

# Modeling Security-Enhanced Linux Policy Specifications for Analysis\*

Myla Archer Elizabeth Leonard

Code 5546, Naval Research Laboratory, Washington, DC 20375

{archer, leonard}@itd.nrl.navy.mil

Matteo Pradella

CNR IEIIT-MI, Politecnico di Milano, Milano ITALY

pradella@elet.polimi.it

## Abstract

*Security-Enhanced (SE) Linux is a modification of Linux initially released by NSA in January 2001 that provides a language for specifying Linux security policies and, as in the Flask architecture, a security server for enforcing policies defined in the language. To determine whether user requests to the operating system should be granted, the security server refers to an internal form of the policy compiled from the policy specification. Since the most convenient description of the policy for user understanding is its “source” specification in the policy language, it is natural for users to expect to be able to analyze the properties of the policy from this source specification. However, though specifications in the SE Linux policy language avoid implementation details, the policy language is very low-level, making the high level properties of a policy difficult to deduce by inspection. For this reason, tools to help users with the analysis are necessary. The goal of the NRL project on analyzing SE Linux security policies is to first use mechanized support to analyze the specification of an example policy, and then to customize this support for use by practitioners in the open source software community. This paper summarizes how we have modeled an example security policy in the analysis tool TAME, the kinds of analysis we can support, and prototype mechanical support to enable others to model example security policies in TAME. (For an extended version of this paper, see [5].)*

## 1 Introduction

Security-Enhanced (SE) Linux [12, 8] is a modification of Linux initially released by NSA in January, 2001 that extends Linux with a flexible capability for security. SE Linux provides a language for specifying Linux security policies that cover all aspects of the system, including process control, file management, and network communications. The SE Linux release includes an example policy specification. Policy enforcement uses the method in the Flask architecture [13], where a security

server makes policy decisions concerning whether to grant user requests to the operating system. To make decisions, the security server refers to an internal form of the policy compiled from the policy specification.

Since the most convenient description of the policy for user understanding is its “source” specification in the policy language, it is natural for users to expect to be able to analyze the properties of the policy from this source specification. However, though specifications in the SE Linux policy language are independent of implementation details, the language is very low-level and detailed, making the high-level properties of a policy difficult to check by inspection. Our experience as well as that of others (e.g., [10]) is that mechanized formal methods can uncover errors that humans miss in inspecting even the most carefully crafted specifications. For a user to analyze a typically intricate policy specification, mechanized tools are a practical necessity. Tools such as *Apol* from Tresys Technology and *Tebrowse* from the University of North Texas allow one to observe simple properties of a policy essentially by browsing the policy. For analyzing a policy for deep properties, more powerful tools are needed.

To answer this need, we have taken some initial steps to develop tool support for analyzing SE Linux security policies using the tool TAME (Timed Automata Modeling Environment) [2, 3]. These steps include 1) creation of an abstract SE Linux model in TAME with policy-independent and policy-dependent parts, 2) design and implementation of algorithms for extracting a) a subset of a specified security policy on which to focus analysis and b) the policy-dependent parts of the model from a policy specification, and 3) use of the results to model an example policy based on the policy in the SE Linux release. Accurately modeling SE Linux plus a security policy requires both an understanding of Linux and a clear definition of the semantics of the policy language.

The ultimate goal of modeling SE Linux<sup>1</sup> in TAME is to determine whether the security policy has desired

\*This work is funded by DARPA.

<sup>1</sup>Here and below, a reference to modeling SE Linux implies that some security policy is included in the model.

properties. However, it is also of interest to check properties related to the well-formedness of the model and the accuracy of the model’s representation of the security policy. Thus, we will use theorem proving in stages to check 1) a set of standard well-formedness conditions for the model, 2) that the assertions checked by the SE Linux policy compiler `checkpolicy` hold for our model of the security policy, and 3) whether certain desirable security properties hold for the model. Stage 3) is predicated upon having a reasonable example for a security policy and understanding what its intended properties are. Unfortunately, finding a reasonable example policy is not as simple as using the example policy in the release (or a subset of this policy). This is because our initial analysis focuses on the system after initialization. At least with our system configuration, when SE Linux is run with the example policy enforced, no effective user actions are permitted [6], and hence no user actions can change the state in security-relevant ways. For this reason, there are no interesting properties of system behavior involving user actions after initialization to prove for the example policy. Thus, one of our tasks is to find a reasonable extension of the example policy to analyze.

The remainder of the paper is organized as follows. Section 2 describes the policy language, discusses its semantics, and explains why the nature of its semantics leads us to represent the operating system itself in our abstract model of an SE Linux policy. Section 3 describes how we constructed an example policy for analysis. Section 4 gives a brief overview of TAME, and then describes how we modeled an example policy in TAME and how we have organized the model for reuse with other policies. Section 5 describes our progress with implementing mechanized support for reusing our model. Section 6 describes both simple and deep properties which we hope to verify for our model, and our approach to the verification. Finally, Section 7 provides some suggestions as to how, with appropriate enhancements, the SE Linux policy language could better support policy analysis.

## 2 The SE Linux policy language

The SE Linux security policy language is described in [8], part of the documentation accompanying the SE Linux release. We note that this language has changed over time. In this paper, we deal primarily with the language and example policy from the initial release of January 2001, since our initial efforts towards modeling policies were based on this language.<sup>2</sup> However, our

---

<sup>2</sup>We make an exception for our recent experiments with SE Linux, which have of necessity involved the version of the language available in the June 2002 release. Because dealing with a moving target is difficult, we have retained the original language as the basis of our model rather than continuously adapting the details of our approach to changes in the language.

policy analysis approach is valid for any version of the language.

The language description in [8] is somewhat informal, and is mostly given by example. Some of the language constructs are not fully defined in [8]; however, most of the constructs used in the example policy accompanying the release have reasonably complete descriptions. Although the language permits definition of policies based on *type enforcement* (TE), *role based access control* (RBAC), and *multi-level security* (MLS), we have focused on analyzing policies that use only TE and RBAC features. Below, we describe the syntax of the TE and RBAC language constructs mentioned in [8], and discuss how the semantics of these constructs influences how we model policies in TAME.

**Policy language syntax.** The SE Linux policy language has four kinds of statements: *declarations*, *rules*, *constraints*, and *assertions*. *Declarations* include *role declarations* and *type declarations*. *Rules* include *access vector rules*, which govern decisions made by the security server about access requests, and *transition rules*, which govern possible role changes of an object and type enforcement (TE) type assignments to newly created objects. *Constraints* constrain the manner in which various access permissions can be applied to various objects. *Assertions* are statements about whether or not certain kinds of access permissions are ever allowed by the policy. Once proved, the assertions can be used as simple properties of the security policy that are available as lemmas in the proof of deeper properties closer to the high-level security goals of the policy.

Each language statement consists of a keyword (e.g., `allow` for most access vector rules) followed by arguments using other language elements such as *type names*, *role names*, *object classes*, *attributes*, and *permissions*. The particular sets of representatives of these elements can depend on the particular policy being defined (and the particular Linux configuration for which it is being defined—e.g., the particular kernel modules present). The sets are typically large. In the example policy with the SE Linux release, there are 3 role names, 28 object classes, 22 attributes, 115 permissions, and 253 type names of which 21 are parameterized—hence a potentially unbounded number of type names. Thus, policy specifications tend to be lengthy, complex, and full of low-level detail.

The complexity of policy specifications is, in practice, somewhat reduced by the use of *macros*. Macros can be either *set macros* that represent sets of permissions, sets of object classes, etc., or *rule macros* that represent sets of rules and associated declarations.

**The policy language semantics and its implications.** Individual constructs in the SE Linux policy language, unlike those in higher-level programming languages and specification languages such as Z [14]

and the B language [1] do not have a fixed or uniform semantics. Although every object class has an associated set of permissions with names suggestive of their intended meanings, the actual semantics of any SE Linux permission is determined by how that permission is used to control system transitions. For example, a successful `write` system call by a process can affect the content of a file, but `write` permission to the file is not equivalent to guaranteed success: the process must also have `setattr` permission to a file descriptor for the file. Similarly, the form of an `allow` rule:

```
allow <type_s> <type_t>:<obj_class> <perm>
```

(where `<type_s>` is the “source type” and `<type_t>` is the “target type”) suggests a direct interpretation for many of its instances, e.g., “a process of type `<type_s>` can be granted permission `<perm>` to an object of class `<obj_class>` and type `<type_t>`”. However, there are many exceptions in which `<type_s>` is not the type of a process or the type `<type_t>` is not associated with an object in class `<obj_class>`. Hence, the significance of any instance of an SE Linux policy rule varies with the nature of the arguments to the rule. And ultimately, like permissions, allow rules are given their actual semantics by their use in the permissions checks controlling system transitions.

We note that because multiple permissions can be needed for an actual flow of information and because the semantics of `allow` rules depend upon how they are used in the system, precisely analyzing the policy for information flows is more complex than simply checking for the existence of a path between security contexts by tracing through `allow`, `type_transition`, and other rules in the policy. Because the meanings of the policy rules are so intertwined with the operating system, one cannot reason precisely about the effectiveness of a policy without modeling the system to which it is to be applied. Therefore, to model an SE Linux policy, we also must model the SE Linux operating system on some level.

### 3 Choosing an example policy

The example policy that accompanies the SE Linux release is not a good example to aid in developing our analysis methods because 1) it does not contain sufficient `allow` rules to make SE Linux usable when it is enforced, and 2) it is too large and complex for an initial feasibility study. Thus, to obtain a good example policy for analysis, we need first to extend that policy “judiciously” so that it allows nontrivial user behavior after system initialization, and then to extract a subset of the extended policy.

**Extending the original policy.** The manner in which the original SE Linux example policy must be extended to be usable is platform dependent. Because

it is so low-level, it must be customized to work for the configuration (e.g., the installed packages and daemons) of the machine on which it is installed. This must be done carefully to ensure that only permissions necessary for correct operation of the system are added. We have obtained a policy usable with the newer version of SE Linux on our system by adding around 30 `allow` rules. For verification purposes, we have used these rules as guidance for extending the original policy into a reasonable policy allowing nontrivial user behavior after system initialization.

**Choosing a subset.** Security policy specifications in the SE Linux policy language are generally large and complex, requiring a possibly prohibitive amount of space and time for modeling and analysis. It is difficult to prove properties of the policy without modeling the full policy. However, modeling and proving properties of a subset can help develop confidence that the policy achieves its goals. Subsets are useful for policy debugging: If the property does *not* hold for the subset, it will not hold for the full policy either. When the property *does* hold for the subset, evidence has been accumulated about its validity for the full policy.

Our extraction algorithm, described in more detail in Section 5, slices a security policy based on the selection of sets of types and system calls. The extracted policy slice uses only the selected types and the permissions associated with the selected system calls.

For our initial experimental analysis, we consider the portion of the operating system necessary for file management and process control. Subsets that include the types associated with hardware interfaces, networking, or initialization of the system could be modeled similarly. Another consideration in our choice of an initial policy subset for analysis is the lack of full documentation of some of the policy language constructs. Our chosen subset avoids those constructs.

## 4 Modeling SE Linux in TAME

**A TAME overview.** TAME is an interface to the theorem prover PVS [11] that simplifies specifying, and proving properties of, automata models. To support specifying automata, TAME provides templates that allow the user to specify the standard parts of an automaton—its state space, its start state(s), and its transitions. To support reasoning about the specified automata, TAME provides a set of standard supporting theories and a set of strategies that support proving automaton properties either automatically (if possible) or using proof steps resembling the natural steps used in high-level hand proofs.

The TAME model for SE Linux that we have been developing is based on the I/O automata model [9]. The TAME template organizes the specification of an I/O automaton by using a standard set of constructs. The state space is represented by a record

type `MMTstates`, and the start states are specified by a predicate `start`. The transitions are specified by a datatype `actions` describing the set of actions that can trigger transitions, a predicate `enabled` to describe the preconditions on the actions, and a function `trans` to describe the effect of each action `a` on a state `s`. The *transitions* of the automaton are the prestate-poststate pairs  $(s, \text{trans}(a, s))$  with  $\text{enabled}(a, s) = \text{true}$ .

The proof support provided by TAME is mainly aimed at proving invariant properties of automata, such as *state invariants* (properties of all reachable automaton states) and *transition invariants* (properties of all reachable transitions). As noted in [4], most high-level security properties of an SE Linux policy can be represented as either state or transition invariants. The existing TAME proof support will be useful in proving such properties; however, it can be anticipated that advantage can be taken of the common features of TAME models of SE Linux to add proof steps especially geared to these models. This issue is discussed further in Section 6.

**A TAME model of SE Linux.** To model SE Linux abstractly in TAME, one must choose an appropriate state space, set of initial states, and set of transitions. In our TAME model<sup>3</sup>, the state space is determined by a set of variables of which the principal variable is `objects`, the set of objects (such as processes, files, directories, file descriptors, etc.) managed by the operating system, and there is a single initial state. We chose to model the system from the point after the system has been initialized, and the initial state in our model reflects this. As in any TAME model, transitions are the result of actions in the datatype `actions`; in our model, actions correspond to system calls issued by processes.

The values of the state variable `objects` are sets of members of the datatype `OBJECT`. The constructors in `OBJECT` provide a way to construct an object of every class. The formal parameters of the constructors behave like fields in a record. For each object class, the choice of which parameters to include is determined by three factors: 1) the need to tag every object with its security context (or security label)<sup>4</sup>, 2) represent (abstractly) the effects of system calls on the object, and 3) in some cases, the need to be able to state system properties of interest.

In addition to the variable `objects`, there are two additional kinds of state variables: *shadow variables*

<sup>3</sup>Our TAME model can be found on the NRL SE Linux project page at <http://chacs.nrl.navy.mil/SoftwareEng>.

<sup>4</sup>In SE Linux, every object has a *security context* that contains such information as an associated user, TE type, RBAC role, and possibly an MLS security level. The integer-valued SID (security identifier) actually used as the security label in SE Linux is a session-specific hash encoding of the security context. This implementation detail is not necessary in our model.

and *indexing* variables. Shadow variables add no information about the state to that provided by `objects`, but are used to provide more direct access to information difficult to express directly in terms of `objects`. Indexing variables are used in the management of numerical IDs (such as `Pid`) and version numbers.

The abstract model of SE Linux has two significant aspects: a fixed aspect that depends only on the operating system, and a variable aspect that depends on the particular security policy imposed on the operating system and, to some extent, on the choice of policy subset to model. Based on this categorization of parts of the model, we have developed and partially implemented an approach that can greatly simplify the modeling process for SE Linux with new policies. The fixed parts of our model can be reused in new models. As discussed in Section 5, construction of the variable parts of the model, including the choice of a policy subset to model, can be supported by a combination of extraction algorithms and definitions libraries.

The fixed part of the model includes the state space. It also includes much of the description of actions. In particular, the definition of the datatype `actions` essentially consists of declarations of the various system calls and their arguments. Those parts of `enabled` involving checks of arguments (e.g., if a process `p` issues a `write` system call with file descriptor argument `fd`, then `fd` must be one of `p`'s file descriptors) and those parts of `trans` that do not involve type or role transitions are also fixed. Besides checking arguments, the predicate `enabled` determines whether `PermissionGranted` holds for a given action in a given state. The function `trans` handles type transitions by calling whichever of the functions `Newfiletype` or `Newproctype` is appropriate.

The variable parts of the model include the definitions of the predicate `PermissionGranted` used by `enabled` and the functions `Newfiletype` and `Newproctype` used by `trans`. The predicate `PermissionGranted` determines whether the required permissions for system calls can be granted. It is defined in terms of the predicate `Allowed`, which is directly derived from the `allow` rules in the policy, and the predicate `PathAllowed`, which applies `Allowed` recursively over the ancestor directory names of a file name `pn`. There are fairly straightforward algorithms for compiling `Newfiletype`, `Newproctype`, and `Allowed` from a policy specification. Because the policy specification language does not yet support description of the required permissions associated with particular system calls, we cannot yet compile the full definition of `PermissionGranted` automatically.

The choice of initial state is also variable. E.g., in our initial example model of a policy subset, our choice of initial state is affected by the elimination of that part of the full policy controlling what happens during

system initialization. The choice of initial state can be aided by a library of objects whose states reflect a particular status of the operating system.

## 5 User support for modeling policies

Because of the size and complexity of policy specifications and SE Linux itself, developers using our analysis methods will need tool support for creating the variable parts of a TAME model for SE Linux. We plan to offer two types of support: automatic extraction tools and libraries. Algorithms for automatically extracting a policy subset and for then creating the policy-dependent portions of a TAME model for SE Linux from the (possibly reduced) policy have been implemented using Python [16, 15].<sup>5</sup> Libraries can be developed to aid in the construction of the policy-independent portions of the model.

**Automatic extraction tools.** Our policy slicing algorithm allows a user to specify a policy subset by specifying a set of TE types  $T$ , and a set of permissions  $P(oc)$  for each object class  $oc$ . As noted in Section 3, these sets are chosen based on the types and system calls that are to be analyzed. In particular, the permissions in each  $P(oc)$  are those needed for the system calls to be analyzed. The algorithm removes all permissions from any rule involving an object class  $oc$  that are not in  $P(oc)$ , and then removes all declarations and rules in TE files that either reference types not in  $T$  or have no remaining permissions.

In addition to the slicing algorithm, we have developed an algorithm that extracts all allow rules from a set of TE files and converts them into the form of the `Allowed` predicate, and a similar algorithm that extracts all the type transition information and converts it into the functions `Newproctype` and `Newfiletype`.

**Library support.** The actual choice of system calls included in the model can be considered a variable part of the model. However, the definitions of the preconditions and effects of system calls are policy-independent, and hence fixed, down to the level of `PermissionGranted`, `Newfiletype`, and `Newproctype`. Thus, they can be written just once for all policies. We have begun to develop a TAME library of action declarations and the fixed parts of action preconditions and effects for SE Linux models. This library can eventually be used to support the automatic construction of the top level definitions of the actions in a model, once a user selects the system calls to include.

The initial state is another part of the model that the user may wish to vary. For this variable aspect of the model, another library can be developed that allows users to select the processes and other objects to automatically include in their desired initial state.

<sup>5</sup>We have implemented the algorithms for both the January 2001 and June 2002 versions of the SE Linux policy language.

## 6 Checking properties of models

As noted in the introduction, we are checking SE Linux properties in stages, starting with simple properties, and advancing to deeper properties.

**Simple properties.** There are two types of simple properties: well-formedness properties and policy assertion properties. The well-formedness properties are policy-independent, while policy assertion properties are policy-dependent.

Well-formedness of the TAME specification as a PVS specification is checked simply by applying the PVS type checker and proving any type correctness conditions that the type checker generates. Additional well-formedness conditions include *shadow variable properties*, which assert that the shadow variables have the intended relation to the variable objects, and *object type properties*, which show that the OBJECT components of “reachable” objects have the expected object class. Such properties are not true in every state  $s$  in the state space but are expected to hold in every *reachable* state of the model. Therefore, they must be proved as state invariants. A strategy for using action preconditions in induction proofs can be designed that omits expanding `PermissionGranted` in order to simplify proving such policy-independent properties.

Policy assertion properties are derived directly from the `neverallow` statements that comprise the assertions of a policy. Such a property can be checked directly simply by expanding the `Allowed` predicate and using the result to check that no case forbidden by the associated `neverallow` statement is allowed. Checking these assertions provides some assurance that the definition of `Allowed` in the model is consistent with the specification.

**Deeper properties.** The deeper properties of greatest interest derive from the security goals that the policy designer wishes to achieve for a Linux system. A set of eight general goals for the example policy in the SE Linux release is given in [12]. These goals are stated at a very high level, e.g., “protect the integrity of the kernel”, and “protect the administrator role and domain from being entered without user authentication”. Determining the precise properties SE Linux should have to achieve the high-level goals is difficult without more explicit input from the policy designer.

Here are two possible deeper properties of interest:

1. Only a process whose initial TE type is `klogd.t`, or one of its descendents, ever gets permission to execute a `write` system call to kernel log files.
2. Any process that has `search` permission in a directory has `search` permission in all ancestors of the directory.

Property 1, which can be formulated as a transition invariant, may be one of the properties desired for protecting kernel integrity. Property 2, which can be formulated as a state invariant, is interesting for a different reason: if this property holds, then every use of `PathAllowed` involving the `search` permission can be changed to a simple (non-recursive) use of `Allowed`, which is easier to reason about.

Other properties of interest may be the information flow properties being checked by Herzog and Guttman [7]. As noted in Section 2 precise checking of such information flow properties cannot be done by straightforward reasoning from the policy rules. These information flow properties can likely be checked in the TAME model, since this model contains more system detail than is embodied in the policy rules alone. Currently, the feasibility of doing so is an open question.

**Feedback from unfinished proof goals.** Every unfinished proof goal occurring in the course of the proof of a state or transition invariant corresponds to a state transition that, if it is reachable, is a counterexample: a transition that either fails to preserve the state invariant or fails to satisfy the transition invariant. To handle an unfinished proof goal, one can often introduce additional facts that show that the prestate in the transition is not reachable. These facts may be facts about the specification that have not yet been used in the proof (e.g., the inductive hypothesis), or they may come from separately proved invariant lemmas or lemmas about the data types used in the specification. However, sometimes the unfinished goal will correspond to a real counterexample. In such cases, it is useful to be able to simulate the system to discover whether the prestate is indeed reachable. Creating such a simulation capability is work for the future.

## 7 Discussion

Several minor modifications to the SE Linux policy language would make it more friendly to the analysis of policy specifications. For example, our algorithms are unnecessarily complicated because information about the significance of certain macros has to be computed. This problem could be solved by replacing the macro construct by a similar construct containing additional information to (e.g.) 1) distinguish rule macros from set macros, 2) distinguish macros for permissions sets from macros for object class sets, and 3) identify the object class or classes associated with a permissions set macro.

Other aspects of the macro construct can make a policy difficult to analyze by inspection. One example is the use of parameterized types in a macro where these types are not locally declared. This is safe only if the macro is used only in an environment where these types are defined. The need for a safety check would go

away if macros were replaced by functions not involving global variables.

Modifying the language into something closer to a strongly typed programming language would permit the use of functions rather than macros, and facilitate other consistency checks that one gets “for free” from type checking in such a language.

## Acknowledgements

We wish to thank Ross Godwin and James Pak for explaining many details of the implementation of SE Linux and its behavior in practice. We also thank Constance Heitmeyer and Ralph Jeffords for helpful comments on an earlier version of this paper.

## References

- [1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Math. and Artif. Intel.*, 29(1-4):139–181, 2000. Published Feb., 2001.
- [3] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.
- [4] M. Archer, E. Leonard, and M. Pradella. Towards a methodology and tool for the analysis of Security-Enhanced Linux security policies. Technical Report NRL/MR/5540-02-8629, NRL, Wash., DC, August 16 2002.
- [5] M. Archer, E. Leonard, and M. Pradella. Analyzing Security-Enhanced Linux policy specifications. Technical Report NRL/MR/5540-03-8659, NRL, Wash., DC, 2003.
- [6] R. Godwin, J. Pak, M. Archer, and E. Leonard. Documenting aspects of SE Linux. Draft report, 2002.
- [7] A. L. Herzog and J. D. Guttman. Achieving security goals with Security-Enhanced Linux. Extended abstract of a presentation at the IEEE Symp. on Security and Privacy, 2002.
- [8] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. Technical report, National Security Agency, Jan. 2, 2001.
- [9] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.
- [10] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP’98)*, 1998.
- [11] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. The PVS prover guide. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1998.
- [12] S. Smalley and T. Fraser. A security policy configuration for Security-Enhanced Linux. Technical report, National Security Agency, Jan. 2, 2001.
- [13] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. of the Eighth USENIX Sec. Symp.*, pages 123–139, Aug. 1999.
- [14] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1991.
- [15] G. van Rossum. *Python Library Reference, Release 2.2.1*. PythonLabs, April 2002.
- [16] G. van Rossum. *Python Tutorial, Release 2.2.1*. PythonLabs, April 2002.