CONFIDDENT: A model-driven consistent and non-redundant layer-3 firewall ACL design, development and maintenance framework

S. Pozo [*], R.M. Gasca, A.M. Reina-Quintero, A.J. Varela-Vaca

Department of Computer Languages and Systems, ETS Ingeniería Informática, University of Seville, Avda. Reina Mercedes S/N, 41012 Sevilla, Spain

A B S T R A C T

Keywords:

MDD

MDA

Firewall

Maintenance

Development

Diagnosis

Design, development, and maintenance of firewall ACLs are very hard and error-prone tasks. Two of the reasons for these difficulties are, on the one hand, the big gap that exists between the access control requirements and the complex and heterogeneous firewall platforms and languages and, on the other hand, the absence of ACL design, development and maintenance environments that integrate inconsis-tency and redundancy diagnosis. The use of modelling languages surely helps but, although several ones have been proposed, none of them has been widely adopted by industry due to a combination of factors: high complexity, unsupported firewall important features, no integrated model validation stages, etc. In this paper, CONFIDDENT, a model-driven design, development and maintenance framework for layer-3 firewall ACLs is proposed. The framework includes different modelling stages at different abstraction lev-els. In this way, non-experienced administrators can use more abstract models while experienced ones can refine them to include platform-specific features. CONFIDDENT includes different model diagnosis stages where the administrators can check the inconsistencies and redundancies of their models before the automatic generation of the ACL to one of the many of the market-leader firewall platforms currently supported.

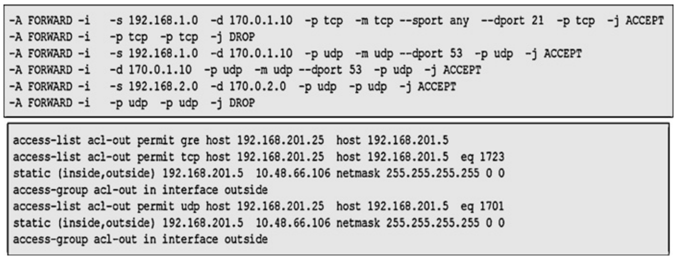
1. Introduction

A firewall is a network element that controls the traversal of packets across different network segments. It is a mechanism to enforce an access control policy, represented as an Access Control List (ACL), or *rule set*. Developing and managing firewall ACLs are tedious, time-consuming and error-prone tasks for several reasons (Chapple et al., 2009; Wool, 2004). Two of the most important problems firewall administrators have to face are: (1) the high complexity of firewall-specific ACL development and maintenance, and (2) ACL inconsistencies (contradictions) and redundancies introduced during these life-cycle tasks.

Networks have different access control requirements which must be translated by a firewall administrator into firewall ACLs. Firewall-specific languages are, in general, hard to learn, use and understand. Each firewall platform has its own language, which has to be known by the firewall administrator in order to implement the access control requirements. Fig. 1 presents two fragments of ACLs written in IPTables and Cisco PIX, respectively. They give an idea of the complexity and differences between firewall languages. Note that the number of rules in Firewall ACLs may range between a few ones and 5000, with an average of about 800 (Taylor, 2005) in new deployments in 2009. This average number of rules is doubling every year (Chapple et al., 2009). Moreover, ACL maintenance implies that about 10% of rules could change every month (Chapple et al., 2009).

Besides that, inconsistencies and redundancies could be introduced in ACL development and maintenance life-cycle stages, as they are the two most frequent sources of faults in firewall ACLs (Chapple et al., 2009; Wool, 2004). A firewall ACL with inconsistent rules indicates, in general, that the firewall is accepting traffic that should be denied or vice versa and represents severe security problems such as unwanted accesses to services, denial of service, overflows, etc. (Pozo et al., 2009b). ACL consistency is of extreme importance in several contexts, such as highly sensitive applications (e.g. health care). A firewall ACL with redundancies implies matching engine performance degradation and firewall platform memory waste. In a recent survey (Chapple et al., 2009) 91% of administrators answered that they felt that at least one fault was introduced during their last ACL update, while half of them answered that the ACLs they developed surely contained undetected faults. In the same survey, administrators answered that they do not have enough resources to manually diagnose and correct these faults, and that they do not have found automated tools to address these problems during ACL development life-cycle.

* Corresponding author.
E-mail addresses: sergiopozo@us.es (S. Pozo), gasca@us.es (R.M. Gasca), reinaqu@us.es (A.M. Reina-Quintero), ajvarela@us.es (A.J. Varela-Vaca).



```
-A FORWARD -i   -s 192.168.1.0  -d 170.0.1.10  -p tcp  -m tcp --sport any  --dport 21  -p tcp  -j ACCEPT
-A FORWARD -i   -p tcp  -p tcp  -j DROP
-A FORWARD -i   -s 192.168.1.0  -d 170.0.1.10  -p udp  -m udp --dport 53  -p udp  -j ACCEPT
-A FORWARD -i   -d 170.0.1.10  -p udp  -m udp --dport 53  -p udp  -j ACCEPT
-A FORWARD -i   -s 192.168.2.0  -d 170.0.2.0  -p udp  -p udp  -j ACCEPT
-A FORWARD -i   -p udp  -p udp  -j DROP
```

```
access-list acl-out permit gre host 192.168.201.25  host 192.168.201.5
access-list acl-out permit tcp host 192.168.201.25  host 192.168.201.5  eq 1723
static (inside,outside) 192.168.201.5  10.48.66.106 netmask 255.255.255.255 0 0
access-group acl-out in interface outside
access-list acl-out permit udp host 192.168.201.25  host 192.168.201.5  eq 1701
static (inside,outside) 192.168.201.5  10.48.66.106 netmask 255.255.255.255 0 0
access-group acl-out in interface outside
```

Fig. 1. Example firewall ACLs: (1) IPTables; (2) Cisco PIX.

These faults have traditionally been addressed in two ways: (1) using modelling languages which permit an abstraction from the underlying firewall platform and language syntax, and (2) using inconsistency and redundancy diagnosis algorithms over ACLs once implemented or derived from high-level models. Although the use of modelling languages surely helps, none of the proposed ones in the research community has been widely adopted by industry due to a combination of factors: they usually have a similar complexity to firewall-specific languages, none of them support the full set of firewall platforms important features, unsupported firewall important features, no integrated model validation stages, etc. Furthermore, models could also include inconsistencies and redundancies, which must be diagnosed and corrected before automated ACL generation and deployment. If not, diagnosis and correction of faults must be done at later stages of the development process, losing model traceability, increasing costs, and reducing the reliability and robustness of the ACL (Douglas et al., 1996).

The tools and methods that aim to help administrators during ACL development process should gain in functionality and ease of use at rates to match the increase in firewall ACL development complexity. In this paper CONFIDDENT, a CONsistent and non-redundant FIrewall Design, DEvelopment, and maiNTenance framework is proposed. In CONFIDDENT, simple and abstract platform-independent ACL models (which might satisfy a large base of inexperienced firewall administrators) can be combined with more complex and less abstract platform-specific ACL models (which can satisfy experienced administrators) through a series of automatic model transformations. CONFIDDENT takes into account inconsistencies and redundancies that can be introduced during modelling stages and integrates different model verification stages. Models can automatically be transformed to firewall-specific ACLs. CONFIDDENT currently supports a wide range of firewall platforms, which represent market-leaders: Linux IPTables, Cisco PIX, FreeBSD IPFilter, FreeBSD IPFirewall, OpenBSD Packet Filter, and Checkpoint.

To the best of our knowledge, this is the first published work that addresses the design, development, and maintenance of consistent and non-redundant firewall ACLs at the design stage using a model-driven approach. With CONFIDDENT, it is possible to create tools that effectively fill the gap between current modelling languages and firewall-specific ACLs, providing firewall administrators with tools that represent a real alternative for the whole life-cycle of ACL management.

The paper is structured as follows: in Section 2, related works are provided, with special focus on languages and modelling tools for developing and managing abstract firewall ACLs. In Section 3 background and concepts on MDD paradigm and MDA view are given. In this section CONFIDDENT architecture is also proposed. In Section 4, CONFIDDENT platform-independent meta-model is proposed with its specification and an example. A model diagnosis stage is included prior to the next modelling stage. In Section 5, IPTables firewall platform is described and a platform-specific meta-model for it is proposed as a proof of concept. Then, a model to model transformation specification is given in order to transform the PIM to the PSM. Furthermore, another model verification stage and a discussion about its necessity are included prior to the final code generation stage of the process. In Section 6, a model to text transformation into IPTables code is given with its specification. In Section 7 we discuss and evaluate the results achieved. In Section 8, we make some concluding remarks and propose future works. In Appendix I, AFPL DSML is showed (which is closely related with the platform-independent meta-model). In Appendices II and IV PIM and PSM meta-classes descriptions are respectively presented. In Appendix III an example scenario is given and used for the examples of the paper. In Appendix V, the M2M transformation rules between the PIM and IPTables PSM are given. In Appendix VI, a file containing attribute default values is presented. Finally, Appendix VII shows the M2T transformation rules for IPTables.

2. Related works

In Bartal et al. (2004) proposed an abstract language, Firmato, a firewall domain-specific modelling language (DSML). Models, which are based on a textual description of entity-relationship diagrams (ERDs), can be automatically transformed to firewall-specific ACLs. However, the complexity of Firmato is very similar to firewall-specific languages. Firmato has two major limitations: (1) it does not support network address translation (NAT) and (2) it can only represent knowledge in positive logic (*allow* rules). These two limitations complicates the specification of exceptions (a rule with a general *allow* action, immediately preceded by a more restrictive rule with a *deny* action). This could result in the need of writing several rules to express one exception. However, as a lateral effect, rules are always consistent (although not necessarily non-redundant) and order-independent.

In Damianou et al. (2001) provided another abstract language, Ponder, to model network policies (in general). A re-engineered version, Ponder2, is also available. Network policies include a superset of access control related concepts (for example, routing). Neither Ponder nor Ponder2 can be automatically transformed to firewall-specific ACLs. In theory, a language that can express any network policy could also express a firewall ACL. However, the complexity of Ponder2 surpasses the modelling needs for firewall ACLs. Furthermore, firewall specific concepts such as NAT cannot be modelled.

FLIP (Zhang et al., 2007) is a recently proposed firewall DSML. Models can also be automatically transformed into several firewall-specific languages, although no more information about this feature is provided in the paper where it was presented. Their authors claim that ACLs expressed in FLIP are always consistent. In fact, they are because of one of its limitations: it does not support overlapping rule selectors in different rules. Preventing the use of overlaps is a major limitation, since it is impossible to express exceptions. In addition, its syntax is even more complex than Firmato's one. However, due to this lack of expressiveness, FLIP ACLs are order-independent. Finally, NAT is not supported in FLIP.

AFPL (Pozo et al., 2008, 2009c) is the most recent firewall DSML. Contrarily to the other proposals, it has been designed after an analysis of the features of six major firewall-specific languages in a bottom-up process, supporting most of their features but with less complexity. AFPL can model *stateful* and *stateless* rules (although an administrator does not need to know these kind of details, since complexity is hidden in the modelling language), positive and negative logic rules, filtering field overlapping, exceptions, and can be automatically compiled to six market-leader firewall languages.

However, none of the reviewed modelling languages is extensible by users, and no one integrates a fault diagnosis stage for diagnosing inconsistencies and redundancies at the development life-cycle.

Table 1
Survey of the main features of access control modelling languages.

	Firmato	FLIP	AFPL	Ponder2	SRML	Rule-ML	PCIM	XACML
DSML	√	√	√	×	×	×	×	×
User extensible	×	×	×	×	×	√	Partial	√
Inconsistency diagnosis	N/A	N/A	×	×	×	×	×	×
Redundancy diagnosis	N/A	N/A	×	×	×	×	×	×
Stateful rules	√	√	√	N/A	N/A	N/A	N/A	N/A
NAT	×	×	√	N/A	N/A	N/A	N/A	N/A
Positive logic	√	√	√	√	√	√	√	√
Negative logic	Partial	√	√	√	√	√	√	√
Rule field overlap	√	×	√	√	√	√	√	√
User-controlled rule order	N/A	N/A	√	√	×	×	√	√
Relative complexity	High	High	Low	High	Low	Low	Medium	Medium
Transformation to Firewall-specific ACLs	√	√	√	×	×	×	×	×
Firewall-specific ACL import	×	×	Partial	×	×	×	×	×

Some organizations have proposed languages to model access control policies as XML documents, such as XACML, PCIM, Rule-ML, and SRML. However, none of these languages is specific enough for firewall ACLs, resulting in a high complexity to model firewall concepts, or in an impossibility to model them at all (this is the case of NAT for all these modelling languages). UML also has been proposed to model access control policies (Basin et al., 2006; Jürjens, 2002). In general, these modelling languages are very generic and are not intended for the area of any particular access control problem. Table 1 presents a survey of the most important features of the reviewed languages (related to their ability to model firewall ACLs). There are other good surveys of access control policy languages available in the bibliography (De Capitani Vimercati et al., 2007; Ardagna et al., 2004; El-Atawy, 2006).

All of the reviewed modelling languages (domain-specific or not) have many trade-offs that have motivated their existence. However, if firewall ACL design, implementation and maintenance are set as the objectives for their design, we think that the number and utility of available concepts that the modelling language supports must be one of the most important considerations to take during language design. A large number of available concepts guarantee a high expressiveness, although the complexity of the resulting language could rise to levels near to the ones of firewall-specific languages. This is the case of many of the reviewed languages, especially the DSML ones. Generic policy languages, such as Ponder2 or XACML, aim to address this complexity issue raising the abstraction level with a more general access control model for representing any network policy. However, the reality is that with these more general languages it is not possible to model important features of firewall platforms, such as NAT.

With respect to commercial or Open Source ACL development and maintenance tools and, to the best of our knowledge, the most important ones are Firewall Builder, Cisco ASDM, Checkpoint Blades Software, and LogLogic ChangeManager. Table 2 presents a survey of the reviewed tools.

Firewall Builder (FWB, from now on) is an Open Source tool that is able to model ACLs for different firewall platforms. An important fact about FWB is that it hides platform-specific firewall details with a graphical representation (FWB is not based on models). In fact, the particular final firewall platform to which generate code must be specified as the first step of the design process. If the platform is changed during ACL development, information may be lost if the new platform does not support the same features of the initial one. Models are not extensible by users, since a modification in the model would also require GUI and compilers modification. FWB allows the administrator to define objects to represent network elements such as services, computers, network segments, etc., allowing the separation of logic from network topology. FWB allows the importing of both, Cisco IOS and IPTables ACLs. With regard to fault diagnosis, FWB only supports a very primitive form of rule shadowing detection. A rule is shadowed when there is another, previous rule that covers the matching space of the second one, and both have contradictory actions.

Cisco ASDM is a GUI provided by Cisco to their customers in order to assist them in the process of network device configuration. This tool supports the development of ACL for firewalls and configuration files for other Cisco network devices. As with FWB, it is not possible to model platform-specific details, but only hides them with a GUI. Contrarily to FWB, this is not important here, since ASDM can only generate code for Cisco devices. For the same reason, models are fixed and users are not allowed to modify them. Importing Cisco devices configuration files is possible, as well as a direct ACL deployment from the tool. It is also object-oriented, providing separation between logic and network topology. Unfortunately this tool does not provide any kind of fault diagnosis.

Checkpoint Blades Software (CBS, from now on) is the new denomination of Checkpoint software. CBS is based on a modular design, where different modules support different devices (e.g., firewall ACL development and maintenance software, intrusion detection systems signature development and maintenance software, etc.). Since CBS is designed with only one platform in mind, it shares the same features as Cisco ASDM, with the only difference of firewall-specific ACL import, which CBS does not support, since Checkpoint firewalls are always designed using a GUI.

Finally, LogLogic ChangeManager (CM, from now on) is a commercial tool that is able to model ACLs for different firewall platforms. This tool supports the development of firewall ACLs and configuration files for other network devices. As with the other tools, CM hides platform-specific firewall details with a graphical representation. However, in contrast with FWB, the development process is visual. Through this process, it is possible to effectively abstract the administrator from some of the platform-specific features. As with FWB, the firewall platform must be specified at the first step of the development process, since each firewall platform supports a different feature set. If the platform is changed during ACL development, information may be lost if the new platform does not support the same feature set of the initial one. Models are not extensible by users. CM allows the administrator to define objects to represent network elements such as services, computers, network segments, etc. allowing a real separation from topology and logic. CM also allows the importing of firewall-specific ACLs of the supported platform. Finally, this tool does not provide any kind of fault diagnosis.

Although AFPL supports most of the features of six market-leader firewall platforms (Linux IPTables, Cisco PIX, FreeBSD IPFilter, FreeBSD IPFirewall, OpenBSD Packet Filter, and Checkpoint) and appears to be the most equilibrated DSML for firewalls from the ease of use/modelling features perspective, it also suffers from some drawbacks. Firstly, AFPL has been designed using a bottom-up methodology (Pozo et al., 2008, 2009c). This implies

Table 2
Survey of high-level ACL design tools.

	Firewall Builder	Cisco ASDM	Checkpoint Blades	LogLogic ChangeManager
Focus is on syntax abstraction or in functionality abstraction	Syntax	Syntax	Syntax	Hybrid
Topology/logic separation	√	√	√	√
Firewall-specific language import	Partial	√	N/A	√
Models are user-extensible	×	×	×	×
Able to model other devices such as IPS	×	√	√	√
Multi-vendor compilation	√	×	×	√
Inconsistency or redundancy diagnosis	Only shadowing	×	×	×
FW platforms supported	Cisco, IPTables, HP ProCurve, BSD PF, IPFW, IPFilter	Cisco	Checkpoint	Cisco, Juniper, Checkpoint, Fortinet, IPTables

that when a new firewall platform is going to be supported, AFPL must be revised in order to check if the feature set of the platform can implement AFPL concepts. We think of this possibility as being very difficult to happen, since the start-up point of AFPL was the *common set* of features of six market-leader firewall platforms. For the same reason, it is very difficult to extend or modify AFPL by end users. Secondly, there are features supported in the considered firewall platforms that are not present as concepts in AFPL. This implies that experienced firewall administrators could need to use real-world firewall features which cannot be modelled. As a lateral effect, an import of an ACL which has not been modelled with AFPL is only partially possible, in the worst case. However, this reduction in the available set of concepts in AFPL is the key to guarantee that models can be transformed into any of the considered firewall platforms. Finally, recall that the use of a modelling language like AFPL does not guarantee that the resulting ACL is consistent and non-redundant, although it usually minimizes these problems. Although AFPL models can be diagnosed for inconsistencies and redundancies using slightly adapted versions of existing algorithms (Pozo et al., 2009a), it does not have fault diagnosis facilities which can be used during design stages. We think that these drawbacks may be enough to prevent firewall administrators from using AFPL as a real alternative to firewall-specific languages or the available development tools.

The reviewed development tools also have some important drawbacks. Firstly, most of them are platform-specific, preventing the use of ACL models. Secondly, although some multi-platform tools exist, they do not abstract the firewall administrator from the firewall platform features, but only from the syntax of the target firewall-specific language. In fact, destination platform must be chosen as the first step of the modelling process. This implies that the abstraction level is linked to the destination platform, and thus it is relatively poor. In fact, models are not available to users to be modified or extended. One major lateral effect of this lack of abstraction is that each model is specific for a firewall platform, and thus it cannot be reused for others. Another important lateral effect is that models cannot be easily diagnosed for inconsistencies and redundancies, since the available algorithms must be adapted for each different firewall vendor and version.

Due to these drawbacks, we think that it is not possible to fulfil the modelling requirements of all firewall administrators and all sceneries with only one modelling language or tool. In this paper we propose CONFIDDENT, a multi-platform CONsistent and non-redundant FIrewall Design, DEvelopment, and maiNTenance framework. We propose to replace ad-hoc development methods with well-grounded models that can represent the essential features of the firewall platforms, and automatic reasoning processes for the analysis of engineering decisions for those administrators concerned with the efficient and timely production of quality Firewall ACLs. The framework supports several user-selectable abstraction levels for modelling, depending on user needs or expertise, enabling automatic transformations through different abstraction levels at any time. In CONFFIDENT, models can be automatically transformed into firewall-specific ACLs for a wide representative variety of firewall platforms. There have been other proposals which use MDD as the basis for modelling networking testbed configurations (Galán et al., 2010) but, to the best of our knowledge, this is the first one to model firewall configurations and ACLs.

3. CONFIDDENT specification and architecture

CONFIDDENT specification has been built using the foundations of Model-Driven Development (MDD) paradigm, focusing on a particular view, the OMG's Model-Driven Architecture (MDA) (OMG MDA, 2003). Although there are other views of MDD such as Software Factories (Greenfield et al., 2004) and Model-Integrated Computing (Sztipanovits and Karsai, 1997), the one provided by MDA seems to be the most prevalent at present. Model-Driven Architecture (MDA) development aims at generating systems from high-level system models and requirements models, taking away much of the concurrent manual changing of artefacts at the different stages of software development. It promises better leverage on building quality (i.e., stakeholder value) into the software products and should support the measurement of software quality at different stages of the development life cycle. The three primary goals of MDA are portability, interoperability and reusability.

In MDA, models can be built at different abstraction levels. In this context, a model is an abstract representation of a system structure, function or behaviour. Transformations are the way of obtaining one model in one level (target model) from another model or set of models from another level (source model). Models are specified with concepts that are described in a meta-model, as a consequence, it is said that a model has to conform to a meta-model. The meta-model determines the constructs that can be used and the rules that must be followed to build a model. Fig. 2 depicts the main artefacts involved in a simple model transformation. The source and target meta-models can be the same or not. If the resulting model of the transformation (the target model) is expressed in the same language than the source model, then source and target meta-models have to be the same. Otherwise, they differ. Transformations are described in a language that also has to conform to its own meta-model. The artefact in charge of executing the transformation is known as the transformation engine. Usually transformation definitions refer to meta-models instead of models. As a consequence, the transformation engine needs the source model, the source meta-model, the target meta-model and the transformation definition as input, and it will produce the target model that will conform to the target meta-model. One transformation is considered to be a model-to-model transformation (M2M) if it takes a set of models as input and produces a set of models as output. However, if the result is a set of textual artefacts or a code implementation, it is called a model-to-text transformation (M2T).

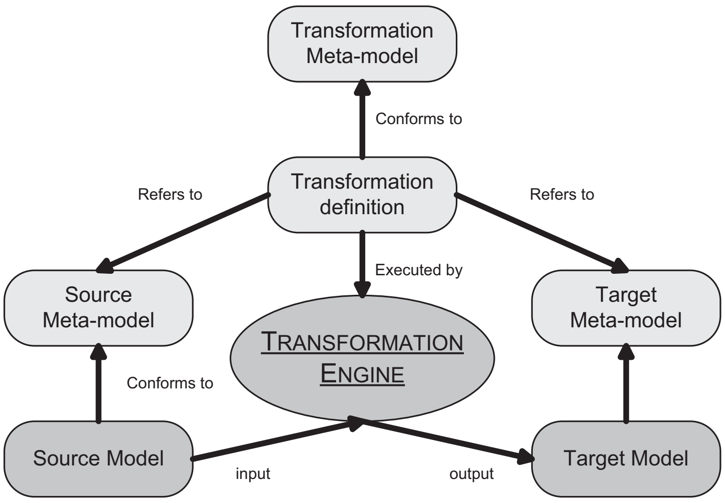


Fig. 2. Artefacts involved in a simple transformation in MDA.

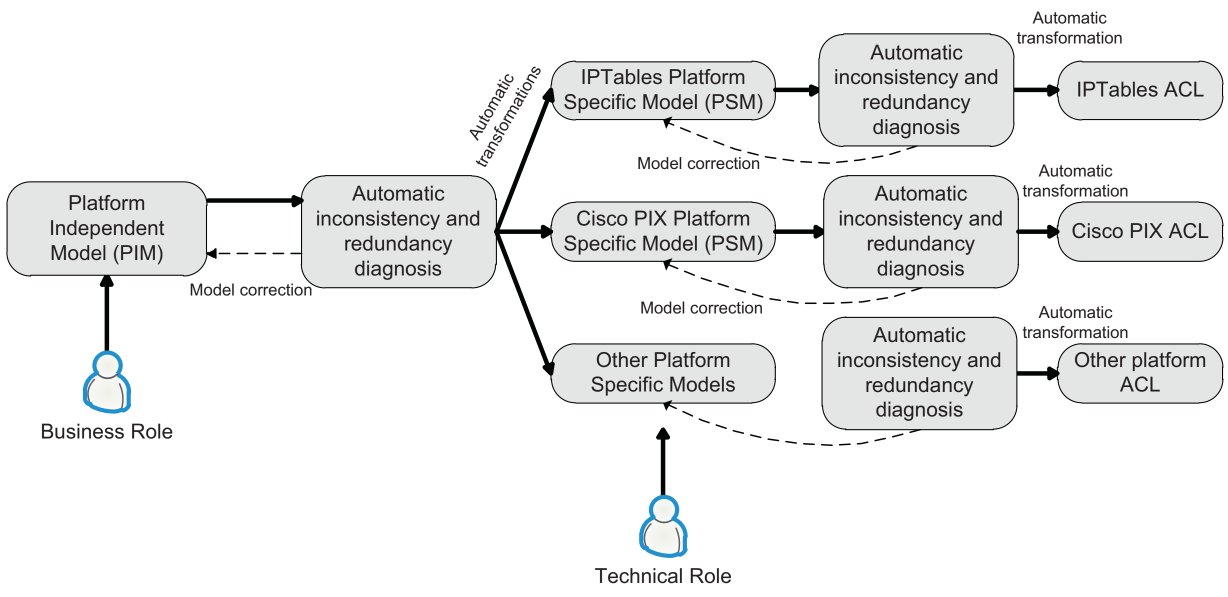
Simultaneously to the development of MDA, the Eclipse Open Source community has been working on giving support to this OMG framework with the Eclipse Modelling Framework (EMF; Budinsky et al., 2003) (i.e., the meta-modelling framework), which nowadays can be considered as the *de facto* standard. The Eclipse Modelling Project (EMP) is the project devoted to the evolution and promotion of model-based development technologies.

3.1. *CONFIDDENT architecture*

The CONFIDDENT framework consists in several modelling stages, each with a different abstraction level (Fig. 3) and has been heavily inspired by the MDA view.

The first stage consists on defining platform-independent models. These models conform to a platform-independent meta-model, which represents the highest level of abstraction. Through this meta-model it must be possible to model all concepts that are available as features in *all* of the supported firewall platforms. Because of this, platform-independency is obtained at this modelling stage.

The second modelling stage is where the administrator selects the target platform for the final ACL, and where platform-specific details are modelled, if any, building a platform-specific model (PSM). Several platform-specific meta-models (at least one for each target platform) are also necessary at this modelling stage. Ideally, an administrator must be able to model the full feature set of the selected firewall platform by means of its meta-model, and thus the meta-model must contain enough concepts to model the features. However, this will surely result in a very complex meta-model where some of its concepts may never be used by less experienced administrators. To prevent this, CONFIDDENT uses modularized platform-specific meta-models: a target platform meta-model is built from small parts, where each of these parts defines a platform-specific feature. The administrator is thus free to select the trade-off between abstraction level/features available, and thus this modelling stage can be understood as a variable abstraction level one. Following this approach, existing platform-specific meta-models can be extended or modified, and new ones can be created to fulfil any administrator needs. Even repositories of platform-specific meta-models can be provided to CONFIDDENT users. In fact, if the modelling of platform-specific features is not needed, the administrator can avoid PSM modelling stage and enter the last one.



Platform Independent Model (PIM)
Business Role
Model correction
Automatic inconsistency and redundancy diagnosis
Automatic transformations
IPTables Platform Specific Model (PSM)
Automatic inconsistency and redundancy diagnosis
Automatic transformation
IPTables ACL
Model correction
Cisco PIX Platform Specific Model (PSM)
Automatic inconsistency and redundancy diagnosis
Automatic transformation
Cisco PIX ACL
Model correction
Other Platform Specific Models
Automatic inconsistency and redundancy diagnosis
Automatic transformation
Other platform ACL
Technical Role

Fig. 3. CONFIDDENT architecture.

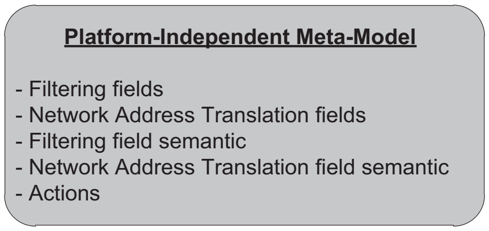

Platform-Independent Meta-Model
- Filtering fields
- Network Address Translation fields
- Filtering field semantic
- Network Address Translation field semantic
- Actions

Fig. 4. CONFIDDENT platform-independent metal-model concepts.

The last modelling stage is where the firewall-specific ACL (i.e. the implementation) is obtained through an automatic M2T transformation by means of predefined transformation rules. Although the implemented ACL can be modified by an administrator, it is not recommended, since traceability with the models may be missed. Furthermore in order to accomplish any modification, administrators need to know the details of the particular firewall-specific language and platform.

If during the modelling stages an inconsistency or redundancy fault is found, then models must be corrected before entering the following modelling stage.

Since there are at least two conceptually different abstraction levels (PIM and PSM), firewall ACL concepts should be decomposed in two disjoint sets. For the division, some design decisions must be made since firewall platforms are very different from one vendor to another, and even among the available Open Source platforms. These differences range from different number, type, and syntax of selectors (or filtering fields) that each platform's filtering algorithm can handle, to huge distinctions in rule-processing algorithms that can affect the design of the ACL. In a previous analysis (Pozo et al., 2008, 2009c) it was identified that the vast majority of filtering concepts can be expressed with any of the most representative firewall platforms (Linux IPTables, Cisco PIX, FreeBSD IPFilter, FreeBSD IPFirewall, OpenBSD Packet Filter, and Checkpoint). These concepts are the basis for the platform-independent meta-model in CONFIDDENT, since they completely fulfil the objectives at this first abstraction level (Fig. 4).

However, there are filtering related concepts that are not going to be included in the platform-independent meta-model. There are basically two options for these concepts: to include them in another (lower) level of the framework, or to completely remove them. If they are removed, experienced administrators that may need to use them must modify the automatically generated firewall-specific ACL. However, if these concepts are introduced at the platform-specific meta-model, experienced administrators can model the necessary concepts without directly modifying the implemented ACL. For this reason, we propose to consider these platform-specific filtering-related features as modules that are part of the PSM for each platform. Each of these modules can represent a disjoint set of concepts. By the composition of these models the needed features of the firewall platform can be modelled.

Again, there are other features not related with filtering that also present in firewall platforms. These other features may also be needed by administrators. These features may, for example, be how to manage malformed packets, connection tracking issues, if packet mangling is available the rules to configure platform behaviour, how to manage content inspection, how to configure logging, etc. In CONFIDDENT all of these features that are closely related with firewall platforms must also be considered at platform-specific meta-models. Each (disjoint) set of related features for a firewall platform is considered also as a different module for the platform meta-model. Thus platform-specific modelling is done through module combination. This methodology results in a trade-off between abstraction level and features available which the user can decide.

Fig. 5 shows an example platform-specific meta-model for a fictitious firewall platform. Transformations between different firewall platform models, although possible through a M2M transformation, may have loss of information, since different platform-specific meta-models may represent a different set of concepts for each platform.

Again, firewall platforms have other specific features related to how each platform executes the ACL, and that administrators cannot modify this behaviour during modelling stages. These features are, for example, how each platform threats connection tracking (that is, *stateful* or *stateless* connections), how rule processing is

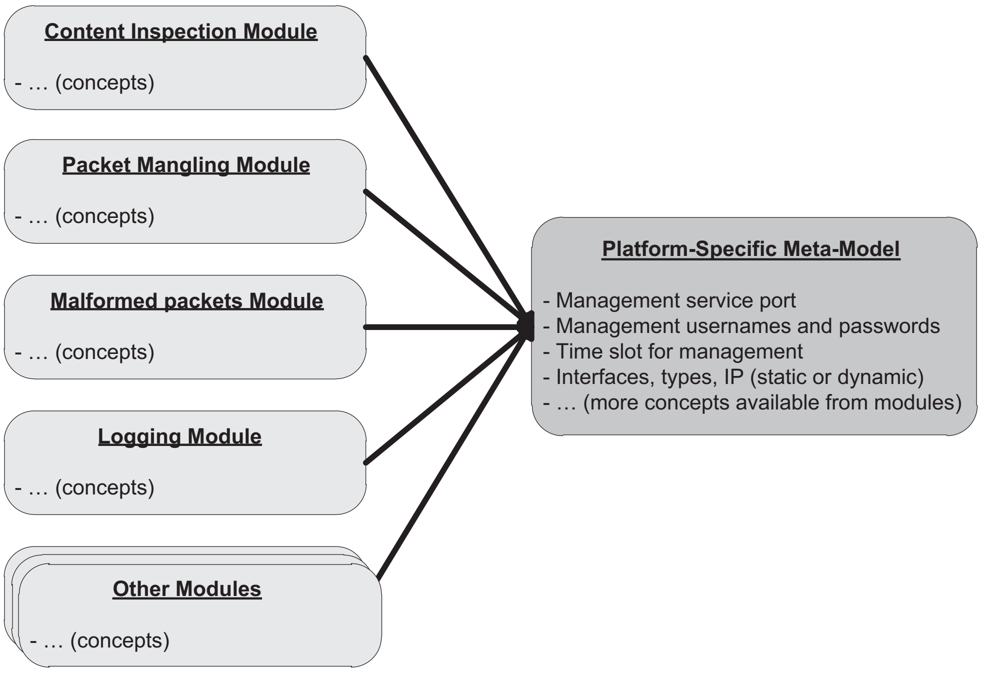

Content Inspection Module
- … (concepts)
Packet Mangling Module
- … (concepts)
Malformed packets Module
- … (concepts)
Logging Module
- … (concepts)
Other Modules
- … (concepts)
Platform-Specific Meta-Model
- Management service port
- Management usernames and passwords
- Time slot for management
- Interfaces, types, IP (static or dynamic)
- … (more concepts available from modules)

Fig. 5. CONFIDDENT platform-specific metal-model concepts (modular approach).

performed (forward, backward, with jumps), etc. For this reason, these features do not need to be modelled and are only considered during M2T transformation (to target firewall-specific ACL).

Following the methodology proposed in CONFIDDENT, the platform-independent meta-model is kept as simple as possible, serving for the vast majority of administrators, while the modular approach for the platform-specific meta-models facilitates the use of more specific features to more experienced ones with an adjusted abstraction level. Note that if *all* features of a firewall platform are available as modules for its PSM, a lossless inverse transformation (from a firewall-specific ACL implementation to models) is also possible.

3.1.1. *Model validation and diagnosis in CONFIDDENT*

The use of the proposed architecture does not yet guarantee the absence of faults in the models. Both model validation and diagnosis techniques are used in CONFIDDENT. Validation is to test if the model is well constructed (i.e. testing if the model is correct or not). Validations can be applied to different parts of the model, like structure or semantics. Diagnosis is used to explain why a model is not correct, and of course implies a validation stage. Like validations, diagnosis can be applied to different parts of the model, such as structure or semantics. In addition to validations, it is possible to look for explanations to different problems, like inconsistency and redundancy, identifying which components of the model are faulty. In complex, real world, problems it is usually a good idea to give a minimal diagnosis (i.e. the fewer possible number of faulty components). The *Parsimony Principle* says that the preference for a diagnosis of a problem is to give the least complex explanation. This is especially important for big models and in models with a lot of inconsistencies, since not giving it could result in an overwhelming number of components to be corrected. In our problem, these components are the modelled rules. The minimal diagnosis problem is an optimization problem which, unfortunately, is NP-hard in many problem domains.

In CONFIDDENT, structural and semantic model validations (at both PIM and PSM levels) are implemented as OCL constraints. However, model inconsistency and redundancy diagnosis are implemented as an external library. This has been done in this way because OCL is precisely a validation language. However, OCL does not have primitives to solve optimization problems, which are needed to solve inconsistency and redundancy diagnosis faults with the best possible results (in terms of completeness, minimality and performance). In fact, thanks to the high performance of many of the available inconsistency and redundancy diagnosis algorithms, the ACL model can be diagnosed on-line. However, we think that is less intrusive from the modeller point of view, to run the diagnosis when needed. In Pozo et al. (2009a) it was suggested the possibility of using existing firewall-specific ACL inconsistency and redundancy diagnosis algorithms over ACL models when the ACL modelling language is sufficiently expressive. This possibility will be analysed in this paper during following sections, when the platform-independent and platform-specific meta-models are proposed.

ACL inconsistency and redundancy diagnosis is a very complex problem that is receiving considerable attention from the scientific community since 1999. During the first research years, the main focus of these works was to define the problem and to provide complete solutions to it (Hari et al., 2000; Eppstein and Muthukrishnan, 2001; Al-Shaer and Hamed, 2004; Al-Shaer et al., 2005; Hamed and Al-Shaer, 2006; Yuan et al., 2006; García-Alfaro et al., 2008). There were proposals that used logic languages and constraint solvers. However, the performance of these solutions was not appropriate for solving real-life problems. Later on, the focus was in providing approximate solutions for the problem (Pozo et al., 2009d; Baboescu and Varguese, 2003). However, experimental results showed that the variability of the problem was too high to generalize an approximation algorithm. Finally, in recent works (Pozo et al., 2009b, 2010) it has been described an algorithm which can solve the problem in polynomial time and space complexity. The algorithm is complete and minimal, and thus it has enabled the solution of real-life problems. Even constraint or SAT solvers performance is not usually on par with specialized algorithms and data types for this problem.

The diagnosis stages can be used at each modelling stages, and should always be used before automatic code generation. If faults were diagnosed over the implemented firewall-specific ACL, mappings between ACL rules and models would be necessary in order to trace back the ACL and make corrections to models. Then, models must be transformed to firewall-specific ACL again, and diagnosis algorithms must be re-run in order to know if the faults have been corrected and to guarantee that new ones have not been introduced during correction. Fault diagnosis and correction at the final stages of any development process implies, in general, a lower quality and of the generated code, and that a lot of budget will be spent on these tasks (Douglas et al., 1996). Integrating fault diagnosis at the modelling stages can reduce budget spent in this task and accelerate the development cycle, increasing overall ACL quality. This is even more important in MDD methodologies, since models are their core.

3.1.2. *Modelling issues discussion*

PSM modularization by features is not the only possibility to address this problem. PIM modularization is also possible, as shown in Galán et al. (2010). In fact, feature modularization is possible at any abstraction level.

When the PIM is not modularized (the taken approach), then it specifies a set of stable concepts that are shared over all the supported platforms. Note that in order to obtain a stable PIM meta-model, a sufficiently representative set of destination platforms must be analysed first. We provide this analysis in previous works, where different methodologies to design an abstract firewall modelling language, AFPL, were discussed (Pozo et al., 2008, 2009c). Concepts that are tied to a particular platform or set of platforms are specified at PSM level. However, due to the fact that these concepts are tied to a particular platform, and in order to be aligned with the MDA primary goals (portability, interoperability and reusability), they have been represented as modules. This also enables the user to select the exact number of concepts which he or she needs to model, and thus to adjust the abstraction level to its needs. In fact, users with different roles can act as modellers at the different modelling stages of the framework. Deployment information has not been considered in our work, but can be specified at a second PSM modelling level or taken from a file during M2T transformation. The main benefits of our approach are:

- The taken approach allows a refinement of the initial model (PIM) by adding new concepts in a per-need basis, which effectively results in a complete control over the complexity of the modelling process. The new concepts modelled during refinements do not intersect with previously modelled concepts, allowing an easy concept composition. For this reason, once the PIM validity has been tested, there is no need to do it again during subsequent refinements. In fact, this separation of concepts allows the use of different user-roles at each part of the modelling process: a more business-oriented user specifies the PIM, and a firewall-specialist models the PSM (if needed).
- The PIM meta-model is very stable. In fact, if the PIM is designed with a bottom-up methodology, and only contains concepts that are available in all platforms, any model specified at this level can be transformed to any platform. If the features specified at PSM

are not needed by the user, the PIM can be directly transformed to code, without user intervention at PSM level.

- Modularization at PSM level does not limit that more features can be added to the PIM (also as modules). For example, IPv6 addresses cannot be specified with our meta-model. However, this feature is nowadays supported by most firewall vendors. Thus an extension to the PIM can be specified. However, note the following point.
- The PSM meta-model is the one that contains most of the variability, since new platform versions periodically arise which add, remove or modify existing features which could require the modification of the platform specific meta-model. Modularizing at PSM level also provides portability, interoperability and reusability advantages for coping with this variability even within the same platform, since the platform specific concepts have been isolated one from each other. It is easier and less prone to errors the modification of one or more independent modules than the modification of a meta-model of a full-fledged platform.
- The modularization at PSM level does not limit the importing of existing ACLs. The PSM can contain all the modules needed to model all the features available for a given platform, and thus a direct inverse transformation can be easily be done without user intervention. This has been discussed in Section 5.4.
- A diagnosis stage can be introduced at PIM modelling, since this part of the full model is not going to be modified through the rest of the modelling process. Of course another one may be necessary at PSM modelling stage, since the semantic of the concepts of each meta-model does not necessarily affect the consistency and redundancy characteristics of the models: it is something that will entirely depend on the set of concepts modelled by each particular PSM.The main drawbacks of our approach are
- The user may have to interact several times during the modelling process. For us, this is not a problem, but an opportunity to allow different user-roles interact at the different modelling stages
- If a new platform is going to be supported, an analysis of the PIM meta-model is needed in order to check if the features available in the new platform can be modelled by the existing PIM concepts. If the modelling is not possible, it may be needed to modify the PIM in addition to the PSM if the features fall within the PIM ones in our conceptual separation. However, we think of this possibility as being very difficult to happen, since the start-up point of the proposed framework has been AFPL domain-specific modelling language. AFPL was born from an analysis of the features available in six market leader firewall platforms, and allows the modelling of their common set of features (Eclipse Modeling Project, 2007; Wool, 2004). Anyway, if more concepts are needed, they can always be added as modules in the PIM.

If the PIM is modularized instead of the PSM, it implies that it is composed of different parts. One part represents the core of concepts which enable the modelling of all the features that are supported by all platforms considered as the real systems (i.e. the common ones). Surely, this core is not going to have enough concepts to model very specific features of these platforms, and thus modules may be created in order to fulfil this modelling need. These modules represent the concepts which enable the modelling of a new set of features which are only available for certain platforms. The PSM will then specify all the needed concepts to model the deployment of the model. This approach (Galán et al., 2010) has some important drawbacks over the taken one for our problem. Since each approach is addressing MDD from a different angle, the problems they solve are different in nature.

- The PIM specifies concepts to model features which are not necessarily available in all real platforms. During PIM to PSM transformations what to do with this part of the model is not an easy question to answer. It may be simply pruned in order to enable the transformation without user intervention. However, in some problem domains this may be unacceptable, as is the case of our problem and in general in any problem related to software. For example, removing a rule during a transformation because a filtering selector is not available in the destination platform will surely modify the semantic of the ACL. Because of this, the generated ACL will not conform to the model, which may imply security faults such as accepting traffic that would be denied or vice versa. Note that the semantic of firewall ACLs is not simply represented by a set of unrelated rules, but by a set of related ones.
- Inconsistency and redundancy diagnosis stages problems. Firstly, platform-specific model may also contain inconsistencies and redundancies (as the PIM). That is, if a meta-model specifies concepts related to deployment features, do not necessarily imply that the derived models are consistent. Again, this is something that must be analysed in a per-meta-model basis. Secondly, if a diagnosis stage has been introduced in the first modelling stage, then the diagnosed model cannot be automatically modified for two reasons: (1) the modification may change the semantic of the model, and (2) it can introduce new inconsistencies and redundancies that are not going to be detected at the first modelling stage (i.e. a second diagnosis stage may be necessary, after the transformations that may prune parts of the model).
- The specification of concepts that are not available in all the supported platforms (present or future) at PIM level could cause problems during M2T transformations. The PIM should only specify concepts which are platform-independent. All the concepts that are specifically tied to a particular platform should be specified at PSM level. This decoupling can offer future benefits. For example, different user roles can model different parts of the final model. Note that the three primary goals of MDA (a view of MDD) are portability, interoperability and reusability. In fact, if the PSM is removed in our work or in the another proposal (Galán et al., 2010), the MDA architecture has no benefit at all, since choosing a declarative language also offers the benefit of separating the model specification from the generated code.

However, this approach also has some benefits:

- The whole set of concepts which are available as features in a given platform are specified at PIM level, allowing a direct inverse transformation from a configuration (or ACL in our problem) without user intervention. If more features are needed, new modules are added. Migrations are not the focus of our work. However, it is a very important topic for us, since it completes our work, and is considered a topic for future research.
- Since all the concepts are modelled at PIM level, consistency and redundancy diagnosis could only be needed at this abstraction level. However, the deployment model may also contain inconsistencies and redundancies. That is, the semantic of the concepts of each meta-model does not necessarily affect the consistency and redundancy characteristics of the models. This is the reason why the diagnosis stage may or may not be necessary at PSM level: again, it is something that will entirely depend on the set of concepts modelled by each particular PSM.

3.1.3. *Summary*

In conclusion, we have addressed many of the drawbacks that could prevent firewall administrators from using modelling languages for the design and maintenance of firewall ACLs. CONFIDDENT sits in a niche of firewall modelling languages and tools (Fig. 6). Note that with CONFIDDENT, we are also implicitly proposing a development and maintenance methodology for firewall ACLs

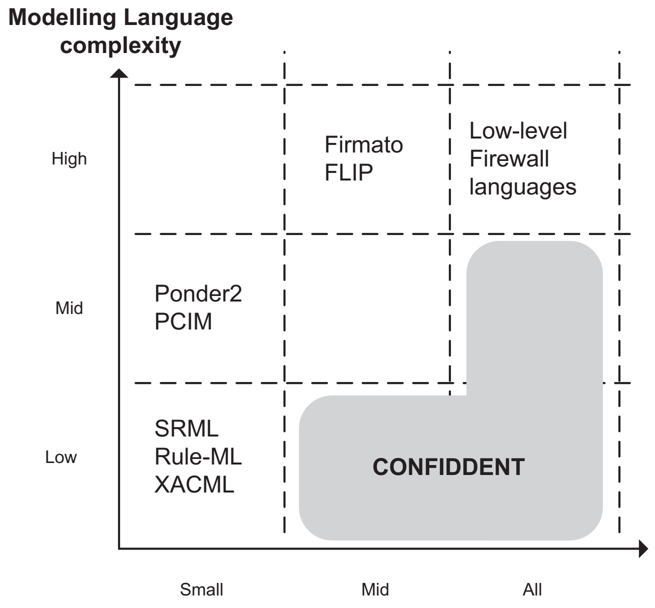


Fig. 6. CONFIDDENT positioning.

that overcomes the drawbacks of the reviewed modelling languages and tools.

4. CONFIDDENT PIM meta-model

In this section CONFIDDENT PIM meta-model (from now on, PIM) is specified and described. This PIM is designed on the basis of a previously proposed DSML language, AFPL (Pozo et al., 2008, 2009c). AFPL is the most recent firewall ACL DSML. AFPL can model *stateful* and *stateless* rules (although an administrator does not need to know these kind of details, since complexity is hidden in the modelling language), positive and negative logic rules, filtering field overlapping, exceptions, and can automatically be transformed to six market-leader firewall languages. Table 3a presents AFPL modelling concepts. AFPL RelaxNG Schema Definition is provided in Appendix I. The design issues of this language have been described in previous papers and will not be repeated here. We refer the interested reader to the papers where the language was proposed.

When AFPL was first proposed, it was identified that the most representative firewall platforms (Linux IPTables, Cisco PIX, FreeBSD IPFilter, FreeBSD IPFirewall, OpenBSD Packet Filter, and Checkpoint) can express a similar set of concepts related with basic filtering. Since these concepts completely fulfil the abstraction objectives for CONFFIDENT at this first abstraction level (taken from the previous section), we propose to directly use AFPL as the PIM for CONFIDDENT. However, since the concepts managed by AFPL are specified in the XML technological space, a transformation to *modelware* one is needed. Fig. 7 presents CONFIDDENT PIM meta-model.

According to Kurtev et al. (2002), a technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. There are a few technological spaces that have been well identified, such as XML, the grammar technological space (also known as *grammarware*) or the meta-model technological space (known as *modelware*).

The PIM is composed of a structure of meta-classes. The root element is the Policy which represents the ACL concept. An instance of the Policy meta-class is composed of one or more Rule meta-classes and zero or more DstNATrule and/or SrcNATrule rules. Thus AFPL (Pozo et al., 2009b) supports three kinds of rules: filtering (also present in AFPL), SNAT, and DNAT. SNAT and DNAT are not mandatory, but at least one filtering rule must be specified (in order to set the policy default action).

Each instance of the Rule meta-class represents a condition/action rule of the ACL. A rule can be applied to a particular interface of the firewall platform (*interface* attribute), and with a particular direction of the flow of packets (*direction* attribute). These two attributes of the Rule meta-class are optional, since if no interfaces are defined, the rule is applied to all interfaces in all directions (*in* and *out*). The *comment* attribute is also optional and represents the documentation for a rule. Furthermore, the Rule meta-class has an *action* attribute representing the action that the firewall should take if a packet matches the condition part. Note that there are three possible actions (*allow*, *deny*, *reject*). However, from a semantic point of view, *reject* and *deny* actions represent the same thing (i.e. to block a packet). The only difference is if with the denied packet an ICMP error message must be sent to the origin (*reject*) or not (*deny*).

The concepts regarding the condition part are represented in the Matches meta-class, which is a component of the Rule meta-class. Each Rule can have only one condition part and, for that reason, the cardinality is one. The Matches meta-class has a set of attributes representing the concepts available for filtering. These concepts correspond to the fields which are considered during the filtering process: source and destination IP addresses, source and destination ports, protocol, and ICMP type (only if protocol is ICMP). A data type has been defined for each one of these attributes. The data types restrict the values that attributes can have. Thus, for example, the *IpType* constrains the valid String pattern to specify an IPv4 address and an optional CIDR value (netmask). The data types are depicted at the right part of Fig. 7 diagram (stereotyped as «*datatype*»).

At the same level of Rule meta-class there are the DstNATrule and SrcNATrule meta-classes. These meta-classes represent DNAT and SNAT rules respectively. Note that NAT rules are optional. Following (Table 3c) description, a DNAT rule can be applied to an interface. Note that no information regarding rule direction can be modelled, since DNAT rules are always applied with incoming direction. In the same way, a SNAT rule (Table 3b) can only be applied with outgoing direction. Again, for both kind of rules, it is possible to specify the characteristics of the packet being translated using the NatOrigPacket meta-class, which has the same attributes as the Matches meta-class, but with different cardinalities. However, translation information differs between SNAT and DNAT.

Table 3a
AFPL DSML (filtering).

Selector	Obligation	Dependencies	Syntax
Source and destination IP address	Mandatory		- IP, CIDR Block, Identifier, Wildcard
Interface	Optional		- IP, Identifier, Wildcard
Interface direction	Optional		- In, Out, Wildcard
Protocol	Mandatory		- TCP, UDP, ICMP, Identifier, Number, Wildcard
Source and destination port	Optional	Only if protocol is TCP or UDP	- Number, Interval [p1,p2], Identifier, Wildcard
ICMP type	Optional	Only if protocol is ICMP	- Number, Identifier
Action	Mandatory		- Allow, Deny, Reject

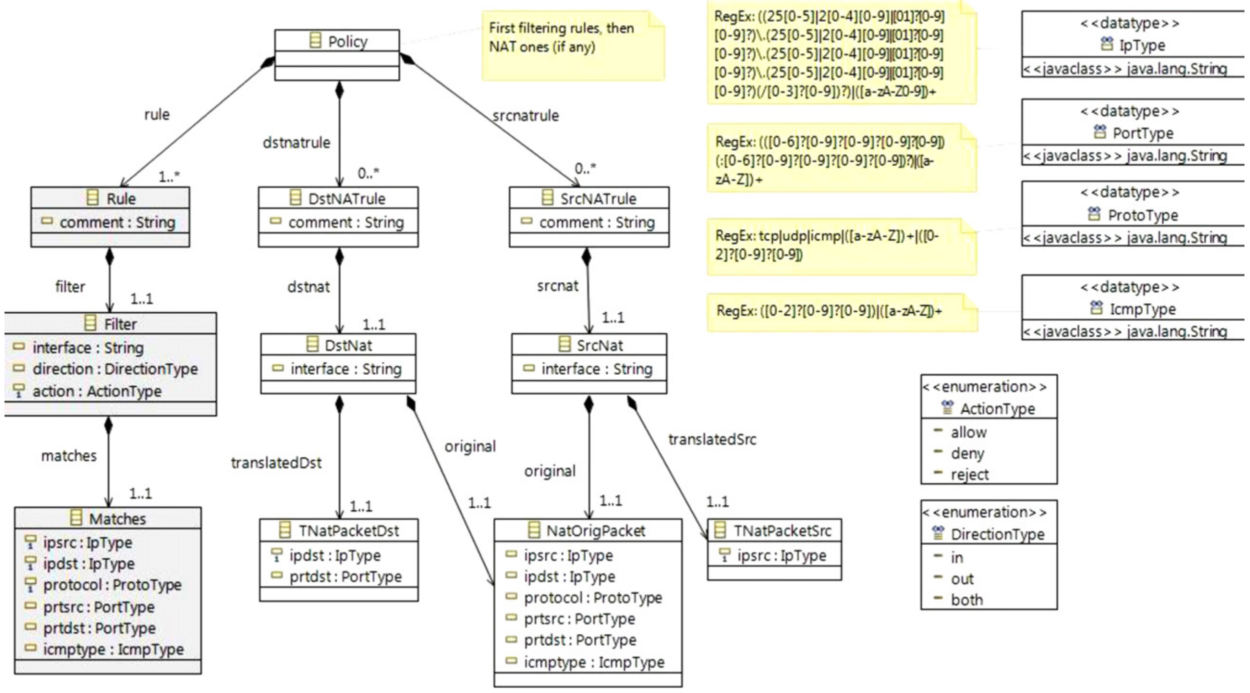


Fig. 7. CONFIDDENT PIM meta-model.

Table 3b
AFPL DSML (Source NAT).

Translated selector	Obligation	Syntax	Comments
Source IP Address	Mandatory	- Host IP, interface name (identifier)	If the interface name is given, the interface IP is used (it could be a dynamic link like PPP)

Note that there is no way to explicitly represent rule priorities. The reason is that a rule priority is implicitly represented in the model using rule-order definition in the PIM. Although a new attribute to represent the order could be added in the Rule meta-class, we have preferred to use the implicit order to keep the meta-model as simple as possible.

It is also worth noting that there is no information regarding how state information is represented on a per-ACL basis. That is, there is no way to model if the firewall is *stateful* or *stateless* with this PIM. The reason is twofold. First, *stateless* firewalls are old technology that is being abandoned in recent firewall platforms (as is the case of Cisco PIX). Second, the only difference between *stateless* and *stateful* firewall ACLs is that in the later ones information regarding source ports of a connection do not have to be explicitly represented in a rule using the source port selector, since this information is automatically managed by firewall platforms during ACL execution. However, *stateful* firewalls also allow the specification of source port information in rules. In *stateless* firewalls, this information must be explicitly specified in each rule and, if not, no return connections will be allowed. Since the usage of the source port at PIM modelling is optional, the meta-model is state-agnostic, in the sense that it allows to represent both kinds of firewall platforms (or configurations). Finally, this decision makes the PIM more consistent, since in CONFIDDENT, the PIM should not model concepts regarding firewall platforms execution details (Fig. 4).

Table 3c
AFPL DSML (destination NAT).

Translated selector	Obligation	Dependencies	Syntax
Destination IP address	Mandatory		- Host IP
Destination port	Optional	Destination port must be specified in the original packet	- Number, Interval [p1,p2]

A detailed description of the PIM meta-model, with attribute cardinalities and attribute data types, is presented in Appendix II.

4.1. Example PIM instance

In Fig. 8 an example of an instantiated PIM is presented. This model conforms to Fig. 7 meta-model, and models Appendix III scenario. In this example, no information regarding interfaces has been modelled for simplicity reasons (both interface and direction are optional). As explained before, this PIM cannot represent other characteristics than the ones needed to make filtering decisions and NAT translations. However, this model could have (and if fact, it has) inconsistency and redundancy faults. Thus, the use of fault diagnosis algorithms is justified at this modelling stage.

4.2. PIM inconsistency and redundancy diagnosis stage

In Fig. 4 the concepts available at platform-independent modelling stage were shown. These concepts are available in the major firewall platforms and in all CONFIDDENT supported ones. The available NAT types are also the ones supported in major firewall platforms and in CONFIDDENT supported ones. However, apart from the higher level of abstraction gained from this modelling stage with respect to using firewall-specific languages, there is no mechanism to avoid the introduction of faults in models. Because of this, and in order to guarantee that the automatic transformation to the PSM is done from a fault-free model, a diagnosis stage is necessary. Once the diagnosis has been run and if the model contains faults, they must be corrected before the automatic PIM to PSM transformation stage. If the diagnosis algorithms used are fast

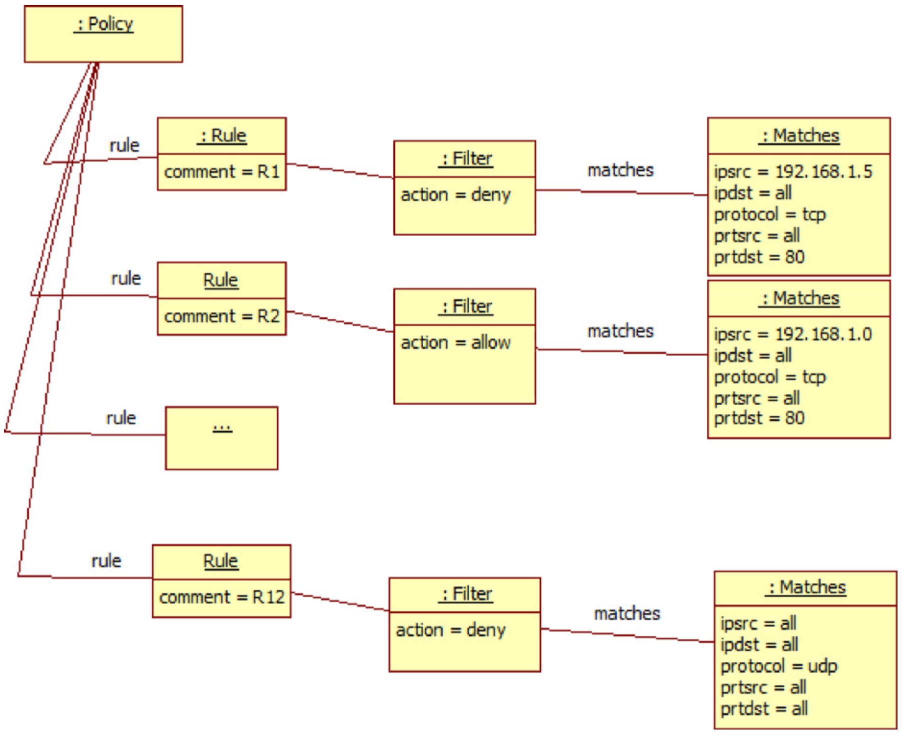


Fig. 8. Example PIM instance from Appendix III scenario.

enough, the diagnosis stage can also be run interactively while the model is being built, and faults can be given on a per-concept basis. Furthermore, since modifications to the model during fault correction can introduce new faults, the diagnosis algorithms must be re-run each time the instance is modified (interactively or when modifications are finished).

Since the PIM meta-model has been derived from AFPL (no new concept have been added), any of the fault diagnosis algorithms available in the scientific community (Pozo et al., 2009a) which can be applied to AFPL, can also be applied during CONFIDDENT PIM modelling. To the best of our knowledge, all proposed inconsistency and redundancy diagnosis algorithms can be run for AFPL models, since AFPL features the basic filtering selectors available in all firewall platforms (Chapple et al., 2009). However, since the diagnosis stage may also be run at the same time the model is built (i.e. on-line), fast diagnosis algorithms are necessary.

To the best of our knowledge, Liu and Gouda (2008) redundancy diagnosis algorithm, and Pozo et al. (2009b, 2010) inconsistency diagnosis algorithm are the best proposals to solve these problems. Both have been used in the proposed framework. The execution of these fault diagnosis algorithms over Appendix III scenario return the results presented in Table 4.

Model consistency and redundancy diagnosis have been implemented in Java as a library. The model to be diagnosed is passed to the library. The library takes the useful parts of the model for diagnosis purposes, runs the diagnosis algorithms and returns a minimal diagnosis. This result may be then interpreted by the modelling language again in order to mark the diagnosis result over the initial model. This part can be implemented using graphical modelling languages like GMF.

Table 4
Fault diagnosis results over Appendix III scenario.

Diagnosed rule	Fault type	Other rules implied in the same fault
R1	Inconsistency	R2, R3
R4	Inconsistency	R2, R3
R5	Inconsistency	R6, R7
R8	Inconsistency	R2, R3, R6, R7
R12	Inconsistency	R9, R10, R11
R6	Redundancy	R7
R9	Redundancy	R10

5. CONFIDDENT Netfilter IPTables PSM meta-model. M2M transformations

At this point, a PIM for CONFFIDENT has been specified, and at least one PSM for each supported platform is still needed. In this section a PSM meta-model specification for the IPTables firewall platform (from now on, PSM) is proposed, along with its EMF (Budinsky et al., 2003) implementation. The PSM has been designed on the basis of an analysis of the IPTables platform-specific details. Note that the proposed IPTables PSM specification is only one of the multiple possibilities that exist to model the platform, and for that reason should only be considered as a proof of concept. A set of M2M transformations between the proposed PIM and the PSM are also specified. The transformations have been implemented in ATL (ATLAS, 2007). There are many model transformation languages and approaches. In Czarnecki and Helsen (2006) a feature-based survey of these approaches can be found, and ATL is one of the most widely supported and extended. The possibility of PSM inconsistencies is analysed, and another model verification stage is proposed before the last stage of the modelling process is achieved (M2T transformations). However, this diagnosis stage is not always necessary as it depends on the possible relations between the PSM and PIM meta-models, as will be explained below. An example over previous section one is also given.

IPTables is an evolution of *ipchains*, the previous Linux Kernel firewall. It is a command-line oriented platform. That is, its configuration and ACL must be written in a *shell script*, where commands are introduced into a command interpreter (and to the kernel) one by one. However, most recent versions incorporate a feature which allows a file to be used as input and output to load and download, respectively, an ACL to or from the kernel. IPTables distinguishes among three types of traffic: incoming packets, which have as destination the firewall platform (*input* traffic); outgoing packets from the firewall platform (*output* traffic); and, finally, traffic that traverses two firewall interfaces (*forward* traffic). *Input* traffic is usually used for firewall maintenance purposes, while *output* traffic is usually used when the firewall needs to access external services. *Forward* traffic is the most common one, and it is used when a client at one interface of the firewall needs to access a service offered by a server at another interface.

IPTables defines a set of *chains* where rules are inserted and executed. Chains are composed of lists of rules. The order in which these rules are executed depends on the chains. That is, rules in some chains are executed before rules in others. The ordering between rules in the same chain is imposed by their position in the ACL. The first rule is the one with the highest priority and it will be matched at the first place (that is, a classical firewall execution engine). Chains are grouped in tables, where each table is associated with a different type of packet processing. IPTables has three pre-defined tables:

- **Filter.** This table contains three chains, each one associated to the three types of standard traffic: *input*, *output*, and *forward*.
- **NAT.** This table contains the rules associated to the network address translation protocol. This is one of the first tables that a packet must traverse. It is composed of three chains: *prerouting*, *postrouting*, and *output*. At each of these chains rules regarding destination and source NAT operations are also executed.
- **Mangle.** This table contains rules that modify some parameters of the packet header reaching the firewall, such as QoS. It is the first table that a packet must traverse and has five predefined chains: *prerouting*, *input*, *forward*, *output*, and *postrouting*.

All chains have at the same time input and output directions, depending on the source or destination IP addresses of the rules. In addition, IPTables allows the administrator to define its own chains and tables, and to define jumps between chains as an action when a packet matches a rule.

Once a packet arrives at an interface of the firewall, it must follow the IPTables flow (Fig. 9). Firstly, the packet is processed at the *prerouting* chain, where the firewall engine executes packet mangling, if it is activated. The packet is also transformed using destination NAT, if there is any rule in the NAT table that matches the packet. Then, the packet is processed in the *route* chain, where the engine decides (using routing information) if the packet must be forwarded or otherwise directed to the firewall itself. If the packet is to be forwarded, it is derived to the *forward* chain, where the matching engine checks if there is a rule in the filter table that matches the packet. If the packet is not allowed to pass, then it is dropped. If the packet is allowed to pass, then the packet is processed at the *postrouting* chain in order to match it in the NAT table (destination NAT in this case). Finally, the packet leaves the firewall.

If the packet is directed to the firewall itself, it is processed at the *input* chain, where the matching engine checks if the packet should pass or not (using the *filter* table). Before matching, mangling is executed if it is activated. If the packet is not allowed to pass, it is dropped at this point. If the packet is allowed, it finally reaches the firewall.

If the firewall itself sends a packet, it is processed at the *output* chain, where mangling is executed if it is activated, and where the matching engine checks if the packet can pass or not. If not, processing stops here. If it is allowed to pass, then the packet is processed at the *postrouting* chain in order to match it in the NAT table (destination NAT in this case). Finally, the packet leaves the firewall.

IPTables is one of the most versatile and complex firewall platforms available in the Open Source community as well as in the commercial one and in fact, its feature set can only be compared to the ones provided by BSD firewall platforms. An analysis of IPTables features related to all aspects of the platform has been presented in a previous paper (Pozo et al., 2009c). However, a summary is presented below.

- **Filtering selectors**. IPTables can filter packets according to TCP Flags, TCP Options, MAC source address, Time To Live (TTL), Type of Service (TOS), TCP Maximum Segment Size (TCPMSS), and of course all the concepts available in the PIM specification. Even the state of connections can be used to filter rules. Furthermore, IPTables supports a richer syntax (more data types) for each of the filtering selectors that can be modelled with the PIM.
- **NAT modes and selectors**. IPTables supports two NAT modes. These modes are exactly the same ones that can be modelled with the PIM. However, IPTables has more transformation selectors than the ones that can be modelled with the PIM. Furthermore, IPTables supports a richer syntax for each of the NAT selectors that can be modelled with the PIM.
- **Packet mangling**. IPTables allows changing the TTL and TOS header fields of TCP packets in the mangle table. Mangling is considered to be a filtering related feature, since mangling is applied before the IPTables standard filtering tables. Mangling related concepts are not available in the PIM.
- **Actions**. IPTables actions are the same three ones that can be modelled in the PIM. However, IPTables can use a special kind of action in order to jump between tables. In fact, IPTables users can split an ACL into different tables and jump between them using this special action. However, this action cannot be modelled with the PIM.
- **Logging**. IPTables allows different types of logging, on a per-rule, and on a per-chain basis. In the first case, each time a rule marked with logging is executed by the matching engine, it will be logged. In the second case, logging is defined for one or more chains and for one or both directions (in or out), logging all rules which are executed in the chain for which logging has been defined. Note that it is allowed to mix both kinds of logging. Logging related concepts are not available in the PIM.
- **Maximum rule hit frequency**. In IPTables, the number of times a rule can be matched can be restricted. Different parameters can be considered, such as a frequency limit (specified by an absolute number and optionally a time measure), and a maximum burst within a specified time interval. These concepts are not available in the PIM.
- **Filtering of malformed packets**. This feature allows to automatically filter (deny or reject) packets with invalid values in their header, such as invalid checksums. This concept is not available in the PIM.
- **Rule processing order**. IPTables processes rules in a forward-checking way (that is, try first the first rule in the ACL). This is also the way most firewall platforms process rules. This feature cannot be controlled by the user, and for this reason cannot be considered in the PIM and neither in the PSM.

5.1. IPTables PSM meta-model

Recall that CONFIDDENT PSM meta-modelling follows a modular methodology. Each (disjoint) set of related features for a firewall platform is considered as a different module for the platform meta-model, and thus platform-specific modelling is done through module combination. This methodology results in a user-selectable abstraction level/features available trade-off. For this reason, IPTables meta-model can be specified with all the modules needed to model all the features the platform has, or only with some modules

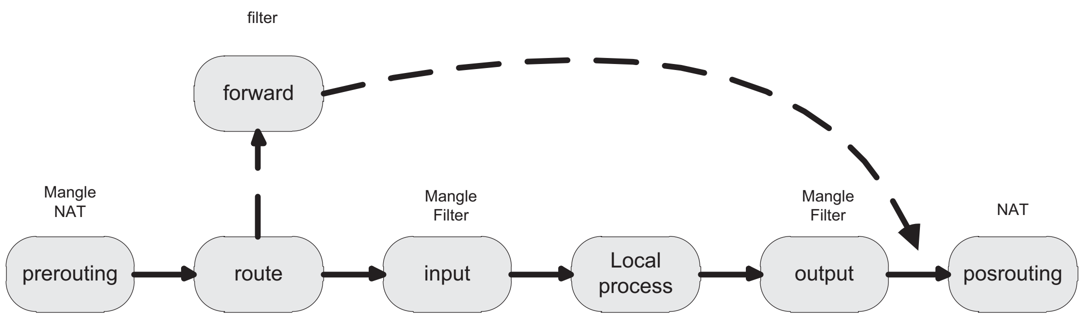


Fig. 9. IPTables platform packet flow diagram.

in order to be able to model only a reduced set of features. CONFIDDENT imposes absolutely no constraints on the number of modules that can be defined for the same target platform or the concepts available at each of them, provided that they are accompanied by its corresponding M2M and M2T transformations. In fact, this is one of CONFFIDENT main strengths. Note that if *all* features of a firewall platform are available as modules for its PSM, a full inverse transformation (from a firewall-specific ACL implementation to models) is also possible. In this paper, we have selected a few IPTables features in order to specify some modules as a proof of concept. These modelled features are chain information, logging and filtering of malformed packets. Note that many of these platform-specific features have a relation with the meta-classes that were defined in the PIM (which are represented in the PIM module). However, there are not relations between the concepts modelled at each of the modules. Fig. 12 depicts the proposed PSM meta-model for IPTables, which has been structured in four modules.

- **PIM module**. This module models the full set of concepts that has been modelled in the PIM.
- **Firewall Configuration Module**. This module models the concepts related with the configuration of the firewall management interface, as well as filtering-engine configuration features (like the filtering of malformed packets).
- **Chains Module**. This module models the three chains of IPTables platform.
- **Logging Module**. This module models the two different types of logging and existing relations with chains, filtering rules and NAT rules.

As with the PIM, the PSM is composed of a structure of related meta-classes and data types (Fig. 10). The right part of the figure represents the data types used in the meta-model. It is also important to remember that data types have associated some regular expressions that restrict the possible values each particular data type can have. However, these constraints have been omitted in Fig. 10 for readability reasons.

This PSM proposal includes all PIM meta-classes and attributes, but also includes several new meta-classes and data types to model the three platform-specific modules. The majority of these new meta-classes have some kind of relation with the PIM ones, because a platform-specific model of a firewall includes concepts that are modifiers or to PIM ones (used to extend functionality).

The PSM is composed of by the four described modules. The Firewall Configuration Module includes the concepts regarding the firewall platform by means of the IP_Firewall meta-class, which models the management IP address (*ip* attribute), if this IP is static or obtained from a server o a dial-up interface (*op* attribute), the interface identifier or name (*interface* attribute), and the filtering-engine configuration features like the filtering of malformed packets (*filtermalformed* attribute).

The Chains Module is composed by the Chains meta-class, and more precisely by the Input, Output and Forward meta-classes. At each of these chains it is possible to define the default action to be taken by the firewall-engine for both directions (*inflow* and *outflow*) for chains. For each of these chains the logging information is modelled by means of the Logging meta-class (Logging Module). Its *action* attribute represents if the logs should be sent to kernel or user space, and the *prefix* attribute represents a comment for each log register.

The PIM module includes all meta-classes described in the previous section and models all concepts related to filtering and NAT. However, single rules can also be associated with logging information, using the same Logging Module meta-classes.

A detailed description of the proposed IPTables meta-model with attribute cardinalities and attribute data types is given in Appendix IV. Let us recall from the previous section that there is no information regarding how state information is represented on a per-ACL basis because the meta-models are state-agnostic. Furthermore, as in the PIM meta-model rule priority is implicitly represented in the model using rule-order definition in the PSM.

The selection of one or another feature from the firewall platform to be included in the meta-model is related with the number and complexity of modules that would be necessary. Since modules concepts are disjoint, in general there would be no relations between PSM modules meta-classes. However, depending on the features modelled these modules could have relations with the part of the meta-model that comes from the PIM (i.e. the PIM Module). In some cases, these relations are not going to affect the concepts modelled at PIM level from their consistency point of view. This is the case of the IPTables modules proposed in this section. However, there may be other cases where the new concepts could modify previously modelled concepts. This is the case of modules that could model concepts like new actions for filtering rules, new NAT modes and/or transformation selectors, packet mangling, etc. In fact, the introduction of these new concepts must allow the administrator to modify the concepts that were previously modelled at the PIM. However, the PIM was diagnosed for inconsistencies and redundancies before entering PSM modelling stage, and if it is modified at the PSM modelling stage, models must be diagnosed again. For this reason, the selection of one or another feature to be modelled in the PSM has important implications that may affect the entire modelling process, including the diagnosis stages.

Note that the selected features for the proposed IPTables PSM do not affect the previously modelled part, since they do not affect the filtering or NAT rules. A collateral effect of the platform-specific features modelled in the proposed PSM is that the administrator is not allowed to modify the concepts modelled in the PIM Module, guaranteeing that new inconsistencies or redundancies are not introduced at the platform-specific modelling stage. One way to implement this is by means of an editor where these operations are forbidden.

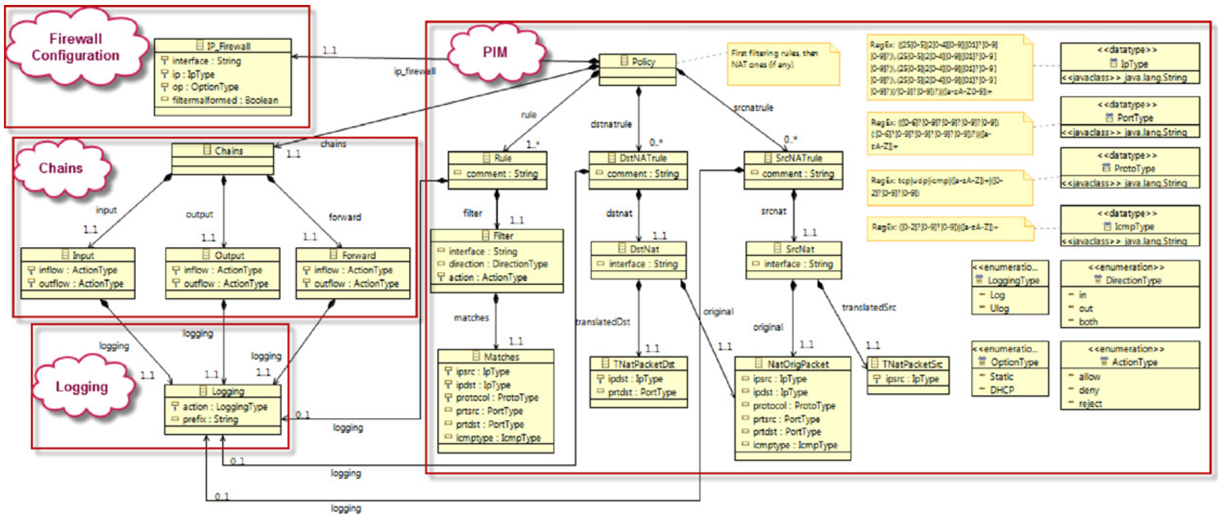


Fig. 10. CONFIDDENT IPTables PSM meta-model proposal (does not include all IPTables platform features).

5.2. *Model-to-model transformation*

In this section, transformation rules between the PIM and the proposed PSM are specified. Note that these rules are specific for each PSM. The proposed PSM consists of two differentiated parts. On one hand, the PSM models the concepts regarding the policy (as it was modelled in the PIM), and on the other hand, the platform-specific details, such as logging, firewall platform data, etc.

The concepts related to the policy remains with no modifications, since no new selectors have been considered in the PSM meta-model. Note that the administrator is not allowed to modify this information in the PSM in order to preserve the consistency and non-redundancy of the policy obtained from the PIM diagnosis stage. This first part of the PSM is, in fact, a direct transformation from the PIM.

The platform-specific details are generated in two stages. The first one is automatic, and it consists in the generation of the IP_Firewall and Chains meta-classes instantiated with their attributes set to their default values. This decision of assigning default values to attributes has been taken in order to help less experienced administrators with the modelling. In fact, if no platform-specific concepts are going to be modelled, the second stage is completely transparent for them. The second stage is manual, and is where experienced administrators can modify the values of these default attributes. Even new instances of the meta-classes can be introduced. Suppose, for example, that an administrator wants to activate the logging option for some rules. In this case, the administrator has to create instances of the meta-class Logging, and assign them to the instances of the Rule or one of the two NAT meta-classes for which logging must be activated.

An ATL implementation of the M2M transformation is given in Appendix V. The M2M transformation is composed of several parts. Although all of them are placed in the same file, we have preferred to separate them in order to improve the readability of the code.

The *pim2psm* rule adds in the PSM model classes and attributes initialized to their default values. Although, the default values can be hardcoded directly into the transformation, the provided reference implementation in Appendix V makes use of the concept of parameterization of the transformations. Hence, the transformation receives as input an external file (incorporated in the header of the transformation as "*parameters: XML*") where the default values of the transformation are specified. It is also possible to specify the parameters in a full-fledged model and implement a merging operation with tools for merging models such as Atlas Model Weaver (AMW) (Didonet Del Fabro et al., 2006) or Epsilon Merging Language (Engel et al., 2006). However, we have preferred to use an XML file, since for our problem model weaving is a too heavy solution, in the sense that too many resources are needed for it. The transformation rules read the these values using the statement *thisModule.getParameter('name')* where the ATL helper *getParameter* read the parameter *name* from the external file. An example of external file is given in Appendix VI. Using the parameterization of the transformation makes the implementation more maintainable since in the case of changing default values or adding new parameters, the transformation template will not need modifications.

Firstly, variables for assigning these default values are defined at *using* section. At *to* section, IP_Firewall meta-class is instantiated using the default values for its attributes. Next, Input, Output, and Forward meta-classes attributes are also instantiated to their default values, with logging activated. Next, Chains class is instantiated with Next, Input and Output instances. Finally, Policy meta-class is instantiated with IP_Firewall, Chains, Rules, DstNATrule and SrcNATrule. The final section has some ATL-specific checks for checking the values of optional attributes of classes, and thus contains no modelling or transformation details.

The *pim2psmRule* rule defines how Rule meta-classes defined in the PIM are transformed into the PSM ones. It takes Filter, DstNATrule and SrcNATrule instances from the PIM and transforms their attributes into the PSM ones. Note that as PSM attributes for the Filter meta-class are the same as in the PIM, this transformation rule just makes a direct mapping from the PIM. This transformation rule is also the responsible of setting the default values for the Logging meta-class. The final section is responsible for checking optional attributes, and has the same function as in previous rule.

The *pim2psmFilter* and *pim2psmMatches* transformation rules deal with the transformation details of the attributes of class Filter and Matches respectively. Both are direct mappings of PIM attributes, and both have a final section with the same purpose as in the previous transformation rules.

The pim2psmDstNatrule, *pim2psmDstNat* and *pim2psmTnatpacketdst* rules are used to transform DNAT rules from the PIM to the PSM. Again, they are a direct mapping from the PIM values to the PSM. *pim2psmDstNatrule* transforms the comment and calls *pim2psmDstNat* in order to transform DstNat class. *pim2psmDstNat* transforms TNatPacketDst and NatOrigPacket classes calling *pim2psmTnatpacketdst* and *pim2psmNatorigpacket* transformation rules. Then, *pim2psmTnatpacketdst* rule transforms TnatPacketDst class attributes. Finally, *pim2psmNatorigpacket* rule

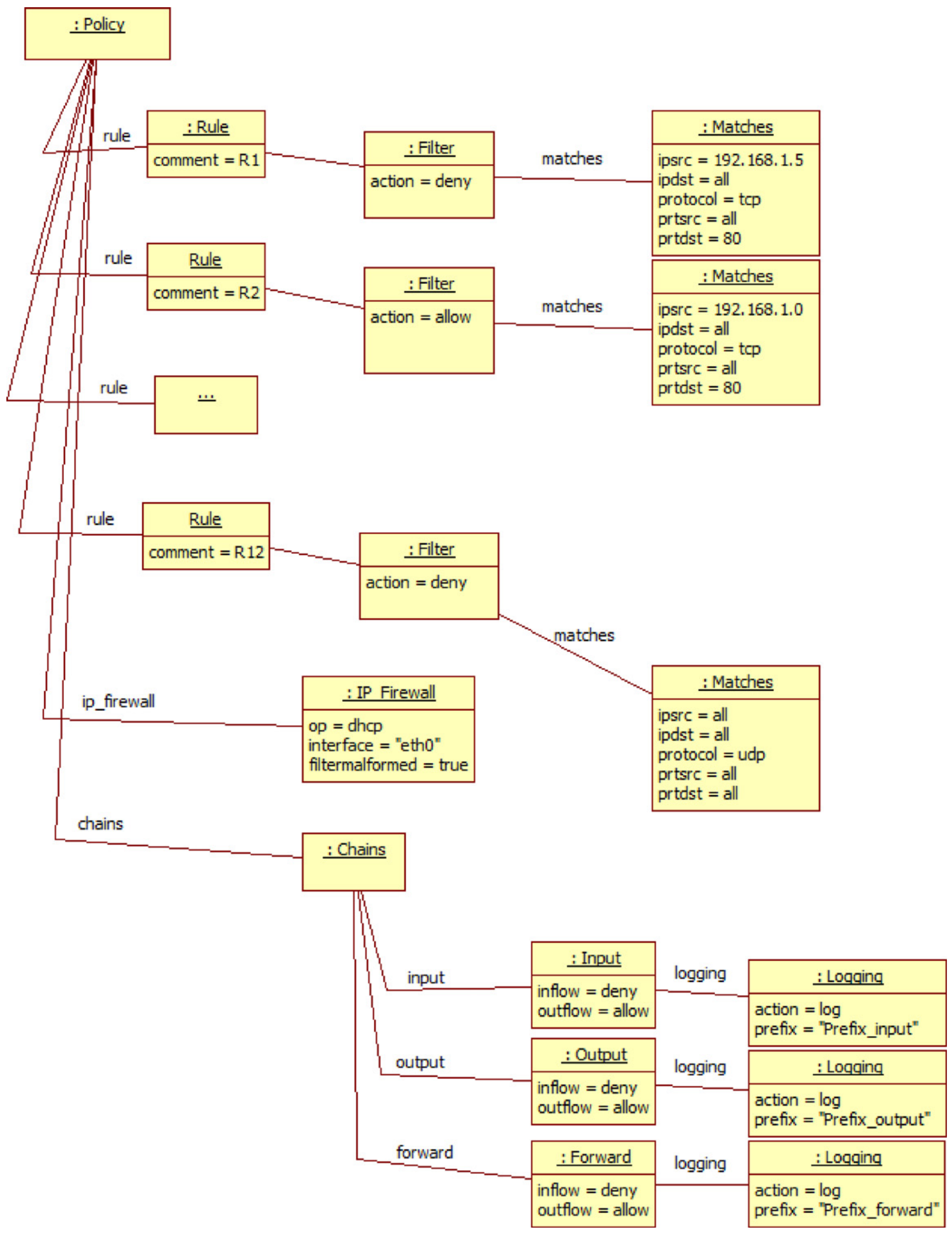


Fig. 11. Example PSM instance automatically obtained from Fig. 8 PIM instance.

transforms NatOrigPacket class attributes and is very similar to *pim2psmMatches* rule. All described transformation rules have a final section with the same purpose as in the previous ones. Finally, *pim2psmSrcNatrule*, *pim2psmSrcNat* and *pim2psmTnatpacketsrc* transformation rules do very similar transformations for SNAT and they are not going to be described.

5.3. *Example PSM instance*

In Fig. 11 an example of the automatically generated PSM instance is presented. This PSM instance conforms to the PSM meta-model, and it is the result of the M2M transformation presented in Appendix IV. Note that this instance contains concepts regarding the three modules available in the meta-model (Logging, Chains and Firewall configuration).

Again, rule order is implicit in the model (the first rule is the one specified first). This ordering will also be used for the M2T transformation that will be introduced in the next section. Note that, since this PSM has been automatically generated from the PIM, the values of the meta-classes attributes have been set to their default values. In addition, the information regarding individual logging of rules has not been modelled since, as it was mentioned in the previous subsection, this information must be directly modelled by an administrator, if needed.

However, since the default values in the PSM can be manually modified by an administrator, an analysis to know if new inconsistencies can be introduced is necessary.

5.4. PSM inconsistency and redundancy diagnosis stage

In any part of our framework where the administrator can interact with models, modifying their meta-classes and attributes, inconsistencies and redundancies can be introduced. In general and depending on each particular PSM, two possible types of faults can be introduced at this modelling level. The first one appears when the PSM modelling affects the concepts modelled at the previous stage and that have been transformed into the PSM through a M2M transformation. This is, if inconsistencies or redundancies can be introduced between the values that can take the PSM meta-classes that represent platform-specific features and values that can take the meta-classes that represent the concepts that were modelled at PIM stage. Surely this can be controlled in the automatic M2M transformation if defaults values are used, but when the administrator can modify the default values, inconsistent and redundant values may appear. The second type of fault appears when contradictory or redundant values can be introduced between the values that can take any of the PSM meta-classes that represent concepts that were not available at PIM modelling stage. In this paper it is assumed that M2M and MT2 transformations are trusted (that is, they cannot introduce inconsistencies or redundancies).

With respect to the proposed IPTables PSM, the M2M transformation is semi-automatic in the sense that the attributes and meta-classes of the PSM are automatically set to their default values, which the administrator can modify to fulfil its real requirements. An analysis of the meta-classes and valid attribute values of the PSM is needed in order to know if new inconsistencies or redundancies have been introduced when manual modifications have been made to these attributes. This analysis is also valid in order to test if any particular care may be taken for the M2M transformation when setting the default values.

Since the administrator cannot modify the instances of any kind of rule meta-classes at PSM modelling, then no new inconsistency or redundancy faults regarding the policy model can be introduced. Thus, no new consistency analysis of the PSM policy is needed for this module. However, the rest of modules need an analysis.

- **IP_Firewall**. This meta-class models information regarding the firewall configuration. The management IP address of the firewall could be dynamic or static (*op* attribute). If a dynamic IP is used, then the *ip* attribute must be empty, as it is set in the automatic M2M transformation. These two attributes could be mutually inconsistent, because setting an IP is inconsistent with the value *dynamic* of the *op* attribute, and vice versa. For example, the administrator can set an IP for the firewall and leave the *op* attribute set to dynamic. This inconsistency is checked via an OCL constraint at the PSM meta-model (Appendix IV, IP_Firewall meta-class, Constraint: "self.op = dhcp implies self.ip = null").
- **Chain, Input, Output, Forward**. These meta-classes model the default policies for the three standard chains of IPTables. The default policy is set with the *inflow* and *outflow* attributes using the ActionType data type. Thus, no inconsistency or redundancy faults are possible.
- **Logging**. Logging information can be added to chains in order to log each time the default rules are executed, or to single rules, where a rule associated with a logging action is registered each time it is executed. Since IPTables does allow the administrator to mix logging actions for single rules and for the default action in chains, it is not possible to introduce inconsistencies with the Logging meta-class.

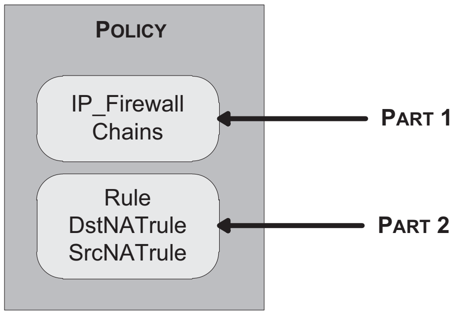


Fig. 12. IPTables transformation process parts.

As can be observed from the analysis of the meta-classes, attributes, and possible values of the proposed IPTables PSM meta-model, the only inconsistency that can be introduced in the model is related to the firewall IP address configuration information, which can be identified directly in the PSM meta-model using an OCL constraint which prevents the introduction of the invalid value. For the proposed PSM there are no possible cases, since the PIM Module cannot be modified by the administrator, and is a direct transformation from the PIM model. One way to implement this is by means of an editor where these operations are forbidden.

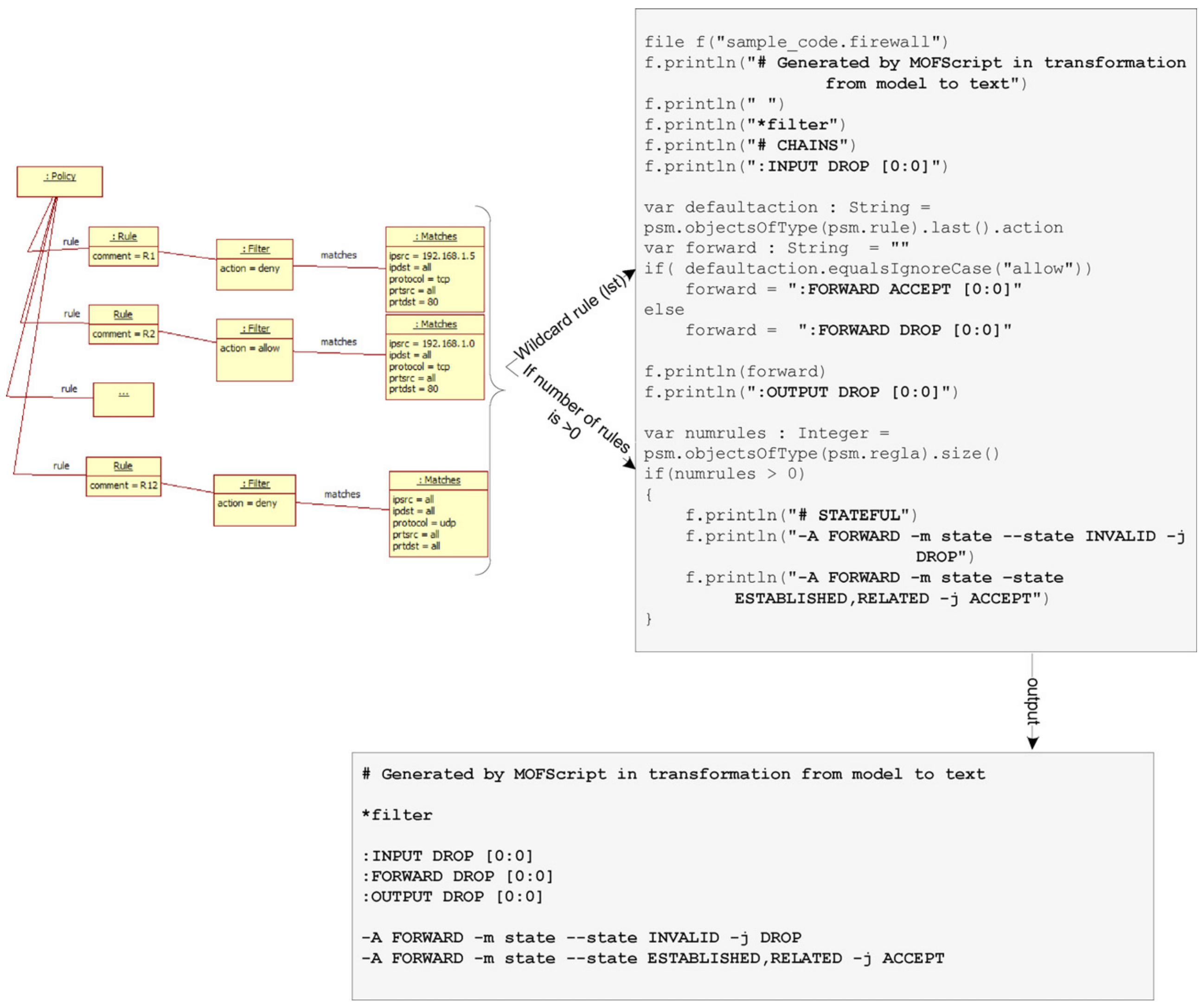
6. Automatic code generation. M2T transformation

In this section, a M2T transformation from the PSM to IPTables SAVE format is described. The transformations have been implemented in MOFScript (MOFS, 2007).

The proposed transformation for IPTables can be divided into two different parts (Fig. 12). On the one hand, all PSM classes not related to the filtering and NAT rules are transformed. On the other hand, Filtering, NAT rules and their logging settings (on a per-rule basis) are transformed. Note that the order in which information appears in the output file depends entirely on the particular firewall platform. The structure of an IPTables file imposes that in the first place it should appear the information not related to filtering and NAT rules, and then all the rules.

Transformations in MOFScript are composed of a set of transformation rules (the full implementation is presented in Appendix VII). For the proposed M2T transformation, we have defined two rules, corresponding to the two parts of the process mentioned above. The *main()* rule is the transformation entry point. The first lines deal with the header of the IPTables file, and they set the actions for the three default chains (Input, Output, Forward) as they were defined in the PSM (that is, with default values, or with values redefined by the administrator). In this proof of concept, only the Forward chain has been considered (the other chains are very similar). The default action value for this chain is the same one as the action of the last filtering rule. Next, the firewall engine is set to *stateful* by default. This is also the way other firewall engines like Cisco PIX and Checkpoint FW-1 work. Next, filtering rules are transformed to code through the transformation rule *generaterules()*. If there were NAT rules in the PSM, they must be transformed also at this point. Finally, a footer instructing IPTables to commit changes and to execute the ACL is written into the file, finishing the transformation.

For simplicity reasons, all rules are introduced into the *forward* chain, but the use of other chains is straightforward (by checking rules source and destination IP addresses). Next, the *interface* and *direction* where to apply the rule are transformed. Next, transformations of source and destination IP addresses, as well as source and destination ports are done, checking if they represent wildcards



```
file f("sample_code.firewall")
f.println("# Generated by MOFScript in transformation
              from model to text")
f.println(" ")
f.println("*filter")
f.println("# CHAINS")
f.println(":INPUT DROP [0:0]")

var defaultaction : String =
psm.objectsOfType(psm.rule).last().action
var forward : String  = ""
if( defaultaction.equalsIgnoreCase("allow"))
    forward = ":FORWARD ACCEPT [0:0]"
else
    forward =  ":FORWARD DROP [0:0]"

f.println(forward)
f.println(":OUTPUT DROP [0:0]")

var numrules : Integer =
psm.objectsOfType(psm.regla).size()
if(numrules > 0)
{
    f.println("# STATEFUL")
    f.println("-A FORWARD -m state --state INVALID -j
                      DROP")
    f.println("-A FORWARD -m state -state
         ESTABLISHED,RELATED -j ACCEPT")
}
```

```
# Generated by MOFScript in transformation from model to text

*filter

:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]

-A FORWARD -m state --state INVALID -j DROP
-A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Fig. 13. IPTables file header generation (from *main()* transformation rule).

or not (value *all* for the attribute). If the protocol is ICMP, then the ICMPType is also considered in the transformation. Finally, the rule action along with logging information is considered together. At this point, a call to *generateForwardLogging()* is done in order to transform the default logging action of the *forward* chain.

6.1. *Example*

In this section we show how our M2T transformation is executed on the PSM instance of Fig. 11. In the first part of the M2T transformation, the headers of the IPTables file are generated. The three typical chains in IPTables are Input, Output and Forward. By default all chains have a *drop* action, but in case that a default action has been specified in the Policy model, its action is taken. After that, the firewall is configured in its *stateful* mode only if there is at least one rule in the model. This has been set as an OCL constraint, and translated into a simple if-statement (*if (numrule* > 0, from Fig. 13) when the number of rules in the model is greater than zero. An example is presented in Fig. 13.

In the second part, the instances of the Rule meta-class are transformed iterating over them. The method *generaterules()* is called for each rule instance. The first instance of a rule in the PSM (the one whose attribute comment is R1) has been taken as an example to illustrate the code generation. The first lines of *generaterules()* copy into a set of variables the information related to the rule instance for its later processing. Each attribute value is filtered and analysed to generate a correct sentence in IPTables format. The output text is generated in a String variable which will be dumped over a text file at the end of the transformation procedure.

In Fig. 14(a) and (b) a transformation for this first rule is showed. In order to produce a human-readable code, it has been included a comment before each rule in the file. The generated comment has been obtained from the attribute *comment*. After that, all rules are introduced in the *FORWARD* chain, as a consequence we have to add "*-A FORWARD*" to the sentence. Depending on the values stored in the variables, different sentences will be formed. The *direction* variable is filtered in order to check the direction of the packet, if the *interface* attribute has been set. If the value is *in* a "*-i*" flag followed by the value of the *interface* attribute is added to the file. The same is done if the value is *out*, but with a "*-o*" flag. Otherwise, we have to add both flags. The *ipsrc* and *ipdst* variables are checked for the value *all*. If the variables do not have a value *all*, the IP information needs to be included in the rule. For the source IP an "*-s*" followed by the value of the *ipsrc* attribute is written to the file, and for the destination IP a "*-d*" flag followed by the value of (the) *ipdst* (attribute). The *protocol* variable is filtered to check if it is a TCP value or not. In the case that it is working with a TPC protocol, we add a "*-p*" flag followed by the protocol name. After that, it is necessary to check the source port (*prtsrc*) and the destination port (*dstprt*). In both cases, the associated flags are added to the output file in the right format.

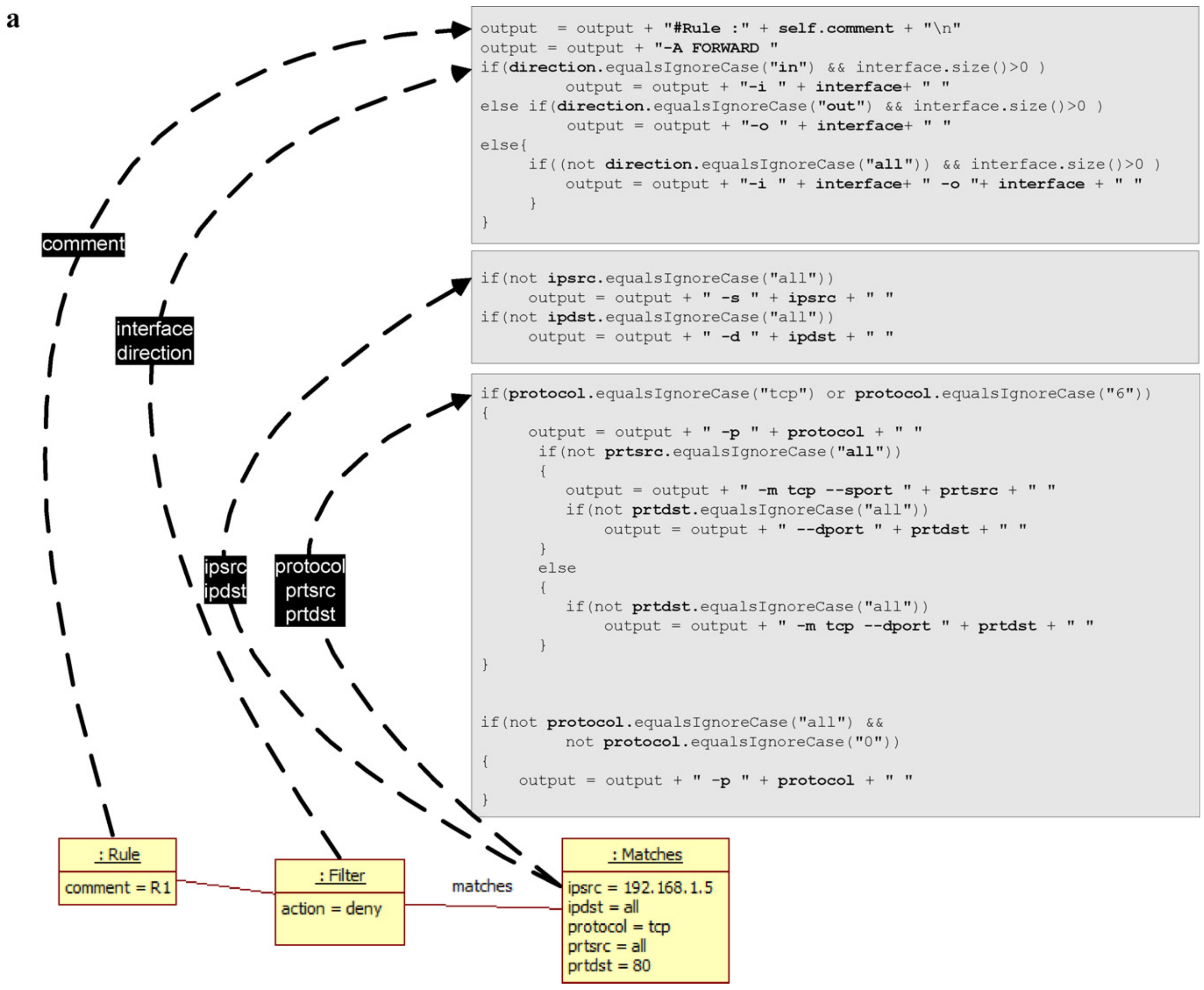
In Fig. 14(b) the *action* variable is transformed. In this case, the treatment for the three cases: *deny*, *allow* and *reject* have not been included in the example in order to improve legibility (Appendix VII contains the full code), but only for the *deny* action (the case for R1). The process is the same for *allow* and *reject* with the only difference being that the action is transformed to the output file. The *deny* value must be transformed into a DROP action, *allow* value to ACCEPT, and *reject* value to REJECT. Then it is necessary to check if logging has been specified in the rule or not. In order to improve legibility, rules in this example do not have their own logging information. Instead, the logging specified in the chain where the rule is associated to, is used. This is done by calling *generateForwardLogging()*. The logging information included there is transformed as the log part in the output file.

A complete output generated is presented here in Fig. 15. The implicit rule order in the model has been used to generate the ordering for the rules in the code. This file can be directly executed in any IPTables firewall platform, with no user intervention at code-level.

7. Discussion of results

One of the main aims of CONFIDDENT is to provide a model-driven design and maintenance framework which can satisfy a wide spectrum of firewall administrators. CONFIDDENT is based on several modelling stages, each with a different abstraction level, and has been heavily inspired by the MDA view of the Model-Driven Engineering paradigm. Model-Driven Architecture (MDA) development aims at generating systems from high-level system models and requirements models, taking away much of the concurrent manual changing of artefacts at the different stages of software development. It promises better leverage on building quality (i.e., stakeholder value) into the software products and should support the measurement of software quality at different stages of the development life cycle. The three primary goals of MDA are portability, interoperability and reusability. The first stage consists on defining platform-independent models. These models conform to a platform-independent meta-model (this meta-model is fixed in CONFIDDENT), which represents the highest level of abstraction. With this meta-model, an administrator is able to model the vast majority of features of the market-leader firewall languages supported in CONFIDDENT.

a



```
output  = output + "#Rule :" + self.comment + "\n"
output = output + "-A FORWARD "
if(direction.equalsIgnoreCase("in") && interface.size()>0 )
         output = output + "-i " + interface+ " "
else if(direction.equalsIgnoreCase("out") && interface.size()>0 )
         output = output + "-o " + interface+ " "
else{
     if((not direction.equalsIgnoreCase("all")) && interface.size()>0 )
         output = output + "-i " + interface+ " -o "+ interface + " "
     }
}
```

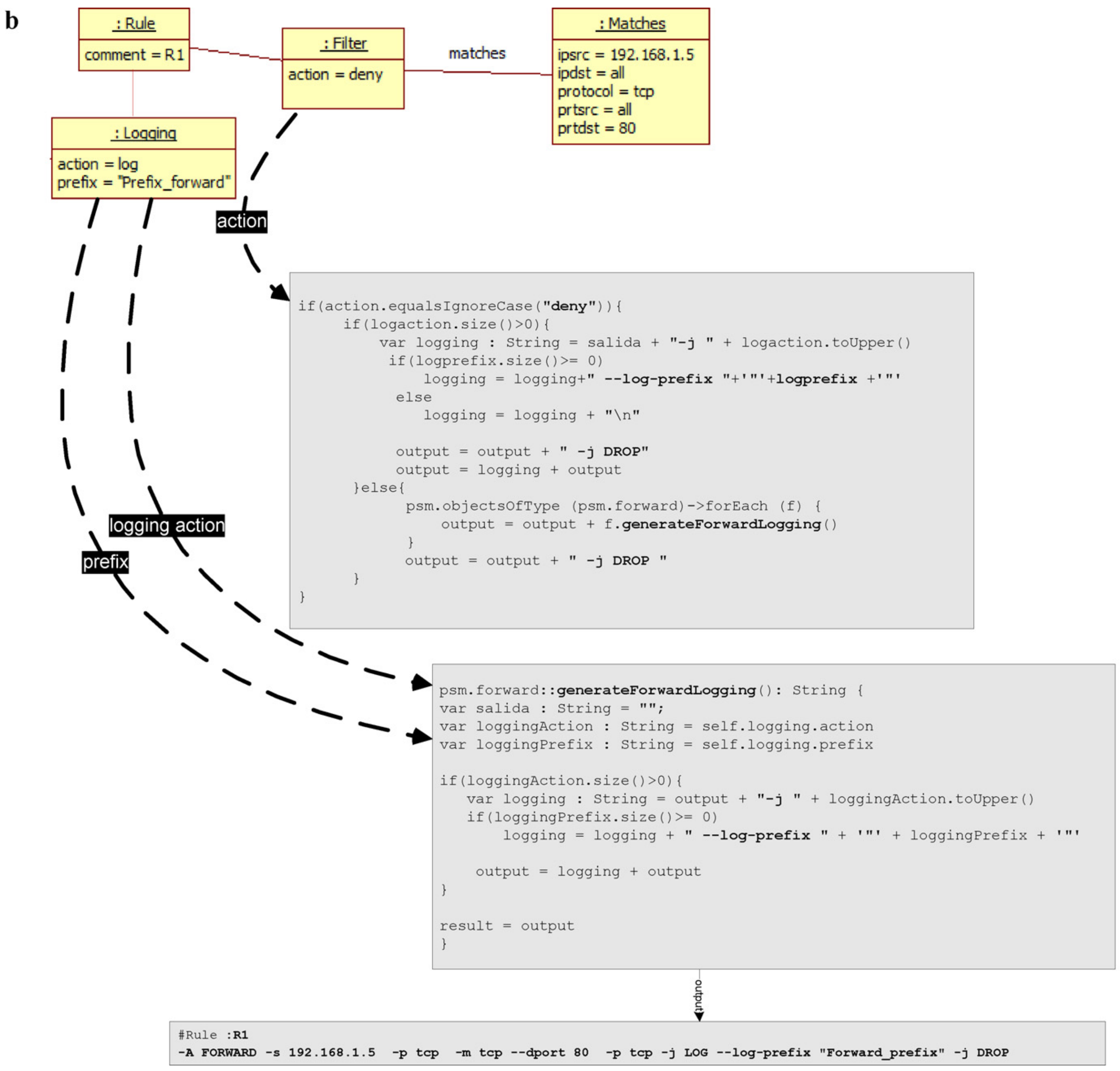
```
if(not ipsrc.equalsIgnoreCase("all"))
     output = output + " -s " + ipsrc + " "
if(not ipdst.equalsIgnoreCase("all"))
     output = output + " -d " + ipdst + " "
```

```
if(protocol.equalsIgnoreCase("tcp") or protocol.equalsIgnoreCase("6"))
{
     output = output + " -p " + protocol + " "
      if(not prtsrc.equalsIgnoreCase("all"))
      {
         output = output + " -m tcp --sport " + prtsrc + " "
         if(not prtdst.equalsIgnoreCase("all"))
             output = output + " --dport " + prtdst + " "
      }
      else
      {
         if(not prtdst.equalsIgnoreCase("all"))
             output = output + " -m tcp --dport " + prtdst + " "
      }
}


if(not protocol.equalsIgnoreCase("all") &&
         not protocol.equalsIgnoreCase("0"))
{
    output = output + " -p " + protocol + " "
}
```

Fig. 14. (a, b) Rule R1 transformation example.

b



```
if(action.equalsIgnoreCase("deny")){
     if(logaction.size()>0){
         var logging : String = salida + "-j " + logaction.toUpper()
          if(logprefix.size()>= 0)
             logging = logging+" --log-prefix "+'"'+logprefix +'"'
           else
             logging = logging + "\n"

           output = output + " -j DROP"
           output = logging + output
      }else{
            psm.objectsOfType (psm.forward)->forEach (f) {
                output = output + f.generateForwardLogging()
            }
            output = output + " -j DROP "
      }
}
```

```
psm.forward::generateForwardLogging(): String {
var salida : String = "";
var loggingAction : String = self.logging.action
var loggingPrefix : String = self.logging.prefix

if(loggingAction.size()>0){
   var logging : String = output + "-j " + loggingAction.toUpper()
   if(loggingPrefix.size()>= 0)
       logging = logging + " --log-prefix " + '"' + loggingPrefix + '"'

    output = logging + output
}

result = output
}
```

```
#Rule :R1
-A FORWARD -s 192.168.1.5  -p tcp  -m tcp --dport 80  -p tcp -j LOG --log-prefix "Forward_prefix" -j DROP
```

Fig. 14. (Continued)

The second modelling stage is where the administrator selects the target platform for the final ACL, and where platform-specific details are modelled, if any, building a platform-specific model (PSM). Several platform-specific meta-models (at least one for each target platform) are also necessary at this modelling stage. Ideally, an administrator must be able to model the full feature set of the selected firewall platform by means of its meta-model, and thus the meta-model must contain enough concepts to model the features. However, this will surely result in a very complex meta-model where some of its concepts may never be used by less experienced administrators. To prevent this, CONFIDDENT uses modularized platform-specific meta-models: a target platform meta-model is built from small parts, where each of these parts defines a platform-specific feature. The administrator is thus free to select the trade-off between abstraction level/features available, and thus this modelling stage can be understood as a variable abstraction level one. Following this approach, existing platform-specific meta-models can be extended or modified, and new ones can be created to fulfil any administrator needs. Even repositories of platform-specific meta-models can be provided to CONFIDDENT users. However, the MT2 transformation has not modularized in the current reference implementation, modifying the PSM to add new modules will involve modifying the (monolithic) M2T transformation. In fact, the platform-specific modelling stage may completely be transparent if the PIM fulfil administrators' modelling needs.

The last modelling stage is where the firewall-specific ACL (i.e. the implementation) is obtained through an automatic M2T transformation by means of predefined transformation rules. Although the implemented ACL can be modified by an administrator, it is not recommended, since traceability with the models may be loosed. Furthermore in order to accomplish any modification, administrators need to know the details of the particular firewall-specific language and platform.

With respect to the reviewed high-level languages and commercial and Open Source ACL design tools, CONFIDDENT provides abstraction in both platform functionality and language syntax. We

```
# Generated by MOFScript in transformation from model to text

*filter

:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]

-A FORWARD -m state --state INVALID -j DROP
-A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT

#Rule :R1
-A FORWARD  -s 192.168.1.5  -p tcp  -m tcp --dport 80  -p tcp -j LOG --log-prefix "Forward_prefix" -j DROP
#Rule :R2
-A FORWARD  -s 192.168.1.0  -p tcp  -m tcp --dport 80  -p tcp -j LOG --log-prefix "Forward_prefix" -j ACCEPT
#Rule :R3
-A FORWARD  -d 170.0.1.10  -p tcp  -m tcp --dport 80  -p tcp -j LOG --log-prefix "Forward_prefix" -j ACCEPT
#Rule :R4
-A FORWARD  -s 192.168.1.0  -d 170.0.1.10  -p tcp  -m tcp --dport 80  -p tcp -j LOG --log-prefix "Forward_prefix" -j DROP
#Rule :R5
-A FORWARD  -s 192.168.1.60  -p tcp  -m tcp --dport 21  -p tcp -j LOG --log-prefix "Forward_prefix" -j DROP
#Rule :R6
-A FORWARD  -s 192.168.1.0  -p tcp  -m tcp --dport 21  -p tcp -j LOG --log-prefix "Forward_prefix" -j ACCEPT
#Rule :R7
-A FORWARD  -s 192.168.1.0  -d 170.0.1.10  -p tcp  -m tcp --sport any  --dport 21  -p tcp -j LOG --log-prefix "Forward_prefix" -j ACCEPT
#Rule :R8
-A FORWARD  -p tcp  -p tcp -j LOG --log-prefix "Forward_prefix" -j DROP
#Rule :R9
-A FORWARD  -s 192.168.1.0  -d 170.0.1.10  -p udp  -m udp --dport 53  -p udp -j LOG --log-prefix "Forward_prefix" -j ACCEPT
#Rule :R10
-A FORWARD  -d 170.0.1.10  -p udp  -m udp --dport 53  -p udp -j LOG --log-prefix "Forward_prefix" -j ACCEPT
#Rule :R11
-A FORWARD  -s 192.168.2.0  -d 170.0.2.0  -p udp  -p udp -j LOG --log-prefix "Forward_prefix" -j ACCEPT
#Rule :R12
-A FORWARD  -p udp  -p udp -j LOG --log-prefix "Forward_prefix" -j DROP

COMMIT
# COMPLETED
```

Fig. 15. Generated code for Appendix III scenario.

believe the CONFIDDENT PIM meta-model is the most complete model of those reviewed. As we have explained in the paper, this modelling stage would be enough for most ACL developments, and the model obtained is valid for these six platforms. This contrasts with the reviewed multi-vendor ACL developments tools, where the target platform must be specified before ACL modelling. Furthermore, in CONFIDDENT specific details for a particular target platform can be modelled through a platform-specific meta-model. These meta-models are designed in a modular way, assuring that complexity is always adjusted to experience and modelling needs of administrators. Even transformations between different platforms PSM are possible (possibly with information loss). This also contrasts with multi-vendor design tools, where models are fixed and no transformations are allowed between different platform models.

Through the use of complete meta-models for each of the supported firewall platforms, a direct import of firewall-specific ACLs with no information loss is possible via an inverse transformation. However, this is out of the scope of this paper.

However, the use of the proposed architecture does not yet guarantee the absence of faults of the models. For this reason, CONFIDDENT includes one fault diagnosis stage at each modelling level. Diagnosis can even be run interactively while modelling (even during model maintenance) if algorithms are efficient enough. Due to these stages, the administrator is able to correct these faults during modelling, and not in the generated ACL, which contributes to reduce the time and budget spent on this task. This feature is not supported by any of the analysed high-level languages or ACL design tools.

In summary (Table 5), CONFIDDENT provides both a new level of administrator productivity and a new level of confidence on the developed code. CONFIDDENT represents the first proposal of abstraction to design and manage firewall ACLs which allows administrators to work both on a platform independent model, and on a platform specific model for a particular target firewall platform, and where models are diagnosed for faults before entering in any new modelling stage.

8. Conclusions and future works

In this paper, two of the traditional firewall ACL problems have been revisited: complexity of ACL design, development, and maintenance, and ACL inconsistency and redundancy diagnosis. Although several works listed in the bibliography deal with solutions to these two problems, we found none of them completely satisfactory. When languages have a large feature set, they are more complex to be used than a firewall-specific one. When they have a small feature set (and usually are more abstract and not firewall-specific), some very specific and advanced features could be needed by experienced administrators and not available in them. Furthermore, the use of models does not guarantee that the resulting generated code is consistent and non-redundant.

The tools and methods that aim to help administrators during ACL development process should gain in functionality and ease of use at rates to match the increase in firewall ACL development complexity. In this paper CONFIDDENT, a CONsistent and non-redundant FIrewall Design, DEvelopment, and maiNTenance framework has been proposed. In CONFIDDENT, simple and abstract platform-independent ACL models (which might satisfy a large base of inexperienced firewall administrators) can be combined with more complex and less abstract platform-specific ACL models (which can satisfy experienced administrators) through a series of automatic model transformations. CONFIDDENT takes into account inconsistencies and redundancies that can be introduced during modelling stages and integrates different model verification and diagnosis stages. Models can automatically be transformed to firewall-specific ACLs. CONFIDDENT currently supports a wide range of firewall platforms, which represent market-leaders: Linux IPTables, Cisco PIX, FreeBSD IPFilter, FreeBSD IPFirewall, OpenBSD

Table 5
CONFIDDENT comparison with reviewed tools.

	Firewall Builder	Cisco ASDM	Checkpoint Blades	LogLogic ChangeManager	CONFIDDENT
Focus is on syntax abstraction or in functionality abstraction	Syntax	Syntax	Syntax	Hybrid	Both
Topology/logic separation	√	√	√	√	√
Firewall-specific ACL import	Partial	√	N/A	√	√
Models are user-extensible	×	×	×	×	√
Able to model other devices such as IPS	×	√	√	√	×
Adjustable abstraction level	×	×	×	×	√
Transformations possible between platform models	×	×	×	×	√
Multi-vendor compilation	√	×	×	√	√
Integrated inconsistency and/or redundancy diagnosis	Only shadowing	×	×	×	√
FW platforms supported	Cisco, IPTables, HP ProCurve, BSD PF, IPFW, IPFilter	Cisco	Checkpoint	Cisco, Juniper, Checkpoint, Fortinet, IPTables	Cisco, IPTables, BSD PF, IPFW, IPFilter, Checkpoint

Packet Filter, and Checkpoint and its reference implementation is based in the Model-Driven Architecture.

To the best of our knowledge, this is the first published work that addresses the design, development, and maintenance of consistent and non-redundant firewall ACLs at the design stage using a model-driven approach. With CONFIDDENT, it is possible to create tools that effectively fill the gap between current modelling languages and firewall-specific ACLs, providing firewall administrators with tools that represent a real alternative for the whole life-cycle of ACL management.

The present work can be extended in several ways. For example, UML can be used as a modelling language to represent access control requirements (the Computation Independent Model, CIM), as well as the automatic transformation into the PIM. This will add another, higher, abstraction level to the framework. This new abstraction level could include risk information in order to automatically generate specific parts of the model in order to assess the protection of the higher risk assets. In addition, multiple firewalls can also be considered in the meta-model, which would require the specification of the network topology in order to assign specific rules to firewalls. At a lower level, optimizations can be made in the M2T transformations in order to generate more efficient code.

Acknowledgement

We would like to thank the anonymous reviewers for their constructive comments on the early version of this paper. This work has been partially funded by Spanish Ministry of Science and Education project under grant TIN2009-13714, and by FEDER (under ERDF Program).

Appendix I. AFPL XSD

AFPL is composed of a policy, which consists of list of Condition/Action rules. AFPL should have at least one filtering rule, although NAT rules are always optional. This is because there is a required rule at the end of firewall ACLs that represents the default policy that should be taken if no other match is found in it.

Filtering rules are composed of the firewall interface on which the rule is going to be applied and the direction of the flow of packets (*interface* and *direction* tags, which are optional), the condition part (*matches* tag), the action to be taken when a packet matches the rule, and finally a comment which represents the documentation of the rule. The condition part of the rule uses the selectors presented in Table 3, which also have specific syntaxes. These selectors are Source and Destination IP (mandatory), Source and Destination Ports (only if protocol is TCP or UDP), Protocol (mandatory), ICMP Type (only if protocol is ICMP), and finally interface and direction (optional).

NAT rules are of two types: source NAT and destination NAT. Each kind of rule *(srcnatrule, dstnatrule)* changes (translates) source or destination TCP/IP headers respectively. Both NAT rules can have a comment and an interface where the rule is applied (the interface direction can be inferred automatically depending upon the NAT mode). Again, both NAT rules need an original packet *(NatorigPacket)* which has the same selectors as the ones used in filtering rules) and a set of selectors to be translated. Note that this set of translated selectors is the main difference between the two NAT modes.

The presented XSD has been implemented in RelaxNG Compact, which is a schema language based on XML. Currently, it is in the final stage of standardization (ISO/IEC 19757-2).

```
#Firewall Policy Schema RelaxNG Compact version 11/4/08
#Abstract Firewall Policy Language
#V2.11 (23/01/09)

#Each rule may be a Filter Rule or Nat Rule. At least one filter
#rule is mandatory

grammar {
  start =
    element policy {
      element rule {
        element filter { filterType },
        element comment { text }?
      }+,
      element srcnatrule {
        element srcnat { srcNatType },
        element comment { text }?
      }*,
      element dstnatrule {
        element dstnat { dstNatType },
        element comment { text }?
      }*

    }

  filterType =
    element interface { text }?,
    element direction { string "in" | string "out" | string "both" }?,
    element matches { matchesType },
    element action { string "allow" | string "deny" | string "reject" }

  srcNatType =
    element interface { text }?,
    element original { natOrigPacket },
    element translatedSrc { TnatPacketSrc }

  dstNatType =
    element interface { text }?,
    element original { natOrigPacket },
    element translatedDst { TnatPacketDst }

  matchesType =
    element ipsrc { ipType },
    element ipdst { ipType },
    element protocol { protoType },
    element prtsrc { portType }?,
    element prtdst { portType }?,
    element icmptype { icmptType }?

  natOrigPacket =
    element ipsrc { ipType }?,
    element ipdst { ipType }?,
    element protocol { protoType }?,
    element prtsrc { portType }?,
    element prtdst { portType }?,
    element icmptype { icmptType }?

  TnatPacketSrc =
         element ipsrc { ipType }

  TnatPacketDst =
         element ipdst { ipType },
         element prtdst { portType }?

  ipType = xsd:string { pattern = "((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(/[0-3]?[0-9])?)|([a-zA-Z0-
9])+" }
  protoType = xsd:string { pattern = "tcp|udp|icmp|([a-zA-Z])+|([0-2]?[0-9]?[0-9])" }
  portType = xsd:string { pattern = "(([0-6]?[0-9]?[0-9]?[0-9]?[0-9])(:[0-6]?[0-9]?[0-9]?[0-9]?[0-9])?)|([a-zA-
Z])+" }
  icmptType = xsd:string { pattern = "([0-2]?[0-9]?[0-9])|([a-zA-Z])+" }
}
```

Appendix II. PIM description

In this appendix all the elements of CONFIDDENT PIM meta-model are described by means of a template. There are three kinds of templates: meta-classes, data types and enumeration templates. Each template is divided in different parts which represent the main features of the element that is being described in the template. Thus, a meta-class template is composed of four areas (description, attributes, relations, and constraints), while data types and enumeration templates only have a description and a constraint area. The description area contains a brief explanation of the element purpose. The attributes area holds the set of attributes that have been defined in a meta-class. Each attribute has a name, a cardinality, a type, and a brief explanation of its purpose. The relations area includes the different relationships that a meta-class has. Each relation is specified by means of a name, its cardinality, the source and target meta-classes, and a brief description. Finally, the constraints area contains some constraints associated to the particular element that is being modelled. In the meta-class templates, the constraints are expressed in OCL, while in the data type templates, they are specified by means of regular expressions. The different values which an enumeration can have are also considered as a constraint in the enumeration template, because they limit the set of possible values in an enumeration. It is interesting to highlight that constraints have not been depicted in the graphical meta-model (Fig. 7) for readability reasons.

It is necessary to clarify that MDA does not require the use of UML to specify PIMs or PSMs, but it is just a recommendation. When a developer has to define a meta-model, he has to choose the meta-modelling technique: a UML-based profile (also named lightweight extension) or a MOF-based meta-model (or heavy-weight extension). There are different reasons for selecting one of them (Desfray, 2000).

MetaClass	Policy[a]
Description	The ACL of a Firewall
Attributes	
Relations	**rule [1..*]:** (Policy → Rule) Through this relation it is possible to obtain all the rules associated to the policy **dstnatrule [0..*]:** (Policy → DstNATRule) Through this relation it is possible to obtain all the destination NAT rules associated to the policy **srcnatrule [1..*]:** (Policy → SrcNATRule) Through this relation it is possible to obtain all the source NAT rules associated to the policy
Constraints	context Policy inv: (self.rules → size() ≥ 1)

[a] Policy is declared as root of meta-model, hence this instance must appear only once in the models.

MetaClass	Rule
Description	Condition/action rule
Attributes	**comment [0..1]:** String Documentation for a rule
Relations	**filter [1..1]:** (Rule → Filter) Through this relation it is possible to obtain the general filtering parameters of a filtering rule
Constraints	context Rule inv: (self.matches.size() = 1)

MetaClass	Filter
Description	Models the general parameters of a filtering rule
Attributes	**interface [0..1]:** String Firewall interface where the rule is applied **direction [0..1]:** Direction Type Direction of packet flow **action [1..1]:** Action Type Action to be taken when a packet matches a condition
Relations	**matches [1..1]:** (Filter → Matches) It defines the match parameters of a rule
Constraints	context Rule inv: (self.filter.size() = 1)

MetaClass	Matches
Description	Models filtering parameters of a Rule
Attributes	**ipsrc [1..1]:** IpType IP source of a rule **ipdst [1..1]:** IpType IP destination of a rule **protocol [1..1]:** ProtoType Protocol of a rule **prtsrc [0..1]:** PortType Port source of a rule. Only if the protocol is TCP or UDP **prtdst [0..1]:** PortType Port destination of a rule. Only if the protocol is TCP or UDP **icmptype [0..1]:** IcmType Type of the ICMP protocol. Only if the protocol is ICMP
Constraints	context Matches inv: (self.protocol = tcp or self.protocol = udp) implies self.prtdst <> null xor (self.protocol = icmp) implies self.icmptype < > null

MetaClass	DstNATRule
Description	Destination NAT rule
Attributes	**comment [0..1]:** String Documentation for a rule
Relations	**dstnat [1..1]:** (DstNATRule → DstNat) Through this relation it is possible to obtain the general parameters of a destination NAT rule

MetaClass	SrcNATRule
Description	Source NAT rule
Attributes	**comment [0..1]:** String Documentation for a rule
Relations	**srcnat [1..1]:** (SrcNATRule → SrcNat) Through this relation it is possible to obtain the general parameters of a source NAT rule

MetaClass	DstNAT
Description	Models general parameters of a destination NAT rule
Attributes	**interface [0..1]:** String Firewall interface where the rule is applied
Relations	**translatedDst [1..1]:** (DstNAT → TNatPacketDst) Through this relation it is possible to obtain the NAT rule translation parameters **original [1..1]:** (DstNAT → NatOrigPacket) Through this relation it is possible to obtain the filtering rule parameters associated to this NAT rule (i.e. the original rule where the translation is applied)
Constraints	context DstNAT inv: (self.dstnat.size() = 1)

MetaClass	SrcNAT
Description	Models general parameters of a source NAT rule
Attributes	**interface [0..1]:** String Firewall interface where the rule is applied
Relations	**translatedSrc [1..1]:** (SrcNAT → TNatPacketSrc) Through this relation it is possible to obtain the NAT rule translation parameters **original [1..1]:** (SrcNAT → NatOrigPacket) Through this relation it is possible to obtain the filtering rule parameters associated to this NAT rule (i.e. the original rule where the translation is applied)
Constraints	context SrcNAT inv: (self.srcnat.size() = 1)

MetaClass	TNatPacketDst
Description	Models translation parameters of destination NAT rules
Attributes	**ipdst [1..1]:** IpType Translated destination IP address **prtdst [0..1]:** PortType Translated destination port
Constraints	context TNatPacketDst inv: (self.tnatpacketdst.size() = 1)

MetaClass	TNatPacketSrc
Description	Models translation parameters of source NAT rules
Attributes	**ipsrc [1..1]:** IpType Translated source IP address
Constraints	context TNatPacketSrc inv: (self.tnatpacketsrc.size() = 1)

MetaClass	NatOrigPacket
Description	Models filtering parameters of a NAT rule
Attributes	**ipsrc [1..1]:** IpType IP source of a rule **ipdst [1..1]:** IpType IP destination of a rule **protocol [1..1]:** ProtoType Protocol of a rule **prtsrc [0..1]:** PortType Port source of a rule. Only if the protocol is TCP or UDP **prtdst [0..1]:** PortType Port destination of a rule. Only if the protocol is TCP or UDP **icmptype [0..1]:** IcmType Type of the ICMP protocol. Only if the protocol is ICMP

MetaClass	NatOrigPacket
Constraints	context NatOrigPacket inv: (self.protocol = tcp or self.protocol = udp) implies self.prtdst <> null xor (self.protocol = icmp) implies self.icmptype < > null

DataType	IpType
Description	This data type models an IP address, The only constraint is that it is defined by means of a regular expression that limits the possible values that an IP address can have. An example of valid IP is: 192.168.1.1/16
Constraints	Regular expression: ((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(/[0-3]?[0-9])?)|([a-zA-Z0-9])+

DataType	ProtoType
Description	It models a protocol. Examples of valid protocols are: tcp, udp, 80
Constraints	Regular expression: tcp|udp|icmp|([a-zA-Z])+|([0-2]?[0-9]?[0-9])

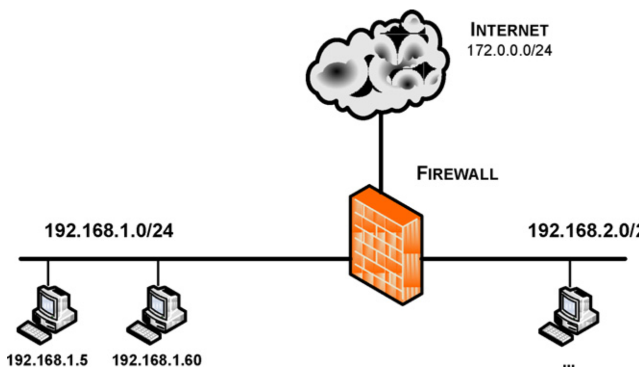
DataType	PortType
Description	It models a port. Examples of ports are: 80, 80:65535, Port1
Constraints	Regular expression: (([0-6]?[0-9]?[0-9]?[0-9]?[0-9])(:[0-6]?[0-9]?[0-9]?[0-9]?[0-9])?)|([a-zA-Z])+

Enumeration	DirectionType
Description	It describes the possible direction of a packet when it is crossing the Firewall interface
Constraints	Enumeration: • IN • OUT • BOTH

Enumeration	ActionType
Description	It describes the possible actions to execute when the packet has been matched with a rule
Constraints	Enumeration: • ALLOW • DENY • REJECT

Appendix III. Example scenario

This example scenario consists in a single firewall connected to three network segments. An example ACL for this firewall is presented in the following table. Note that this ACL deliberately contains faults (inconsistencies and redundancies).



Priority/ ID	Protocol	Source IP	Src Port	Destination IP	Dst Port	Action
R1	tcp	192.168.1.5/32	any	*.*.*.*/0	80	deny
R2	tcp	192.168.1.*/24	any	*.*.*.*/0	80	allow
R3	tcp	*.*.*.*/0	any	172.0.1.10/32	80	allow
R4	tcp	192.168.1.*/24	any	172.0.1.10/32	80	deny
R5	tcp	192.168.1.60/32	any	*.*.*.*/0	21	deny
R6	tcp	192.168.1.*/24	any	*.*.*.*/0	21	allow
R7	tcp	192.168.1.*/24	any	172.0.1.10/32	21	allow
R8	tcp	*.*.*.*/0	any	*.*.*.*/0	any	deny
R9	udp	192.168.1.*/24	any	172.0.1.10/32	53	allow
R10	udp	*.*.*.*/0	any	172.0.1.10/32	53	allow
R11	udp	192.168.2.*/24	any	172.0.2.*/24	any	allow
R12	udp	*.*.*.*/0	any	*.*.*.*/0	any	deny

The following table contains the results of a complete inconsistency and redundancy diagnosis algorithm over the example ACL. A complete taxonomy of faults is provided in Hamed and Al-Shaer (2006).

Diagnosed rule	Fault type	Other rules implied in the same fault
R1	Inconsistency	R2, R3
R4	Inconsistency	R2, R3
R5	Inconsistency	R6, R7
R8	Inconsistency	R2, R3, R6, R7
R12	Inconsistency	R9, R10, R11
R6	Redundancy	R7
R9	Redundancy	R10

Appendix IV. IPTables PSM description

In this appendix the elements of the proposed IPTables PSM are described. The templates used to describe the PSM elements are the same as the previous appendix ones, so to have a complete description of the meaning of the different areas and kinds of templates, take a look to the introduction paragraph given at it. As some metaclasses, data types and enumerations are common to PIM and PSM, in this appendix we only have included those elements that have been modified for any relations or those ones that are completely new.

MetaClass	Policy
Description	The ACL of a Firewall
Relations	**chains [1..1]:** (Policy → Chains) Models the chains of an IPTables Firewall

MetaClass	Rule
Description	Condition/action rule
Relations	**logging [0..1]:** (Rule → Logging) Logging associated to a filtering rule

MetaClass	DstNATrule
Description	Destination NAT rule
Relations	**logging [0..1]:** (DstNATrule → Logging) Logging associated to a Destination NAT rule

MetaClass	SrcNATrule
Description	Source NAT rule
Relations	**logging [0..1]:** (SrcNATrule → Logging) Logging associated to a Source NAT rule

MetaClass	IP_Firewall	
Description	Firewall configuration concepts	
Relations	**ip_firewall [1..1]:** (Policy → IP_Firewall) This relation joins the policy with the configuration information	
Attributes	**interface [1..1]:** String The name of the firewall administrative interface	**Default value:** “eth0”

MetaClass	IP_Firewall	
	ip [1..1]: IpType The IP address of the firewall administrative interface, if it has an static IP	**Default value:** (empty)
	op [1..1]: OptionType It represents if the IP of the firewall is static or dynamic (obtained through DHCP or a dial-up link). In the second case the attribute *ip* is ignored	**Default value:** DHCP
	filterMalformed [0..1]: Boolean If filtering (denying) of malformed packets is activated or not	**Default value:** true
Constraints	context IP_Firewall inv: self.op = dhcp implies self.ip = null	

MetaClass	Chains
Description	Composite of input, output, forward
Relations	**input [1..1]:** (Chains → Input) It represents the input chain
	output [1..1]: (Chains → Output) It represents the output chain
	forward [1..1]: (Chains → Forward) It represents the forward chain

MetaClass	Input	
Description	Default actions for packets directed to the firewall (both directions)	
Relations	**logging [0..1]:** (Input → Logging) If there is a relationship between Input and Logging, it means that the rules associated to the input chain must be logged.	
Attributes	**inflow [1..1]:** ActionType Action for input packets	**Default value:** DENY
	outflow [1..1]: ActionType Action for output packets	**Default value:** ALLOW

MetaClass	Output	
Description	Default actions for packets leaving the firewall (both directions)	
Relations	**logging [0..1]:** (Output → Logging) If there is a relationship between Output and Logging, it means that the rules associated to the output chain must be logged.	
Attributes	**inflow [1..1]:** ActionType Action for input packets	**Default value:** DENY
	outflow [1..1]: ActionType Action for output packets	**Default value:** ALLOW

MetaClass	Forward	
Description	Default actions for packets that are forwarded by the firewall (both directions)	
Relations	**logging [0..1]:** (Forward → Logging) If there is a relationship between Forward and Logging, it means that the rules associated to the forward chain must be logged.	
Attributes	**inflow [1..1]:** ActionType Action for input packets	**Default value:** DENY
	outflow [1..1]: ActionType Action for output packets	**Default value:** ALLOW

MetaClass	Logging	
Description	Logs sent to kernel or user space	
Attributes	**action [1..1]:** LoggingType Type of logging (user or kernel space)	**Default value:** LOG
	prefix [0..1]: String Comment string used in the output log	**Default value:** “User_prefix”

Enumeration	OptionType
Description	It represents how an IP interface is assigned
Constraints	Enumeration: • STATIC • DHCP

Enumeration	LoggingType
Description	Represents the logging types supported by IPtables
Constraints	Enumeration: • ULOG • LOG

Appendix V. PIM to IPTables PSM M2M transformation (ATL)

```
uses XMLHelpers;
module pimnat2psmnatModule; -- Module Template
create OUT : psm from IN : pim, parameters : XML;

rule pim2psmRule{
from
        r: pim!Rule
using{
        -- Rule logging data
        action: psm!LoggingType =
        thisModule.getParameter('log_action_rule');
        prefix: String =
        thisModule.getParameter('prefix_log_rule');
}
to
        s: psm!Rule(
                Filter <-  r.Filter
        )
        do{

        if(r.comment.asSet()->notEmpty())
                s.comment <- r.comment;
        else
                false;

        if(r.dstnatrule.asSet()->notEmpty())
                s.dstnatrule <- r.dstnatrule;
        else
                false;

        if(r.srcnatrule.asSet()->notEmpty())
                s.srcnatrule <- r.srcnatrule;
        else
                false;
        }
}

rule pim2psmFilter{
from
        r: pim!Filter
to
        s: psm!Filter(
                action <-  r.action,
                matches <- r.matches
        )
        do{
        if (r.interface.asSet()->notEmpty())
                s.interface <- r.interface;
        else
                false;

        if(r.direction.asSet()->notEmpty())
                s.direction <- r.direction;
        else
                false;
        }
}

rule pim2psmMatches{
from
    m1: pim!matches
to
    m2: psm!matches(
        ipdst <- m1.ipdst,
        ipsrc <- m1.ipsrc,
        protocol <- m1.protocol

        )
        do{
        if (m1.prtsrc.asSet()->notEmpty())
                m2.prtsrc <- m1.prtsrc;
        else
                false;

        if (m1.prtdst.asSet()->notEmpty())
                m2.prtdst <- m1.prtdst;
        else
                false;

        if (m1.icmptype.asSet()->notEmpty())
                m2.icmptype <- m1.icmptype;
        else
                false;
        }
}
```

```
rule pimnat2psm{
from
        p: pim!Policy
using{
        -- IP_Firewall data
        inter: String = thisModule.getParameter('interface');
        -- If it is dhcp attribute is no used
        ipfw: psm!IpType = thisModule.getParameter('ip_firewall');
        op: psm!OptionType = thisModule.getParameter('option');
        fmf: Boolean = thisModule.getParameter('fmf');

        -- Chain logging data - Input
        action_i: psm!LoggingType = thisModule.getParameter('ioption');
        prefix_i: String = thisModule.getParameter('ioption');

        -- Chain logging data - Ouput
        action_o: psm!LoggingType = thisModule.getParameter('ooption');
        prefix_o: String = thisModule.getParameter('ooption');

        -- Chain logging data - Fordward
        action_f: psm!LoggingType = thisModule.getParameter('ooption');
        prefix_f: String = thisModule.getParameter('ooption');

        -- Chain Action data - Input
        iinflow: String = thisModule.getParameter('iinflow');
        ioutflow: String = thisModule.getParameter('ioutflow');
        -- Chain Action data - Ouput
        oinflow: psm!LoggingType = thisModule.getParameter('oinflow');
        ooutflow: String = thisModule.getParameter('ooutflow');
        -- Chain Action data - Fordward
        finflow: psm!LoggingType = thisModule.getParameter('finflow');
        foutflow: String = thisModule.getParameter('foutflow');
        }
to

        ip_fw: psm!IP_Firewall(
                interface <- inter,
                op <- op
        ),
        lci: psm!Logging(
                action <-  action_i,
                prefix <- prefix_i
        ),
        lco: psm!Logging(
                action <-  action_o,
                prefix <- prefix_o
        ),
        lcf: psm!Logging(
                action <-  action_f,
                prefix <- prefix_f
        ),
        i: psm!Input(
                inflow <- iinflow,
                outflow <- ioutflow,
                logging <- lci
        ),
        o: psm!Output(
                inflow <- oinflow,
                outflow <- ooutflow,
                logging <- lco
        ),
        f: psm!Forward(
                inflow <- finflow,
                outflow <- foutflow,
                logging <- lcf
        ),
        chains: psm!Chains(
                input <- i,
                output <- o,
                forward <- f
        )
        ,
        pol: psm!Policy(
                ip_firewall <- ip_fw,
                chains <- chains,
                rules <- psm!Rule->allInstances(),
                dstnatrule <- psm!DstNATrule->allInstances(),
                srcnatrule <- psm!SrcNATrule->allInstances()
                )
 do{
        if (op = #DHCP)
                true;
        else
                ip_fw.ip <- ipfw;

        if (fmf = true or fmf = false)
                ip_fw.filtermalformed <- fmf;
        else
                false;
 }
}

helper def : getParameter(name : String) : String =
        XML!Element.allInstancesFrom('parameters')->select(e |
                e.name = 'param'
        )->select(e |
                e.getAttrVal('name') = name
        )->first().getAttrVal('value');
```

```
rule pim2psmDstNatrule{
from
        m1: pim!dstnatrule

to
        m2: psm!dstnatrule(
                dstnat <- m1.dsnat
        )
        do{
        if(m1.comment.asSet()->notEmpty())
                m2.comment <- m1.comment;
        else
                false;
        }
}

rule pim2psmDstNat{
from
        m1: pim!dstnat

to
        m2: psm!dstnat(
        tnatpacketdst <- m1.tnatpacketdst,
        natorigpacket <- m1.natorigpacket
        )
        do{
        if (m1.interface.asSet()->notEmpty())
                m2.interface <- m1.interface;
        else
                false;
        }
}

rule pim2psmTnatpacketdst{
from
        m1: pim!tnatpacketdst

to
        m2: psm!tnatpacketdst(
                ipdst <- m1.ipdst
        )
        do{
        if (m1.prtdst.asSet()->notEmpty())
                m2.prtdst <- m1.prtdst;
        else
                false;
        }
}

rule pim2psmNatorigpacket{
from
        m1: pim!natorigpacket

to
        m2: psm!natorigpacket(
        )
        do{
        if (m1.ipsrc.asSet()->notEmpty())
                m2.ipsrc <- m1.ipsrc;
        else
                false;

        if (m1.ipdst.asSet()->notEmpty())
                m2.ipdst <- m1.ipdst;
        else
                false;

        if (m1.protocol.asSet()->notEmpty())
                m2.protocol <- m1.protocol;
        else
                false;

        if (m1.prtsrc.asSet()->notEmpty())
                m2.prtsrc <- m1.prtsrc;
        else
                false;

        if (m1.prtdst.asSet()->notEmpty())
                m2.prtdst <- m1.prtdst;
        else
                false;

        if (m1.icmptype.asSet()->notEmpty())
                m2.icmptype <- m1.icmptype;
        else
                false;
        }
}
```

```
rule pim2psmSrcNatrule{
from
        m1: pim!srcnatrule

to
        m2: psm!srcnatrule(
                srcnat <- m1.srcnat
        )
        do{
        if(m1.comment.asSet()->notEmpty())
                m2.comment <- m1.comment;
        else
                false;
        }
}

rule pim2psmSrcNat{
from
        m1: pim!tnatpacketdst

to
        m2: psm!tnatpacketdst(
                tnatpacketsrc <- m1.tnatpacketsrc,
                natorigpacket <- m1.natorigpacket
        )
        do{
        if (m1.interface.asSet()->notEmpty())
                m2.interface <- m1.interface;
        else
                false;
        }
}

rule pim2psmTnatpacketsrc{
from
        m1: pim!tnatpacketsrc

to
        m2: psm!tnatpacketsrc(
                ipsrc <- m1.ipsrc
        )
}
```

Appendix VI. External file example containing the parameters of the ATL M2M transformation

```
<parameters>
       <param name="interface" value="eth0"/>
       <param name="ip_firewall" value=""/>
       <param name="option" value="#DHCP"/>
       <param name="fmf" value="true"/>
       <param name="log_action_rule" value="#LOG"/>
       <param name="prefix_log_rule" value="Prefix_logging_rules"/>
       <param name="aioption" value="#LOG"/>
       <param name="pioption" value="Prefix_input"/>
       <param name="aooption" value="#LOG"/>
       <param name="pooption" value="Prefix_output"/>
       <param name=afoption" value="#LOG"/>
       <param name="pfoption" value="Prefix_forwardput"/>
       <param name="iinflow" value="#ALLOW"/>
       <param name="ioutflow" value="#DENY"/>
       <param name="oinflow" value="#DENY"/>
       <param name="ooutflow" value="#ALLOW"/>
       <param name="finflow" value="#ALLOW"/>
       <param name="foutflow" value="#DENY"/>
</parameters>
```

Appendix VII. IPTables to code M2T transformation (MOFScript)

Please note that the meta-class IP_Firewall has not been included in the transformation rules. The configuration of the platform depends entirely on the version of the Linux Kernel and the distribution it is running on, and is out of the scope of this paper.

```
/**
 * MOFScript model to Text transformation
 * from PSM model to iptable-save code.
 */

texttransformation MultipleMetaModels (in psm:"psm") {
main () {

         file f("sample_code.firewall")
         f.println("# Generated by MOFScript in transformation from model to text")
         f.println(" ")
         f.println("*filter")
         f.println("\n#********************************")
         f.println("# CHAINS")
         f.println("#********************************\n")
         f.println(":INPUT DROP [0:0]")

         var defaultaction : String = psm.objectsOfType(psm.regla).last().action
         var direction : String = psm.objectsOfType(psm.regla).last().filter.direction
        var ipsrc : String = psm.objectsOfType(psm.regla).last().matches.ipsrc
        var ipdst : String = psm.objectsOfType(psm.regla).last().matches.ipdst
        var prtsrc : String = psm.objectsOfType(psm.regla).last().matches.prtsrc
        var prtdst : String = psm.objectsOfType(psm.regla).last().matches.prtdst
        var protocol : String = psm.objectsOfType(psm.regla).last().matches.protocol


         var forward : String  = ""

         if(direction.equalsIgnoreCase("all") && ipsrc.equalsIgnoreCase("all") &&
                  ipdst.equalsIgnoreCase("all") && prtsrc.equalsIgnoreCase("all") &&
                  prtdst.equalsIgnoreCase("all") && protocol.equalsIgnoreCase("all")){
                  defaultaction =   psm.objectsOfType(psm.regla).last().action
                           if( defaultaction.equalsIgnoreCase("allow"))
                                    forward = ":FORWARD ACCEPT [0:0]"
                           else
                                    forward =  ":FORWARD DROP [0:0]"

         } else
                  forward =  ":FORWARD DROP [0:0]"

         f.println(forward)
         f.println(":OUTPUT DROP [0:0]")

         var numrules : Integer = psm.objectsOfType(psm.rules).size()
         if(numrules > 0)
         {
                  f.println("\n#********************************")
                  f.println("# STATEFUL")
                  f.println("#********************************\n")
                  f.println("-A FORWARD -m state --state INVALID -j DROP")
                  f.println("-A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT")
         }

         f.println ( "\n#********************************")
         f.println ( "# RULES")
         f.println ( "#********************************\n")
         psm.objectsOfType (psm.rules)->forEach (r) {
                  r.generaterules()
                  stdout.println (" ")
         }

         f.println("\nCOMMIT")
         f.println("# COMPLETED")
}
  /* Method used to transform rules, hard-work*/
  psm.rule::generaterules(){
   file f("outputfile.firewall")

   var output : String = ""
   var direction : String = self.filter.direction
   var interface: String = self.filter.interface
   var action: String = self.filter.action
   var logaction : String = self.logging.action
   var logprefix :String = self.logging.prefix
   var ipsrc : String = self.matches.ipsrc
   var ipdst : String = self.matches.ipdst
   var prtsrc : String = self.matches.prtsrc
   var prtdst : String = self.matches.prtdst
   var protocol : String = self.matches.protocol
   var icmptype : String = self.matches.icmptype

   output = output + "#Rule :" + self.comment + "\n"
   output = output + "-A FORWARD "

   if(direction.equalsIgnoreCase("in") &&  interface.size() >0)
                  output = output + "-i " + interface+ " "
   else if(direction.equalsIgnoreCase("out") &&  interface.size() >0)
                  output = output + "-o " + interface+ " "
   else{
                  if((not direction.equalsIgnoreCase("all")) &&  interface.size() >0 )
                           output = output + "-i " + interface+ " -o "+ interface + " "

   }
```

```
if(not ipsrc.equalsIgnoreCase("all"))
                          output = output + " -s " + ipsrc + " "
   if(not ipdst.equalsIgnoreCase("all"))
                          output = output + " -d " + ipdst + " "

   if(protocol.equalsIgnoreCase("tcp") or protocol.equalsIgnoreCase("6"))
   {
                 output = output + " -p " + protocol + " "
                 if(not prtsrc.equalsIgnoreCase("all"))
                 {
                          output = output + " -m tcp --sport " + prtsrc + " "
                          if(not prtdst.equalsIgnoreCase("all"))
                                   output = output + " --dport " + prtdst + " "
                 }
                 else
                 {
                          if(not prtdst.equalsIgnoreCase("all"))
                                   output = output + " -m tcp --dport " + prtdst + " "
                 }
   }

   if(protocol.equalsIgnoreCase("udp") or protocol.equalsIgnoreCase("17"))
   {
                 output = output + " -p " + protocol + " "
                 if(not prtsrc.equalsIgnoreCase("all"))
                 {
                          output = output + " -m udp --sport " + prtsrc + " "
                          if(not prtdst.equalsIgnoreCase("all"))
                                   output = output + " --dport " + prtdst + " "
                 }
                 else
                 {
                          if(not prtdst.equalsIgnoreCase("all"))
                                   output = output + " -m udp --dport " + prtdst + " "
                 }
   }

   if(protocol.equalsIgnoreCase("icmp") or protocol.equalsIgnoreCase("1"))
   {
                 output = output + " -p " + protocol + " "
                 if(not icmptype.equalsIgnoreCase("all"))
                          output = output + " -m icmp --icmp-type " + icmptype + " "

   }

   if(not protocol.equalsIgnoreCase("all") && not protocol.equalsIgnoreCase("0"))
   {
         output = output + " -p " + protocol + " "
   }
   if(action.equalsIgnoreCase("deny"))
   {
                  if(logaction.size()>0)
                  {
                           var logging : String = output + "-j " + logaction.toUpper()

                           if(logprefix.size()>= 0)
                                    logging = logging + " --log-prefix " + '"' + logprefix
+ '"' +  "\n"
                           else
                                    logging = logging + "\n"

                           output = output + " -j DROP"
                           output = logging + output
                  }
                  else{
                           psm.objectsOfType (psm.forward)->forEach (f) {
                                    output = output + f.generateForwardLogging()
                           }
                           output = output + " -j DROP "
                  }

   }

   if(action.equalsIgnoreCase("reject"))
   {
                  if(logaction.size()>0)
                  {
                           var logging : String = output + "-j " + logaction.toUpper()
                           if(logprefix.size()>= 0)
                                    logging = logging + " --log-prefix " + '"' + logprefix
+ '"' +  "\n"
                           else
                                    logging = logging + "\n"
                           output = output + " -j REJECT"
                           output = logging + output
                  }
                  else{
                    psm.objectsOfType (psm.forward)->forEach (f) {
                                    output = output + f.generateForwardLogging()
                           }
                           output = output + " -j REJECT "
                  }

   }

   if(action.equalsIgnoreCase("allow"))
   {
                  if(logaction.size()>0)
                  {
                           var logging : String = output + "-j " + logaction.toUpper()
                           if(logprefix.size()>= 0)
                                    logging = logging + " --log-prefix " + '"' + logprefix
+ '"' +  "\n"
                           else
                                    logging = logging + "\n"
                           output = output + " -j ACCEPT"
                           output = logging + output
                  }
                  else{
                    psm.objectsOfType (psm.forward)->forEach (f) {
                                    output = output + f.generateForwardLogging()
                           }
                           output = output + " -j ACCEPT"
                  }
   }


   f.println(output)
}
```

```
psm.forward::generateForwardLogging(): String {
 var output : String = "";
 var loggingAction : String = self.logging.action
 var loggingPrefix : String = self.logging.prefix

         if(loggingAction.size()>0)
         {
                  var logging : String = output + "-j " + loggingAction.toUpper()
                  if(loggingPrefix.size()>= 0)
                           logging = logging + " --log-prefix " + '"' + loggingPrefix +
'"'

                   output = logging + output
         }

         result = output
}
}
```

References

Al-Shaer, E., Hamed, H., 2004. Modeling and management of firewall policies. IEEE Transactions on Network and Service Management 1 (April (1)).

Al-Shaer, E., Hamed, H., Boutaba, R., Hasan, M., 2005. Conflict classification and analysis of distributed firewall policies. IEEE Journal on Selected Areas in Communications 23 (October (10)), 2069–2084.

Ardagna, C.A., Damiani, E., De Capitani di Vimercati, S., Samarati, P., 2004. XML-based Access Control Languages, Elsevier Information Security Technical Report (Online), pp. 35–46.

ATLAS Transformation Language, 2007. http://www.eclipse.org/m2m/atl/.

Baboescu, F., Varguese, G., 2003. Fast and scalable conflict detection for packet classifiers. Computers Networks 42 (6), 717–735.

Bartal, Y., Mayer, A., Nissim, K., Wool, A., 2004. Firmato: a novel firewall management toolkit. ACM Transactions on Computer Systems 22 (4), 381–420.

Basin, D., Dorser, J., Lodderstedt, T., 2006. Model driven security: from UML models to access control infrastructures. ACM Transactions on Software Engineering and Methodology 15 (1), 39–91.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J., 2003. Eclipse Modelling Framework: A Developer's Guide. Addison-Wesley.

Cisco Adaptive Security Device Manager. Available at: http://www.cisco.com/en/US/products/ps6121/index.html.

Chapple, M.J., D'Arcy, J., Striegel, A., 2009. An analysis of firewall rulebase (mis)management practices. Information Systems Security Association Journal (February).

Checkpoint Software Blades. Available at: http://www.checkpoint.com/products/softwareblades/architecture/index.html.

Checkpoint Software Technologies LTD. Available at: http://www.checkpoint.com/.

Cisco PIX Firewall Software. Available at: http://www.cisco.com/en/US/products/sw/secursw/ps2120/index.html.

Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. IBM Systems Journal 45 (3).

Damianou, N., Dulay, N., Lupu, E., Sloman, M.,2001. The ponder specification language. In: Workshop on Policies for Distributed Systems and Networks (POLICY). HP Labs, Bristol, UK, pp. 29–31.

De Capitani di Vimercati, Foresti, S., Jajodia, S., Samarati, P., 2007. Access control policies and languages. International Journal of Computational Science and Engineering 3 (2).

Desfray, P., 2000. UML profiles versus metamodeling extensions. An ongoing debate. In: COM00, Proceedings of the First Workshop on UML in the COM Enterprise: Modeling CORBA, Components, XML/XMI and Metadata.

Didonet Del Fabro, M., Bézivin, J., Valduriez, P., 2006. Weaving models with the Eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe 2006, October 2006, http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium2_WeavingModels.pdf.

Douglas, P., Alliger, G., Goldberg, R., 1996. Client-server and object-oriented training. IEEE Computer 9 (6), 80–84.

Eclipse Modeling Project, 2007. http://www.eclipse.org/modeling/.

Eclipse Modeling Framework. http://eclipse.org/emf/.

El-Atawy, A., 2006. Survey on the Use of Formal Languages/Models for the Specification, Verification, and Enforcement of Security Policies, DePaul University Technical Reports CTI 06-005.

Engel, K.D., Paige, R.F., Kolovos, D.S., 2006. Using a model merging language for reconciling model versions. In: Rensink, A., Warmer, J. (Eds.), ECMDA-FA, Vol. 4066 of Lecture Notes in Computer Science. Springer, pp. 143–157.

Eppstein, D., Muthukrishnan, S., 2001. Internet packet filter management and rectangle geometry. In: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), January 2001.

Firewall Builder. http://www.fwbuilder.org/.

FreeBSD FreeBSD IPFilter. Available at: http://coombs.anu.edu.au/~avalon/.

FreeBSD IPFirewall, Reference Manual. Available at: http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls-ipfw.html.

Galán, F., Fernández, D., Jorge, E., López de Vergara, R., 2010. Using a model-driven architecture for technology-independent scenario configuration in networking testbeds. IEEE Communications Magazine 48 (12), 132–141.

García-Alfaro, J., Boulahia-Cuppens, N., Cuppens, F., 2008. Complete analysis of configuration rules to guarantee reliable network security policies. International Journal of Information Security 7 (2).

Greenfield, J., Short, K., Cook, S., Kent, S., 2004. Software Factories. Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley Publishing, Inc.

Hamed, H., Al-Shaer, E., 2006. Taxonomy of conflicts in network security policies. IEEE Communications Magazine 44 (3).

Hari, B., Suri, S., Parulkar, G., 2000. Detecting and resolving packet filter conflicts. In: Proceedings of IEEE INFOCOM, March 2000.

Jürjens, J., 2002. UMLsec: extending UML for secure systems development. In: 5th International UML, Springer-Verlag LNCS 2460, Dresden, Germany, pp. 1–9.

Kurtev, I., Bézivin, J., Aksit, M., 2002. Technological spaces: an initial appraisal. In: International Federated Conf. (DOA, ODBASE, CoopIS), Industrial Track.

Liu, A.L., Gouda, M.G., 2008. Complete redundancy removal for packet classifiers in TCAMs. IEEE Transactions on Parallel and Distributed Systems 24.

LogLogic Change Manager. Available at: http://www.loglogic.com/products/security-change-management.

Moore, B., Ellesson, E., Strassner, J., Westerinen, A., 2001. Policy Core Information Model (PCIM), IETF RFC 3060.

Netfilter IPTables. http://www.netfilter.org.

OASIS eXtensible Access Control Markup Language (XACML), http://www.oasis-open.org/committees/xacml/.

OMG MDA Guide Version 1.0. Technical Report omg/2003-05-01. OMG, May 2003.

PacketFilter User Guide Reference. Available at: http://www.openbsd.org/faq/pf/.

Pozo, S., Ceballos, R., Gasca, R.M.,2008. AFPL: an abstract language model for firewall ACLs. In: 8th International Conference on Computational Science and Its Applications (ICCSA), Lecture Notes in Computer Science (LNCS), vol. 5073, Part 2. Springer-Verlag, Perugia, Italy.

Pozo, S., Ceballos, R., Gasca, R.M., 2009a. Model based development of firewall rule sets: diagnosing model faults. Information and Software Technology Journal 51 (5), 894–915.

Pozo, S., Varela-Vaca, A.J., Gasca, R.M., Ceballos, R.,2009b. Efficient algorithms and abstract data types for local inconsistency isolation in firewall ACLs. In: 4th International Conference on Security and Cryptography (SECRYPT). IEEE Computer Society Press, Milan, Italy.

Pozo, S., Varela-Vaca, A.J., Gasca, R.M.,2009c. AFPL2: an abstract language for firewall ACLs with NAT support. In: 2nd International Conference on Dependability and Security in Complex and Critical Information Systems (DEPEND). IEEE Computer Society Press, Athens, Greece.

Pozo, S., Ceballos, R., Gasca, R.M., 2009d. A heuristic process for local inconsistency diagnosis in firewall rule sets. Journal of Networks 4 (8), 698–710.

Pozo, S., Varela-Vaca, A.J., Gasca, R.M., Quadratic, A., 2010. Complete, and minimal consistency diagnosis process for firewall ACLs. In: Advanced Information Networking and Applications (AINA). IEEE Computer Society Press, Perth, Australia.

Rule Markup Language (RuleML). http://www.ruleml.org/.

2001. Simple Rule Markup Language (SRML): A General XML Rule Representation for Forward-chaining Rules. IBM.

Sztipanovits, J., Karsai, G., 1997. Model-integrated computing. Computer 30 (October (4)), 110.

Taylor, D.E., 2005. Survey and taxonomy of packet classification techniques. ACM Computing Surveys 37 (3), 238–275.

The MOFScript Home Page, 2007. Available at: http://www.eclipse.org/gmt/mofscript/.

Wool, A., 2004. A quantitative study of firewall configuration errors. IEEE Computer 37 (6), 62–67.

Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C., Mohapatra, P.,2006. FIREMAN: a toolkit for FIREwall Modelling and ANalysis. In: IEEE Symposium on Security and Privacy (S&P). IEEE Computer Society Press, Oakland, CA, USA.

Zhang, B., Al-Shaer, E., Jagadeesan, R., Riely, J., Pitcher, C., 2007. Specifications of a high-level conflict-free firewall policy language for multi-domain networks. In: ACM Symposium on Access Control Models and Technologies (SACMAT), Sophia Antipolis, France, pp. 185–194.

Sergio Pozo holds a PhD in Computer Engineering from the University of Seville, in Spain, where he is a full-time Senior Lecturer with the Computer Languages and Systems Department. He is part of the QUIVIR Research Group. His main research interests are network and security devices and software modelling, model-based diagnosis, auto-recovery, and applications of the model-driven development paradigm to information security. More precisely, he is focused in firewall ACL languages and models, inconsistency/redundancy/conformance diagnosis in firewall ACLs. He is also reviewer and organizer of computer security conferences and journals.

Rafael M. Gasca holds a PhD in Computer Science from the University of Seville, in Spain, where he is a full-time Reader at the Computer Languages and Systems Department since 1991. He is the head of the QUIVIR Research Group, where has been the advisor of several fundamental research projects as well as applied RD projects in cooperation with the industry. His main research interests are domain-specific languages and models for computer security devices, techniques for diagnosing security models faults (mainly inconsistency and redundancy), auto-recovery techniques for diagnosed faults (autonomic computing), and applications of the Model-Driven Paradigm to computer security. He is also a frequent reviewer for security conferences and journals, and organizer of artificial intelligence and model-based diagnosis conferences.

Antonia M. Reina Quintero holds an MSc in Computer Engineering from the University of Seville. She works as a full-time Lecturer at the Computer Languages and Systems Department from the University of Seville since 2000, although she also has worked as a computer engineer for a leading company in traffic control systems. Her current research is focused on aspect-oriented programming, advanced separation of concerns and Model-Driven Architecture applied to web-based systems.

A.J. Varela-Vaca holds an MSc in Computer Engineering from the University of Seville, in Spain. Currently, he holds a full-time research grant and works under the supervision of R.M. Gasca and S. Pozo at the Computer Languages and Systems Department. His main research interests are computer and network dependability issues, and models for security. More precisely, he is focused in the application of Model-Based Engineering paradigm to IT Security, and dependability issues in business processes.