Combining MAL-based DSMLs for multi-domain cyber threat modelling

Johan Apelgren
Abstract

Combining MAL-based DSMLs for multi-domain cyber threat modelling

Johan Apelgren

The Meta Attack Language (MAL) is a meta language used to define domain-specific modelling languages (DSMLs) for cyber threat modelling. These modelling languages define the objects and properties of a specific domain (say, a cloud service) which can then model real-life systems built in that domain (say, a website hosted in that cloud service). You can then simulate cyber attacks on these models to find vulnerabilities.

However, in reality single-domain systems are very rare, meaning that single-domain simulations often miss vital attack paths. This paper explores how to integrate two single-domain MAL-based languages into a coherent modelling language capable of modelling both domains at once. It finds that in this context the preferred method of integration is extending one language with the objects of the other, allowing not only multi-domain modelling but also reducing code duplication and allowing feature-inheritance. Additionally, the paper explores how to handle arbitrary combinations of MAL-based languages. It suggests an addition to the MAL-syntax to accommodate such combinations, and discusses how to address various corner cases and pitfalls.
Table of contents

1 Introduction ............................................................................................................ 5
2 Scope ....................................................................................................................... 6
3 Method .................................................................................................................... 7
   Step 1: Background .............................................................................................. 7
   Step 2: Requirements ........................................................................................... 7
   Step 3: Overview of coreLang & azureLang ....................................................... 7
   Step 4: Discussion ............................................................................................... 7
4 Background ............................................................................................................. 8
   4.1 SecuriCAD ........................................................................................................ 8
   4.2 Meta Attack Language .................................................................................... 9
      4.2.1 Associations .......................................................................................... 10
      4.2.2 Attack steps ......................................................................................... 11
5 Requirements ......................................................................................................... 13
   5.1 Interview with Joar Jacobsson ....................................................................... 14
      5.1.1 Joar’s requirements ............................................................................. 15
   5.2 Interview with Andreas Gylling ..................................................................... 15
      5.2.1 Andreas’s requirements ...................................................................... 16
6 Overview of coreLang and azureLang ............................................................... 16
   6.1 coreLang ....................................................................................................... 17
   6.2 azureLang ..................................................................................................... 18
7 Discussion ............................................................................................................... 21
   7.1 Strategies for DSML composition .................................................................. 21
      7.1.1 Embedding ........................................................................................... 21
      7.1.2 Merging ................................................................................................. 22
      7.1.3 Extension .............................................................................................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Implementation</td>
<td>23</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Propagating changes between peer languages</td>
<td>28</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Hybrid logic</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>Bibliography</td>
<td>40</td>
</tr>
</tbody>
</table>
1 Introduction

Cyber threat modelling is the process of using a model to identify potential adversarial cyber threats to the real-life IT system that the model represents [1]. SecuriCAD is a suite of software developed by Foreseeti AB to do this kind of cyber threat modelling. SecuriCAD performs attack simulations on a model of a system, analyzing how an attacker could move through the system to reach high-value targets. The insights gained from these attack simulations can then be used by stakeholders to inform themselves of potential risks so that they can either minimize their exposure to the threat or outright prevent it from ever being realized.

In order to deliver accurate and actionable information, the models used in SecuriCAD must be able to represent the real-life system as accurately as possible from the viewpoint of cybersecurity. While there are general-purpose modelling languages that could be used to define such models (e.g. UML, the “unified modelling language”), Foreseeti instead uses the Meta Attack Language (MAL), a meta-language specifically designed to model cybersecurity. Today, Foreseeti primarily uses three different domain-specific modelling languages (DSMLs) written using MAL: coreLang, azureLang and awsLang, with more DSMLs being developed.

These three languages are designed to model three different kinds of systems. AzureLang and awsLang are designed to model systems hosted within the Azure and AWS cloud infrastructure respectively, while coreLang is designed to be a general-purpose language that can model the basic features of virtually any IT-system.

However, in practice most organizations employ a combination of on-premise and cloud infrastructure. In order to accurately model these real-life scenarios SecuriCAD needs to be able to use multiple modelling languages in an interconnected fashion. This paper will therefore attempt to integrate coreLang with azureLang in order to create a sort of hybrid language, with the hope of discovering a method for combining other languages in the future.
2 Scope

The goal of this paper is to provide an analytical analysis of how to integrate MAL-based languages with each other, with the specific focus being the integration of azureLang and coreLang. Actually implementing the proposed solution is not within the scope of this paper. This restriction was made in order to prevent the project from being too ambitious in terms of the time required to reach a satisfactory conclusion.

Foreseeti currently uses three languages, of which two will feature prominently in this paper. While awsLang will be mentioned occasionally (usually to compare it to azureLang, or as an arbitrary stand-in for other MAL-languages), it will not be used as a basis for the main analysis and the discussion.

The reason for this restriction is that azureLang and awsLang have a significant degree of overlap. While azureLang and awsLang each contain classes that the other Lang does not (since the classes represent specific services unique to either Azure or AWS), both the general structure and the implementation details are very similar. Therefore, performing an analysis on both Langs was judged to be superfluous. Whatever solution is deemed suitable with regard to one of the Langs should be equally suitable for the other Lang, and any reference to the general capabilities of azureLang (as opposed to specific capabilities like individual classes) should be understood as applying to both azureLang and awsLang.
3 Method

This paper will employ the following method. Each step will correspond to a section of the paper.

Step 1: Background

The paper will begin by giving some necessary background information on what SecuriCAD does, how it uses the three languages written in the Meta Attack Language (MAL), as well as on the workings and syntax of MAL itself.

Step 2: Requirements

After that, we will discuss what requirements a composite language needs to be able to fulfill. The requirements will be based on interviews with two of the primary stakeholders at Foreseeti, who were asked to identify what they see as the problem with the design right now and what they hope to accomplish with a redesign.

Step 3: Overview of coreLang & azureLang

In the third step, we will do a brief overview of the structure of both coreLang and azureLang. The purpose of this is to prepare for the discussion section by explaining more in detail what type of real-life concepts each of the languages is trying to represent, and how they do it.

Step 4: Discussion

The main discussion will be split into two parts.

The first part will look at other academic work in the area of language composition to see what techniques there are for combining modelling languages. We will then identify which of these options work best in this particular case, with the goal of fulfill as much as possible of the requirement specification detailed in step 2.

The second part begin by discussing how to implement a composite azureLang-coreLang hybrid using the selected technique. Once the specific case of azureLang and coreLang integration has been resolved, we will move on discuss what a general implementation method for arbitrary combinations of languages could look like, what problems each such method has, and how to solve them.
4 Background

4.1 SecuriCAD

SecuriCAD is a software suite intended to, as the name implies, assist in the design of secure IT systems. By using SecuriCAD it is possible to model IT systems by combining various objects, similar to how CAD blocks are used in other CAD software to model e.g. a bridge or a house. Once such a model has been created, SecuriCAD can then run attack simulations on the model in an attempt to find vulnerabilities in the design of the system. The details of how these attack simulations work are not integral to the scope of this paper, but a brief explanation will be provided for context in section 4.2.

The set of objects available inside SecuriCAD depends on which language, or “Lang”, you load into the software. (A “Lang” is defined in this paper to mean a MAL-based DSML.) Specifically, what you load into SecuriCAD is the compiled version of some collection of files adhering to the MAL-specification (again, see section 4.2), which together make up the Lang. Today, Foreseeti primarily uses three Langs: coreLang, azureLang and awsLang. Which of these Langs you want to use will depend on what kind of IT system you’re trying to model.

For general IT systems, the most appropriate Lang would be coreLang [2]. CoreLang is an open-source project maintained and developed mainly by researchers at KTH Royal Institute of Technology and at Foreseeti itself. The purpose of coreLang is to define a set of generalized objects that can be used to model the core features of abstract IT systems, while still being well-defined enough to provide a meaningful representation of real-life systems. For example, some of the objects available in coreLang are Vulnerability, Network, Data, Application, Identity and Privileges. Loading coreLang into SecuriCAD gives access to these objects (and many more), which can then be used to build a high-level approximation of an IT system.

Since most companies and institutions today rely on cloud hosting services to provide large parts of their IT infrastructure, Foreseeti have also developed two proprietary Langs, azureLang and awsLang, designed to model environments in Microsoft Azure and Amazon Web Services, respectively. Compared to coreLang, these Langs are highly specialized, containing objects unique to the specific platform the Lang is intended to model. For example, azureLang contains the object
AKSCluster (see figure 5 in section 6.2), which represents a collection of Kubernetes nodes from the Azure Kubernetes Service, while awsLang contains the object EC2Instance, which represents an instance of the Amazon Elastic Compute Cloud service.

These Langs form the backbone of the SecuriCAD product, providing not only the building blocks used to create the models themselves, but also the logic used by the attack simulations to find vulnerabilities in the models once they have been created. In order to understand the details of how this works, the next section will explain the most important features of the metalanguage used to write these Langs.

4.2 Meta Attack Language

The Meta Attack Language [3], or ‘MAL’ as it will be referred to throughout this report, is a metalanguage intended to be used to create domain-specific modelling languages within the domain of cybersecurity. Within the context of this domain the languages written with MAL can vary significantly in scope, from rather general languages like coreLang to highly specific languages like the two cloud Langs.

The general structure of MAL will be largely familiar to anyone with a background in object-oriented languages, and includes features such as classes, inheritance, and a more specified method-esque feature called an attack step. Designing a new domain-specific language in MAL consists of specifying a set of classes and rules for how these classes relate to each other. These classes will represent whatever concepts are required in order to adequately represent a real-life system within the domain in question. The ‘objects’ referred to in the previous section are simply the instantiated versions of these classes, much like in a normal object-oriented language.

However, creating a model is only one half of what MAL is trying to simplify. MAL-based languages are also meant to contain the information necessary to perform attack simulations on the models. Aside from the modelling interface itself, this is the primary feature that SecuriCAD provides to its users.

SecuriCAD performs these simulations using attack graphs. An attack graph is a graph where the nodes, each of which represents an object, are connected to each other as described by the model. The simulation then uses an algorithm to find the shortest path between the ‘Attacker’ node and any node designated as a ‘high value asset’. (How the Attacker node is connected to the rest of the system will depend on
what kind of attack you’re interested in simulating). If such a path can be found, a vulnerability has been detected in the system. All such vulnerabilities are then reported at the end of the simulation.

In order to facilitate the creation of attack graphs, MAL expects the Lang designer to define several characteristics of the classes the designer decides to implement into their language. The two most important characteristics are *associations* and *attack steps*. Together, these two characteristics describe how the attacker is allowed to move through the system.

### 4.2.1 Associations

Associations define how different classes relate to each other [3, pp. 4–5]. Here is an example of a rather straightforward association, taken from coreLang, to illustrate how this works.

```
Group [memberOf] * <-- MemberOf --> * [groupIds] Identity
```

*Figure 1: Group to Identity association*

The word between the arrows, *MemberOf*, is the name of this association. The capitalized words at each end specify the classes of the associated objects. In this case, the association connects a *Group* object to an *Identity* object. The words inside the square brackets is the handle by which an object can refer to all the objects connected with it through this association. For example, referring to *memberOf* (note the lower-case m to differentiate it from the association name) while in the Identity class will identify all groups that the identity object is currently connected to. Finally, the cardinality of the relation is specified between the bracketed words and the arrows. In this case, the cardinality is many-to-many, represented by the two asterisks.

This association gives a formal description of a real-life security concept, namely that an identity (i.e., a user account) can be a member of zero or more groups, and that a group can contain zero or more identities. Using this association, a group can refer to all identities that are members of it, and an identity can refer to all groups that the identity is a member of.

What associations exist in the language define how the objects may be connected to each other in the models created using the language. However, an association simply
says that there is a relation of some kind between the two classes. For the attack simulations to give meaningful results, the designer of the language must define the logic by which an attacker can move from one associated object to another. This is done through attack steps.

### 4.2.2 Attack steps

Attack steps can be seen as a MAL-equivalent to methods, i.e., they specify some procedure associated with a specific instantiated object belonging to the class in which they are defined. Specifically, an attack step defines how the attacker may move from that attack step (the “parent step”) to other attack steps (the “child step(s)”). The child steps can either belong to the same object as the parent step, or to another object it has an association with [3, pp. 4–6].

However, to be able to move through an attack step the attacker must first compromise that attack step. How to accomplish that depends on the type of the attack step, as it can be either an AND-step and an OR-step. Analogous to AND/OR-gates, an OR-step becomes compromised as soon as the attacker compromises any of its parent step(s), while an AND-step becomes compromised if and only if an attacker manages to compromise all of its parent steps. In other words, a compromised parent steps sends the value True to all its child steps, while an uncompromised parent step sends the False.

When doing a simulation on a model, an Attacker object will be placed in the model and connected to a designated object (which specific object will depend on what scenario you wish to test) by allowing it to compromise an attack step belonging to that object. When the simulation starts, the attacker will begin moving from the initial attack step to any child step(s) it links to. If it manages to compromise any of those attack steps, they will in turn lead to new attack steps, and so on, until the attacker has exhausted all paths through the system and the attack has either succeed or not, which is measured by whether the attacker managed to reach one or more objects defined as a high value asset.

To make this more concrete, let us look at an example of the IAMObject class from coreLang (figure 2). Since classes in MAL don’t have attributes, the majority of a class definition will be made up of attack steps.
The first line of the class definition specifies the name of the class, as denoted by the ‘asset’ keyword. The asset keyword is in this case prefixed with the ‘abstract’ keyword to create an abstract class, i.e. one which cannot be instantiated. Inside the braces we find four attack steps: one defense (denoted by ‘#’), two OR-steps (denoted by ‘|’), and one AND-step (denoted by ‘&’).

A defense is (despite the name) a subtype of attack step. The difference is that the status of a defense does not depend on the status of its parent steps (and therefore no step will ever lead to a defense), but is instead set by the modeler. If a defense is enabled, the attack steps specified by the defense (in this case, the attack step `successfulAssume`) cannot be compromised by the attacker [3, p. 4]. In IAMObject the defense is disabled by default (indicated by ‘Disabled’ in square brackets after the defense name), but it is available to a modeler should they choose to use it.

\[\text{abstract asset IAMObject} \}
\begin{verbatim}
# disabled [Disabled]
-> successfulAssume

| attemptAssume
-> successfulAssume

& successfulAssume @hidden
-> assume

| assume
-> execPrivApps.authenticate,
    highPrivApps.authenticate,
    lowPrivApps.specificAccessAuthenticate,
    readPrivData.identityAttemptRead,
    writePrivData.identityAttemptWrite,
    deletePrivData.identityAttemptDelete,
    managedIAMs.attemptAssume
\end{verbatim}

Figure 2: coreLang IAMObject class definition.

\[1\] In the attack graph, active defenses work just like uncompromised attack steps, preventing the attacker from progressing by sending False to any AND-step it leads to. As a note, this means that a defense should never lead to an OR-step, since an OR-step is compromised as soon as one path leading to it is True. In other words, the presence of an active defense on an OR-step is irrelevant to whether it becomes compromised or not.
As mentioned before, once an attack step has been compromised the attacker will be able to move to all of its child attack steps. The child steps are what’s listed after the arrow, ‘->’. (There is also a “plus arrow”, ‘+>’, which signifies that we want to override an attack step inherited from a superclass.) If the child step belongs to a different object, the attack step will be prefixed by the association tag corresponding to that object. If the child step belongs to the same object, it will be written without prefix (compare line 4, 7, 10 with line 13-19 in figure 2).

To illustrate how this would work in a simulation, let’s imagine an example where an attacker has compromised an attack step which led to attemptAssume on an object of the IAMObject class. (Let’s also assume that the disabled defense is, indeed, disabled.) Since attemptAssume is an OR-step, the attacker would instantly compromise it, leading the attacker to successfulAssume. SuccessfulAssume is an AND-step, meaning that the attacker must clear all parent steps to compromise it. In this case, the parent steps are attemptAssume, which was just compromised, as well as the defense which we decided to leave inactive. SuccessfulAssume has thus been compromised, which lets the attacker continue on to assume, another OR-step.

From assume, the attacker can move on to other objects. All the child steps to assume belong to other objects, as indicated by the presence of association tags prefixing the name of the attack steps. If the IAMObject the attacker just managed to assume happens to have an object connected to it through one of the associations that are specified, the attacker may attempt to compromise the specific attack step on that object. Hopefully, some of those attack steps are OR-steps, allowing the attacker to keep progressing immediately. If they are AND-steps with uncompromised parent steps, the attacker will have to progress through other parts of the system first to see if it can compromise those parent steps as well.

5 Requirements

In order to know how we should integrate the Langs, we need to know what the idea behind re-designing the Langs are. To that end interviews were conducted with two Foreseeti employees, Joar Jacobsson and Andreas Gylling.

Joar Jacobsson is head of the Cyber Domain and Process Development (CDPD) team at Foreseeti, which is the team responsible for Lang development as well as direct customer contact and support. He is not primarily a software developer, but rather a
systems engineer and security expert. In this capacity he also participates in the development of coreLang. For these reasons, he is the person best positioned to know both what features are desirable from the customer and what the composite Lang needs to be capable of from a purely technical standpoint. In addition to this, it should be noted that Joar is the supervisor for this thesis.

Andreas Gylling is a cyber security and threat modelling specialist in the CDPD team whose primary task is to develop tools for customers and Lang attack logic. He wrote large parts of azureLang, making him an excellent person to consult not only for ideas on how the composite Lang should work but also to avoid potential pitfalls in the language design.

5.1 Interview with Joar Jacobsson

Joar’s motivation for wanting to combine coreLang and azureLang is to remedy two limitations with the current Lang design. The first limitation is that each Lang has a distinct area of concern. CoreLang is designed to contain the objects that are found in virtually any IT environment. It contains very advanced logic for these basic features, and is actively reasoned about and updated by a core group of scientists and experts (primarily from Foreseeti and KTH). In comparison, azureLang is designed to model only the services provided by Azure, and the capabilities granted within the Azure ecosystem. This means that Azure does not take advantage of the developments of coreLang, and cannot model with the same granularity as coreLang is capable of.

For example, a Virtual Machine (VM) object in coreLang covers all basic functions and threat vectors of a general VM, while a VM in azureLang covers the Azure-specific functions of an Azure VM (e.g. Azure’s role-based access control (RBAC) actions). However, an Azure VM still possesses all the features (and vulnerabilities) of a general VM, and should reflect that as well. This includes the capability to represent whatever non-Azure-specific software might be installed on a VM, coreLang is designed to do but azureLang is not.

The second limitation is the inability to model multi-domain systems. As mentioned in the section 4.2, you load a Lang file into SecuriCAD to gain access to the modelling objects in that Lang. However, you can only load one Lang file at a time, meaning that you cannot model an environment that belongs to two domains at the same time. (Note that simply changing SecuriCAD to allow you to load two Langs at once
you would be meaningless, since you would not have any associations bridging the gap from one Lang to the other. Therefore, the solution has to come at the Lang design level.)

This is a problem since while virtually all organizations today use a cloud provider (or multiple), it’s hard to imagine a company not also having on-premise infrastructure of some kind, even if it’s only a few laptops and phones and a Wi-fi network. Since their laptops, phones and Wi-fi will be used by the employees to connect (directly or indirectly) to their cloud system, they are a threat vector. This means that modelling only the things hosted on the Azure service provides an incomplete threat assessment when compared to real-life attack scenarios.

5.1.1 Joar’s requirements

The limitations Joar lays out can be summarized as two requirements on the final solution:

1. Specialized objects should have all the capabilities of their more general counterparts defined in more general languages.
2. It should be possible to combine multiple Langs into hybrid-Langs, retaining the objects of both Langs.

It can be noted that the solution to the second requirement should preferably be general enough that it solves not just the problem at hand, but also applicable to any other combination of Langs that may be relevant in the future. As mentioned, many companies use multiple cloud providers, meaning we will need the ability to combine not only coreLang and azureLang but azureLang and awsLang, (a future) gcpLang, etc. You might, for example, want to create a hybrid between androidLang (which does not yet exist) and vehicleLang (which exists as a prototype) to model the cyber threat of connecting your phone to your car. The solution presented in this paper will therefore, if possible, accommodate the creation of arbitrary hybrid Langs.

5.2 Interview with Andreas Gylling

Andreas’s interests in the redesign primarily come down to wanting to minimize duplicate effort. For example: Azure and AWS are very similar platforms, so much of the vulnerability and attack logic in azureLang and awsLang will naturally also be very similar. But since awsLang and azureLang are separate entities maintained and
developed by different developers, each language cannot automatically make use of work done on the other language. Even copying logic from the other Lang still carries development costs, since the namespace is different and the Langs work slightly differently. Additionally, even if you take the effort to copy the logic, changes and updates to the logic do not automatically propagate, meaning you have to do additional development work to keep the languages at par. As a practical example of this, Andreas notes that the network logic in awsLang has been extended to include attack logic that deals with reverse lookups. However, since this was added after the network logic was adapted to fit into azureLang, this feature does not currently exist in azureLang.

The fact that there is no shared basis for Langs in the same domain also means that the Langs grow further and further apart, making this problem worse over time. These small differences also increases the learning curve when a developer switches from one Lang to the other.

The same problem exists between coreLang and azureLang, but with regard to more general features rather than specific attack logic. For example, when coreLang recently changed its vulnerability logic to a more comprehensive system, these changes did not automatically propagate to aws-/azureLang, but instead have to be manually adapted to work with each of the cloud-Langs.

5.2.1 Andreas’s requirements
Andreas’s thoughts can be summarized as a third requirement on the final solution:

3. It should be possible to automatically propagate new features and logic from one Lang within the same domain to another, without breaking the logic that already exists in the Lang.

The degree to which a solution fulfills these three requirements will be the primary metric by which we identify it as a good solution during the discussion.

6 Overview of coreLang and azureLang

Before trying to find a solution that fits the requirements, we will do a quick overview of the current state of the two Langs. The reason for this is to have a broad overview of the structure of the languages, so that we can identify what entities each language is trying to represent. While this has already been touched upon in previous sections,
going through some parts of it will hopefully make the design idea of the languages more tangible for the reader.

6.1 coreLang

Figure 3 is a UML schema showing the inheritance structure of coreLang. As we can see, coreLang is quite small. It contains only 19 classes spread over 4 modules, three of which consist of only a single class. This is, as we will see in the next section, very different from azureLang.

Since the three single-class modules all deal with the same concept (different types of vulnerabilities), let’s put those aside and focus on the main module, coreLang.mal. Here, we find 16 classes meant to represent a variety of concepts used to construct IT-systems, including everything from physical objects (e.g. Hardware), software (e.g. Application), security concepts (e.g. Group, Identity, Privileges), infrastructure (e.g. Network), and information (e.g. Data).
For the purposes of this paper, what classes are available and how they work is not strictly speaking all that important. Instead, the primary takeaway here should be that coreLang is, like the name implies, meant to represent the core cybersecurity features of IT-systems. The intent is that it should be capable of representing, on a high level of abstraction, any object that can be expected to be found within an arbitrary IT-system and which has some security implications.

### 6.2 azureLang

As mentioned, azureLang is significantly bigger than coreLang. With 16 modules and over 200 classes, it is in fact so big that it is infeasible to display the entire language visually as we did with coreLang. Instead, we will try to give limited overview by focusing on some specific parts of azureLang and draw conclusions from them. In figure 4 we see the main module, core.mal.
In this module, we find a couple of classes that we recognize from coreLang, such as Data, Firewall and Credentials. We also find some new concepts, like AccessKey, Subscription and Account, along a few Azure-specific concepts like ServicePrincipalContainer. However, the most important class in core.mal is the abstract class BaseComponent. This class is used to propagate basic logic used by a wide variety of other classes. 13 classes inherit directly from BaseComponent, including the Resource class (an abstract class used to represent Azure services) which in turn has another 31 classes which directly inherit from it. In total, 107 classes inherit from BaseComponent, making it the ultimate superclass of about half of the language. How to let classes keep the logic they inherit from BaseComponent will be discussed in further detail in section 7.2.

Figure 5 shows an expanded example of how a module might be structured. Here we have added the entire kubernetes_services.mal module, as well as 4 classes (three from vm.mal and one from vulnerability.mal) which are intermediate steps between classes in core.mal and classes in kubernetes_services.mal. The Kubernetes module intends to model the Azure Kubernetes Service, and contains objects that relate directly to that service. Vulnerability.mal intends to model the general concept of different vulnerabilities, and thus contains a more general class, VulnerableApplication, representing not a specific service but rather the concept of a vulnerable application. Vm.mal intends to model the Azure version of a general concept (a virtual machine), and thus contains classes representing general concepts which are then extended to represent the specific types of Azure VMs available in Azure.
From these examples we can see that azureLang is both more specific and less cohesive of a language than coreLang. While coreLang focuses on modelling only the most fundamental building blocks used to construct IT systems, azureLang tries to model both fundamental IT concepts and a specific incarnation of those concepts. There is a good reason for this: in order to model the Azure version of a virtual machine, you first need to be able to model a non-Azure virtual machine. Likewise with most other specialized concepts: the model of an Azure application is not complete if it only captures the Azure-specific traits of that application, missing the security traits it gains by virtue of being an application. In other words, azureLang needs what coreLang has, but the opposite is not true.
7 Discussion

7.1 Strategies for DSML composition

To get an idea of how to best combine two different DSMLs we will consult previous academic work within the same topic. This section will be based primarily on [4], which lists three primary categories of techniques for combining domain specific modelling languages: embedding, merging, and extending [4, p. 5].

7.1.1 Embedding

Language embedding is the technique of building a DSML using a preexisting language [5]. Essentially, the concepts of the new language is mapped to concepts within a host language, constructing a new language inside of it [4, p. 7]. The advantages of this approach are fairly obvious: you don’t have to invent your own syntax, and you get access to all of the host language’s features, tooling, libraries, etc.

In our case this approach is not interesting, not least since we already have a host language: MAL. Rewriting azureLang by using coreLang makes little sense: coreLang only contains sets of modelling objects, it does not contain the necessary functionalities we require to define the Azure objects we need - those are MAL-features. Simply put, coreLang is not expressive enough to embed another DSML inside of it.

At most, one could perhaps define azureLang objects in terms of coreLang objects. A modeler could, for example, place coreLang Hardware, Data, and Application objects into the model, connect them together, and use that to represent an Azure VM. The modeler could then do similar things to represent Azure users, networks, and other desired modelling objects. However, this would move the language definition from the language design stage to the modelling stage. Instead of having azureLang, we would have “azure compositions” using coreLang. Again, this does not seem desirable. First of all, we want to retain the higher level of abstraction provided by the more specialized Lang. But, more importantly, doing it this way means we would lose the ability to define attack steps specific to Azure services (since we are

---

2 Note that the original names were “Language embedding”, “Metamodel Merge” and “Metamodel Extension”. They were altered both to avoid the inconsistent language and to avoid confusion, since we have primarily talked in terms of languages and do not want to introduce the term metamodel at this point (not least since the term seems to have no firm definition).
merely approximating azureLang services using coreLang objects), something that is absolutely vital for the language to properly represent reality. For these reasons, embedding is not a viable alternative.

7.1.2 Merging

Language merging is, as the name implies, the process of merging two languages into one [4, p. 6]. The merging is done using shared join points to combine the two languages, where each join point is a pair of objects, one from each language, that are analogous to each other [6, p. 4]. A naïve implementation might simply merge on the basis of identical class names. This is the approach used by the “Package Merge” operation defined in UML2 [7, p. 3]. Using such an implementation, the azureLang Network class and the coreLang Network class would become a single class, also named Network, that inherits all capabilities of both classes. Such a naïve solution would obviously not be desirable in our case, since coreLang Network and azureLang Network are not intended to represent the same concepts.

However, even a more sophisticated implementation would likely not suit our needs, because just like the coreLang and azureLang Network classes are not intended to represent the same concepts, coreLang and azureLang are not intended to model the same domain. While one of the goals of the redesign is to be able to retain the capabilities of more general objects in the specialized azureLang objects (see requirement #1), the same is not true the other way around. While the merged Network class might be an improvement over the azureLang Network class, it would not be an improvement over the coreLang Network. It should still be possible to model a non-Azure environment using coreLang without having a bunch of Azure operations attached to your classes. That this would be a problem becomes even more apparent when you consider that azureLang is not the only Lang that might want to be combined with coreLang. Eventually, you’d end up with one enormous Lang containing the logic for every domain, which is clearly undesirable.

7.1.3 Extension

Language extension is the approach of combining two DSMLs by extending one of the DSMLs with the concepts of the other [4, p. 5]. The objects of the child DSML can either directly inherit from objects in the parent DSML or simply reference them. This approach seems like a great option in our case, for multiple reasons.
First, it presumes a hierarchy of languages, which is exactly what we have: a general language (coreLang) and a specialized language (azureLang). Additionally, it presumes a hierarchy of objects, which we also have: the explicit intent is that coreLang should be able to represent the fundamental building blocks of an arbitrary IT-system, which of course includes the Azure platform and its services.

Second, it would allow changes to coreLang to automatically propagate to azureLang, since azureLang objects would either be extended directly from coreLang objects or connect to coreLang objects (or their children) through associations.

Third, it would not affect coreLang’s ability to exist as a stand-alone language, since the original coreLang objects are not modified.

As such, we can see that a functional implementation of this approach would fulfill all three of our requirements (see 5.1.1 and 5.2.1).

Requirement #1 is satisfied since the azureLang objects would automatically inherit the capabilities of their coreLang parent object, which will be the more general version of themselves.

Requirement #2 is at least partially satisfied, since the new azureLang would automatically be a hybrid Lang consisting of the old azureLang and coreLang, and would therefore by necessity include all coreLang objects. However, to fully satisfy the requirement we want to present a general approach to combining Langs, not just with coreLang but with each other. This will be discussed in further detail in section 7.2.2.

Finally, requirement #3 is also partially satisfied since any new features that are added to coreLang would automatically propagate to azureLang by virtue of azureLang being based on coreLang. However, this solution does not automatically propagate changes and features between e.g. azureLang and awsLang. This will be discussed further in section 7.2.1.

### 7.2 Implementation

Having settled on an implementation strategy, let’s begin sketching how the actual implementation would be done. The basic idea is straightforward: we will simply make each azureLang object an extension of the coreLang object that most closely corresponds to it. If there is no corresponding coreLang object, there is no reason to
change anything. However, since the ambition is for coreLang to contain all objects necessary to model a basic IT-system, this should be a rare occurrence.

As an example, let’s look at the current `SecurityPrincipal` class from the azureLang RBAC (role-based access control) module:

```plaintext
asset SecurityPrincipal extends BaseComponent
{
    let possibleRbacActions = roleAssignments...
    let keyVaultAccessPolicies = accessPolicies...
    let allActions = possibleRbacActions...

    | assume @traceLeft
      -> performActions,
      allActions().resources.authenticated,
      possibleRbacActions().resources[ContainerRegistry].acrLogin,
      groups.groups*.assume,
      roleAssignments.calledByPrincipal,
      roleAssignments.roles.satisfy

    | performActions @hidden
      -> allActions().performAction
}
```

*Figure 6: AzureLang SecurityPrincipal definition. Edited for clarity and space.*

To integrate this object into coreLang, we need to find the object which most closely mirrors its intended purpose. In this case that would be the Identity object, a sub-object of the IAMObject class which we previously saw in section 4.2.2. The definition of these objects can be found below.

Note that both the IAMObject class and the Identity class have attack steps named assume. Note that the Identity assume step uses a plus-arrow (\(\rightarrow\)) rather than the normal arrow (\(\rightarrow\)). This indicates that the logic of the assume step in the child class will be added to the logic of the assume step in the parent class, rather than overwrite it. Since SecurityPrincipal also has a step named assume, we will need to change this from using the normal arrow to using the plus-arrow in order to not lose the logic from the assume steps belonging to IAMObject and Identity.
This means that the altered class definition for SecurityPrincipal will be very similar to the original one, with one potential caveat. The caveat is that SecurityPrincipal currently extends from the BaseComponent object. As mentioned in the section 6.2, most objects in azureLang extends either BaseComponent or Resource (which in turn extends BaseComponent). When we change SecurityPrincipal to extend from Identity rather than BaseComponent, we will need to incorporate that logic in some other way to avoid losing current azureLang functionality.
One solution would be to just add the logic here, straight into the SecurityPrincipal definition. However, that would mean that we would also have to add the logic to every other object that extends from either BaseComponent, Resource, or any of their children. That both clutters the class definitions and requires a lot of extra work so we would prefer to find a solution that lets us assign this logic to multiple objects at once, if possible.

One solution that lets us do that would be adding multiple inheritance to MAL, allowing us to extend from two classes at the same time and gaining all capabilities of both parent classes. However, multiple inheritance is not without pitfalls (notably the diamond problem [8, p. 109]), and adding that functionality would require significant amounts of work on the MAL compiler instead, likely more work than it would be to manually add the logic to the current classes.

Another option would be to add a new class higher up in the hierarchy which contains this logic. As seen in figure 5, IAMObject actually does not extend from anything, and in fact most coreLang objects don’t, as can be seen in the coreLang UML schema. This means that it would be possible to add a class to coreLang which contains the missing logic, from which IAMObject and other classes would then extend. This new class would essentially fulfill the same purpose as BaseComponent currently does in azureLang, but from a few steps higher in the hierarchy. Of course,
this solution is not without downsides. Since CoreLang is meant to act as a base for other Langs (e.g. awsLang), and therefore should contain only the core objects and logic that is generally applicable to any IT system, we can’t justify adding it to coreLang just because it seems convenient in the moment. We can only justify adding the logic if it is so general that it actually qualifies on its own merits. Looking at the definition of BaseComponent below, we can see that the logic is, in fact, very general. While authenticated might be somewhat less general, read, write and delete are features that are applicable to virtually anything software related. Whether that is enough of a reason to justify adding it to coreLang is open for debate, and it would still exclude some classes which couldn’t reasonably include these attack steps (e.g. the Hardware class).

```java
abstract asset BaseComponent
{
    | attemptWrite @hidden
        -> write
    | attemptDelete @hidden
        -> delete
    | attemptRead @hidden
        -> read
    | read
    | write
    | delete
    | authenticated @hidden
}
```

*Figure 9: BaseComponent definition*

Putting aside the issue of incorporating the BaseComponent logic, we at the very least have a basic schema for how to extend azureLang classes from coreLang classes. However, as noted previously (notably by requirement #3), we would like to be able to automatically propagate changes between peer languages (e.g. between awsLang and azureLang) in order to minimize the amount of duplicate implementation work the current design requires. Extending azureLang directly from coreLang would mean we have to do it all over again when tackling awsLang, and doesn’t do anything to solve this problem in the future.
7.2.1 Propagating changes between peer languages
The solution to this problem is quite simple: we introduce an intermediate language between coreLang and azure-/awsLang. This new Lang, which would reasonably be named cloudLang, will contain any and all logic that is not specific to either cloud platform. That is, any logic that doesn’t depend on the intricacies of a specific service within Azure or AWS should be moved to cloudLang.

Introducing this intermediate language solves the problem of having to re-implement logic that has previously been implemented to the other language. Instead of adding a new feature to awsLang and having to manually move it to azureLang, we would now add it to cloudLang and have the change propagate downwards to both sub-Langs.

Since awsLang and azureLang have a lot of similar (but not always identical) logic for similar services, it is highly recommended that both Langs are reworked at the same time. For each section, the relevant logic from both awsLang and azureLang should be analyzed. If there are discrepancies between the two implementations, one (preferably the best one) should be kept, and the other discarded. Note that this might be trickier than it sounds, since keeping the logic from one Lang might break logic that relies on this implementation in the other Lang.

Since most (if not all) services offered on one of these cloud platforms will have a counterpart in the other cloud platform, it follows that most (if not all) logic used to describe one cloud platform will also be relevant in describing the other cloud platform. This means that once this process is complete, the vast majority of logic will be located in cloudLang rather than in aws-/azureLang. In fact, it is quite possible that a lot of classes in aws-/azureLang will end up being empty re-definitions just for the sake of re-naming a generic object to the AWS/Azure-specific service name.

7.2.2 Hybrid logic
While the primary goal of this paper was to find a way to integrate coreLang and azureLang, a secondary goal was to find a generalized formula for integrating multiple languages into one model, since the reality is that most real-life systems do not contain themselves to a single domain. This means that we want the ability to create models that contain both azureLang and awsLang, or a model that contains
e.g. azureLang and adLang (a planned Lang for modelling the Windows *Active Directory* service), or any other arbitrary combination.

To properly model such systems, it seems inevitable that we will need logic in one language (say, azureLang) that references objects or attack steps from another language (say, awsLang). (We will call such logic “hybrid logic” going forward.) However, this presents us with a new problem.

Say, for example, that we add an association which connects an azureLang class to an awsLang class. We then place this association in one of the azureLang files, and try to compile the language. At this point, one of two things will happen. If the compiler finds the awsLang files in the current directory it’s trying to compile, everything will work just fine. A compiled language file will be created, which can then be loaded into SecuriCAD where you will now be able to connect the two classes to each other through the new association. However, if the awsLang files are not present in the current directory the compiler will return an error since the association refers to an object that it cannot find. This is an issue because it should be possible to use each language as a stand-alone. You shouldn’t need to always compile it together with all other languages it happens to share hybrid logic with. So, we need a solution more sophisticated than simply putting new hybrid logic into existing languages.

### 7.2.2.1 Option 1: Indirect references

The first option would be to refer to other languages only indirectly. Assuming that azureLang and awsLang are redesigned, and that future Langs will be based on coreLang to begin with (which seems prima facie reasonable, since all the advantages gained from basing azureLang on coreLang will be equally applicable to any future Langs), these Langs will all share coreLang as a common frame of reference. All the associations and attack steps that exist in coreLang can be referenced in the sub-Langs, without creating any compilation problems. And since child classes inherit the attack steps and associations of their parent classes, it would be possible to connect child classes to each other using their parent’s associations.

To give an example, let’s look at the *CanAssume* association from coreLang.

![Identity](parentId) * <-- CanAssume --> * [childId] Identity

*Figure 10: coreLang *CanAssume* association.*
As we can see, this association allows us to connect one identity object (denoted parentId) to another Identity object (denoted childId). When we re-defined the azureLang SecurityPrincipal class earlier in this section (figure 6), it was the Identity class we extended from, meaning that SecurityPrincipal can utilize this association by virtue of being a subclass of Identity.

This means that if we have an awsLang class that is analogous to SecurityPrincipal (let’s call it awsPrincipal) we can connect a SecurityPrincipal object to an awsPrincipal object. Using this connection we can then refer to attack steps in the other language. As seen in the Identity class definition (figure 5), the assume attack step leads to to `parentId.attemptAssume`. If we have a SecurityPrincipal object connected to an awsPrincipal object, where the awsPrincipal object is indicated as the parentId-side of the association, reaching assume on the SecurityPrincipal object will direct us through the association to the awsPrincipal object and perform attemptAssume (which in turn leads to the assume step). Specifically, it should be noted that the assume triggered is not the one in Identity (despite it being reached through an Identity-association) but that of the actual object type, so it would include any logic specific to awsPrincipal’s assume step.

Using this method, we can see that it is indeed possible to create hybrid logic without ever referring to any objects that aren’t a part of the language natively. However, this method also has several significant drawbacks.

The first drawback is that this system will be very hard to maintain. When changes are made to one language it might affect the functionality of this kind of hybrid logic. Often, this effect might not be immediately obvious, both because the logic will be located in another Lang and because the logic only refers to the objects indirectly. Rigorous testing would help detect unintentional changes to some degree, but would require testing all combinations of languages every time a change is done which also isn’t desirable.

The second drawback is that you are limited to the associations and attack steps that are already defined in coreLang. For example, the CanAssume association refers to the participating Identities as a parent and child identity. While you can use this association even if the two identities you want to connect aren’t actually a parent and child identity, it would be semantically confusing. Furthermore, what if you want to
connect two objects that coreLang doesn’t have an association for? You would be forced to either add this association to coreLang (without coreLang actually having any use for it), or to refrain from adding that logic altogether.

For attack steps, the issue is even bigger. Even if you have an appropriate association to the object you want to interact with, that object might not have an appropriate attack step in coreLang. For example, say the object has two attack steps in coreLang, both of which trigger multiple other attack steps, but you only want to trigger a specific attack step on the sub-language object. Or, say that you want to trigger the assume step when a SecurityPrincipal object is connected to an awsPrincipal object, but not when connected to an adPrincipal object (assuming such an object would exist). That kind of fine-tuning would be unavailable to you with, but will almost certainly be required in order to write languages that properly reflect reality.

Given these issues, it seems clear that indirectly referring to objects through coreLang is not a viable alternative either. To avoid unintentionally triggering unwanted attack steps, we need to be able to write custom logic that directly refers to objects and attack steps in other languages.

7.2.2.2 Option 2: Make compiler less strict

Another option would be to change the MAL-compiler itself. Since the original problem was that the compiler will not allow us to refer to things that are not present in the language we’re currently trying to compile, why not loosen this restriction in some fashion? That would let us simply add hybrid logic as we want, with full control of what objects we refer to and in what way.

Of course, there is a reason the compiler crashes when we refer to nonexistent objects. We want compile-time crashes because they inform us of mistakes made during development. Changing the compiler to no longer react to logic that refers to objects that don’t exist will just lead to poorer code. What if the compiler thinks we’re referring to an object in another language, but we actually just made a typo? We could have the compiler warn us every time it finds a reference to something that doesn’t exist, but warnings are easily ignored – especially if you expect a number of warnings to pop up, as you would if your language contains hybrid logic.

A better way of doing it would be to add a flag that would indicate to the compiler that an object may or may not be present at compile time. If it encounters a reference
to a nonexistent object without this flag, the compiler will crash just like before, giving us the advantage of compile-time crashes but with increased flexibility.

```plaintext
SecurityPrincipal [azPrincipal] * <-- example --> * [awsPrincipal] $awsPrincipal
```

*Figure 11: example of how a nonexistent-flag ($) could be used in an association.*

In the above figure, $ is used as the flag symbol on a theoretical azureLang-to-awsLang association. When the compiler encounters such a flag, it will know that whatever follows is hybrid logic that isn’t always going to be relevant. If during a particular compilation it cannot find the awsPrincipal class, it will simply discard this association and proceed, perhaps outputting a warning for the sake of transparency. It would work the same for references to objects and attack steps.

However, this option also has its downsides. The most obvious downside is having to change the compiler, rather than solving the problem through language design. The inner workings of the MAL-compiler is outside of the scope for this paper, but this kind of change is likely less trivial than it seems.

Another downside is that this approach would clutter the base languages quite significantly. As stipulated, we want to be able to create hybrid languages out of any arbitrary combination of Langs. This means that each language will eventually contain hybrid logic relating to multiple other languages. At that point, a class might contain four hybrid attack steps and one native attack step, for example. While the nonexistent-flag will tell the compiler to ignore this logic, it will nonetheless still be there. Whenever we compile a language as a stand-alone, the compiler will have to sift through a bunch of hybrid logic that isn’t currently relevant, increasing the time it takes to compile the language file.

Additionally, it makes it much harder to read the code and quickly get a grasp of what’s going on. If the reader is only interested in the base Lang, the hybrid logic is just distracting. But even if they are interested in some specific hybrid logic, it’s likely still confusing. While the presence of a flag would tell you a specific line refers to hybrid logic, it might not be immediately apparent what combination of Langs it’s referring to. (This info could for example be provided with a comment, but that would require all developers to diligently comment every line of hybrid code they write.)
So, while a compiler change could fix the problem it doesn’t seem like a silver bullet. Preferably, we want to be able to easily separate the hybrid logic from the base logic when it’s not relevant. Let’s explore a solution that does just that.

7.2.2.3 Option 3: Separate files

The previous solution attacked the problem from the compiler end. This solution attacks it from the other end: if the problem is that the compiler won’t let us have hybrid logic in the base language, why put it there in the first place?

Instead, the hybrid logic could be defined in a separate file, to be included only when you actually want to compile two languages together. This file would, in practice, constitute the actual “hybrid language,” keeping both base languages clean of any references to external objects.

For associations, this would solve the problem quite neatly, and in fact be a significant upgrade over putting the association into either of the base languages since there would be no ambiguity about where such associations can be found. (It makes no difference for the MAL-compiler whether an azureLang-to-awsLang association is defined in azureLang or in awsLang, but gathering them in one designated place makes it much clearer for a developer.)

For attack steps, it’s not quite as straightforward. To add hybrid logic to class A, you would have to re-define class A in the hybrid file, including both the current logic and the new hybrid logic, and then remove the original class A from the base Lang. Otherwise, the compiler will detect two definitions of the same class and, not knowing what to do, promptly abort.

This is obviously not a very elegant solution. Not only is it very cumbersome to have to manually edit out portions of the base languages every time you want to compile with the hybrid file, it would also be a nightmare to maintain. Every time you change a class in a Lang, you will have to check the hybrid file (or more specifically, *every* hybrid file that concerns that Lang) to see if the class has a hybrid definition and alter that too. While it would work, it is clearly not an optimal solution. However, using a separate file does have multiple advantages. If we can make the process of adding logic to existing classes a bit less cumbersome, it seems like quite a good solution. And since we already considered a compiler change, why not another one?
7.2.2.4 Option 4: Separate files + compiler change
Instead of changing the compiler to loosen the restrictions on what objects we are allowed to refer to, we could instead change the compiler to accommodate adding hybrid logic in a separate file. The problem with the separate file solution was that the compiler doesn’t allow us to re-define a class without first eliminating the old definition. Obviously, only allowing a single definition of each class is a very reasonable limitation, but what if we instead of redefining a class we were allowed to expand a class that previously exists?

To illustrate, let’s once again use SecurityPrincipal as an example. First, let’s remind ourselves what its (coreLang-integrated) class definition looked like.

```asset SecurityPrincipal extends Identity
{
    let possibleRbacActions = roleAssignments...
    let keyVaultAccessPolicies = accessPolicies...
    let allActions = possibleRbacActions...

    | assume
      + performActions,
      allActions().resources.authenticated,
      possibleRbacActions().resources[ContainerRegistry].acrLogin,
      groups.groups*.assume,
      roleAssignments.calledByPrincipal,
      roleAssignments.roles.satisfy

      | performActions @hidden
        -> allActions().performAction
}
```

Figure 12: coreLang-integrated SecurityPrincipal definition

Now, say we want to add the same type of SecurityPrincipal-to-awsPrincipal association we used in the flag example (but without the flag, of course), and use that association to trigger the assume attack step on any awsPrincipal objects connected to our SecurityPrincipal objects if their own assume step has been compromised. We define this logic in a separate file, using expand as a keyword before the asset declaration to flag to the compiler that this an expansion of a class that already exists (if it doesn’t, something has gone wrong and the compiler should crash). Note that we use the plus-arrow (+>) to add the logic to the existing assume logic, rather than overwrite assume.
This method of combining a separate file with a compiler change solves both of the issues that we identified with each method individually. Firstly, this approach means that all hybrid code is contained in a single location, where it can be easily located and maintained, and where it doesn’t clutter the original language with logic that often won’t be relevant. Secondly, it means you don’t have to re-write the entire class definition, but instead just add the hybrid portions. This significantly reduces the amount of code you have to both write and read when developing hybrid languages, and means you don’t have to update every hybrid definition of a class whenever you change the base language version of the class. Finally, it also makes it obvious which logic is hybrid logic and which belongs to the base Lang, whereas before you would have had to compare the hybrid definition to the original definition to see the difference.

These advantages seems to make this the preferred solution. The only true downside seems to be, once again, the need for a compiler change. However, while this solution seems very promising, there are still some question marks we need to clarify, specifically with regards to conflicts between multiple hybrid-files.

7.2.2.5 Resolving Conflicts

7.2.2.5.1 Namespace conflicts

First, we have the issue of namespace conflicts. As discussed, the MAL-compiler doesn’t allow duplicate names for objects. Additionally, it also imposes restrictions on duplicate association-tags, since it needs to keep track of which association we are using when we connect objects or refer to attack steps. This is not a big problem within a single language, since there isn’t really a reason for a language to have multiple objects that share names. However, the same is not true when we want to use multiple languages in the same model, especially when talking about more general objects like Files, Networks, Data, etc. Indeed, awsLang and azureLang as
currently designed both contain “Network” objects. Even if this object will almost certainly be removed in favor of the analogous coreLang object (or at least moved to cloudLang), it seems quite likely that there will be namespace conflicts between objects contained in different languages in the future. Since we want to be able to create arbitrary language constellations, we should try to preempt that issue before it arises.

There are two obvious solutions to this problem: either to adopt a strict naming convention, or to implement package functionality into MAL. A naming convention would simply mean that developers name all designated objects (at a minimum file names, classes and attack steps, possibly including associations) according to a pre-defined ruleset which ensures that no conflicts will occur. Such a convention can of course take a variety of forms, but the simplest version would be to just take the language prefix (so core for coreLang, azure for azureLang, etc.) and use that as a prefix for all objects within that Lang.

A package system is essentially an implemented version of the same idea: the developer would define a package (say, the azure package), and namespace conflicts can only occur with regard to objects defined within the same package. This means that the package system takes care of keeping the namespaces separate, instead of having the developer manually name each object according to the convention.

The primary difference between these solutions is where the workload is located. Implementing a package system will take some non-trivial amount of initial effort, but once implemented will handle namespace conflicts with almost no day-to-day effort on the part of Lang developers (except for needing to define packages once per file). On the other hand, a naming convention requires no implementation at all but instead requires that the Lang developer constantly follow the naming rules. A naming convention might also offend some aesthetic sensibilities, since each object name would be somewhat cluttered by having it include the name of the Lang (or some similar identifier) which the package system does not require.

7.2.2.5.2 Definition conflicts

The second issue is that of definition conflicts. Say for example that we want to compile three Langs together, A, B and C. In the folder we have all three Langs, as well as A_B_hybrid.mal, B_C_hybrid.mal and A_C_hybrid.mal. If A_B_hybrid.mal
and A_C_hybrid.mal both expand the same object, which should take precedence?
What if one hybrid file overwrites the base logic of an attack step (i.e. uses the normal arrow, ->, rather than the plus arrow, +>), when the other extends it? What if both hybrid files overwrite the base attack step? Or what if both hybrid files add attack steps with the same name?

There are likely even more corner-case scenarios than the ones mentioned above (not least because the complexity increases the more languages you have in your hybrid constellation), but let’s at least try to give answers to these ones. It seems necessary to allow multiple hybrid files to expand the same object, since it is virtually guaranteed that the same objects will be relevant in multiple hybrid combinations. If multiple hybrid files expand the same object and there is no conflict (e.g. if they one adds an attack step and the other expands one that already exists), there is no problem: we simply combine the additions with the base class and move on.

If there is a conflict, it gets more complicated. If one file expands a base attack step with additional logic while the other file overwrites that same attack step, it might often be fine to combine these two changes, but the risk is that the hybrid Lang that expanded the attack step relies on the base logic still being present and thus might break when you overwrite it with the logic from the other hybrid Lang. In that sense, the question of what to do when two hybrid languages overwrite the same attack step might be easier, since we know that both rely solely on the logic present in the hybrid file, and you could therefore combine it and know that everything they need is present.

On the other hand, the main reason to overwrite an attack step would be to remove some basic logic with a more hybrid-tailored version, or some basic logic that for whatever reason isn’t relevant in the hybrid language. So, combining two overwritten attack steps might re-introduce some of the logic you wanted to overwrite in the first place. The fact that both alternatives have downsides seems to imply that it might be more reasonable to disallow overwriting attack steps in hybrid languages altogether, and instead only allow expanding existing attack steps.

Whether this is a reasonable restriction largely depends on whether or not its actually desirable to overwrite existing attack steps. In reality this feature might be relevant only rarely, in which case the best choice is likely to not allow it at all. As for
what to do when two hybrid files add attack steps with the same name, it seems clear that this should not be allowed because combining the logic of both means that triggering the attack step from one language will invoke logic from the other language that isn’t guaranteed to be related (they might have similar names without referring to the exact same thing). For this reason, it seems prudent to include hybrid-defined attack steps (and perhaps all hybrid logic, to be consistent) in the naming convention.

8 Conclusion

In this paper we have analyzed how to combine two currently separate DSMLs, written in the Meta Attack Language (MAL), into an integrated system. The specific MAL-based DSMLs analyzed was coreLang, a general-purpose cybersecurity modelling language, and azureLang, a language designed to model specifically the security aspects of the Azure cloud platform. The specific purpose was to find a way to integrate these as well as future DSMLs into the same coherent system, allowing arbitrary combinations of languages designed to model any cybersecurity domain.

Given the hierarchical structure of the DSMLs and their objects, we found that an inheritance approach was the preferred integration technique. However, given that the specialized language has a peer language, awsLang, we suggest a three-tiered approach instead of directly inheriting from coreLang. This three-tiered approach would have coreLang at the top, an intermediate language (cloudLang) that contains all logic shared between the specialized languages in the middle, and azureLang along awsLang at the bottom. This solution enriches azureLang (and awsLang) with all the capabilities of coreLang, allows azureLang and coreLang to be used in the same model, makes any updates to coreLang automatically propagate downwards to both coreLang and the peer languages, and makes any update to the shared cloud logic propagate downwards to both peer languages.

Additionally, we propose a generalized process for creating future hybrid languages. Through a change to the MAL compiler that allows a language designer to define new attack steps to an existing object, we can define any additional logic needed for some arbitrary combination of languages in a separate file. This file will be the de facto hybrid language, containing all logic not already present in the base languages. This separation makes it easy to keep track of, change, or extend whenever needed. As a
part of this process we also propose either the implementation of a package system, or the adoption of a naming convention, in order to preemptively avoid the namespace conflicts that would likely arise when trying to compile multiple Langs together.
9 Bibliography


