Invariant Preservation In Iterative Modeling (Extended Version)

Levi LÚCIO, Eugene SYRIANI, Moussa AMRANI, Qin ZHANG and Hans VANGHELUWE

Laboratory for Advanced Software Systems (LASSY)
University of Luxembourg
6, rue R. Coudenhove-Kalergi
L-1359 Luxembourg

TR-LASSY-12-13

January – October 2012

Invariant Preservation In Iterative Modeling (Extended Version)

Levi LÚCIO[‡], Eugene SYRIANI[*], Moussa AMRANI[†], Qin ZHANG[†] and Hans VANGHELUWE[‡§]
[†] University of Luxembourg (Luxembourg), {Moussa.Amrani, Qin.Zhang}@uni.lu
[‡] McGill University (Canada), {Levi, hv}@cs.mcgill.ca
[*] University of Alabama (Tuscaloosa, AL, USA), esyriani@cs.ua.edu
[§] University of Antwerp (Belgium), Hans.Vangheluwe@ua.ac.be

***Abstract*—In a Model-Driven Development project, models are typically built iteratively to better satisfy a set of requirements. Therefore it is crucial to guarantee that one iteration of a model evolution does not hinder the previous version. In this paper, we focus on invariant preservation of behavioral models expressed in Algebraic Petri Nets. The theory developed is applied to a Multi-Level Security File System modeled iteratively. We also discuss how this approach can be applied on Domain-Specific Languages that are translated to Algebraic Petri Nets.**

I. INTRODUCTION

Iterative development [12] is often adopted in modern software projects [4] as it allows for rapid validation of the developed features at a finer granularity. In Model-Driven Engineering (MDE), *iterative modeling* [10] is often witnessed in model evolution. During the development of a MDE project, given a fixed set of requirements, the modeler typically produces a first model that satisfies an initial subset of the requirements. Then at the following iteration, he either produces a model that satisfies a larger subset of the requirements or revises the model to more correctly satisfy the previous subset of requirement. The model will therefore undergo several iterations until all requirements are satisfied correctly.

The problem with this iterative modeling process is that there is no guarantee that the model M_{i+1} resulting after an iteration $i+1$ does not break properties of the previous model M_i that correctly satisfied requirements. Therefore from a regression point of view, it is crucial that satisfied and unchanged properties of an evolving model are preserved at each iteration.

In this paper we report on ongoing work for building a framework for iterative development for Domain-Specific Languages (DSLs). Given a metamodel for describing behavioral models in the domain of interest and a set of requirements, a modeler iteratively specifies models (metamodel instances) until fulfilling those requirements. We aim at assisting the modeler in the iterative process, by ensuring that each iteration actually preserves previously satisfied properties. Our proposal to tackle this problem is based on invariant preservation on Algebraic Petri Nets (APNs). In previous work we have presented a partial translation from StateCharts into APNs [15], meaning StateCharts can be used as a front end for our approach. However, any language for which such a translation exists could be plugged into our framework. We present in this paper some illustrated preliminary results demonstrating our approach's feasibility.

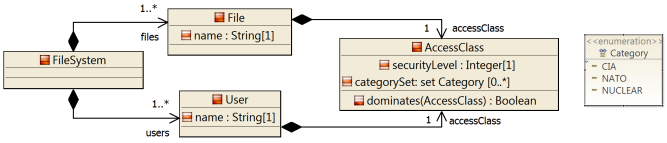

Fig. 1. Simplified MLS metamodel [3], as a Kermeta Class Diagram [17].

In Section II we present a concrete example together with desirable requirements that will be addressed along later iterations. This example starts from a metamodel that is transformed into an APN for performing the necessary verifications. Section III presents a formalisation of APNs and proves that the iterations actually preserve invariant properties. Then in Section IV, we present an iterative modeling process applied to our example and show how the initial requirements are gradually and iteratively enforced. Finally, Section V discusses our approach and related work, and Section VI concludes by presenting our future work.

II. RUNNING EXAMPLE (NAIVE VERSION)

Our example is inspired from [16]: it is an UNIX-like operating system that includes a Multi-Level Security (MLS) File System (FS). We first present the expected requirements on the file system, then explain its implementation in terms of APN, and finish with the requirements formalisation in terms of three properties expressed over the APN. Section IV will then present two manual evolution that iteratively enforce more properties on the system.

A. Requirements

In MLS [3], two concepts are essentially relevant: *objects* designate system resources or data repositories that must be protected (*e.g.,* files, directories or terminals); *subjects* denote entities capable of requesting services from system resources (*e.g.,* users, or processors). Both are associated to an *access class*: they classify objects and subjects according to their confidentiality, and responsability degree, respectively. Intuitively, an object associated with a high access class can only be seen or manipulated by a highly trusted subject.

For simplification and space limitation purposes, we present a simplified version of a file system consisting only of *files* as possible objects, and *users* as possible subjects. Files can

	Name	SL	CR
Files	f1	2	NATO
	f2	0	CIA
	f3	1	∅
Users	levi	3	NATO, CIA
	eugene	0	NATO, CIA

```
M0(files)=[file(f1, ac(s(s(ZERO)),
            list(NATO, empty)));
          file(f2, ac(ZERO,
            list(CIA, empty)));
          file(f3, ac(s(ZERO),
            empty))]
M0(users)=[user(levi, ac(s(s(s(ZERO))),
            list(NATO, list(CIA, empty)))));
          user(eugene, ac(ZERO,
            list(NATO, list(CIA, empty))))]
```

Fig. 2. A simple FS model and the corresponding initial marking.

either be read or written by an user, or idle (which implies no simultaneous readings by different users).

An important quality of such file systems is *confidentiality*, especially against Trojan horses [1]: they consist in code executed by a trusted user without knowing it or consenting for it; they can also pass confidential data by copying sensitive data into files accessible to untrusted users. One common technique for preventing this attack is *data confinement*: a trusted user cannot open simultaneously two files with the most confidential one in read mode and the less confidential in write mode, thus preventing data leaks by copy from the former to the latter.

Therefore in our running example, we aim at enforcing the two following requirements: (**R1**) a file can be open, either in read or in write mode, only by an user with sufficient access rights; (**R2**) an user always respects the data confinement condition.

B. Model-Checking Algebraic Petri Nets with AlPiNA

AlPiNA (Algebraic Petri Net Analyzer) is a model-checker developed by the SMV team in Geneva [7]. AlPiNA takes two inputs: an APN, *i.e.,* a Petri Net (PN) whose tokens are terms of an equational algebraic specification (*i.e.,* sorts, associated operations and equations), and whose behaviour is graphically defined; and a set of first-order formulæ over the specification terms expressing invariants over the PN. It allows the verification of such formulæ by exhaustively exploring the PN state space. If they are not satisfied, AlPiNA returns a possible marking that violates the formulæ.

In our context, AlPiNA is used at each iteration step to check that new properties are satisfied. In case of a revision, the model-checker is run on each previously satisfied properties to ensure they are still satisfied by the revised model.

C. The Naive File System FS

A running example is used to illustrate the approach. It consists in a small DSL representing a simplified version of an MLS File System. We first describe a possible metamodel describing the necessary data structures for representing File Systems, then the corresponding algebraic specification, and finally the behavioural semantics in terms of PNs.

1) An MLS *File System Metamodel:* Figure 1 presents a simplified metamodel for MLS File Systems: it contains only files as objects and only users as subjects. A File and an User consists in a name and an associated AccessClass. An AccessClass is composed of an integer representing a securityLevel and a Category set. A Category is one of the following credential: CIA, NATO or NUCLEAR (represented as an enumeration). Access classes can be compared using the dominates operation: if $\mathsf{ac} = (\mathsf{s}, \mathsf{C})$ and $\mathsf{ac}' = (\mathsf{s}', \mathsf{C}')$ are two access classes, where $\mathsf{s}, \mathsf{s}' \in \mathbb{N}$ represent security levels and C, C' represent sets of credential categories, then ac *dominates* ac' (noted $\mathsf{ac} \triangleright \mathsf{ac}'$) iff ac is "*stronger*" than ac', *i.e.,* formally, if $\mathsf{s}' \leq \mathsf{s}$ and $\mathsf{C}' \subseteq \mathsf{C}$). Note that this notion of access class domination is different from the one classical in PN, which operates on markings.

Consider now an MLS FS model corresponding to the naive FS, as described in Fig. 2: it consists in three files and two users (SL stands for security level and CR for credential). Here, the user levi dominates all files whereas eugene only dominates file f2.

For now, any MLS FS model (like the one depicted in Fig. 2, left) conforming to the metamodel of Fig. 1 is translated by hand in two steps into an APN:

- the first step consists of a translation of the data structures specified by the metamodel into Abstract Data Types (ADTs);
- the second step takes care of the behavioural semantics.

We detail each step in the following sections, providing an overview of how the translation is operated, giving further information about how to automatise it.

*2) Algebraic Data Types (*ADT*s):* Classes File and User both contains an attribute name used to identify a file / user. Instead of using string, they are translated into sorts: all we really need is a way to compare such identifiers. In AlPiNA, we use Generators to specify the initial marking (as described in Fig. 2 (left).

```
0 Adt FileName
    Sorts fileName;
2   Generators
      f1 : fileName;
4     f2 : fileName;
      f3 : fileName;
```

```
0 Adt UserName
    Sorts userName;
2   Generators
      levi   : userName;
4     eugene : userName;
```

This sorts are then used for creating sorts file and user following the same schema. First, a generator is defined for building a term of each sort, by combining a name with an accessClass. Then, two operations name and class return respectively the name and the class of such a term with the help of variables prefixed in AlPiNA with $.

```
0  import "fileName.adt"
   import "accessClass.adt"
2  Adt File
     Sorts file;
4    Generators
       file: fileName,accessClass->file;
6    Operations
       name: file -> fileName;
8      class: file -> accessClass;
     Axioms
10     name(file($fn,$acl)) = $fn;
       class(file($fn,$acl)) = $acl;
12   Variables
       fn  : fileName;
14     acl : accessClass;
```

```
0  import "userName.adt"
   import "accessClass.adt"
2  Adt User
     Sorts user;
4    Generators
       user: userName,accessClass->user;
6    Operations
       name: user -> userName;
8      class: user -> accessClass;
     Axioms
10     name(user($un,$acl))  = $un;
       class(user($un,$acl)) = $acl;
12   Variables
       un  : userName;
14     acl : accessClass;
```

The enumeration Category is translated into a sort containing only three elements. An ADT CategorySet is defined to take care of the attribute categorySet in AccessClass: it makes use of the predefined generic ADT List, and contains only one operation subset for checking the access class' domination.

```
Adt Category
  Sorts category;
  Generators
    NATO    : category;
    CIA     : category;
    NUCLEAR : category;
```

```
import "boolean.adt"
import "category.adt"
import "list.gadt"
Adt CategorySet Is List[category]
  Operations
    subset : list[category], list[category] -> bool;
  Axioms
    empty subset $l = true;
    if contains($h,$l) = false then
      list($h,$t) subset $l = false;
    if contains($h,$l) = true then
      list($h,$t) subset $l = $t subset $l;
  Variables
    h : category;
    t : list[category];
    l : list[category];
```

In the class AccessClass, the attribute securityLevel is an Integer on which only an order is necessary for comparison. Therefore, it is encoded using a Peano-like sort representation, with `ZERO` for 0 and `s` for the successor operation.

```
import "boolean.adt"
Adt securityLevel
  Sorts securityLevel;
  Generators
    ZERO: securityLevel;
    s: securityLevel -> securityLevel;
  Operations
    le: securityLevel,securityLevel -> bool
        ;
  Axioms
    ZERO le ZERO = true;
    ZERO le s($x) = true;
    s($x) le ZERO = false;
    s($x) le s($y) = $x le $y;
  Variables
    x : securityLevel;
    y : securityLevel;
```

Now, only the AccessClass class needs to be translated into an ADT. A generator ac is defined to build accessClass's terms from terms of securityLevel and Category list. Then, the dominates operation is defined according to the specification of Section II-C2.

```
import "boolean.adt"
import "categorySet.adt"
import "securityLevel.adt"
import "list.gadt"
import "category.adt"
Adt accessClass is list[category]
  Sorts accessClass;
  Generators
    ac : securityLevel, CategorySet -> accessClass;
  Operations
    dominates : accessClass, accessClass -> bool;
  Axioms
    if ($s2 le $s1)=true & ($c2 subset $c1)=true then
      ac($s1,$c1) dominates ac($s2,$c2)= true;
    if ($s2 le $s1)=false & ($c2 subset $c1)=true then
      ac($s1,$c1) dominates ac($s2,$c2)=false;
    if ($s2 le $s1)=true & ($c2 subset $c1)=false then
      ac($s1,$c1) dominates ac($s2,$c2)=false;
    if ($s2 le $s1)=false & ($c2 subset $c1)=false then
      ac($s1,$c1) dominates ac($s2,$c2)=false;
  Variables
    s1 : securityLevel;
    s2 : securityLevel;
    c1 : CategorySet;
    c2 : CategorySet;
```

An extra ADT is defined for the purpose of the behavioural semantics. Named FileUserPair, it uses the generic ADT Pair to build a pair with a file and an user, and introduces an operation userHasPermissionForFile that checks if the user's access class dominates the file's access class.

```
import "boolean.adt"
import "pair.gadt"
import "file.adt"
import "user.adt"
Adt FileUserPair is pair[file,user]
  Operations
    userHasPermissionForFile : pair[file,user] -> bool;
  Axioms
    userHasPermissionForFile(pair($f,$u)) =
      getAccessClass($u) dominates getAccessClass($f);
  Variables
    f : file;
    u : user;
```

3) Behavioural Semantics: In the second step, we define the naive translational semantics of MLS systems *i.e.,* opening/-closing files without credential checkings. The resulting APN contains four places and four transitions. Places `files` and `users` contain files and users tokens, representing respectively the files and users objects corresponding to the initial marking of Fig. 2. When an user wants to read or write a file, the corresponding transition (`openR` or `openW`, respectively) is fired and consumes tokens `$f` from place files and `$u` from place `users`; then produces a pair $\$p = (\$f, \$u)$ stored in the corresponding place. When `$u` closes `$f`, the corresponding `close` transition is fired, which checks that `$f` was actually opened by the user that opened it, and produces back tokens `$f` and `$u` into their respective places. Consuming file tokens every time a file is opened prevents another user from opening the same file.

Figure 3 depicts the PN that corresponds to the naive FS. The marking of places `reading` and `writing` is empty (and denoted by empty brackets `[]`, and the marking for the two other places follows the initial marking of Fig. 2 (right).

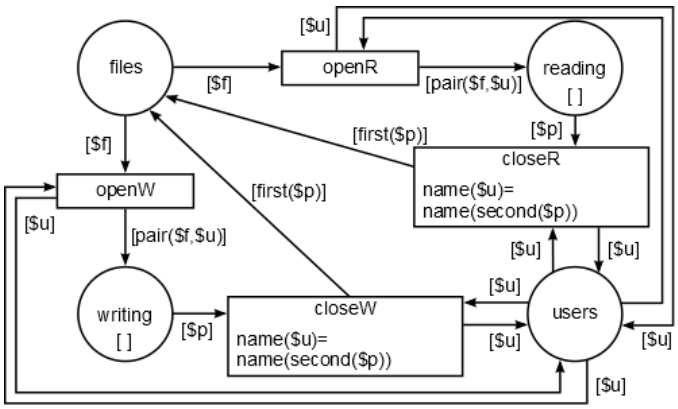

Fig. 3. The naive File System FS with its initial marking M.

D. Requirements Properties

The requirements of Section II-A can now be formally expressed on the previous PN as invariant formulæ. Requirement **R1**, concerned with file opening by a sufficiently trusted user, is split into two properties P1 and P2, corresponding to each opening mode (resp. write / read mode).

$\mathtt{P1} \triangleq \forall\ \$p = (\$f, \$u) \in \mathtt{write} \cdot \mathtt{ac_u} \rhd \mathtt{ac_f}$

$\mathtt{P2} \triangleq \forall\ \$p = (\$f, \$u) \in \mathtt{read} \cdot \mathtt{ac_u} \rhd \mathtt{ac_f}$

Here, requirement **R1** holds if for any pair $\$p = (\$f, \$u) \in \mathtt{File} \times \mathtt{User}$ in the `write` or `read` place, `$u`'s access class dominates `$f`'s access class. Written in AlPiNA, these two properties lead to the following expressions:

	P1	P2	P3
FS	✗	✗	✗
FS_s	✓	✓	✗
FS_c	✓	✓	✓

TABLE I
PROPERTY SATISFACTION FOR THE THREE *File System* VERSIONS.

```
Expressions
  P1: forall($p in write: userHasPermissionForFile($p)=true);
  P2: forall($p in read:  userHasPermissionForFile($p)=true);
Variables
  p : pair[file,user];
```

Requirement **R2** is implemented by a third property P3 that ensures the confinement property on the PN.

$$\texttt{P3} \triangleq \forall\ \$\texttt{p} = (\$\texttt{f}, \$\texttt{u}) \in \texttt{read}, \forall\ \$\texttt{p}' = (\$\texttt{f}', \$\texttt{u}') \in \texttt{write} \cdot \$\texttt{un} = \$\texttt{un}' \Longrightarrow \$\texttt{ac}_\texttt{f} \rhd \$\texttt{ac}'_\texttt{f}$$

This property says that for all pairs of files simultaneously opened by the same user, when one is on read mode and the other on write mode, the read file's access class dominates the write file's one. Written in AlPiNA, it gives the following expression:

```
Expressions
  P3: forall($p in read:
        forall ($p1 in write:
          (name(second($p1)) = name(second($p)))
          =>
          !((class(first($p)) dominates class(first($p1))=true) &
          !(name(first($p)) = name(first($p1))))
        )
      );
Variables
  p : pair[file,user];
  p1 : pair[file,user];
```

Running AlPiNA on the naive FS shows, as summarised in Tab. I, that none of these properties are satisfied: this FS only implements as simple FS mechanism without access control.

III. INVARIANTS PRESERVATION ALONG EVOLUTION

This Section establishes sufficient conditions for evolving an APN model while preserving *invariant* properties. The theoretical result and its rationale is explained in terms of APNs' iterations, illustrated on our running example.

Let us consider a set of properties P holding on a model $\mathtt{M}_i$, noted $\mathtt{M}_i \models P$. We want to check that an evolution $\mathtt{M}_{i+1}$ still ensures the same properties (transposed to $\mathtt{M}_{i+1}$), and eventually satisfies new properties Q, namely $\mathtt{M}_{i+1} \models \bar{\tau}(P) \wedge Q$ (where $\bar{\tau}$ applies sequentially to each formula of P an adequate formulæ transformation τ w.r.t. the iteration). A naive solution would be to perform again all the checkings with all properties on $\mathtt{M}_{i+1}$. In the case of arbitrary changes on M are permitted, three situations can occur: (i) in a lucky situation, the iteration is safe: P is still satisfied on $\mathtt{M}_{i+1}$; (ii) some (or all) of the properties within P are not satisfied anymore on $\mathtt{M}_{i+1}$; (iii) some (or all) of the properties cannot be checked, because the evolution changed so deeply $\mathtt{M}_i$'s structure that these properties become meaningless on $\mathtt{M}_{i+1}$. Furthermore, the model-checking costs can sometimes become an important issue, and running again and again these analysis can be time consuming.

A better solution consists in identifying sufficient conditions under which those properties are preserved by construction, by restraining the kinds of iterations allowed on a model. In [18], [19], Padberg, Gajewski and Ermel proposed a mechanism for preserving safety properties on Algebraic Petri-Nets based on place preserving morphisms. However, this result does not allow changes on the transitions guards, which is required in order to deal with the kind of iterations we are targeting. Therefore, we propose here to extend this result by extending the property preserving morphisms in a way that it becomes possible to stregthen guards without loosing previous behaviours.

In the two following definitions, we formalise the notion of APN and formulæ for expressing (invariants) requirements, and illustrate with our naive FS.

Definition III.1 (APN & Marking)**.** *An Algebraic Petri Net $N = (\Sigma, P, T, pre, post, cond, A) \in \mathcal{N}$ consists of an algebraic specification $\Sigma = (S, O, E)$ with S a set of sorts, O a set of many-sorted operations and E a set of equations defining the meaning of O's operations, and a Σ-algebra A; two sets P and T of places and transitions; two functions $pre, post : T \to [\mathcal{T}_\Sigma() \times P]$ labeling resp. input and output arcs of transitions with terms over Σ; a function $cond : T \to \wp(\mathcal{E}(\Sigma))$ assigning to each transition a finite set of equational conditions.*

For $N \in \mathcal{N}$, a N-marking $\mathtt{M} \in \mathbb{M}(N)$ is a function associating to each place a multiset of elements of A. The set of all M-follower *markings $\mathbb{M}_>(N, \mathtt{M})$ contains all markings obtained after firing all possible transitions from* M. □

Notationally, different APNs will be noted with superscripts, which are reflected componentwise: *e.g.,* if $N' \in \mathcal{N}$, then simply $N' = (\Sigma', P', T', pre', post', cond', A')$.

Example 1 (Naive FS)**.** *We define the* APN $\mathtt{FS} \in \mathcal{N}$ *for the naive* FS*, explaining how the mathematical structures are derived from AlPiNA's syntax, as depicted in Fig. 3 and the* ADT*s of Section II-C2. We detail each component of* FS*'s structure:*

- Σ_{FS} *is defined according to the* ADT*s, where S, O and E correspond respectively to* Sorts*,* Operations *and* Axioms*;*
- $A = (A_s)_{s \in S}$ *is an indexed (initial) algebra over each sort of S;*
- *Obviously from Fig. 3,*

$$P_{\mathsf{FS}} = \{\mathtt{files}, \mathtt{users}, \mathtt{writing}, \mathtt{reading}\}$$
$$T_{\mathsf{FS}} = \{\mathtt{openR}, \mathtt{openW}, \mathtt{closeR}, \mathtt{closeW}\}$$

- *the transitions labelling and the place guards are defined as follows:*

$$pre_{\mathsf{FS}} = \left\{ \begin{array}{r} \mathsf{openR}, \mathsf{openW} \mapsto [(\$\mathsf{f}, \mathsf{files}); (\$\mathsf{u}, \mathsf{users})]; \\ \mathsf{closeR} \mapsto [(\$\mathsf{p}, \mathsf{read}); (\$\mathsf{u}, \mathsf{users})]; \\ \mathsf{closeW} \mapsto [(\$\mathsf{p}, \mathsf{write}); (\$\mathsf{u}, \mathsf{users})] \end{array} \right\}$$

$$post_{\mathsf{FS}} = \left\{ \begin{array}{r} \mathsf{openR} \mapsto [(\mathsf{pair}(\$\mathsf{f}, \$\mathsf{u}), \mathsf{read}); (\$\mathsf{u}, \mathsf{users})]; \\ \mathsf{openW} \mapsto [(\mathsf{pair}(\$\mathsf{f}, \$\mathsf{u}), \mathsf{write}); (\$\mathsf{u}, \mathsf{users})]; \\ \mathsf{closeR}, \mathsf{closeW} \mapsto [(\mathsf{first}(\$\mathsf{p}), \mathsf{files}); (\$\mathsf{u}, \mathsf{users})] \end{array} \right\}$$

$$cond_{\mathsf{FS}} = \left\{ \begin{array}{r} \mathsf{openR}, \mathsf{openW} \mapsto \emptyset \\ \mathsf{closeR}, \mathsf{closeW} \mapsto \{\mathsf{name}(\$\mathsf{u}) = \mathsf{name}(\mathsf{second}(\$\mathsf{f}))\} \end{array} \right\}$$

The initial marking $\mathtt{M}_0$ corresponds to Fig. 2. □

Definition III.2 (Formulæ)**.** *Static formulæ are syntactically built with atomic formulæ denoting the marking of one place $p \in P$ with the data element $a \in A$ and the usual boolean connectors. If ϕ is such a formula, then $\Box\phi$ is an invariant formula. For a marking $\mathtt{M}$ defined on $N \in \mathcal{N}$, ϕ holds if it is included in $\mathtt{M}$, and $\Box\phi$ holds if ϕ holds in each state reachable from $\mathtt{M}$. We note $\mathtt{M} \models_N \Box\phi$.* □

Example 2. *Let $\mathtt{M}'_0$ be the marking $\mathtt{M}_0$ without user* eugene*. Then $\mathtt{M}'_0 \models_{\mathtt{FS}}$ P1 (*i.e., P1 *holds under $\mathtt{M}'_0$) but $\mathtt{M}_0 \not\models_{\mathtt{FS}}$ P1, because* eugene*'s security level is lower than* f2*'s.* □

Definition III.3 (Guard Strenghtening)**.** *Let $N, N' \in \mathcal{N}$ be componentwise equal* APNs *except for transitions conditions $cond$ and $cond'$ respectively. N'* strengthens *N if for all transitions $t \in T$, $cond'(t) \Longrightarrow cond(t)$.* □

Note that from an MDE point of view, guards are strengthened by adding new guards in conjunction to the previous ones, which makes the implication trivially holding.

Relating successive iterations in such a way that structure is not completely lost is an important feature for validating iterations' correctness. This is achieved with APN morphisms: an APN *morphism* from N to N' is a tuple $f = (f_\Sigma, f_P, f_T, f_A) \in \mathcal{M}(N, N')$ of functions between the algebraic specifications $f_\Sigma : \Sigma \to \Sigma'$, the places $f_P : P \to P'$, the transitions $f_T : T \to T'$ and the algebræ $f_A : A \to A'$.

The following definitions specify two different morphisms: the first one is the classical *place preserving* (PP) morphism used in [19]; the second takes care of guards streghtening (GS).

Definition III.4 (Place Preserving / Guard Strengthening Morphisms)**.** *A morphism $f \in \mathcal{M}(N, N')$ is* place preserving *(noted $f \in \mathcal{M}_{\mathtt{PP}}(N, N')$) if:*

- *Firing conditions are preserved;*
- *Places are preserved, meaning no new arcs are added to mapped places;*
- *f_T, f_P and f_Σ are injections, and f_Σ is persistent, meaning Σ' is a correct extension of Σ;*
- *N' embeds N, meaning all arcs are mapped, more precisely, there can be more places in the pre and $post$ domains of a mapped transition than in the corresponding domains of the original transition;*
- *A' is merely extended for new parts in A by f_A, or it is merely renamed.*

Precise (mathematical) definitions can be found in [18].

f is a place preserving guard strengthening *(noted $f \in \mathcal{M}_{\mathtt{PPGS}}(N, N')$) if there exists $N'' \in \mathcal{N}$ and $f_{\mathtt{PP}} \in \mathcal{M}_{\mathtt{PP}}(N, N'')$ and N' strengthens N''.* □

Notice that because in Def. III.3, we require implication of conditions, a place preserving guard strengthening iteration is still place preserving, since only the firing conditions are affected by the strengthening (first condition).

We will note $f_{\mathtt{M}} : \mathbb{M}(N) \to \mathbb{M}(N')$ the extension of f to markings of these nets; and $\tau_f(\phi)$ the transformation of invariant formulæ w.r.t. f.

We now extend the classical result for invariant preservations under PP morphisms to the case where guards are also strengthened. This allows to better reflect iterative development practices by also changing transitions guards in a controlled way.

Theorem III.1 (Invariant preservation under PP morphisms)**.** *Let $N, N' \in \mathcal{N}$, $f \in \mathcal{M}_{\mathtt{PP}}(N, N')$, and $\mathtt{M} \in \mathbb{M}(N)$, $\mathtt{M}' \in \mathbb{M}(N')$ be markings such that $\mathtt{M} = \mathtt{M}'|_f$ (meaning $\mathtt{M}$ is the restriction of $\mathtt{M}'$. to $f_{\mathtt{M}}(\mathtt{M})$). The following implication holds for any invariant formula $\Box\phi$:*

$$\mathtt{M} \models_N \Box\phi \Longrightarrow \mathtt{M}' \models_{N'} \tau_f(\Box\phi)$$

Proof: The proof is given in [18, Theorem 3.17]. ■

Proposition III.2 (Marking Inclusion under GS)**.** *Let $N, N' \in \mathcal{N}$ such that N' strengthens N. Then, N'-markings are included in N-markings. More formally,*

$$\mathtt{M}' \in \mathbb{M}(N') \subseteq \mathtt{M} \in \mathbb{M}(N) \Longrightarrow \mathbb{M}_>(N', \mathtt{M}') \subseteq \mathbb{M}_>(N, \mathtt{M})$$

Proof: By induction on the transitions firings. The base case is trivial: starting from the same initial marking $\mathtt{M}_0 \in \mathbb{M}(N)$ and also $\mathtt{M}_0 \in \mathbb{M}(N')$, because N' is strengthened, less transitions are enabled, producing a included follower markings set. Let assume now that we have $\mathtt{M} \in \mathbb{M}(N)$ and $\mathtt{M}' \in \mathbb{M}(N')$ such that $\mathtt{M}' \subseteq \mathtt{M}$. What happens for each follower markings? Follower markings sets are built by considering all fireable transitions (non-fireable ones do not play any role). We prove that the markings resulting from each such transition are included, making the global marking corresponding to the follower marking set is included. For such a transition $t \in T$, there is two possibilities: (i) if t is enabled in N and also in N', then the follower marking for t just propagates; (ii) if t is enabled in N but not in N', then a new marking $\mathtt{M}_{\mathtt{new}}$ is produced for N, and the follower marking for t is indeed included because it already was and the new N-marking contains one new marking. ■

Proposition III.3 (Invariant preservation under PP-GS morphisms)**.** *Implication of Thm. III.1 holds if $f \in \mathcal{M}_{\mathtt{PPGS}}(N, N')$.*

Proof: By Def. III.4, there exists $N'' \in \mathcal{N}$ and $f_{PP} \in \mathcal{M}_{\mathtt{PP}}(N, N'')$ and N' strengthens N''. Then successively applying Thm. III.1 and Prop. III.2 finishes the proof. ■

IV. EVOLVING SYSTEMS

The File System of Fig. 3 was not ensuring any property we were interested in. This Section presents two interesting iterations: the first step leads to a *simple File System* $\mathtt{FS_s}$ that matches Requirement **R1**, *i.e.,* files can be opened only by users with sufficient access rights; the second iteration leads to a *Confined File System* $\mathtt{FS_c}$ that matches Requirement **R2**, *i.e.,* it further ensures the confinement property on file opening.

A. Simple File System $\mathtt{FS_s}$

The FS of Fig. 2 was naive in the sense that it simply implements the basic mechanisms for opening / reading and closing files, without access control. We make FS evolve into

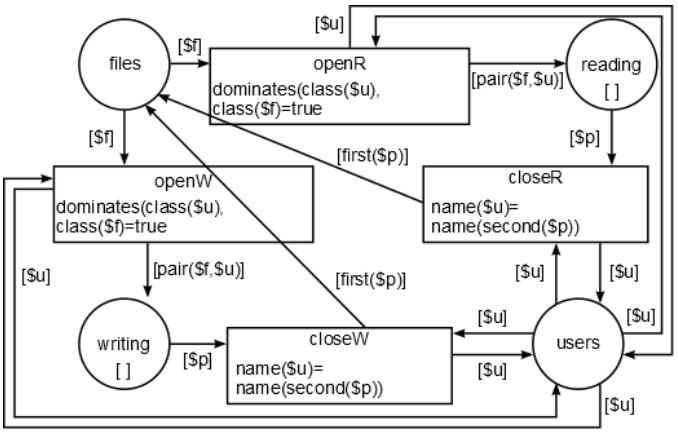

Fig. 4. Simple security filesystem. Same initial marking as Fig. 3.

$\mathtt{FS_s}$ to meet Requirement **R1** by guarding each file use (either reading or writing) by a condition that checks the access rights: in Fig. 4, each transition (`openR`, `openW`, `closeR` or `closeW`) is now guarded to prevent the firing if the `$u`'s access class does not dominate `$f`'s access class.

From a specification point of view, the corresponding APN $\mathtt{FS_s} \in \mathcal{N}$ is componentwise identical to $\mathtt{FS} \in \mathcal{N}$, except for the guards. More formally, we have:

$$cond_{\mathsf{FSS}} = \left\{ \begin{array}{l} \mathsf{reading, writing} \mapsto \{\mathsf{dominates(class(\$u), class(\$f))} = \mathsf{true}\} \\ \mathsf{closeR, closeW} \mapsto \{\mathsf{name(\$u)} = \mathsf{name(second(\$f))}\} \end{array} \right\}$$

In this iteration, the PN's structure does not change: no places are added or removed; therefore the morphism is simply the identity: $f_{\mathsf{PP}} = id$. Nevetheless, the transitions are strengthened by new guards that straightforwardly imply the previous empty guards ($cond_{\mathsf{FSS}} \Longrightarrow cond_{\mathsf{FS}}$). Table I (Row 2) summarises the iteration when model-checking all properties on $\mathtt{FS_s}$: `P1` and `P2` now hold.

B. Confined File System $\mathtt{FS_c}$

We now make $\mathtt{FS_s}$ evolve into $\mathtt{FS_c}$ to prevent Trojan horses attacks happening, thus meeting Requirement **R2**. Two new places `logRead` and `logWrite` are added: they log a list of pairs $(\$\mathtt{f}, \$\mathtt{u})$, adding an element each time `$u` opens `$f` and deleting it when closed, for each opening mode. Every time a *new* file is opened in read mode, `openRead` checks from `logWrite` if no file with smaller access rights is open by the same user in write mode. We introduce a new function $\mathtt{min}(\$\mathtt{lfu})$ that computes the minimum access class (for the domination) in the list `$lfu` of pairs $(\$\mathtt{f}, \$\mathtt{u})$. The new guard for the `openRead` transition is now a conjunction of the previous guard with a new condition stating that the minimal access class in `logWrite` dominates the newly opened file. The reverse principle is applied for `openWrite` with a function `max` retrieving the maximal access class logged in `logRead`. The corresponding ADT is defined as follows:

```
0  import "file.adt"
   import "user.adt"
2  import "accessClass.adt"
   import "pair.gadt"
4  import "list.gadt"
   import "fileUserPair"
6  Adt listOfFiles is list[FileUserPair]
     Operations
8      maxAccess : list[FileUserPair] -> accessClass;
       calcMaxAccess : accessClass, list[FileUserPair] -> accessClass;
10
       minAccess : list[FileUserPair] -> accessClass;
12     calcMinAccess : accessClass, list[FileUserPair] -> accessClass;
     Axioms
14     maxAccess($l) = calcMaxAccess(first($l),$l);

16     calcMaxAccess($acl,empty) = $acl;
       if (class(first($p)) dominates $acl) = true then
18         calcMaxAccess($acl,list($p,$l))=calcMaxAccess(class(first($p)),$l);
       if ($acl dominates getAccessClass(first($p))) = true then
20       calcMaxAccess($acl,list($p,$l)) = calcMaxAccess($acl,$l);

22     minAccess($l) = calcMinAccess(first($l),$l);

24     calcMinAccess($acl,empty) = $acl;
       if (class(first($p)) dominates $acl) = true then
26       calcMinAccess($acl,list($p,$l)) = calcMinAccess($acl,$l);
       if ($acl dominates class(first($p))) = true then
28       calcMinAccess($acl,list($p,$l)) = calcMinAccess(class(first($p)),$l);
     Variables
30     l   : list[FileUserPair];
       p   : FileUserPair;
32     acl : accessClass;
```

The APN $\mathtt{FS_c} \in \mathcal{N}$ is defined as follows:

- Σ_{FSC} is the same as Σ_{FS} with the additional ADT listOfFiles, and A_{FSC} the corresponding initial algebra;
- From Fig. 5, we have

$$P_{\mathsf{FSC}} = P_{\mathsf{FS}} \cup \{\mathtt{logRead}, \mathtt{logWrite}\}$$
$$T_{\mathsf{FSC}} = T_{\mathsf{FS}}$$

- the input/output arcs for each transition are those previously defined for FS, plus those coming from respectively logWrite and logRead:

$$pre_{\mathsf{FSC}} = \left\{ \begin{array}{l} \mathsf{openR} \mapsto pre_{\mathsf{FS}}(\mathsf{openR}) \oplus [(\$\mathsf{lpw}, \mathsf{logWrite})]; \\ \mathsf{openW} \mapsto pre_{\mathsf{FS}}(\mathsf{openW}) \oplus [(\$\mathsf{lpw}, \mathsf{logRead})]; \\ \mathsf{closeR} \mapsto pre_{\mathsf{FS}}(\mathsf{closeR}) \oplus [(\$\mathsf{lpr}, \mathsf{logRead})]; \\ \mathsf{closeW} \mapsto pre_{\mathsf{FS}}(\mathsf{closeW}) \oplus [(\$\mathsf{lpr}, \mathsf{logWrite})]; \end{array} \right\}$$

$$post_{\mathsf{FSC}} = \left\{ \begin{array}{l} \mathsf{openR} \mapsto post_{\mathsf{FS}}(\mathsf{openR}) \oplus [(\mathsf{list(pair(\$f, \$u), \$lpr)}, \mathsf{logRead})]; \\ \mathsf{openW} \mapsto post_{\mathsf{FS}}(\mathsf{openW}) \oplus [(\mathsf{list(pair(\$f, \$u), \$lpw)}, \mathsf{logWrite})]; \\ \mathsf{closeR} \mapsto post_{\mathsf{FS}}(\mathsf{closeR}) \oplus [(\mathsf{delete(\$p, \$lpr)}, \mathsf{logRead})] \\ \mathsf{closeW} \mapsto post_{\mathsf{FS}}(\mathsf{closeW}) \oplus [(\mathsf{delete(\$p, \$lpr)}, \mathsf{logWrite})] \end{array} \right\}$$

- The guards for transitions closeR and closeW are the same as previously; those for the added transitions logRead and logWrite are empty; and the guards for openR and openR are enforced accordingly:

$$cond_{\mathsf{FSC}} = \left\{ \begin{array}{l} \mathsf{logRead} \mapsto \emptyset \\ \mathsf{logWrite} \mapsto \emptyset \\ \mathsf{openR} \mapsto \left\{ \begin{array}{l} \mathsf{dominates(class(\$u), class(\$f))} = \mathsf{true} \ \ \& \\ \mathsf{dominates(minAccess(\$lpw), class(\$f))} = \mathsf{true} \end{array} \right\} \\ \mathsf{openW} \mapsto \left\{ \begin{array}{l} \mathsf{dominates(class(\$u), class(\$f))} = \mathsf{true} \ \ \& \\ \mathsf{dominates(class(\$f), maxAccess(\$lpr))} = \mathsf{true} \end{array} \right\} \\ \mathsf{closeR} \mapsto cond_{\mathsf{FS}}(\mathsf{closeR}) \\ \mathsf{closeW} \mapsto cond_{\mathsf{FS}}(\mathsf{closeW}) \end{array} \right\}$$

We now define the morphism $f \in \mathcal{M}(\mathtt{FS_s}, \mathtt{FS_c})$ for this iteration. Since $\mathtt{FS_c}$ had been defined by extending $\mathtt{FS_s}$, the morphism only consists of mapping the previous structure into the extended ones:

- f_Σ maps sorts, operations and equations to the syntactically corresponding ones: sort fileName in $\mathtt{FS_s}$ is mapped to sort fileName in $\mathtt{FS_c}$ and so one, and the same for operations and equations;
- f_A naturally maps terms syntactically;

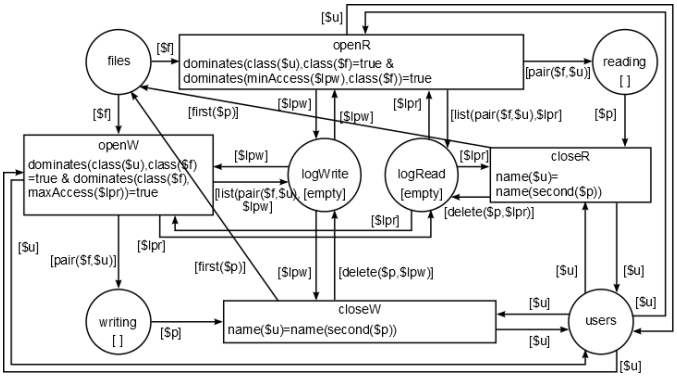

Fig. 5. Confined filesystem. Same initial marking as Fig. 3.

- f_P and f_T map places and transitions using name correspondance: place files in $\mathtt{FS_s}$ is mapped to place files in $\mathtt{FS_c}$, and so on.

We still have to check that conditions from Def. III.4 hold for f. We briefly comment on each of these conditions:

- w.r.t. the note following Def. III.4, firing conditions are preserved due to the guard strengthening;
- comparing Fig. 4 and Fig. 5 and their formal specifications, no new arcs are added to the places already present in $\mathtt{FS_s}$;
- Obviously from their definition, f_T, f_P and f_Σ are injections and f_Σ is persistent since $\Sigma_{\mathtt{FSS}} \subseteq \Sigma_{\mathtt{FSC}}$;
- $\mathtt{FS_c}$ is an embedding of FSC according to the definition of $\mathtt{FS_c}$ by extension of $\mathtt{FS_s}$ for pre and $post$;
- $A_{\mathtt{FSC}}$ is a correct extension of $A_{\mathtt{FSS}}$ with no renaming.

These constitute a safe iteration from $\mathtt{FS_s}$ to $\mathtt{FS_c}$ because the corresponding morphisms match the conditions of Section III. Table I (Row 3) summarises the model-checking of all properties on $\mathtt{FS_c}$: P1 and P2 still hold; P3 is now satisfied.

V. Related Work

Petri Nets and their declinations like Colored PN (CPNs), are widely used for formalising systems' behaviour and analysing their relevant properties. However from an MDE point of view, translating complex models into APNs is easier, because the gap for encoding the necessary data structures a engineer is working with is smaller due to the richness of algebraic specifications. However, a very restricted list of contributions investigated invariant preservation in PNs.

Padberg, together with several other co-authors, published extensively on invariant preservation of APNs, building a full categorical framework for APNs rule-based refinements. Our contribution extends Padberg's result published in [18], [19] but as already mentioned, this result does not consider guard strenghtening. To the best of our knowledge, Padberg's work on this topic has been discontinued after a last survey paper [9] has been published on the topic in 2003.

Around 2000, Cheung and Lu [2] studied five classes of invariant-preserving transformations in CPNs, namely Insertion, Elimination, Replacement, Composition and Decomposition. Most closer to our work are the Insertion and Replacement (of transitions) transformations: for the former, they obtain full preservation but at the price restraining the markings and preserving guards, instead of our strengthening; while the latter is directly related to our guard strengthening, although their result also consider removing adjacent transitions' arcs. Nevertheless, guard strengthening, a necessary iterative process for tackling iterative MDE development, is not explicitly considered. The authors later concentrated in Place/Transition PNs rather than CPNs [5], [8]. Lewis' Ph.D [13] studied morphisms allowing behaviour-preserving refinements, and in particular refinements through bisimulation.

In the MDE community, iterative development is widely used and adopted, with several case studies demonstrating its relevance (cf. *e.g.,* Grau, Joseph, and Sagesser's work on an iterative lifecycle model completing the classical waterfall one [4]). Himsl *et al.* [6] proposed an iterative (meta-) modeling process for enterprise engineering, dealing with metamodels migrations with reflection back on previously designed instances. This is directly related to our engineering process, although their contribution is strictly syntactic, whereas our iterations concern also the behaviour. Konrad *et al.* [10] proposed i^{2}MAP, an UML-based framework for incremental and iterative modeling and analysis process dedicated to embedded systems. Very close to our approach, they derive temporal logic formulæ from natural language specifications to express goals, against which syntactic and behavioural consistencies are checked. Kumazawa and Tamai studied in [11] an interesting semi-automated iterative development case called *model fixing*: it consists in using counterexamples of a model-checker to enhance a model in order to make it satisfy a property, by only using previously checked properties, the original model and counterexamples, without sacrificing too much the original model's integrity. It however would require some additional work to adapt this technique for APNs, which is richer than the labeled transitions systems extension they are using. Uzam and Zhou proposed in [20] an iterative and "easy-to-use deadlock prevention policy" for Flexible Manufacturing Systems, based on reachability analysis on Petri Nets. At each iteration, supposing that an effective solution to deadlock prevention exists, they detect a bad marking that is further used to prevent its reachability of the system, by modifying the Net accordingly. Our work contrasts with theirs in the fact that we deal with APNs, which are more expressive from an MDE perspective, and that our iterations are mathematically proven, whereas their methodology was proved not to be sound (cf. [14] for counterexamples).

VI. Conclusion

In this paper, we presented a preliminary study on invariant preservation of behavioral models expressed in Algebraic Petri Nets, in the context of an iterative modeling process. Given a set of requirements and a metamodel for a Multi-Level Security File System, we developed a first APN model. We also mapped the set of requirements from onto properties expressed as APN invariants. For each property that is not satisfied after running a model checker, we iteratively evolve the model to a next version in order to eventually satisfy that property while

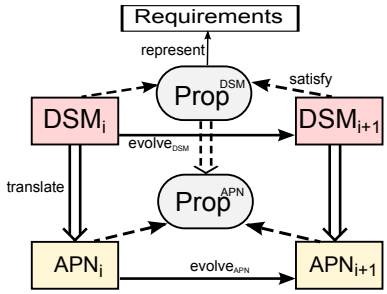

Fig. 6. Generalisation of our approach.

preserving properties that were already implemented. We also provide a formal mathematical proof of our proposed model evolution theory based on invariant preservation. Our case study shows the feasibility of our approach and, to a certain extent, that the evolution constraints we consider are not too restrictive but in fact usable to solve engineering problems.

Our future research is concerned with an extension to enhance our theory to preserve other kinds of properties besides invariants, for example liveness properties [9], temporal properties, *etc* by using existing work in the literature. Furthermore, we would like to support the integration of modeling language with higher level of abstraction than APN, namely Domain-Specific Languages in general. This generalisation of our approach is illustrated in Fig. 6.

The preservation of the properties at the DSM level can be achieved by making use of the commuting diagram in Fig. 6. In order to verify that an iteration at the DSM level is property preserving (formally, $evolve_{DSM} \circ translate$), we check that it's counterpart at the APN level is also property preserving (formally, $translate \circ evolve_{APN}$). Of course, in order to do this we have to be sure that the translations of both the DSLs models into APNs models and properties at the DSL level into properties at the APN level are formally verified. Finally, larger case studies are necessary to validate the usability of our work in engineering environments.

We think the theory presented in this paper contributes to simplifying the complexity of model evolution and can be used as a fundamental "infrastructure" for the safe evolution of behavioral models tackling different domains.

REFERENCES

[1] Marshall D. Abrams, Sushil G. Jajodia, and H. J. Podell, editors. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[2] To-Yat Cheung and Yiqin Lu. Five Classes of Invariant-Preserving Transformations on Colored Petri Nets. In *Application and Theory of Petri Nets*, volume 1639 of LNCS, pages 692–692. Springer, 1999.

[3] David Elliot Bell and Leonard LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, The MITRE Corp., 1973.

[4] R. Grau, B. Joseph, and K. Sagesser. Introducing an Iterative Lifecycle Model at Credit Suisse IT Switzerland. *IEEE Software*, 99, 2012.

[5] Hejiao Huang, To-yat Cheung, and Wai Ming Mak. Structure and Behavior Preservation by Petri Net-Based Refinements in System Design. *Theoretical Computer Science*, **328**(3):245–269, 2004.

[6] M. Himsl, D. Jabornig, W. Leithner, P. Regner, T. Wiesinger, J. Küng, and D. Draheim. An Iterative Process for Adaptive Meta- and Instance Modeling. In *Database and Expert Systems Applications*, volume 4653 of LNCS, pages 519–528. Springer, 2007.

[7] Steve Hostettler, Alexis Marechal, Alban Linard, Matteo Risoldi, and Didier Buchs. High-Level Petri Net Model Checking with AlPiNA. *Fundamenta Informaticæ*, 113(3–4):229–264, 2011.

[8] H. Huang and L. Jiao. *Property-Preserving Petri Net Process Algebra in Software Engineering*. World Scientific Pub., 2012.

[9] Julia Padberg and Milan Urbásek. Rule-Based Refinement of Petri Nets: A Survey. In *Petri Net Technology for Communication-Based Systems*, pages 161–196, 2003.

[10] Sascha Konrad, Heather Goldsby, and Betty Cheng. i^{2}MAP: An Incremental and Iterative Modeling and Analysis Process. In *Model Driven Languages and Systems*, volume 4735, pages 451–466, 2007.

[11] T. Kumazawa and T. Tamai. Iterative Model Fixing with Counterexamples. In *Software Engineering Conference*, pages 369 –376, 2008.

[12] Craig Larman and Victor R. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36(6):47–56, June 2003.

[13] G.A. Lewis. *Incremental Specification and Analysis in the Context of Coloured Petri Nets*. U. of Tasmania, 2002.

[14] Zhi Wu Li and Gai Yun Liu. Comments on "An Iterative Synthesis Approach to Petri Net Based Deadlock Prevention Policy for Flexible Manufacturing Systems". In *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, page 692, 2009.

[15] Levi Lúcio, Qin Zhang, Vasco Sousa, and Tejeddine Mouelhi. Verifying Access Control in Statecharts. In *Journal of ECASST, MPM'11 workshop*, Wellington, October 2011.

[16] Maximiliano Cristiá, Gisela Giusti, and Felipe Manzano. The Implementation of Lisex, a MLS Linux Prototype. In *ASSE*, 2005.

[17] P.-A. Muller, F. Fleurey, and J.-M. Jzquel. Weaving Executability into Object-Oriented Meta-Languages. In *Proc. of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[18] J. Padberg, M. Gajewsky, and C. Ermel. Refinement versus Verification: Compatibility of Net Invariants and Stepwise Development of High-Level Petri Nets. Technical report, Technische Universitat Berlin, 1997.

[19] Julia Padberg, Magdalena Gajewsky, and Claudia Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. *Science of Computer Programming*, 40(1):97–118, 2001.

[20] Murat Uzam and Meng Chu Zhou. An Iterative Synthesis Approach to Petri Net-Based Deadlock Prevention Policy for Flexible Manufacturing Systems. *IEEE Transactions on Systems, Man and Cybernetics – Part A: Systems and Humans*, **37**(3):362–371, 2007.