Towards Language Support for Model-based Security Policy Engineering

Peter Amthor[a] and Marius Schlegel[b]
Technische Universität Ilmenau, Germany

Keywords: Software Engineering, Security Engineering, Security Policies, Security Models, Specification Languages, Domain-specific Languages, Automatic Translation.

Abstract: Software engineering for security-critical systems is based on manual translations between languages from different domains: an informal security policy is translated to a formally verifiable model, and further to actual source code. This is an error-prone task, put at the risk of losing hard-acquired correctness guarantees. To mitigate this problem, we argue for a methodical support by domain-specific languages and tools. We present ongoing work on two languages that substantiate this thesis, including their usage in a practical setting, and discuss the benefits from combining them with appropriate tool support.

1 INTRODUCTION

Building secure application and systems software imposes increasing effort on security engineers, software developers, and experts for the respective application domain. As a common prerequisite among all these stakeholders, security requirements have to be formally guaranteed from early on in order to uphold *security properties* regarding protection of information confidentiality, system integrity and service availability over all levels of abstraction throughout the whole software engineering process. For this purpose, a specialized *model-based security policy engineering* (MSPE) process makes use of formal models (Basin et al., 2011; Vimercati et al., 2005) to unambiguously specify, verify, and implement all security-critical functionality, which we call a system's *security policy*.

We can distinguish three principal steps in MSPE: (1.) In *Model Engineering*, a formal representation of a security policy is designed in a way that allows to reason about security properties; (2.) During *Model Analysis*, formal methods are applied to verify these properties and correct any errors found; (3.) Finally, in *Model Implementation*, the formal security model created and verified so far is translated in actual code for implementation. All of these steps require different, isolated languages to satisfy their individual requirements: While initially a security policy is defined by requirements engineers and application experts, usually based on semi-formal or natural language (Xiao et al., 2012; Mitra et al., 2016), model engineering requires its translation to mathematical artifacts understandable by analysts. Then again, the formal methods used in model analysis (Crampton and Morisset, 2012; Stoller et al., 2007) require a machine-readable representation of the mathematical model (such as for semi-automated model checkers), which finally needs to be translated into a functional software specification (OASIS, 2013; Ben Fadhel et al., 2016), and, finally, a programming language. These translations are required to preserve relevant knowledge from the previous process step; yet up to date, they are largely manually work – and thus error-prone.

In this paper, we propose a holistic approach: the use of interoperating domain-specific languages (DSLs), with syntax and semantics tailored to streamlining the above translations. Ideally, such DSLs enable an automation of these tasks. We substantiate this position by presenting two DSLs to seamlessly integrate in MSPE (see Fig. 1): REAP (*relationship-based language for assisted policy design*), a visual language for model engineering (§2), and DYNAMO (*dynamic security model specification language*), a machine-readable language for model analysis and automated source code generation (§3).

2 REAP

The fundamental motivation behind a specification language for model engineering is to unite the knowledge and skills of both security engineers and application

[a] https://orcid.org/0000-0001-7711-4450
[b] https://orcid.org/0000-0001-6596-2823

Amthor, P. and Schlegel, M.
Towards Language Support for Model-based Security Policy Engineering.
DOI: 10.5220/0009893205130521
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 513-521
ISBN: 978-989-758-446-6

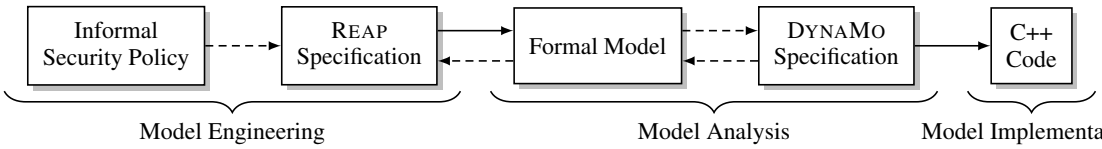


Figure 1: Languages for security policy representation in model-based security policy engineering. Dashed arrows indicate a manual translation, solid arrows an automatic or semi-automatic conversion.

experts. It should allow both groups to easily interact in this part of a software engineering process, while retaining the focus on mathematical formalisms later required by specific MSPE methods. As a first step towards these goals, REAP is a visual specification language that defines the mathematical *structure* for a security policy rather than its concrete *rules*. REAP is supposed to specify the formal framework for any security policy rather than its later functionality. This is because we believe that, once an appropriate model structure has been found, textual representations offer a more suitable language to express policy rules.

In the remainder of this section we discuss the requirements of REAP, which determine both its syntax and semantics (§2.1); the language design decisions following from these requirements (including a running example) (§2.2); and the merits of using the language in the overall MSPE process (§2.3).

2.1 Language Requirements

To devise language requirements, we identified recurring patterns in the established formalizations of common security policy classes. In model terms, most of them fall in the domain of access control (AC) (Basin et al., 2011; Vimercati et al., 2005), including identity-based (Harrison et al., 1976; Sandhu, 1992), role-based (Sandhu et al., 1996; Ferraiolo et al., 2007), or attribute-based (Jin et al., 2012; Servos and Osborn, 2017) models. A specific range of mathematical artifacts is used as building blocks of these models, including (1.) Basic set theory: sets of unique identifiers, their relationships such as subset, power set, difference, union, intersection, cartesian product; (2.) Discrete functions (mappings) based on these sets; (3.) Semantic meta-information about the above artifacts, which is not captured by their structural definition.

The semantics of these artifacts dictate the following functional language requirements with respect to expressiveness: (1.) It is possible to define sets of unique identifiers, which we call *atomic sets*. Their elements (e. g. identifiers of human users) do not have any observable inner structure. (2.) Any language expression is used to define a relationship between sets. This may be e. g. a mapping of user- to role-identifiers, or a relation over roles that indicates a role hierarchy. (3.) An expression may be used to define a new set, called a *derived set* since its elements reference elements of other sets (e. g. a cartesian product).

In addition to these requirements, the intended usage in MSPE demands the following non-functional properties: REAP specifications should (1.) allow for as much precision as needed through alternative syntax for complex expressions, (2.) be as intuitive and readable as possible, (3.) allow to ignore unneeded information about the underlying formalisms to minimize visual complexity, (4.) enable tool-support for modeling decision support and model translation to formal artifacts usable in subsequent MSPE steps.

To satisfy these functional and non-functional requirements, REAP supports an iterative approach to model engineering: it offers three levels of expressiveness, which may be combined as needed. Here is an overview of language features in these levels:

REAP$_0$ Define sets of atomic identifiers and their power sets, connect them through mappings or binary relations.

REAP$_1$ Express complex definitions of sets derived from atomic sets, define aliases for sets.

REAP$_2$ Annotate all visual elements with meta-information, such as logical constraints.

In §2.2, we will discuss syntax and semantics of these language levels by example.

2.2 Language Design

The basic idea of REAP is to visualize formal model components as a graph, consisting of connected subgraphs for individual expressions. As already introduced, we distinguish between three levels of expressiveness that support an increasing amount of model complexity. Since this also means that more expressive policies lead to more complex graphs, our requirement to minimize visual complexity results in policy graphs that are still valid in any of the more expressive language levels. Thus, language design supports a top-down model engineering approach, where expressions for increasingly complex policy semantics may be added to an existing graph on demand.

REAP$_0$. A security policy in REAP$_0$ is a connected, directed graph whose nodes represent sets. These may be either atomic sets, drawn as a circle (◯), or relationships. Relationships may be used to represent either mappings, drawn as a block arrow (⟩), or relations, drawn as a hexagon (⟨⟩). Generally, all nodes in REAP contain up to two labels: a name of the respective model component and a specialization label that adds more information for policy design and refinement. In REAP$_0$, only name labels are used.

Edges connect relationship nodes with atomic set nodes to indicate the operands of a relationship, such as the domain and co-domain of a mapping (expressed by ◯→⟩→◯) or the defining sets of a relation (expressed by ◯→⟨⟩→◯). A REAP$_0$ policy is thus a bipartite graph.

While the presence of an edge is part of the semantics of its adjacent relationship node, its orientation indicates its operand ordering. Based on the observation that most relations used in security models are binary, it suffices to distinguish between a first ("input") operand, given by an incoming edge, and a second ("output") operand, given by an outgoing edge. This also applies for the domain (input) and co-domain (output) of a mapping in a similar way. If required, the power set edge (→→) identifies a power set used in the definition of mappings and relations.

Example 1. Assume a security policy for a hospital information system (HIS). We first identify necessary unique identifiers and their respective base sets: *roles* (Ⓡ) identify organizational functions, *users* (Ⓤ) identify humans, and *permissions* (Ⓟ) identify technical capabilities. The mindset, according the general idea of role-based access control (Sandhu et al., 1996), is that both users and permissions are associated with roles, which may be granted or revoked to control access. Both associations my be represented by relations:

Ⓤ→⟨UA⟩→Ⓡ and Ⓟ→⟨PA⟩→Ⓡ.

This enables a policy engineer to leave unspecified, at least for the moment, how many roles are assigned to a user (and likewise, how many permissions to a role). Another relation is used to relate general (e. g. doctor) to derived roles (e. g. surgeon) in a role hierarchy *RH*.

To enable separation-of-duty semantics, we add the concept of dynamically assigned roles: A mapping

Ⓤ→⟩roles⟩→→Ⓡ

describes subsets of roles available to a user active at a given time. At this point, it becomes obvious that at the same time a user, such as a doctor, may fulfill functions of more than a singular role – such as e. g. as a physician and a medical director at the same time. Consequently, we need the power set edge (→→) that explicitly indicates multiplicity of co-domain-elements, which should be sets of elements from *R*.

Since a REAP specification needs to be connected, we can assemble the graph in Fig. 2a from these subgraphs. With tool support, this graph can be easily translated to mathematical definitions of (1.) sets: U, R, P; (2.) relations: $UA \subseteq U \times R$, $PA \subseteq P \times R$, $RH \subseteq R \times R$; (3.) mappings: $roles : U \to \mathcal{P}(R)$. This example illustrates how the basic language level REAP$_0$ can already express a practically significant range of policies, in particular, RBAC$_1$ (Sandhu et al., 1996).

REAP$_1$. In REAP$_1$, we allow for more complex formal structures: derived sets created by set composition operators, identity relationships to create aliases for sets, and cascading of set composition operators. A REAP$_1$ security policy is a graph composed from valid REAP$_0$ subgraphs and the following expressions.

Derived sets are denoted by a set composition node, which contains a set algebra operator symbol as its specialization label, such as cartesian product ($\times$), partial or total inclusion ($\subset, \subseteq$), union, intersection and difference ($\cup, \cap, \setminus$), or power set ($\mathcal{P}$). Set composition nodes are draw as a diamond (◇) and connected in the same mindset of input and output as with mappings and binary relations. However, since operators such as $\times$ or $\cup$ are not restricted to single input and output nodes, we need to generalize edge orientation semantics: Any incoming edge of a set composition node connects operands, while an outgoing edge references the resulting set.[1] This way operators may be cascaded to allow subgraphs for complex set definitions.

To preserve readability of REAP$_1$ specifications, we also allow for an easy way to hide this complexity whenever undesired: for this purpose, we introduce identity relationships. These are denoted by an edge type that indicates equality of both adjacent nodes (which are also called *aliases*). To distinguish them from regular input-output-edges, we draw them dashed and, due their symmetric semantics, undirected (- - -).

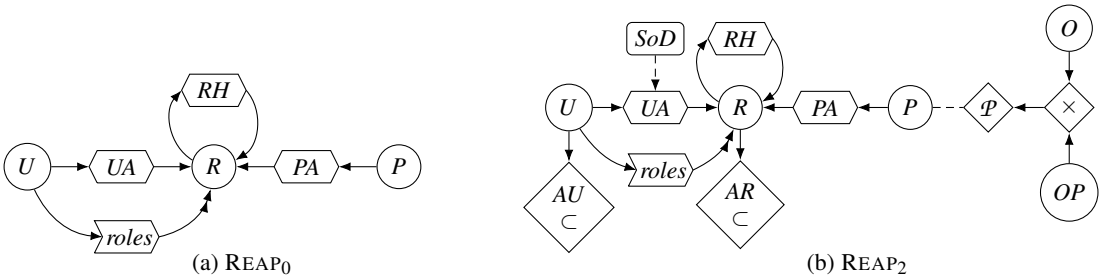
Example 2. We continue the HIS policy introduced in Example 1. To add semantics for role administration, a policy engineer may intend a special subset of roles (Ⓐ) only eligible for a special subset of (administrator-)users (Ⓐ). This results in

Ⓤ→◇⊂ or Ⓤ→◇⊂- - -Ⓐ

and Ⓡ→◇⊂ or Ⓡ→◇⊂- - -Ⓐ.

Note that for both definitions, a policy engineer may opt to declare a node of independent (administrative) identifiers, which are described in more details as subset of legal user/role identifiers in the left part of the

[1] If required, operands are ordered by numerical weights.

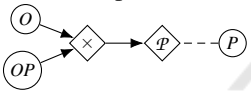


(a) REAP$_0$ (b) REAP$_2$

Figure 2: Exemplary REAP graphs for the hospital information system scenario.

graph, e. g. to enable its reuse as a single atomic set in another, related policy. Alternatively, she may also skip the explicit denomination for the sake of brevity. Both ways are semantically identical.[2]

Finally, again in the mindset of extending parts of the policy by just extending the previous graph, we may go into more detail about what exactly a permission in set P is. In case we conceive a permission as a set of pairs of objects (O) and operation (OP) (Ferraiolo et al., 2007), this may be achieved by cascading a cartesian product and a power set:



While the cartesian product may remain anonymous, we identify the power set resulting from this subgraph with our permissions set P.

Compared to Example 1, this extends our resulting set definitions by $AU \subset U$, $AR \subset R$, O, OP, whereas $P = \mathcal{P}(O \times OP)$ is now a derived set (cf. Fig. 2b). Note that for conciseness, we have decided not to show the implicit relationships between AU and AR (via UA and *roles*) that result from their subset relationship with U and R. Again, drawing these additional edges it left to the discretion of the policy engineer. This is also an example for how potential incompleteness or inconsistencies may be reported to the engineer in a tool-assisted graph drawing workflow.

REAP$_2$. Up to this point, REAP enables the definition and composition of arbitrary sets and their relationships. However, policies may include more abstract requirements that further restrict these definitions, which we call *constraints*. Constraints can be visualized by node annotation, which can also be used for arbitrary meta-information about any part of the model (e. g. for distinguishing mutable from static sets).

In REAP$_2$, we introduce *annotation nodes* for sets of boolean expressions. Such nodes are visually denoted by a capsule (▭) and labeled as usual: by a name and a specialization, where the latter contains an explicit expression if visually capable. Otherwise, the constraint set name serves as a reference to an external textual specification, just as with name labels. They are connected to annotated nodes through visually distinguished scope edges (⇢). A possible use case is shown in Fig. 2b: Constraints to enforce static separation of duties restrict UA, where *SoD* is a reference to a set of boolean expressions such as "$\forall u \in U : \{\langle u, \text{MedMgmt}\rangle, \langle u, \text{NursMgmt}\rangle\} \not\subseteq UA$" which prevents any user from accessing both medical management and nursing services management resources (and possibly other pairs of conflicting roles).

Note that the model resulting from the above examples closely resembles the basic modeling schemes for (administrative) RBAC (Ferraiolo et al., 2007), however specifically tailored to our HIS scenario: for example, since the policy does not mention a session concept nor a separation of administrative from non-administrative roles (Stoller et al., 2007), both are not included in our model specification. This could still be changed if required in later MSPE steps, such as by an efficient model analysis method, using the set-based language elements presented here.

2.3 Usage in Model Engineering

By design, REAP suggests an iterative model engineering workflow. This requires the policy engineer to first make the most fundamental decisions regarding atomic sets and their simplest possible interrelations, which are subject to subsequent refinement later on. This workflow, as well known from the more general software engineering process, reduces the room for errors and increases communication efficiency by hiding information irrelevant for experts from differing fields.

Finally, it should be highlighted that according to our general claim, language support for MSPE must always enable tool-support. For REAP, this comprises two specific tasks: (1.) to support the design part of model engineering through a graphical editor, featuring plausibility checks and decision support (possibly based on a model engineering expert system); (2.) to

[2] A third way, that still names a derived set without an explicit alias by using the name label, is depicted in Fig. 2b.

enable automatic translation to formal algebra required by model analysis methods. While validating the first goal is subject to our ongoing work, the second one has already been demonstrated in the examples above.

3 DYNAMO

Given the critical importance of security policy correctness, our overall goal is to make the entire MSPE process more streamlined and, thus, to avoid error-prone translations of model specifications in transitions between MSPE steps. We argue that this is achievable by comprehensive and uniform language support.

Having engineered a formal model for a security policy in the model engineering step, both subsequent steps, model analysis and model implementation, require a machine-readable model specification as input. Consequently, model specifications can and should be based on a uniform textual specification language usable across the MSPE process steps.

Towards this goal we present DYNAMO – a specification language for dynamic security models based on deterministic automatons. DYNAMO is designed to share goals of all three main steps in MSPE. Hence, a specification in DYNAMO is (1.) close to formal model abstractions (*model engineering step*); (2.) semantically expressive to be able to specify policy dynamics, which enables analyses of dynamic model properties (e. g. model safety), and machine-interpretable by analysis tools (*model analysis step*); (3.) translatable into source code which can be integrated into security architecture implementations (*model implementation step*).

In the following we detail these goals to describe language requirements (§3.1), derive design decisions regarding language paradigms (§3.2), and discuss how tool support embeds DYNAMO into MSPE (§3.3).

3.1 Language Requirements

In general, MSPE involves qualified security engineers which are familiar with mathematical notations of model abstractions. Consequently, a specification language facilitating cross-step support in MSPE should generally support these notations. Access control (AC) policies are one of the most prominent policy classes in practice. As a first step, DYNAMO focuses on language support for dynamic AC models demonstrating the approach to streamline MSPE.

As yielded by the requirements analysis on necessary formal building blocks for expressing AC models (cf. §2.1), basically three abstract data types (ADTs) are required: set, relation, and mapping, each representing its mathematical counterpart. Since we strive for supporting a broad range of dynamic security models, the extension regarding further common mathematical model abstractions, such as matrices[3] (e. g. often used in AC models such as (Harrison et al., 1976), (Sandhu, 1992), and (Zhang et al., 2005)) or lattices (e. g. often found in information flow models (cf. (Vimercati et al., 2005)) should be easily realizable.

The model analysis step may not only aim at analyzing static model properties (e. g. model consistency) but also dynamic model properties (e. g. model safety) regarding the temporal changes an AC system may undergo. In previous work (Amthor et al., 2014; Amthor, 2016; Schlegel and Amthor, 2020), we propose an approach that allows security engineers to configure modeling schemes to application-specific analysis goals. Hence, we require DYNAMO to being able to specify security models with configurable dynamics, which we define based on deterministic automatons (according to the original idea of (Harrison et al., 1976)).

Definition 1 (*Dynamic Security Model*). A *dynamic security model* is a deterministic automaton defined by a tuple $\langle \Gamma, \Sigma, \Delta, \gamma_0, E \rangle$, where

- The *state space* Γ is a set of states defined as a cartesian product of dynamic model components;
- The *input set* $\Sigma = \Sigma_C \times \Sigma_Y^*$ defines possible inputs that may trigger state transitions, where Σ_C is a set of *command* identifiers used to represent operations a policy may authorize and Σ_Y is a set of *values* usable as actual parameters of commands;[4]
- The *state transition scheme (STS)* $\Delta \subseteq \Sigma_C \times \Sigma_X^* \times \Phi \times \Phi$ defines state transition pre- and post-conditions for any input of a command and formal parameters, where Σ_X denotes a set of *variables* to identify such parameters, and Φ represents a set of boolean expressions in first-order logic;
- The *initial state* γ_0;
- The *extension tuple* E is a tuple of static model components which are not part of Γ.

For defining each $\langle c, x, \phi, \phi' \rangle \in \Delta$, the notation $c(x) ::= \mathsf{PRE} : \phi ; \mathsf{POST} : \phi'$ is used. We call the term ϕ the pre-condition and ϕ' the post-condition of any state transition to be authorized via command c. On an automaton level, this means that ϕ restricts which states γ to legally transition from, while ϕ' defines any differences between γ and the state γ' reachable by any input word $\langle c, x \rangle$. To distinguish between the value domains of individual variables in x, we use a

[3] Note that for access control matrices (ACMs) mathematically formalized by mappings, a convenience matrix ADT would just ease access to ACM columns and rows.

[4] We use the Kleene operator to indicate that multiple parameters may be passed.

refined definition of Σ_X to reflect distinct namespaces of variable identifiers for each model component.

In the model analysis step, our goal is to reason about possible state transitions. Hence, we require that only such commands are modeled in Δ that modify their successor state. Additionally, the model implementation step requires a complete model specification also incorporating policy authorization rules which not necessarily modify a current protection state. We assume such rules to be modeled as STS commands with an empty post-condition.

A core motivation for DYNAMO is to prevent and respectively reduce potential errors in transitions between MSPE steps. Hence, we require our language to encompass two well-known concepts of established programming languages. First, by dividing a model specification into two parts, DYNAMO enables reusability of specifications through inheritance: The *model* part describes a certain model class defining policy abstractions through model components as well as macros for pre- and post-condition of STS commands; the *model instance* part describes a concrete initialization of a dynamic model's automaton by using parts of a derived model. Second, by encompassing a strong typing of model components, which is enforced and checked by DYNAMO compilers, security engineers are supported in finding invalid assignments or relations between common model component types.

Model engineering is an iterative approach, model analysis results often have to be incorporated manually. Therefore, DYNAMO must be ergonomic, especially for qualified security engineers, and provide language elements in a way such that specifications are as simple, concise, and unambiguous as possible.

3.2 Language Design

The size and complexity of real-world security policies require model specifications which are as small and simple as possible. Therefore, DYNAMO supports a simple flavor of inheritance known from object-oriented programming languages resulting in hierarchic model specifications. From a high-level perspective, each DYNAMO specification consists of two parts: a **model** and a **model-instance**. A concrete model instance is then always derived from an abstract model.

Generallyin DYNAMO, every statement and every block of statements encapsulated by **begin** and **end** is trailed by a semicolon. The remainder of this section details DYNAMO's language features divided into the two parts of a specification (§3.2.1 and §3.2.2) and extracts of the EBNF notations, which are demonstrated stepwise based on a *dynamic RBAC* (DRBAC) model (according to Def. 1 and (Schlegel and Amthor, 2020)) for a simplified version of the HIS security policy introduced in §2.2.

```
model_spec =
"begin" "model" string ":"
  "begin" "inheritance" ":"
    inheritance_section
  "end" "inheritance" "; "
  "begin" "components" ":"
    components_section
  "end" "components" ";"
  "begin" "pre-clauses" ":"
    pre_clauses_section
  "end" "pre-clauses" ";"
  "begin" "post-clauses" ":"
    post_clauses_section
  "end" "post-clauses" ";"
"end" "model" ";" ;
```

Listing 1: EBNF for **model** specifications.

```
set U, R, P, S;
relation UA(U, R), PA(P, R);
mapping user(S : U), roles(S : 2^R);
```

Listing 2: Example of a **components** section.

3.2.1 Model Specification

In general, a **model** specification defines the abstractions in form of all dynamic and static model components (**components** section) and reusable macros for pre- and post-clauses as building blocks for STS commands (**pre-clauses** and **post-clauses** sections). Listing 1 shows the structure of a **model** specification.

The **components** section defines a model's components which are, later on (in a **model-instance**), divided into dynamic components, belonging to the state space Γ, and static components, belonging to the extension tuple E. Each model component has a type declared by the ADTs **set**, **relation**, or **mapping**, and a string identifier; parametrized types require additional parameters. Listing 2 shows an example for the DRBAC model.

As required for the specification of pre-clauses, which are used in a **model-instance** to specify STS command pre-conditions, DYNAMO defines the intuitive logical operators ==, !=, **in**, **not in**, **not**, **forall**, **exists**, **and**, and **or**, optionally grouped by parentheses **( )**. Listing 3 shows two pre-clauses: First, **check_acf** specifies an *access control function* (ACF) $acf_{RBAC} : S \times P \rightarrow \{\text{true}, \text{false}\}$ which allows to express whether a permission $p \in P$ may be used in a certain session $s \in S$. Second, **can_activate_role** checks whether a role $r \in R$ may be activated in a certain session $s \in S$ by requiring that r is associated with the s's corresponding user u in UA.

```
check_acf(S s, P p):
  exists r in roles(s): [p, r] in PA;
can_activate_role(S s, R r):
  exists u in U: (
    u == user(s) and [u, r] in UA);
```

Listing 3: Example of a **pre-clauses** section.

```
begin activate_role(S s, R r):
  rs = roles(s);
  rs = rs + { r };
  roles = roles + { (s : rs) };
end;
```

Listing 4: Example of a **post-clauses** section.

```
begin inherited_model_1:
  all;
end;
begin inherited_model_2:
  components: {U, R as R_NEW};
  pre-conditions: { };  /* inherit none. */
  post-conditions: all;
end;
```

Listing 5: Example of an **inheritance** section.

Post-clauses are specified as blocks of statements. Each statement modifies a model component by adding or removing elements. To ease the specification, elements of model components may be also assigned temporarily to identifiers, which are local within a clause's scope. Listing 4 exemplarily demonstrates this for the post-clause **activate_role**. Futhermore, it is possible to iteratively execute statements for each element a component contains by using a **for** statement known from common programming languages.

To further improve the reusability of specifications and to reduce the specification effort, a model specification can be inherited (**inheritance** section): By using **all**, a given **model** specification is inherited completely. Parts of a specification can be inherited list-based under indication of the corresponding section. It is also possible to rename inherited components using the keyword **as**. Listing 5 shows two examples.

3.2.2 Model Instance Specification

A **model-instance** specification defines the automaton components of a dynamic model according to Def. 1. Listing 6 shows its structure.

For a concrete **model-instance**, the model components derived from an abstract **model** specification are divided into dynamic and static components. Consequently, the **state-space** statement defines a set of all dynamic model components (see Listing 7).

```
model_instance_spec =
"begin" "model-instance" string ":"
    string ":"
  "state-space" ":"
    "{" state_space "}" ";"
  "input-vector" ":"
    "{" input_vector "}" ";"
  "begin" "state-transition-scheme" ":"
    state_transition_scheme_section
  "end" "state-transition-scheme" ";"
  "begin" "initial-state" ":"
    inital_state_section
  "end" "initial-state" ";"
  "begin" "extension-tuple" ":"
    extension_tuple_section
  "end" "extension-tuple" ";"
"end" "model-instance" ";" ;
```

Listing 6: EBNF for **model-instance** specifications.

```
state-space: {U, S, UA, user, roles};
input-vector: {U, S, R};
```

Listing 7: Example of **state-space** and **input-vector** statements.

The input vector of a dynamic model defines what parameters can be used for STS commands. Accordingly, the **input-vector** section specifies a subset of model components and corresponding power sets (e. g. user set **U** or power set of user set **2^U**).

The **state-transition-scheme** is a list of command specifications where each one either has a precondition part (**pre: ...;**) consisting of logically connected clauses and derived clause macros, a postcondition part (**begin post: ... end post;**) consisting of one or more clauses and clause macros, or both parts. Constants in pre- and post-conditions can be specified by using single quotes. Listing 8 shows an exemplary specification of a command **delegate_treatment_doc_card**.

```
delegate_treatment_doc_card(
         S s_caller, S s_deleg):
  pre: check_acf(s_caller,
         'p_deleg_treatment') and
       is_activated(s_caller,
         'r_doc_card') and
       can_activate_role(s_deleg,
         'r_doc_card');
  begin post: activate_role(s_deleg,
         'r_doc_card');
  end post;
```

Listing 8: Example of a **state-transition-scheme** command specification.

In the **initial-state**, all dynamic model components are assigned their initial values. Listing 9 exemplarily shows assignments of initial values to mo-

```
S = { s1, s2 };
UA = { [u1, r1], [u1, r2], [u2, r1] };
user = { s1 : u1, s2 : u2 };
```

Listing 9: Examples of **set**, **relation** and **mapping** component value assigments used in **initial-state** and **extension-tuple** sections.

del components of types **set**, **relation** and **mapping**.

Complementary to the **state-space** and **initial-state**, in the **extension-tuple** section, all static model components are listed and initialized.

3.3 Tool Support

With the DYNAMO specification language, we aim for uniform language support across the MSPE steps model engineering, model analysis, and model implementation. In order that a specification in DYNAMO is effectively usable, tool support in form of compilers is required, which automate (1.) plausibility and type checking to detect specification errors, (2.) generation of a intermediate language representation for existing model analysis tools, and (3.) source code generation for machine-executable model implementations.

As a first proof-of-concept, we implemented two compilers. First, a DYNAMO-to-XML compiler is able to generate an intermediate XML-based model representation which is compatible with our dynamic model analysis tool (Amthor et al., 2014). Second, a DYNAMO-to-C++ compiler enables an automated generation of an algorithmic model representations in C++ and all functionality necessary for its runtime based on a layered approach.

4 CONCLUSIONS

This paper discusses language foundations for model-based engineering of security policies. Considering the critical and error-prone nature of translating model specifications between the MSPE steps, we argue that MSPE requires holistic specification language and tool support. Towards this goal we present REAP, a visual specification language for formalizing security policies in model engineering, and DYNAMO, a textual specification language enabling model engineering, model analysis and model implementation of dynamic security models. We already tested both languages by proof-of-concept tools.

We consider the work in this paper a first step: Ongoing work focuses on evaluating the methodology within a use case study comprising a real-world security policy, enabling support for further classes of security models in REAP and DYNAMO such as information flow or non-interference models, and secure runtime environments for rigorously enforcing compiler-generated executable model representations within security architecture implementations.

REFERENCES

Amthor, P. (2016). The Entity Labeling Pattern for Modeling Operating Systems Access Control. In *E-Business and Telecomm.: 12th Int. Joint Conf., ICETE 2015, Revised Selected Papers*, pages 270–292.

Amthor, P., Kühnhauser, W. E., and Pölck, A. (2014). WorSE: A Workbench for Model-based Security Engineering. *Comp. & Secur.*, 42(0):40–55.

Basin, D., Clavel, M., and Egea, M. (2011). A Decade of Model-Driven Security. In *Proc. 16th ACM Symp. on Access Control Models and Technol.*, pages 1–10.

Ben Fadhel, A., Bianculli, D., and Briand, L. (2016). GemRBAC-DSL: A High-level Specification Language for Role-based Access Control Policies. In *Proc. 21st ACM Symp. on Access Control Models and Technol.*, pages 179–190.

Crampton, J. and Morisset, C. (2012). PTaCL: A Language for Attribute-Based Access Control in Open Systems. In *Principles of Secur. and Trust: POST 2012*, vol. 7215 of *LNCS*, pages 390–409.

Ferraiolo, D., Kuhn, D. R., and Chandramouli, R. (2007). *Role-Based Access Control*. Artech House. Sec. Ed., ISBN 978-1-59693-113-8.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in Operating Systems. *Comm. of the ACM*, 19(8):461–471.

Jin, X., Krishnan, R., and Sandhu, R. (2012). A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *Data and App. Secur. and Priv. XXVI*, vol. 7371 of *LNCS*, pages 41–55.

Mitra, B., Sural, S., Vaidya, J., and Atluri, V. (2016). A Survey of Role Mining. *ACM Comput. Surv.*, 48(4).

OASIS (2013). eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard.

Sandhu, R. S. (1992). The Typed Access Matrix Model. In *Proc. 1992 IEEE Symp. on Secur. and Priv.*, pages 122–136.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-Based Access Control Models. *IEEE Comp.*, 29(2):38–47.

Schlegel, M. and Amthor, P. (2020). Beyond Administration: A Modeling Scheme Supporting the Dynamic Analysis of Role-based Access Control Policies. In *Proc. 17th Int. Conf. on Secur. and Cryptogr.*, to appear.

Servos, D. and Osborn, S. L. (2017). Current Research and Open Problems in Attribute-Based Access Control. *ACM Comput. Surv.*, 49(4):65:1–65:45.

Stoller, S. D., Yang, P., Ramakrishnan, C. R., and Gofman, M. I. (2007). Efficient Policy Analysis for Administrative Role Based Access Control. In *Proc. 14th ACM Conf. on Comp. and Comm. Secur.*, pages 445–455.

Vimercati, S. D. C. d., Samarati, P., and Jajodia, S. (2005). Policies, Models, and Languages for Access Control. In *Proc. 4th Int. Workshop on Databases in Networked Inf. Syst.*, vol. 3433/2005 of *LNCS*, pages 225–237.

Xiao, X., Paradkar, A., Thummalapenta, S., and Xie, T. (2012). Automated Extraction of Security Policies from Natural-language Software Documents. In *Proc. 20th Int. ACM Symp. on the Found. of Softw. Eng.*, pages 12:1–12:11.

Zhang, X., Li, Y., and Nalla, D. (2005). An Attribute-based Access Matrix Model. In *Proc. 2005 ACM Symp. on Applied Comp.*, pages 359–363.