operations are merged in the similar fashion as the data elements; an embedding is used to establish a subset relation between $O1$ and $O2$, resulting in a new set of operations $O' \subseteq O2$, which also inheirts all of the arguments from $O1$ and $O2$ (Line 47).

For example, given $(\mathsf{SendPage}, \mathsf{Resp}) \in r_O$, $MergeOp$ produces a set $\mathsf{SendPage}_{\mathsf{Resp}}$, which represents the implementation of SendPage as an HTTP response. $\mathsf{SendPage}_{\mathsf{Resp}}$ can be used by NYTimes to send an article back to $\mathsf{Client}_{\mathsf{Browser}}$, but is also treated like any standard HTTP response. Note that $\mathsf{Client}_{\mathsf{Browser}}$ may receive other kinds of HTTP responses besides the ones that carry an article from NYTimes; i.e., there may exist $r \in \mathsf{Resp}$ that is not a member of $\mathsf{SendPage}_{\mathsf{Resp}}$.

The constrain function of the merged module m' is constructed as the union of $m1.cons$ and $m2.cons$ (Line 31). This ensures that m' maintains the behaviors of both $m1$ and $m2$ with respect to the operations that they export and invoke, including the merged operations. In particular, the set of constraints applicable to $o' \in O' = MergeOp(O1, O2)$ can be characterized as follows:

$$m'.cons(o') = \{l \in F \mid l \in m2.cons(o') \vee \exists o1 \in O1 \cdot o' = f_{O1,O2}(o1) \wedge l \in m1.cons(o1)\}$$

In other words, since o' can be considered a member of both $O1$ and $O2$, o' retains all of the constraints that $m1$ and $m2$ impose on $O1$ and $O2$, respectively. Similarly, the set of data that m' stores is constructed as the union of $m1.stores$ and $m2.stores$ (Line 31).

[**TODO**: example with invokes constraints on SendResp]

[**TODO**: Explain why Alice specified GET and GetPage to be equivalent]

Remarks Note that the merging algorithm does not enforce any constraint between the arguments of two operations being merged, or the fields of two data sets. [**TODO**: Say this was intentionally done to make the mapping lightweight]

Step 2: Building a New View Once the different parts of the views have been merged according to the given relationships, constructing the merged view is a straightforward process. Using the function h_M that maps a module from an existing module to its merged counterpart, BuildView simply replaces each of the modules that appear in r_M with the new merged one (Line 15), while leaving those that do not appear in r_M intact (Line 17). Similarly, BuildView uses h_D to replace each of the data sets from $v1$ that appears in r_D with its embedded counterpart (Line 20).

4.3 Implications of Merging

[**TODO**: new interactions when a module invokes a new operation]

[**TODO**: new interactions when two data sets are related, maybe included as arguments]

5 Implementation

To implement our approach, we (1) designed a domain-specific surface language (called sLang), and (2) wrote a compiler that translates models written in sLang to first-order relational logic (namely the Alloy language [1]), which is amenable to fully automated (but bounded) analysis [4].

The main design goal of sLang is to provide a high-level paradigm for writing formal system designs (architectures) which is intuitive and readily accessible to analysts not already experienced in formal methods. sLang, therefore, intentionally hides most of the internal complexities of our declarative formalism; instead, sLang provides a mostly imperative service-oriented paradigm, which allows the analyst to express valid system behaviors simply by specifying service invocations that different components may perform. Furthermore, sLang is fully embedded in Ruby (all sLang programs are syntactically correct Ruby programs), making the extensive set of existing development tools for Ruby readily available, as well as making the learning curve less steep.

In this paper, we do not attempt to evaluate the assertions about the language simplicity and accessibility to non-experts in formal methods; instead, we invite the community to contribute our existing knowledge database by downloading sLang [3] and submitting attack models in their domain of expertise.[**CHECK**: !]

5.1 The sLang Language

Figure 6 shows the implementation of the running "New York Times Paywall" example in sLang. The main concepts from our formalism (views, data, components, and operations) are specified in sLang by using a pseudo-keyword the same name[3] (e.g., lines 1, 2, 3, 5, 7, etc.). The containment relation between them is implicitly defined by the nesting level, much like in Ruby and the traditional object oriented programs in general. For instance, the call to the **`view`** function (line 1) defines the `Paywall` view containing all the definitions inside the **`do ... end`** block. Component definitions can be prepended with the **`trusted`** pseudo-keyword (lines 5, 13) to distinguish trusted from potentially malicious components (if not explicitly marked as trusted, the component is considered malicious, e.g., line 22). Similarly, data definitions can be prepended with **`critical`** (line 2); one of the predefined properties in our framework checks that no critical data ever flows to a non-trusted component. The same syntax (a comma-separated list of name→type pairs between square brackets, i.e., a hash in a singleton array in Ruby terms) is used to specify the data stored in components (lines 5, 13) and operation arguments (lines 7, 14, 23). Operations

[3] All pseudo-keywords in sLang are implemented as plain Ruby methods

```
1  view :Paywall do
2    critical data Page
3    data Link
4
5    trusted component NYTimes [articles: Int ** Page, limit: Int] do
6      creates Article
7      operation GetPage [link: Link, currCounter: Int] do
8        guard    { currCounter < limit) }
9        response { Client::SendPage[articles[link]] }
10     end
11   end
12
13   trusted component Client [counter: Int] do
14     operation SendPage[article: Article] do
15       response { Reader::Display[article] }
16     end
17     operation SelectLink[link: Link] do
18       response { NYTimes::GetPage[link, counter] }
19     end
20   end
21
22   component Reader do
23     operation Display[page: Page]
24     response { Client::SelectLink }
25   end
26 end
```

Fig. 6: Paywall example in sLang

can have guards and responses: the former specifies a precondition which must hold in order for the operation to be allowed (e.g., line 8, only when the given argument (`currCounter`) is less than the stored limit (`limit`)), and the latter specifies which operations and under what conditions the component invokes whenever this operation is triggered (e.g., line 9, every time the `GetPage` operation is invoked, the `NYTimes` component will invoke `SendPage` of the `Client` component, passing the article corresponding to the given link as the operation argument).

sLang supports certain declarative constructs as well, so it is not less expressive than the formalism described in Section 3. For example, operation invocations do not have to be fully specified, as in the `Reader` component which is allowed to invoke the `SelectLink` operation at arbitrary time points (line 24). Passing exact arguments to operations is also unnecessary; arguments can be omitted, or a declarative constraint can be given to arbitrarily constrain argument values. A detailed description of these more advanced features, as well as any discussion of how the embedding of sLang in Ruby is actually implemented, are, however, outside the scope of this paper.

6 Case Study

TBD

7 Related Work

TBD

8 Discussion

TBD

References

1. D. Jackson. *Software Abstractions: Logic, language, and analysis.* MIT Press, 2006.
2. The OAuth 2.0 Authorization Framework. `http://tools.ietf.org/html/rfc6749`.
3. sLang—A Language for writing system models and expressing security properties. `http://people.csail.mit.edu/eskang/slang`.
4. E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications.* PhD thesis, MIT, 2008.

Algorithm 1 Merging Views

```
1: function MergeViews(v1, v2, r)
2:     h ← BuildMapping(v1, v2, r)
3:     v' ← BuildView(v1, v2, h)
4:     return v'

5: function BuildMapping(v1, v2, (r_M, r_O, r_D))
6:     for all (D1, D2) ∈ r_D do
7:         h_D(D1) ← MergeData(D1, D2)
8:     for all (m1, m2) ∈ r_M do
9:         h_M(m1), h_M(m2) ← MergeModules(m1, m2, r_O)
10:    return (h_M, h_D)

11: function BuildView(v1, v2, (h_M, h_D))
12:    M', D' ← ∅
13:    for all m ∈ v1.modules ∪ v2.modules do
14:        if m ∈ dom(h_M) then
15:            M' ← M' ∪ {h_M(m)}
16:        else
17:            M' ← M' ∪ {m}
18:    for all ds ⊆ v1.data ∪ v2.data do
19:        if ds ∈ dom(h_D) then
20:            D' ← D' ∪ h_D(ds)
21:        else
22:            D' ← D' ∪ ds
23:    return (M', D')

24: function MergeData(D1, D2)
25:    D' ← {d2 ∈ D2 | ∃d1 ∈ D1 · d2 = f_{D1,D2}(d1)}
26:    for all d ∈ D1 do
27:        f_{D1,D2}(d).fields ← f_{D1,D2}(d).fields ∪ d.fields
28:    return D'

29: function MergeModules(m1, m2, r_O)
30:    e, i ← m2.exports, m2.invokes
31:    s, c ← m1.stores ∪ m2.stores, m1.cons ∪ m2.cons
32:    for all (O1, O2) ⊆ m1.exports × m2.exports do
33:        if (O1, O2) ∈ r_O then
34:            e ← e ∪ MergeOp(O1, O2)
35:        else
36:            e ← e ∪ O1
37:    for all (O1, O2) ⊆ m1.invokes × m2.invokes do
38:        if (O1, O2) ∈ r_O then
39:            i ← i ∪ MergeOp(O1, O2)
40:        else
41:            i ← i ∪ O1
42:    m' ← (e, i, s, c)
43:    return m'

44: function MergeOp(O1, O2)
45:    O' ← {o2 ∈ O2 | ∃o1 ∈ O1 · o2 = f_{O1,O2}(o1)}
46:    for all o ∈ O1 do
47:        f_{O1,O2}(o).args ← f_{O1,O2}(o).args ∪ o.args
48:    return O'
```