A Precise Definition of Operational Resilience

Levi Lúcio and Nicolas Guelfi
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg

TR-LASSY-11-02

Abstract

Resilience in Information and Communication Technology (ICT) systems was introduced around the seventies and has been more intensively used in the research community in the very last years. If we refer to the literature on the topic we find that the word *resilience* is used with many different definitions and at different levels. This is a problem as it hinders communication between researchers and does not allow for a fundamental theory on the subject.

This paper is a proposal to address this issue theoretically and operationally. In our laboratory we have developed a framework called *DREF* where resilience is essentially defined as a property of an *evolving system* that is considered to improve capabilities to avoid failures. In this paper we formalize operational support for this idea by using artifacts and tools from the modelling world. In particular we define an *evolving system* as a sequence of models and compute if that evolving system is resilient regarding a property of interest by using a model checker.

Our proposal is aimed both at providing a formal definition of the *resilience* concept and at providing the means to produce or analyze a resilient system. Our approach is illustrated by developing a filesystem which is resilient regarding *confidentiality*.

1 Introduction and Problem Statement

We consider software and systems' dependability as a composite quality. While a traditional cartesian approach to engineering would require to consider it strictly scientifically and progressively throughout the whole lifecycle, a loose pragmatic approach tackles it incrementally as soon as its satisfaction level is below the stakeholders' acceptance threshold. The generic problem we address in this paper is to find a solution that would combine those two, a priori, contradictory viewpoints. More precisely, we need an approach that allows for flexible handling of dependability in a scientific framework supported by software engineering tools and techniques.

Resilience was introduced in ICT systems around the seventies [9] and has been more intensively used in the research community in the very last years[1]. By analyzing these references we can notice that the word is used with many different definitions and at different levels [5, 13, 20].

In [10], we have proposed a formal framework called *DREF* designed to ease the development of dependable systems from a software engineering perspective. This framework provides a mathematical definition of resilience and related concepts. In *DREF* the fundamental concepts are: *entities*, *properties*, *satisfiability functions*, *nominal satisfiability*, *tolerance threshold* and *evolution*. *Entities* are anything that is of interest to be considered. It might be e.g. a program, a database, a person, a hardware element, a development process, or a requirement document. *Properties* are the basic concepts to be used to characterise *entities*. It might be e.g. an informal requirement, a mathematical property or any entity that aims at being interpreted over entities. The fact that a *property* is satisfied by an *entity* is defined by a *satisfiability function*[2] having the real numbers as co-domain.

In our context, we want to consider *entities* whose existence (i.e. definition) may vary. Thus change is the difference between two definitions of two entities distributed over a common evolution axis. The intention is to allow for comparison of entities relatively to evolution axes.

Intuitively, we define the concept of *resilience* as the existence of a change for improvement by an evolution process which reduces failures and tolerance needs regarding a property of interest. For instance, if one considers *security* as a property of interest of an evolving e-banking system, that e-banking system would be considered resilient regarding security if the number of successful attacks involving unauthorized money transfers would diminish over the system's lifetime. Examples of practical usage of this definition of resilience could be to provide a methodology to build resilient evolving systems or to evaluate if a given evolving system is resilient regarding a certain property.

[1]on the approximately 1300 citations using the term resilient or resilience registered at DBLP, 90% appeared after 2000 and 75% in the last five years.

[2]We define two categories of satisfiability functions: nominal satisfiability and tolerance threshold. Both partition the satisfiability space in three parts: nominal (from and above nominal satisfiability), failures (below the tolerance threshold) and tolerance in between the two (representing degradation).

Notwithstanding that *DREF* is defined mathematically, the theory is given at a meta level. This paper proposes an instantiation of the *DREF* framework designed to be integrated with operational support to compute *satisfiability* – and consequently *resilience*. In *DREF* the satisfaction of a property by an evolving system is supposed to be provided and is relative to a set of subjective stakeholders (i.e. observers). In this paper we propose to automatically decide on the satisfaction of property of interest. In order to do this we will restrict the properties of interest we are considering to computable ones.

For our operational purposes we use concepts from the modelling world to define and process the notions of *entities*, *properties* and *satisfaction*. In particular *entities* are system models defined in term of Algebraic Petri Nets (APN) [17] and *properties* are defined in terms of safety properties – i.e. invariants regarding places in the APN as defined by the model checker AlPiNA [6].

The paper is structured as follows: in section 2 we present the mathematical theory behind our operational notion of resilience; in section 3 we provide an example of an evolving filesystem exhibiting resilience regarding the confidentiality property; in section 4 we study the theoretical constraints on evolution regarding the choices we have made for the particular modelling and model checking formalisms used in section 3; in section 5 we discuss the open issues of our proposal; in section 6, we present a state of the art focused on existing approaches to resilience; finally section 7 discusses perspectives of this research and concludes.

2 Operational Resilience

In this section we provide formal definitions of the concepts we manipulate in this paper. In the text that follows we consider a universe including the disjoint sets $ENTITY$ and $PROPERTY$. We start by a set of auxiliary definitions.

Definition 2.1 *Evolving System and Discrete Evolution*

An evolving system *is an indexed set of entities* $es = \{ent_1, \ldots, ent_n\} \subseteq ENTITY$*. A* discrete evolution *of es is a pair* $(ent_k, ent_{k+1})(1 \leq k < n)$*. The set of all discrete evolutions of es is written* DE_{es}*.*

Definition 2.2 *Composed Property*

Given an evolving system $es = \{ent_1, \ldots, ent_n\} \subseteq ENTITY$*, a* composed property $C \in \mathcal{P}(PROPERTY)$ *is a set of decidable properties of the elements of es, each of those decidable properties* $p \in C$ *being called a* component property*. The set of all composed properties is called* $C - Property$*. Given a component property* $p \in C$ *and an entity* ent_k $(1 \leq k \leq n)$ *we write* $ent_k \models p$ *to state that* ent_k *satisfies*[3] p*.*

[3]w.r.t. the DREF framework in [10], we have a satisfiability function $\models: es \times c \rightarrow \{0, 1\}$ where $c \in PROPERTY$ is a decidable property. When instantiating the concepts of DREF, nominal satisfiability is 1, failure threshold is 0 and there is no tolerance margin (i.e. no value between 0 and 1).

Intuitively, a *composed property* corresponds to a set of decidable properties which together make up a property regarding which *resilience* can be measured.

Definition 2.3 *Property-Preserving Discrete Evolution*
Given an evolving system $es = \{ent_1, \ldots, ent_n\} \subseteq ENTITY$ *and a composed property* $C \in C{-}Property$*, a discrete evolution* $(ent_k, ent_{k+1}) \in DE_{es}$ $(1 \leq k < n)$ *is* $property-preserving$ *regarding* $p \in C$ *iff* $ent_k \models p \Rightarrow ent_{k+1} \models p$.

Let us now define the concept of *monotonic resilience*, which is the formal definition of *resilience* we will use throughout this paper.

Definition 2.4 *Monotonic Resilience*
Given an evolving system $es = \{ent_1, \ldots, ent_n\} \subseteq ENTITY$ *and a composed property* $C \in C - Property$ *we say that es exhibits monotonic resilience regarding* C *iff for all* $(ent_k, ent_{k+1}) \in DE_{es}$

$$\big|\{p \mid p \in C \ \wedge \ ent_k \models p\}\big| \ \leq \ \big|\{p \mid p \in C \ \wedge \ ent_{k+1} \models p\}\big|$$

and for all $p \in C$ *such that* $ent_k \models p$*,* (ent_k, ent_{k+1}) *is property-preserving.*

In other words, definition 2.4 states that an evolving system exhibits monotonic resilience regarding a C-Property C when any entity in the of that evolving system satisfies at least the same component properties of C as the previous one[4].

As mentioned in section 1, we will be using concepts from the modelling world to represent and reason about the notions of *entity*, *property* and *satisfaction*, in particular: Algebraic Petri Nets [17] (APN) as a modelling langage to represent entities and more generally *evolving systems* (see definition 2.1); the AlPiNA model checker [6, 19] to model decidable *properties* (see definition 2.2) and compute their satisfaction on APN models.

Algebraic Petri Nets are a formalism used for modelling, simulating and studying the properties of concurrent systems. They are based on the well known Place/Transition (P/T) Petri Nets formalism where *places* hold resources – also known as tokens – and *transitions* are linked to places by input and output *arcs*, which can be weighted. Normally a Petri Net has a graphical concrete syntax consisting of circles for *places*, boxes for *transitions* and arrows to connect the two. The semantics of a P/T Petri Net involves the sequential non-deterministic firing of transitions in the net – where firing a transition means consuming tokens from the set of places linked to the input arcs of the transition and producing tokens into the set of places linked to the output arcs of the transition. The algebraic extension allows defining tokens as elements of sets (with associated operations) which are models of algebraic specifications. The arcs of APNs can

[4]Stronger alternatives to *monotonic resilience* would force the number of properties satisfied by ent_{k+1} to be strictly greater than those satisfied by ent_k, or that ent_n would satisfy strictly more component properties than ent_1.

include weights defined by terms of the algebraic specification and the transitions can be guarded by algebraic equations.

The AlPiNA model checker uses as models Algebraic Petri Nets. Specifications in AlPiNA are composed of two parts: an algebraic specification which is a set of abstract definitions of sorts and associated operations; a Petri Net which is represented graphically. AlPiNA and is able to decide on the satisfaction of invariant properties on those nets. The invariants are expressed as conditions on the tokens contained by places in the net at any state of the net's semantics. Invariants are built using first order logic, the operations defined in the algebraic specification and additional functions and predicates on the number of tokens contained by places.

3 Motivational Example – A Confidentiality Resilient Filesystem

In this section we introduce an example of an evolving system presenting *monotonic resilience*. The example is inspired from [8] where a UNIX-like operating system including a multi-level security filesystem is described. In the context of our paper we will describe how to measure resilience regarding the *confidentiality* property across three entities representing three evolutions of the operating system's filesystem.

In Multi-Level Security (MLS) [3] there are two main concepts: *objects* and *subjects*. *Objects* regard system resources or data repositories that must be protected, such as files, directories or terminals. *Subjects* regard entities capable of requesting services from those system resources such as users or processors.

In MLS objects and subjects are associated to an *access class*. Access classes are used to classify objects and subjects according to their confidentiality or responsibility degree respectively. Intuitively, an object having a high access class can only be seen or manipulated by a subject who is highly trusted.

For our studies we will use as example an MLS filesystem. We will only consider *files* from the set of possible *objects* and *users* from the set of possible *subjects*. In order to keep the case study manageable, we will use a simplified model of a filesystem where we consider that a file can be either being read or written by a user, or idle. In particular this means that in our models a file cannot be read by multiple users simultaneously.

3.1 Properties

For our example we are interested in the *confidentiality* property, i.e. the fact that data inside files can only be accessed for reading or writing by users that have enough privileges to do so. An important confidentiality threat which we will take into consideration in this study are trojan horses [1]. Trojan horses consist of code which is executed by a trusted user without his/her knowledge or consent and can pass confidential data to an untrusted user by copying that data to a file an untrusted user can access. In order to prevent such problems from arising we further develop the *confidentiality* property to include the fact that a user cannot have two files opened simultaneously where the more confidential one is open in *read* mode and the less confidential one in *write* mode[5].

Summarizing, and according to the theory we have presented in section 2, we have detailed the confidentiality property in the following *composed property*:

- A file with a certain confidentiality level can only be accessed for reading or writing by a user who is sufficiently trusted;
- A user have cannot two files opened simultaneously where the more confidential one is open in *read* mode and the less confidential one in *write* mode.

3.2 Entities

In order to illustrate a resilient system we will present three entities representing an evolving system which models of a *confidential filesystem*[6].

3.2.1 Naive Filesystem

Let us start with the first entity which we will call *naive filesystem*. The APN model can be observed in figure 1 and represents the semantics of the operation of a filesystem. The Petri net uses several kinds of algebraic tokens[7]: pairs belonging to the set $fileName \times accessClass$ are used in place *filesystem* to represent file names and their respective access classes; pairs belonging to the set $userName \times accessClass$ are used in place *users* to represent a sample of users of the filesystem; finally places *readFiles* and *writeFiles* hold tokens which are pairs $(fileName \times accessClass) \times (userName \times accessClass)$ in order to keep track of which file was opened by which user. We do not explicitly define the names of the variables that act as weights on the arcs given their type can be inferred from the either the origin or the target place of the arc.

The semantics of the *naive filesystem* model in figure 1 is such that it simulates opening files in read or write mode and closing them – by firing the *openForRead*, *openForWrite* and *close* transitions respectively. The variables on the entry arcs of the transitions declare the consumed tokens from the input places and the variables on the output arcs of the transitions declare the produced tokens on the output places. Note that, despite the fact that transitions in APN models may be guarded, in this naive version of the confidential filesystem the *openForRead* and *openForWrite* transitions we do not check if the the user has permission to access a given file.

[5]In the literature this property of a filesystem is called *confinement*.

[6]The interested reader may download the Algebraic Petri Net models and associated properties presented in this paper at [12] and validate the properties against the model using the AlPiNA model checker [19].

[7]For the presentation of the *confidential filesystem* example we use in this paper names of sorts and operations which, despite being self descriptive, for space reasons cannot be formally introduced. A full description of the algebraic specifications used in the *confidential filesystem* example can be found in appendix A.

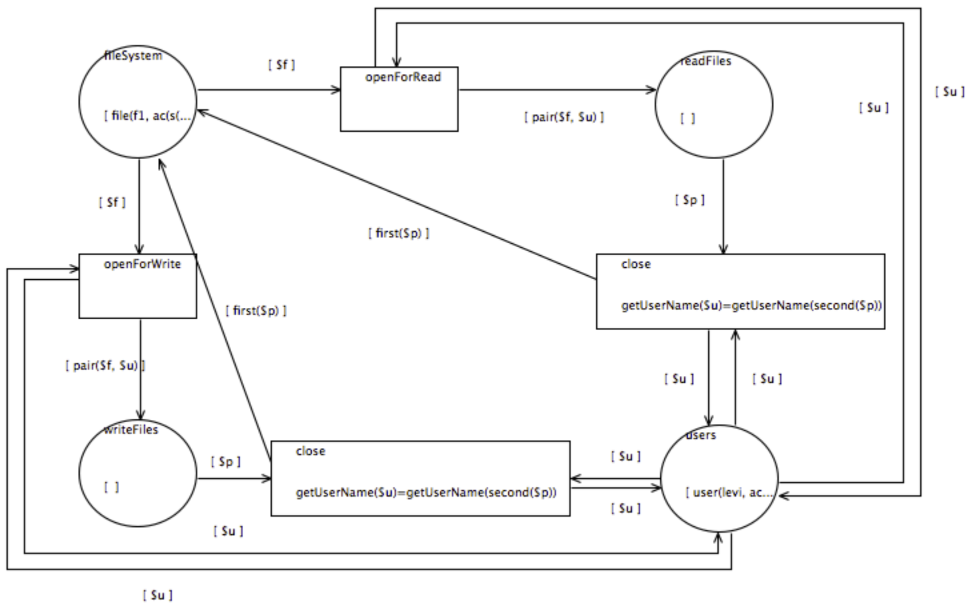

Figure 1: Naive Filesystem

Because we have a first formalization of the filesystem, we can now precisely express the *confidentiality* property we have detailed informally in section 3.1. *Confidentiality* is a *composed property* (see definition 2.2) which can be expressed as three component properties about the APN filesystem model in figure 1. All three properties are safety properties, thus stating what should happen for all states belonging to the models' semantics:

$$\forall p \in writeFiles\ .\ userHasPermissionForFile(p) = true \tag{1}$$

$$\forall p \in readFiles\ .\ userHasPermissionForFile(p) = true \tag{2}$$

$$\begin{aligned} &\forall p_r \in readFiles\ .\ (\forall p_w \in writeFiles\ .\\ &getUserName(proj_2(p_r)) = getUserName(proj_2(p_w)) => \\ &(getAccessClass(proj_1(p_r))\ dominates\ getAccessClass(proj_1(p_w)) = false \end{aligned} \tag{3}$$

- Property 1, which we will call *respectWritePerms*, states that a file in the *writeFiles* place has been opened by a user who is sufficiently trusted. This decision is computed by the *userHasPermissionForFile* predicate which compares the *access class* of the file with the *access class* of the user who opened it;

Table 1: Property Satisfaction for the Naive Filesystem

(1) respectWritePerms	(2) respectReadPerms	(3) noReadWriteFlow
false	false	false

Table 2: Property Satisfaction for the Simple Security Filesystem

(1) respectWritePerms	(2) respectReadPerms	(3) noReadWriteFlow
true	true	false

- Property 2, which we will call *respectReadPerms*, is similar to property 1 but for the files in the *readFiles* place;
- Property 3, which we will call *noReadWriteFlow*, states that if a user has simultaneously a file open in read mode and another in write mode, then the one open in read mode does not *dominate* the one open in write mode.

In table 1 it is possible to observe that the naive filesystem in figure 1 does not respect any of the confidentiality component properties we have defined. The component properties were automatically validated by the AlPiNA model checker.

3.2.2 Simple Security

The second entity which we will call *simple security filesystem* is an evolution of the *naive filesystem* in figure 1 where the transitions *openForRead* and *openForWrite* become guarded. In particular, we add to each of those transitions the following guard:

$$dominates(getAccessClass(u), getAccessClass(f)) = true$$

$$\text{where } u \in (userName \times accessClass) \text{ and } f \in (fileName \times accessClass)$$

This means that in the *simple security* model in figure 2 we are enforcing that, in order for a user to read or write a file, that user has to be sufficiently trusted regarding the access class of the requested file. This is achieved using the *dominates* predicate (see appendix A for the predicate's semantics).

In table 2 the satisfaction of the confidentiality properties by the *simple security* model is shown. We can observe that the introduction of the two guards has rendered confidentiality properties 1 (respectWritePerms) and 2 (respectReadPerms) satisfied.

3.2.3 Confinement

Finally, we introduce the third entity which we will call *confined filesystem.* The *confined filesystem* model can be observed in figure 3 and is an evolution of the *simple security filesystem* in figure 2. Two places *writeFilesLog* and *readFilesLog*

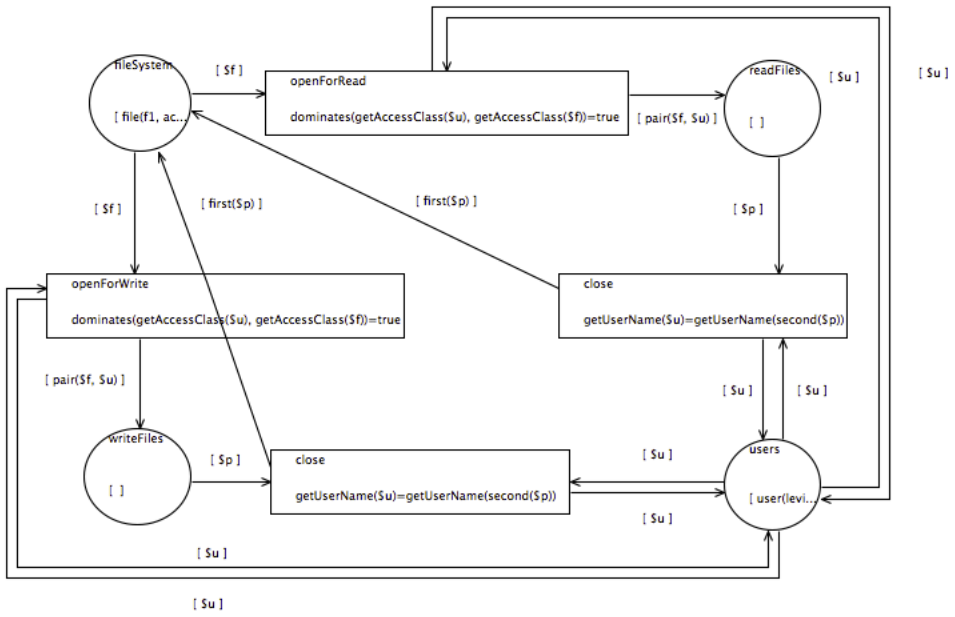

Figure 2: Simple Security Filesystem

have been added to the APN. The function of these places is to hold two lists of pairs $(fileName \times accessClass) \times (userName \times accessClass)$ which log the files which are open for reading and for writing. When a file is open for reading, a check is done on the *writeFilesLog* place to decide if there are no files open for writing by the same user which are less confidential than the file being opened. This is achieved by adding a condition to the guard of the *openForRead* transition (hidden for lack of space in figure 3) as follows:

$$dominates(minAccess(lpw, u), getAccessClass(f)) = true$$

where *minAccess* function returns the minimum access class value of the set of files currently open for writing by user u.

The *dominates* predicate is again used to decide if the confidentiality of the file f being opened by a user u for reading is lower than the minimum confidentiality of the files opened for writing by user u. The reverse principle is applied to opening files for writing.

If we again model check the confidentiality properties on the new model in figure 3 we find that the *noReadWriteFlow* property is now satisfied – as can be observed in table 3. This means that attacks by trojan horses on this new model of a filesystem are made difficult and thus the confidentiality property is made stronger. Note also that properties *respectWritePerms* and *respectReadPerms* remain true when evolving from the *simple security filesystem* entity to the *confined filesystem* entity.

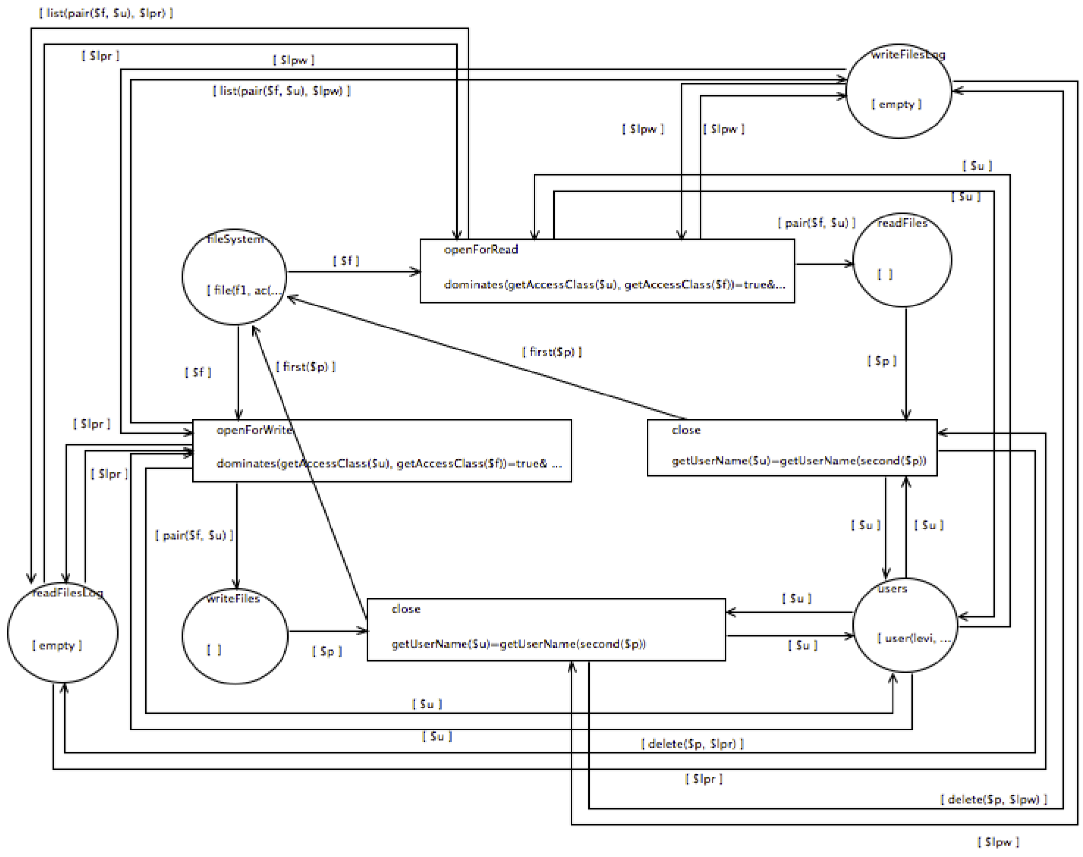

Figure 3: Confined Filesystem

Table 3: Property Satisfaction for the Simple Security Filesystem

(1) respectWritePerms	(2) respectReadPerms	(3) noReadWriteFlow
true	true	true

Table 4: Property Satisfaction for the Simple Security Filesystem

	(1) respectWritePerms	(2) respectReadPerms	(3) noReadWriteFlow
naive	false	false	false
simple security	true	true	false
confined	true	true	true

Finally, in table 4 we present the satisfaction of the confidentiality subproperties for the three entities of the multi-level security filesystem. We can observe that the *naive filesystem* satisfies none of the component properties of confidentiality, the *simple security filesystem* satisfies two of the component properties of confidentiality and the *confined filesystem* satisfies all of the component properties of confidentiality. Because the satisfaction of the *confinement* composed property always increases along the *discrete evolutions* of the confidential filesystem we can claim according to definition 2.4 that the confidential filesystem evolving system presents *monotonic resilience* regarding the confidentiality property.

4 Evolution Conditions

In definition 2.4 we have introduced the notion of monotonic resilience meaning that given an evolving system, represented by a sequence of entities and a composed property C, any entity in the sequence always satisfies at least the same component properties of C as the previous entity in the sequence.

The intuitive approach to checking if a component property is kept during a discrete evolution is to recheck it on the new entity. However if we would admit arbitrary modifications to an APN during a *discrete evolution* (definition 2.1) three outcomes of that discrete evolution would be possible:

1. the discrete evolution would be such that the component properties satisfied by the entity before the evolution would be satisfied by the entity after the evolution;
2. the discrete evolution would be such that some or all of the component properties satisfied by the entity before the evolution would be not be satisfied by the entity after the evolution;
3. the discrete evolution would be such that some or all of the component properties satisfied by the entity before the evolution could not be verified in the entity after the evolution. This would happen in the case where there would be no reasonable mapping between structural parts of the entities in that discrete evolution which are referred to by component properties.

In the context monotonic resilience (definition 2.4) and of the choices we have made in section 3 for *modelling formalism*, *property language* and *satisfaction function*, we are interested in studying under which conditions a discrete evolution can happen between two entities expressed as Algebraic Petri Nets such that points 2 and 3 in the list above are avoided. Such conditions can provide the guidelines for evolution such that monotonic resilience regarding safety properties can be guaranteed in evolving systems expressed as a sequence of APN models.

In [14, 15] we can find a theory for the refinement of Algebraic Petri Nets preserving safety properties. The theory states that if there is a *place preserving morphism* between two APN models then safety properties are kept. Intuitively, a *place preserving morphism* maps the structure of an APN model N_1 into the structure of another APN model N_2 such that:

- each place of N_1 is mapped onto a separate place of N_2 and each transition of N_1 is mapped onto a separate transition of N_2 – formally, the place and transition mapping functions are injective;
- Mapped transitions' guards are at least preserved but can be strengthened[8] when transitions of N_1 are mapped onto N_2;
- Arcs adjacent to places of N_1 are preserved in the corresponding places in N_2 and no new arcs are added;
- There can be more places as input or output of a mapped transition in N_2 than in the corresponding original transition in N_1;
- The mapping of the algebraic specifications used by N_1 are included in those used by N_2.

In practice the conditions above force a structural inclusion of N_1 into N_2 such that the traffic of tokens in the N_1 subnet of N_2 is at most constrained regarding N_1, but no new behaviors are added. Coming back to the three entities we have proposed in section 3.2, we can identify two *discrete evolutions*: *(naive filesystem, simple security filesystem)* and *(simple security filesystem, confidential filesystem)*. We have presented in section 3.2 experimental evidence that the *confidential filesystem evolving system* exhibits *monotonic resilience* regarding the *confidentiality* composed property – formalized as safety properties *respectReadPermissions*, *respectWritePermissions* and *noReadWriteFlow* in section 3.2. We will now show that the the *discrete evolutions* of the evolving system presented in section 3.2 respect the conditions of the Algebraic Petri Net refinement theory described above.

Regarding the *(naive filesystem, simple security filesystem)* discrete evolution (figures 1 and 2), nothing has to be shown because the *naive filesystem* does not satisfy any of the component properties of the *confidentiality property* and thus the evolution in unconstrained.

[8]The strengthening of transition guards is not part of the original theory but an extension provided by us. The formal definition and justification of guard strengthening regarding the original theory can be found in appendix B.

Regarding the *(simple security filesystem, confidential filesystem)* discrete evolution (figures 2 and 3), properties *respectReadPermissions* and *respectWrite-Permissions* are kept. If we apply the APN refinement theory then a morphism mapping places and transitions from the *simple security filesystem* into the $confidentialfilesystem$ has to be found that satisfies the conditions given above. Such a morphism maps places of the entity *simple security filesystem* $filesystem$, $readfiles$, $writefiles$ and $users$ into the places of the same name in entity *confidential filesystem*. It also maps transitions of the entity *simple security filesystem* $openForRead$, $openForWrite$, $close$ and $close'$ into the transitions of the same name in entity *confidential filesystem*.

The identified morphism respects the conditions above as: 1) firing conditions are strengthened in transitions $openForRead$ and $openForWrite$; 2) arcs adjacent to mapped places are not touched; 3) both the place and the transition morphisms are injective.

In appendix B we present a complete and rigorous description of the *place preserving* morphism mapping the *simple security filesystem* APN into the *confined filesystem*. Also in appendix B we formally introduce the *transition condition strengthening* extension to the theory in [14, 15] which is necessary to our approach.

5 Discussion and Threats to Validity

In this paper we have introduced an operational definition of *resilience* based on using the modelling concepts *model* and *decidable property* represent the notions of *evolving system*, the *property* regarding which resilience is measured. In particular and in order to be tool supported, our approach relies on Algebraic Petri Nets for modelling and an associated safety property model checker to compute the satisfaction of safety properties.

Our proposal for an operational definition of resilience raises many questions. We will start by discussing the questions raised by the choice of Algebraic Petri Nets as modelling formalism and then will go on to discuss more general questions independent of the choice of the modelling formalism.

One might ask if the evolution conditions for APN models we have introduced in section 4 are too constraining. In fact, given a *discrete evolution* (ent_k, ent_{k+1}), we impose a particular kind of structural inclusion of ent_k into ent_{k+1}. This structural inclusion ensures that the behavior of ent_{k+1} regarding safety properties expressed over ent_k is included in the behavior of ent_k. If on the one hand this constraint is very taxing on the kind of discrete evolutions that can happen, on the other hand it guarantees that all safety properties verified by ent_k can still be expressed over and verified by ent_{k+1}. If one would like to relax the evolution conditions a possible path would be to take into account the safety properties verified in ent_k when evolving, which might lead to weaker conditions than the ones imposed by the total *place preserving* morphism we have introduced in section 4. There is work from the model checking community on alleviating state space explosion by simplifying specifications using the variables in the properties under check. For example in [4] the authors describe the *cone of reduction* technique used in hardware verification where a specification P is reduced in such a way that the reduced specification – thus with a less expensive state space – P' satisfies a formula ϕ if and only of P satisfies ϕ. More directly related to our research, in [16] a technique is presented for slicing Place/Transition Petri Nets with the goal of easing the verification of LTL formulas. Both these pointers could be used in our research for finding subnets of an APN entity ent_k which satisfy a set of safety properties such that the remaining part of ent_k could be discarded or modified in a discrete evolution – thus relaxing the *place preserving* total morphism constraint.

Another important question regards the usage of *safety properties* as a means of representing properties regarding which resilience can be measured. Other types of properties could be envisaged such as *liveness* or other properties expressed in temporal logics. The preservation of such properties in a discrete evolution would however impose other evolution conditions which could be inspired by [4, 16] as mentioned above.

One might also ask if the decomposition of a property regarding which resilience can be measured into component safety properties is feasible and useful in general. This is a question that can only be answered experimentally.

A more general issue in our approach regards the fact that we are considering *monotonic evolution*. It could be interesting to relax this assumption such that in a discrete evolution (ent_k, ent_{k+1}) the constraint that the number of component properties satisfied by ent_{k+1} can be less than the number of properties satisfied by ent_k – in other words component properties might be dropped within a discrete evolution. Given the evolution conditions are properly studied, this might allow for a larger margin in the fashion an *evolving system* evolves.

Finally, the research we present in this paper is oriented at: concretely defining the *resilience* concept; providing operational means to analyze an evolving system and deciding automatically whether that system is resilient regarding a certain property; provide a set of conditions under which an entity from an evolving system can change such that degradation regarding a certain property does not occur. We have not addressed in this work how to build *discrete evolutions* (ent_k, ent_{k+1}) such that ent_{k+1} satisfies more component properties of a property of interest than ent_k. This is done purposefully as we believe strategies for evolution of a system are domain dependent and thus cannot be given generically in a theory. If however such an evolution strategy is given for a particular domain, it could be possible to expand the concept of *evolving system* from a sequence to a graph of *entities* formed by building the possible evolution paths – given an initial model and an evolution strategy.

6 Related Work

As we have mentioned in section 1, there is a multitude of definitions and usages of the word *resilience* in the information and communication science literature.

Because of this multitude of meanings we provide a relatively broad section on related work. We consider this is important as our definition of resilience is inspired from and tries to concretize existing definitions and usages of the term.

Concerning the definition of resilience coming from the dependability community and the adaptive systems community in information and communication science, many representative researchers [7, 11, 2, 18] share the same definition. From their perspective, resilience is initially defined as "the persistence of service delivery that can justifiably be trusted, when facing changes" and mainly regarded as equivalent to fault-tolerance. However, its use is considered to put emphasis on a notion of "unforeseen events" and to include the effects of evolution through the "change" concept. While no definition, formal or informal, of resilience is proposed, the authors' intuition on resilience show that the *DREF* formal conceptual framework could be used to provide a coherent and more precise definition of resilience from their viewpoint.

Nevertheless, we should remark that only focusing on persistence of service delivery is too restrictive to characterize the concept of resilience. In *DREF* it would mean that the satisfiability function should be always over the nominal satisfiability, never decreasing and, furthermore, using a binary quantification (only correct service delivery or incorrect). In our approach, there is a fundamental difference that forbids us to consider that resilience is a synonym of fault-tolerance, it is the explicit consideration of an evolution axis for satisfiability. Thus if we want to relate to fault-tolerance, resilience would be characterized by the evolution of fault-tolerance capabilities over an evolution axis.

We can also find many studies in which the integration of dependability and evolution concepts are introduced at modelling or meta-modelling level. This is done at a different level of formalization, ranging from informal (natural languages) to formal (mathematical languages). In [13], the authors propose to explicitly add at requirements level non functional properties directly related at abstract level to dependability. Metrics for quantification of these attributes are also informally introduced and can be related to our approach. Nevertheless, the attributes used to characterize resilience are different from the ones in our approach since resilience is defined as availability, reliability and assurance.

In [7] the need for a model based approach to resilience is explicitly foreseen: "to deal with the challenges of adaptation we envisage a model-driven development where models play a key role throughout the development (...)". In fact, models can support estimation of system status, so that the impact of a context change can be predicted. Provided that such predictions are reliable, it should be possible to perform model-based evolution analysis as a verification activity. The work we present in this paper goes in the direction of proving that the hypothesis stated in [7] has an operational counterpart.

7 Conclusions and Future Work

This paper constitutes an attempt at providing a concrete definition for the *resilience* concept. In order to do this we have: considered the abstract definition of resilience in *DREF*; formalized it with the purpose of operationalizing the notion of resilience; provided an instance of an evolving filesystem in Algebraic Petri Nets and showed it is possible to use model checking tools to automatically prove its resilience regarding the *confidentiality* property. We have then formally defined the conditions such that no loss of resilience regarding a set of safety properties occurs in an *discrete evolution* modelled using APN. These conditions provide the basis for a study on how to formally define evolution strategies in this particular formal setting.

The study presented in this paper opens many questions. Given the current state of the art we find that proposing concrete research questions on the semantics of the *resilience* concept is a necessary step. As was explained in section 5, the future work ranges from exploring more flexible conditions than the ones presented in section 4 for defining *discrete evolutions* to expanding the approach to incorporate evolution strategies. As an immediate future work we are planning on instantiating the theory with a real world use case in order to evaluate the practical applicability of the several theoretical and operational layers we present in this paper.

References

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

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[3] D. E. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, The MITRE Corporation, 1973.

[4] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *COMPOS*, pages 81–102, 1997.

[5] P. E. Black and P. J. Windley. Verifying resilient software. pages 262–266, 1997.

[6] D. Buchs, S. Hostettler, A. Marechal, and M. Risoldi. Alpina: A symbolic model checker. In *Petri Nets*, volume 6128 of *Lecture Notes in Computer Science.* Springer, 2010.

[7] B. Cheng and al. Software engineering for self-adaptive systems: A research roadmap. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[8] M. Cristiá, G. Giusti, and F. Manzano. The implementation of lisex, a mls linux prototype. In *Proceedings of ASSE (Argentinian Symposium of Software Engineering)*, 2005.

[9] P. A. Dearnley. An investigation into database resilience. *Comput. J.*, 19(2):117–121, 1976.

[10] N. Guelfi. Technical Report TR-LASSY-10-01, Laboratory for Advanced Software Systems, University of Luxembourg, 2010. Available on request - mail to levi.lucio@uni.lu.

[11] J.-C. Laprie. From dependability to resilience. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, DSN - Fast Abstracts*. IEEE/IFIP, 2008.

[12] L. Lúcio. Model of an evolving confidential filesystem, 2011. `http://hera.uni.lu/~levi.lucio/operational_resilience/evolving_filesystem.zip`.

[13] D. N. J. Mostert and S. H. von Solms. A technique to include computer security, safety, and resilience requirements as part of the requirements specification. *J. Syst. Softw.*, 31(1):45–53, 1995.

[14] 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.

[15] J. Padberg, M. Gajewsky, and C. Ermel. Rule-based refinement of high-level nets preserving safety properties. In *Fundamental Approaches to Software Engineering*, pages 221–238. Springer Verlag, 1998.

[16] A. Rakow. Slicing petri nets. Technical report, University of Oldenburg, Germany, 2007.

[17] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[18] L. Simoncini. Resilient computing: An engineering discipline. *Parallel and Distributed Processing Symposium, International*, 0:1, 2009.

[19] Smv Group. Alpina model checker, 2010. `http://alpina.unige.ch`.

[20] L. Svobodova. Resilient distributed computing. *IEEE Trans. Software Eng.*, 10(3):257–268, 1984.

A Algebraic Specifications for the Secure Filesystem Example

In this appendix we present the complete specifications for the three entities we have introduced informally in section 3.

A.1 Naive Filesystem Specification

```
Adt FileName

  Sorts fileName;

  Generators
    f1 : fileName;
    f2 : fileName;
    f3 : fileName;
```

```
Adt UserName

  Sorts userName;

  Generators
    levi : userName;
    nicolas : userName;
    ayda : userName;
```

```
import "fileName.adt"
import "accessClass.adt"

Adt File

  Sorts file;

  Generators
    file : fileName, accessClass -> file;

  Operations
    getFileName : file -> fileName;
    getAccessClass : file -> accessClass;

  Axioms
    getFileName(file($fn,$acl)) = $fn;
    getAccessClass(file($fn,$acl)) = $acl;

  Variables
    fn : fileName;
    acl : accessClass;
```

```
import "userName.adt"
import "accessClass.adt"

Adt User

  Sorts user;

  Generators
    user : userName, accessClass -> user;

  Operations
    getUserName : user -> userName;
    getAccessClass : user -> accessClass;
  Axioms
    getUserName(user($un,$acl)) = $un;
    getAccessClass(user($un,$acl)) = $acl;

  Variables
    un : userName;
    acl : accessClass;
```

```
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];
```

```
import "boolean.adt"

Adt securityLevel

  Sorts securityLevel;

  Generators
    UNCLASSIFIED : securityLevel;
    s : securityLevel -> securityLevel;

  Operations
    le : securityLevel, securityLevel -> bool;

  Axioms
    UNCLASSIFIED le UNCLASSIFIED = true;
    UNCLASSIFIED le s($x) = true;
    s($x) le UNCLASSIFIED = false;
    s($x) le s($y) = $x le $y;

  Variables
    x : securityLevel;
    y : securityLevel;
```

```
import "boolean.adt"
import "categorySet.adt"
import "securityLevel.adt"
import "list.gadt"
import "category.adt"

Adt accessClass is list[category]

  Sorts accessClass;

  Generators
    top : accessClass;
    bottom : accessClass;
    ac : securityLevel, list[category] -> accessClass;

  Operations
    dominates : accessClass, accessClass -> bool;

  Axioms
    ac($s1,$c1) dominates bottom = true;
    top dominates ac($s1,$c1)  = true;
    top dominates bottom = true;

    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 : list[category];
    c2 : list[category];
```

```
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;
```

A.2 Simple Security Filesystem Specification

The algebraic specification is the same as the one in section A.1.

A.3 Confinement Filesystem Specification

The algebraic specification is the same as the one in section A.1, incremented by the following definitions:

```
import "file.adt"
import "user.adt"
import "accessClass.adt"
import "pair.gadt"
import "list.gadt"
import "fileUserPair"

Adt listOfFiles Is list[pair[file,user]]

  Operations
    maxAccess : list[pair[file,user]] -> accessClass;
    minAccess : list[pair[file,user]] -> accessClass;

    calcMaxAccess : accessClass, list[pair[file,user]] -> accessClass;
    calcMinAccess : accessClass, list[pair[file,user]] -> accessClass;

  Axioms

    maxAccess($l) = calcMaxAccess(bottom,$l);

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

    minAccess($l) = calcMinAccess(top,$l);

    calcMinAccess($acl,empty) = $acl;
    if (getAccessClass(first($p)) dominates $acl) = true then
        calcMinAccess($acl,list($p,$l)) = calcMinAccess($acl,$l);
    if ($acl dominates getAccessClass(first($p))) = true then
        calcMinAccess($acl,list($p,$l)) = calcMinAccess(getAccessClass(
        first($p)),$l);

  Variables
    l : list[pair[file,user]];
    p : pair[file,user];
    acl : accessClass;
```

B Preservation of Safety Properties Algebraic High Level Nets

The main theorem in [14, 15] states that *place preserving* Algebraic High-Level (AHL) Nets morphisms preserve *safety properties*. Algebraic High-Level Nets are equivalent to the Algebraic Petri Nets formalism we are using in the research presented in this report. Place preserving morphisms are a particular class of AHL net morphisms mapping algebraic specifications, places, transitions and algebras. We are interested in such place preserving morphisms as they guarantee we can evolve our models while preserving previously satisfied safety properties. The theory presented in this appendix extends the theory presented in [14, 15] in order to allow transition guard strengthening in a safety property place preserving morphism. Definitions B.1, B.3 and theorem B.5 are lighter versions of the theory presented in [14, 15]. Definitions B.2, B.4, proposition B.6 and lemma B.7 have been introduced by us.

Definition B.1 *Algebraic High-Level (AHL)*
An Algebraic High-Level net is a 7-tuple $\langle SPEC, P, T, pre, post, cond, A\rangle$ *where* $SPEC$ *is an algebraic specification,* P *is a set of places,* T *a set of transitions,* pre *and* $post$ *functions assigning term weighted input and output arcs to transitions,* $cond$ *a function assigning a set of equational conditions to transitions and* A *an algebra which is model of* $SPEC$*.*

Definition B.2 *Guard Strengthened Algebraic High-Level (AHL)*
Let $N = \langle SPEC, P, T, pre, post, cond, A\rangle$ *be an AHL.* $N' = \langle SPEC, P, T, pre, post, cond', A\rangle$ *is a* guard strengthened *version of* N *if for all transitions* $t \in T$ *the set of equations* $cond(t)$ *is included in the set of equations* $cond'(t)$*.*

Definition B.3 *Place Preserving Algebraic High-Level Net Morphism*
Let $N_1 = \langle SPEC_1, P_1, T_1, pre_1, post_1, cond_1, A_1\rangle$ *and* $N_2 = \langle SPEC_2, P_2, T_2, pre_2, post_2, cond_2, A_2\rangle$ *be two AHL Nets.* $f = (f_P, f_T, f_{SPEC}, f_A) : N_1 \to N_2$ *is a* Place Preserving *AHL Net Morphism where* $f_P : P_1 \to P_2$*,* $f_T : T_1 \to T_2$*,* $f_{SPEC} : SPEC_1 \to SPEC_2$ *and* $f_A : A_1 \to A_2$ *are morphisms iff the following is true:*

- *Firing conditions are preserved when transitions of* T_1 *are mapped onto* T_2*;*
- *Arcs adjacent to places of* P_1 *are preserved when those places are mapped onto the places of* P_2 *by* f_p*;*
- f_T*,* f_P *and* f_{SPEC} *are injective and* f_{SPEC} *is persistent, meaning the mapped signatures, terms and equations of* $SPEC_1$ *by* f_{SPEC} *are contained in* $SPEC_2$*;*
- *There can be more places in the pre or post domain of a mapped transition than in the corresponding domains of the original transition;*

- *A_2 merely extends the mapping of A_1 by f_A for the new parts of A_2 or it is merely renamed.*

Definition B.4 *Place Preserving Guard Strengthening Algebraic High-Level Net Morphism*

Let N_1 and N_2 be two AHL Nets. Let also $f : N_1 \rightarrow N_2$ be a Place Preserving *AHL Net Morphism. f is* guard strengthening *if there is a an AHL net N_3 such that $f : N_1 \rightarrow N_3$ is a place preserving morphism and N_2 is a guard strengthened version of N_3.*

Theorem B.5 *Preservation of Safety Formulas*

Let $f : N1 \rightarrow N2$ be a Place Preserving *AHL morphism and M_1 and M_2 be markings of N_1 and N_2 respectively, with $M_{2|f} = M_1$ – meaning the restriction of the marking M_2 to $f(M_1)$. Let ϕ be a formula representing a marking of the AHL network or formulas built by the conjunction or negation of such formulas. Then there is the following equivalence:*

$$M_1 \models_{N_1} \Box\phi \Leftrightarrow M_2 \models_{N_2} \tau_f(\Box\phi)$$

where $M \models_N \Box\phi$ means formula ϕ is satisfied in all markings attainable from marking M by firing all transitions of AHL N. If we consider only one marking M of N, satisfaction of ϕ in M means the marking expressed in ϕ is contained in M. Finally, τ_f is the function translating formulas regarding the place preserving *morphism f.*

Proposition B.6 *Preservation of Safety Formulas under Guard Strengthening*

Let $f : N_1 \rightarrow N_2$ be a place preserving morphism. We know by B.5 that $M_1 \models_{N_1} \Box\phi \Leftrightarrow M_2 \models_{N_2} \tau_f(\Box\phi)$. We thus know that for all markings obtained from M_2 by firing transitions of N_2 (noted $[M2\rangle_{N_2}$) formula $\tau(\phi)$ is satisfied. If the guards of the transitions of N_2 are strengthened then by lemma B.7 the number of markings of the set $[M2\rangle_{N_2}$ is also reduced. Since all the markings on $[M2\rangle_{N_2}$ satisfy $\tau(\phi)$, so do all the markings of any of the subsets of $[M2\rangle_{N_2}$.

Lemma B.7 *Marking Inclusion under Guard Strengthening*

Let N and N' be two AHL nets where N' which is a guard strengthened version of N and M be a marking common to the two nets. We will prove by induction on the firings of N' that the reachable markings of an APN N' from marking M are a subset of the reachable markings of N, i.e. $[M\rangle_{N'} \subseteq [M\rangle_N$. The base case is when we have the initial marking M and any enabled trasition of N' fires. The set of states resulting from these firings, noted $M\rangle_{N'}$, is necessarily included in $M\rangle_N$ because: either a transition t of N has not been strenghtened in N' and the markings resulting from firing t on M are the same for both nets; or a transition t of N has been strengthened in N' in which case $M\rangle_{N'} \subseteq M\rangle_{N'}$. For the induction step we assume that we have a marking of N' which is contained in $[M\rangle_N$. By the same principle of the base case the set of resulting markings from firing the enabled transitions will be contained in $[M\rangle_N$.

We will now refer to the evolution example we have provided in section 3 and validate that the morphisms we use are place preserving, and as such preserve safety properties.

If we take into consideration the discrete evolution *(naive filesystem,simple security filesystem)* (figures 1 and 2), there is no safety property to preserve. In fact, from the three component properties we have presented in section 3.2 that form *confidentiality*, none is verified in by the *naive filesystem* entity as is shown in table 1. However, properties *respectReadPermissions* and *respectWritePermissions* in section 3.2 are experimentally verified by the *simple security filesystem* as can be observed in table 2. We would thus like to prove that properties *respectReadPermissions* and *respectWritePermissions* are indeed preserved by the *confined filesystem* entity in figure 3 in order to confirm the truthfulness of table 4. To do so we will formally define the *simple security filesystem* and *confined filesystem* entities and show that the morphism that defines the discrete evolution from the former to the latter is *place preserving*, and in particular *place preserving guard strengthening.*

By definition B.1 the *simple security filesystem* which we will call ssf is a 7-tuple such that $ssf = \langle SPEC_{ssf}, P_{ssf}, T_{ssf}, pre_{ssf}, post_{ssf}, cond_{ssf}, A_{ssf}\rangle$, where:

- $SPEC_{ssf}$ is given in section A.2
- $P_{ssf} = \big\{fileSystem, readFiles, writeFiles, users\big\}$
- $T_{ssf} = \big\{openForRead, openForWrite, close, close'\big\}$
- $pre_{ssf} = \big\{(openForRead, \{(\$f, fileSystem), (\$u, users)\}),$ $(openForWrite, \{(\$f, fileSystem), (\$u, users)\}),$ $(close, \{(\$p, readFiles), (\$u, users)\}),$ $(close', \{(\$p, writeFiles\}), (\$u, users))\big\}$
- $post_{ssf} = \big\{(openForRead, \{(pair(\$f, \$u), readfiles), (\$u, users)\}),$ $(openForWrite, \{pair(\$f, \$u), readfiles), (\$u, users)\}),$ $(close, \{(first(\$p), fileSystem), (\$u, users)\}),$ $(close', \{(first(\$p), fileSystem\}), (\$u, users))\big\}$
- $cond_{ssf} = \big\{(openForRead, \{dominates(getAccessClass(\$u),$ $getAccessClass(\$f)) = true\}),$ $(openForWrite, \{dominates(getAccessClass(\$u), getAccessClass(\$f)) =$ $true\}),$ $(close, \{getUserName(\$u) = getUserName(second(\$f)\}),$ $(close', \{getUserName(\$u) = getUserName(second(\$f)\})\big\}$
- $A_{ssf} = \textit{the initial algebra of } SPEC_{ssf}$

The *confined filesystem* which we will call cf is a 7-tuple such that $cf = \langle SPEC_{cf}, P_{cf}, T_{cf}, pre_{cf}, post_{cf}, cond_{cf}, A_{cf}\rangle$, where:

- $SPEC_{cf}$ is given in appendix A.3

- $P_{cf} = \{fileSystem, readFiles, writeFiles, users, readFilesLog, writeFilesLog\}$
- $T_{cf} = \{openForRead, openForWrite, close, close'\}$
- $pre_{cf} = \{(openForRead, \{(\$f, fileSystem), (\$u, users), (\$lpw, writeFilesLog)\}),$ $(openForWrite, \{(\$f, fileSystem), (\$u, users), (\$lpr, readFilesLog)\}),$ $(close, \{(\$p, readFiles), (\$u, users), (\$lpr, readFilesLog)\}),$ $(close', \{(\$p, writeFiles\}), (\$u, users), (\$lpw, writeFilesLog))\}$
- $post_{cf} = \{(openForRead, \{(pair(\$f, \$u), readfiles), (\$u, users),$ $(list(pair(\$f, \$u), \$lpw), writeFilesLog)\}),$ $(openForWrite, \{pair(\$f, \$u), writefiles), (\$u, users),$ $(list(pair(\$f, \$u), \$lpr), readFilesLog)\}),$ $(close, \{(first(\$p), fileSystem), (\$u, users), (delete(\mathrm{p}, lpr), readFilesLog)\}),$ $(close', \{(first(\$p), fileSystem\}), (\$u, users), (delete(\mathrm{p}, lpw), writeFilesLog))\}$
- $cond_{cf} = \{(openForRead, \{dominates(getAccessClass(\$u),$ $getAccessClass(\$f)) = true\}),$ $(openForWrite, \{dominates(getAccessClass(\$u), getAccessClass(\$f)) =$ $true\}),$ $(close, \{getUserName(\$u) = getUserName(second(\$f)\}),$ $(close', \{getUserName(\$u) = getUserName(second(\$f)\})\}$
- $A_{cf} =$ *the initial algebra of* $SPEC_{cf}$

We can now build the morphism $f = \langle f_p, f_t, f_{SPEC}, f_A\rangle$ between $ssf$and cf where:

- $f_p = \{(fileSystem, fileSystem), (readFiles, readFiles), (writeFiles,$ $writeFiles), (users, users)\}$
- $f_t = \{(openForRead, openForRead), (openForWrite, openForWrite),$ $(close, close), (close', close')\}$
- f_{SPEC} = morphism mapping signatures, terms and equations of $SPEC_{ssf}$ into the (syntactically) corresponding signatures, terms and equations of $SPEC_{cf}$
- f_A = morphism such that algebra A_{cf} extends $f_A(A_{ssf})$

According to definition B.3 morphism f is *guard strengthening place preserving* and as such the discrete evolution *(simple security filesystem, confined filesystem)* is *property-preserving* (see definition 2.3).

As a remark, note that we have not formally proved any equivalence between the safety property language used for AHL in [14, 15] and that used in AlPiNA. For the concrete properties we have presented in this paper in section 3 this is not necessary as all the used constructs – markings, algebraic operations, variables, logical operators and universal quantifiers – are described or implicitly used in [15].