Model checking grid security

F. Pagliarecci*, L. Spalazzi, F. Spegni
Università Politecnica delle Marche – Ancona, Italy

A R T I C L E   I N F O

Article history:
Received 12 April 2011
Received in revised form 22 November 2011
Accepted 23 November 2011
Available online 9 December 2011

A B S T R A C T

Grid computing is one of the leading forms of high performance computing. Security in the grid environment is a challenging issue that can be characterized as a complex system involving many subtleties that may lead designers into error. This is similar to what happens with security protocols where automatic verification techniques (specially model checking) have been proved to be very useful at design time. This paper proposes a formal verification methodology based on model checking that can be applied to host security verification for grid systems. The proposed methodology must take into account that a grid system can be described as a parameterized model, and security requirements can be described as hyperproperties. Unfortunately, both parameterized model checking and hyperproperty verification are, in general, undecidable. However, it has been proved that this problem becomes decidable when jobs have some regularities in their organization. Therefore, this paper presents a verification methodology that reduces a given grid system model to a model to which it is possible to apply a “cutoff” theorem (i.e., a requirement is satisfied by a system with an arbitrary number of jobs if and only if it is satisfied by a system with a finite number of jobs up to a cutoff size). This methodology is supported by a set of theorems, whose proofs are presented in this paper. The methodology is explained by means of a case study: the Condor system.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Grid computing is one of the leading forms of high performance computing. It enables inter-operability between distributed and heterogeneous resources in a standardized manner. Its architecture is defined in terms of a layered collection of protocols [1] (Fig. 1 reports the related architecture that is often referred to as the hourglass model). Security in a grid environment is a challenging issue because this environment instantiates interactions among a set of possibly unknown entities. Therefore, security in grid computing can be characterized as a complex system involving many subtleties that may lead designers into error [2]. This is similar to what happens with security protocols [3] where, on the other hand, automatic verification techniques (specially model checking) have been proved to be very useful at design time [4].

This paper proposes a formal verification method based on model checking that can be applied to host security verification. More in depth, host security, as part of infrastructure related issues at Fabric Layer of Fig. 1 [5,6], deals with issues that make a host apprehensive about affiliating itself into the grid system, namely: job protection and data protection. In other words, whenever a host is affiliated to the grid, host security consists in protecting the already existing jobs, resources, and data in the host. Indeed, the host submitting the job may be untrusted or unknown to the host running the job. This means that the guest job may contain a malicious or defective code that violates security requirements of already existing jobs and data: the well known CIA Triad – i.e., confidentiality, integrity, availability – as well as authentication, authorization, and non-repudiation (e.g., the reader can refer to the glossary proposed by the Committee on National Security Systems [7]). Nevertheless, this domain has some peculiarities that makes it different from model checking security requirements of security protocols. As a matter of fact, security protocols can be described by means of a finite state model whereas a grid computing system is composed of an arbitrary number of hosts each of them running an arbitrary number of jobs that work on an arbitrary number of data and, thus, it can be described by means of an infinite-state parameterized model. Unfortunately, parameterized model checking is known to be undecidable in general [8]. However, it has been proved that this problem becomes decidable when systems have some regularities in their organization [9,10].

The approach presented in this paper relies on two key ideas: the first one consists in using Emerson and Kahan’s labeled transition systems with disjunctive guards [10], whose verification has been proved to be decidable. The related theorem states that a
requirement is satisfied by a system with an arbitrary number of jobs if and only if it is satisfied by a system with a finite number of jobs up to a cutoff size. The second one consists in proposing a verification methodology that reduces a given grid system model to a model to which it is possible to apply a “cutoff” theorem and, thus, a model checking algorithm. This methodology is supported by a set of theorems, whose proofs are presented in this paper. The methodology is explained by means of a case study: the Condor system [11].

The paper is structured as follows. Section 2 reports the related work on the formal verification of security requirements, parameterized system model checking, and grid systems. For the sake of self-containment, Section 3 reports the theoretical backgrounds that are used in the rest of the paper. Some novel theoretical results are reported in Section 4, whereas the verification methodology built on such results is reported in Section 5. Section 6 provides the application of the methodology to the case study. Finally, Section 7 provides some concluding remarks.

2. Related work

Formal verification of security requirements. Alpern and Schneider [12] proposed a classification of program specifications in terms of trace properties (each property is a subset of the set of all traces). Furthermore, they proved that each trace property can be expressed as either safety (“nothing bad happens”), liveness (“something good eventually happens”), or the conjunction of safety and liveness. Unfortunately, several security requirements cannot be expressed in terms of trace properties [13]. For this reason, Clarkson and Schneider [13] proposed a new classification of program specifications based on the notion of hyperproperties (each hyperproperty is a set of sets of traces), and they proved that each hyperproperty is the intersection of a safety hyperproperty (SH) and a liveness hyperproperty (LH). According to their classification, confidentiality is a combination of SH and LH, integrity should be a safety property (even if not yet proved), availability is sometimes a property (e.g., max. response time) and sometimes a hyperproperty (e.g., the average response time). Furthermore, they suggested that “hyperproperties can be models for branching-time temporal predicates, whereas properties are limited to modeling linear-time temporal predicates” [13]. A temporal logic that joins together branching-time temporal predicates and linear-time temporal predicates is the computation tree logic (CCTL) [14], the temporal logic used in this paper. The part of hyperproperties that is covered by CCTL is still an open problem. Nevertheless, it has been proved [15,13] that k-safety (a subset of safety hyperproperties) can be reduced to a simple safety property by means of self-composition. Maier [16] proposed a different solution for properties. He proposed an intuitionistic semantics for LTL based on Heyting algebra instead of the classical Boolean algebra. According to Maier’s approach, safety and liveness properties receive an elegant characterization. The application of intuitionistic semantics to hyperproperties is still to be studied.

Formal verification of parameterized systems. There exists a vast literature on verifying parameterized systems. The fundamental work on this field states that the verification problem for such kind of systems is in general undecidable [8]. After that result, several authors faced the problem with (sound but incomplete) techniques based on the idea of finding appropriate abstractions and/or invariants [17–23]. The former consists in finding a (relatively small) finite-state abstraction that preserves the property to verify. The latter consists in finding an invariant that represents the common behavior exhibited by all instances of the parameterized system. A requirement is satisfied by the parameterized system when a given relation over the invariant, a generic process, and the property is satisfied. The main limitation of the above techniques is that the process of building an abstraction mapping or an invariant for the system under verification must be guided by the user.

A completely different approach is based on computing a cutoff on the number of process replications or on the maximum length of paths. The former consists in finding a finite number of process instances such that if they satisfy a property then the same property is satisfied by an arbitrary number of such processes. After the seminal work of Emerson and Kahlson [10] on cutoffs for synchronization skeletons, several works proved the existence of cutoffs for further kinds of parameterized systems and specifications: namely token rings [9,24] and shared resources [25]. Recently, Hanna et al. [26] proposed a procedure to compute a cutoff for Input–Output Automata that is independent of the communication topology. On the other hand, computing the cutoff on the maximum length of paths of a parameterized system consists in finding an upper bound on the number of nodes in its longest computation path. When a property is satisfied within the bounded path, then the property holds for a system with unbound paths, i.e., with an arbitrary number of process instances. German and Sistla [27], Emerson and Namjoshi [28] proved that such a cutoff exists for the verification of parameterized systems composed of a control process and an arbitrary number of user processes against indexed LTL properties. Yang and Li [29] proposed a sound and complete method to compute such a cutoff for parameterized systems with only rendezvous actions. In that work, the property itself is represented as an automaton.

Formal verification of grid systems. Some work has been done to formalize grid architectures and their protocols. The proposed approaches can be grouped, according to underlying formal models, based on: Petri Nets (e.g., see [30,31]), process algebras (e.g., see [32–36]), state representations (e.g., see [37,38]), and functional specifications (e.g., see [39]). Concerning Petri Nets, the parameterization of the system is naturally modeled by means
of a parametric amount of tokens in the network. On the other hand, this approach requires that, for each given specification, some auxiliary lemmata must be proved [30]. This means that the verification process is not completely automatic but requires a theorem proving phase. In the approach proposed in this work, this phase is not required (see Section 5). Concerning works on process algebras, state representations, and functional specifications, such approaches do not take into account parameterized systems. This means that they can be applied only to specific problems (e.g., grid protocols [37,32]), grid service chain and composition [34,36], grid optimization algorithms [35,33]), and they are not able to model the intrinsically parameterized nature of grid systems. On the other hand, protocols/workflows verification/composition do not need to be modeled as parameterized systems. Some authors use model checking [34,35,37], and the rest of them use some kind of bisimulation analysis or calculus. In the latter case, the complexity has raised some concerns as some authors [38] admit. However, this paper deals with parameterized systems and proposes an efficient verification methodology (see Sections 4 and 5). Concerning security, it should be noted that only few works are specifically focused on security specification verification (e.g., see [32]).

More in details, Du et al. [30] showed that a grid system model eventually satisfies every request from grid users, and the computing elements that gave their contribution to the result will be payed for. Van der Aalst et al. [31] proposed that the model is validated running the same tasks on the model and on the real grid implementation. Once the model has been validated, they use it as a basis to foresee the load balancing of resources and throughput time of the overall grid. No formal verification is made with it. Dalheimer et al. [37] proposed to formally verify grid resource allocation protocol properties by means of SPIN. Properties are expressed in ltr. Aziz and Hamilton [32] proposed the formal verification, by means of $\pi$-calculus, of non-repudiation in a delegation protocol. Xu et al. [34,35] proposed a model checker for a variant of $\pi$-calculus to be applied to verification [34] and optimization [35] of grid service chain. Zhou and Zeng [36] used another variant of $\pi$-calculus for grid service composition verification, as well. Finally, Mishra et al. [33] verified load-balancing algorithms by means of a static analysis on process algebra expressions. A completely different approach can be found in the work of Bolotov et al. [38] and Prodan [39]. Bolotov et al. proposed [38] an integration of ctl and Deontic logic. The verification of complex systems is based on the natural deduction calculus. In the work of Prodan [39], grid application verification is based on functional specification. A kind of calculus is proposed, as well.

3. Theoretical background

This work deals with formal verification of security requirements of a grid system. According to the proposed approach, this means modeling the system as a parameterized system, representing the safety and security hyperproperties as branching-time temporal formulas, and finding an efficient way to verify the system. This section reports the basic notion in this field that are needed in the rest of the paper.

3.1. Transition systems with guards

According to several authors [27,20,10,26,29], a parameterized system can be modeled as a set of labeled transition systems. A labeled transition system is defined as follows:

**Definition 1 (Labeled Transition System).** A Labeled Transition System definition (or LTS) $U$ is the tuple $(S, \Sigma, \rightarrow, \lambda, L, S_0)$ where:

- $S$ is a finite and nonempty set of states;
- $\Sigma$ is the finite set of actions;
- $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation;
- $\lambda : S \rightarrow L$ is a labeling function on the states;
- $L$ is a finite set of labels;
- $S_0 \subseteq S$ is the set of initial states.

For the sake of simplicity, LTSSs with only an initial state $S_0 = \{ t \}$ will be considered. This reduces notation complexity, but results presented in this paper can be easily generalized to the case with more initial states. Notice that, a Labeled Transition System is in general a non-deterministic system. At this point, it is possible to provide the notion of concurrent system as a set of transition system definitions.

**Definition 2 (Concurrent System).** Let $\{ U_1, \ldots, U_k \}$ be a finite set of Labeled Transition System definitions. Then, a concurrent system $P$ is defined as follows:

$$P = (\{ U_1^{\text{in}}, \ldots, U_k^{\text{in}} \}) = (U_1^1, \ldots, U_1^n, \ldots, U_k^1, \ldots, U_k^n).$$

Intuitively, $P$ denotes a concrete system composed of $n_1$ copies of $U_1$, through $n_k$ copies of $U_k$ running in parallel asynchronously (i.e., the execution of each instance is interleaved with the others). Notice that $P$ is still an LTS where each state is a tuple of instance states [10]. In the following, a Labeled Transition System definition will be referred to as a definition, for short, and any of its running copy will be referred to as an instance. While each definition can be written as $U_i = (S_i, \Sigma_i, \rightarrow, \lambda_i, L_i, S_0)$, the $k$-th instance of $U_i$ ($i \leq n$) is denoted by $U_i^j = (S_i^j, \Sigma_i^j, \rightarrow, \lambda_i, L_i^j, S_0^j)$, thus for each state $s_i \in S_i$, $s_i^j \in S_i^j$ denotes the corresponding state of $U_i^j$.

It should be noted that, in general, nothing is said about how $\Sigma_i \subseteq U_i$ is built. In the synchronization skeleton framework [10], the kind of model considered in this work, some restrictions on $\Sigma$ have been imposed.

**Definition 3 (Emerson and Kahlon’s System [10]).** An Emerson and Kahlon’s System (or Concurrent System with Disjunctive Guards) is the tuple $(U_1^{(m)}, \ldots, U_k^{(n)})$ such that for each definition $d$:

- $L_i = S_i^r$;
- $\lambda_d$ is the identity function (i.e., $\lambda_d(s) = s$);
- $\Sigma_d = \{ g \ | \ g = \bigvee_{j \leq m} \bigwedge_{l \leq n} s_i^{(m)} \land \bigvee_{j \leq m} \bigvee_{l \leq n} \bigvee_{i \leq k} \bigvee_{l \leq n} \bigvee_{i \leq k} \}$, where for each definition $l$ (resp. $j$) and instance $r$ (resp. $h$), $s_i^{(m)} \in S_i^r$ (resp. $s_i^{(m)} \in S_i^j$).

An Emerson and Kahlon’s LTS is an LTS that applies the above restrictions.

Actions such as $\gamma$ are called disjunctive guards because a transition $s \xrightarrow{\gamma} t$ can be applied when $\gamma$ is considered active, that is, if and only if there exists an instance $r \in \{ 1 \ldots n \}$ \{ 0 \} of definition $l$ such that $\forall i \leq l \leq m$ $s_i^{(r)}$ is true (the current state $s_i^{(r)}$ of instance $r$ of definition $l$ is $s_i^{(r)}$ for a given $u$) or there exists an instance $k \in \{ 1 \ldots n \}$ of a definition $j$ such that $\forall i \leq k \leq m$ $s_k^{(j)}$ is true (the current state $s_k^{(j)}$ of instance $k$ of definition $j$ is $s_k^{(j)}$ for a given $u$). An active guard is denoted by $(s_1^{(1)} \ldots s_n^{(1)}) \ldots (s_1^{(n)} \ldots s_n^{(n)}) \equiv g$. Finally, condition $T_i^j = \bigvee_{d \leq e} s_i^{(d)}$ denotes the universal condition which is always true (roughly speaking, it expresses that it is possible to ignore the state instance $r$ of definition $l$ is in).

3.2. Specification language

According to the seminal work by Alpern and Schneider [12] and, more recently, Clarkson and Schneider [13] and Schneider [40], system specifications can be expressed in terms of traces...
as a system is defined as follows:

Definition 4 (Temporal Structure). Let \( P = (U_1^{(n_1)}, \ldots, U_k^{(n_k)}) \) be an Emerson and Kahlion system. Then

\[
P = (S, \Sigma, \rightarrow, \lambda, L, S_0)
\]

where \((S, \Sigma, \rightarrow, \lambda, L, S_0)\) is an LTS such that:

- \( S = S_{U_1} \times \cdots \times S_{U_k} = \{ (s_1^{(n_1)}, \ldots, s_k^{(n_k)}) | s_1^{(n_1)} \in S_{U_1}, \ldots, s_k^{(n_k)} \in S_{U_k} \} \).
- \( \Sigma = \Sigma_{U_1} \cup \cdots \cup \Sigma_{U_k} \).
- \( (s_1^{(n_1)}, \ldots, s_{i-1}^{(n_{i-1})}, s_i^{(n_i)}, \ldots, s_k^{(n_k)}) \xrightarrow{g \in \Sigma} (s_1^{(n_1)}, \ldots, s_{i-1}^{(n_{i-1})}, s_i^{(n_i)}, s_{i+1}^{(n_{i+1})}, \ldots, s_k^{(n_k)}) \)
  if there exist \( u, h \) such that \( s_i^{(n_i)} = s_j^{(n_j)} \) (for each \( v \neq u \) and \( j \neq h \)), and
  \( (s_1^{(n_1)}, \ldots, s_{i-1}^{(n_{i-1})}, s_i^{(n_i)}, \ldots, s_k^{(n_k)}) \xrightarrow{g} g \).

Let \( S^* \) be the set of finite sequence of states. Let \( S^\omega \) be the set of infinite sequence of states. Then

\[ P \subseteq S^* \cup S^\omega \]

is the set of paths of \( P \) if and only if for each \( \pi \in P \), for each \( i \), there exists \( g \in \Sigma \) such that: \( \pi = s_0, s_1, \ldots, s_i, s_{i+1}, \ldots \) and \( s_i \in S \) and \( s_i \xrightarrow{g \in \Sigma} s_{i+1} \).

Intuitively, Definition 4 means that the temporal structure, produced by system \( P \), is obtained straightforwardly computing the asynchronous products of all the instances. As an example, Fig. 3 shows the asynchronous product of the simple system defined in Fig. 2. For a path \( \pi = s_0, s_1, \ldots, s_i, s_{i+1}, \ldots \) and an index \( i \), it is possible to introduce the following notation: \( \pi[i] = s_i; \pi[i..i] = s_0, s_1, \ldots, s_i \) (prefix of \( \pi \)); \( \pi[i..] = s_i, s_{i+1}, \ldots \) (suffix of \( \pi \)). As a consequence, \( \pi[0] \) denotes any path \( \pi \) such that \( \pi[0] = s_0 \). A finite path \( \pi \) is a prefix of another (finite or infinite) path \( \pi' \) (denoted \( \pi \leq \pi' \)) if and only if there exists an index \( i \) such that \( \pi = \pi'[i..] \). Now it is possible to extend the notion of prefix to a set of traces (denoted by \( \leq \), as well) in the following way:

\[ \Pi \leq \Pi' \text{ iff } \forall \pi \in \Pi \cdot \exists \pi' \in \Pi' \cdot \pi \leq \pi'. \]

Notice that, \( \Pi' \) may contain paths that have no prefix in \( \Pi \).

According to the above notions, a hyperproperty is defined as follows:

Definition 5 (Hyperproperty). Let \( S \) be a set of states. Let \( H \) be a hyperproperty. Then,

\[ H \subseteq 2^S. \]

Let \( \Pi^\omega \) be the set of infinite paths of a system \( P \). Then \( P \) satisfies the hyperproperty \( H \) (denoted \( P = H \)) if and only if \( \Pi^\omega \) is in \( H \), formally \( P = H \text{ iff } \Pi^\omega \in H \).

Two noteworthy hyperproperties are: safety hyperproperty or hypersafety ("nothing bad happens") [13] and liveness hyperproperty or hyperliveness ("something good eventually happens") [13]. They are defined as follows:

Definition 6 (Hypersafety and Hyperliveness). Let \( S \) be a set of states.

- \( SH \) is a hypersafety if and only if
  \[
  \forall T \in 2^S \cdot \exists M \in 2^T. (T \not\in SH \Rightarrow (M \leq T \land (\forall T' \in 2^S. M \leq T' \Rightarrow T' \not\in SH))).
  \]
- \( k-SH \) is a k-hypersafety if and only if
  \[
  \forall T \in 2^S \cdot \exists M \in 2^T. (T \not\in SH \Rightarrow (M \leq T \land |M| \leq k \land (\forall T' \in 2^S. M \leq T' \Rightarrow T' \not\in SH))).
  \]
- \( LH \) is a hyperliveness if and only if
  \[
  \forall T \in 2^S \cdot \exists T' \in 2^S. T \leq T' \land T' \in LH.
  \]

Clarkson and Schneider proved that each property can be obtained by the intersection of a hypersafety and a hyperliveness.

Theorem 1 (Hyperproperty Decomposition). For each hyperproperty \( H \), there exists a hypersafety \( SH \) and a hyperliveness \( LH \), such that \( H = SH \cap LH \).

Several Temporal Logics have been introduced in order to give formal specifications of hardware and software systems. This work is based on the Indexed \(-\text{ctl}^\times\chi\), i.e., the Indexed Branching Temporal Logic without the next (X) operator.

Definition 7 (Syntax of Indexed \(-\text{ctl}^\times\chi\)). Let \( AP \) be a set of atomic propositions. Let \( I \) be a set of possible indexes. Then, an Indexed \(-\text{ctl}^\times\chi\) formula can be defined as:

\[
\phi(i) ::= p(i) | \phi(i) \land \phi(i) | \neg \phi(i) | A_i \phi(i) | \bigwedge_{i \in I} \phi(i)
\]

\[
\Phi(i) ::= \phi(i) | \Phi(i) \land \Phi(i) | \neg \Phi(i) | \Phi(i) \cup \Phi(i)
\]

where \( i \in I \) and \( p \in AP \).

Intuitively, \( \phi(i) \) (resp. \( \Phi(i) \)) defines the set of so-called state-formulas (resp. path-formulas) that contain at least an atomic proposition indexed by \( i \). In the following, formulas without path
quantifiers \( (A \land E) \) form \( \text{LTL} \setminus \text{x} \), a subset of Indexed \( \text{CTL}^* \text{x} \). Conventionally, Indexed \( \text{CTL}^* \text{x} \) can be extended as below:

\[
\begin{align*}
\forall i \phi_i & \equiv \neg(\neg\phi_1 \land \neg\phi_2) \\
\forall_i \phi_i & \equiv \neg\bigwedge_i \phi_i(i) \\
\exists_i \phi_i & \equiv \neg(\neg\phi_1 \land \neg\phi_2) \\
\forall_i \phi_i & \equiv \forall \phi_1(i) \quad \text{(in the future } \phi_1) \\
G\phi_1 & \equiv \neg F \neg \phi_1 \quad \text{(in all states } \phi_1).
\end{align*}
\]

In this work, the semantics of Indexed \( \text{CTL}^* \text{x} \) formulas is provided w.r.t. the temporal structure introduced in Definition 4. This means that \( AP = S \cup U \cdots \cup S_{lk} \).

**Definition 8 (Semantics of Indexed \( \text{CTL}^* \text{x} \)).** Let \( \phi \) be a state formula (resp. \( \Phi \) be a path formula). Let \( P \) be an Emerson–Kahlon system, \( S \) be its set of states, and \( \pi^m \) be its set of infinite paths, according to Definition 4. Let \( s \in S \) (resp. \( \pi \in \pi^m \)). Then, \( P \models \phi \) (resp. \( P, \pi \models \phi \)) denotes that \( \phi \) is true w.r.t. a state \( s \) (resp. \( \Phi \) is true w.r.t. a path \( \pi \)) in the temporal structure of \( P \) if and only if:

\[
P, s \models \phi(i) \quad \iff \quad s = (s_1^i, \ldots, s_n^i, \ldots, s_k^i) \quad \text{and} \quad \exists \pi (p \in S_{lk} \text{ and } s^\pi_1 = p).
\]

As suggested by Clarkson and Schneider [13], it can be noted that models for state-formulas are hyperproperties. In fact, the semantics of each state-formula denote a set of sets (one for each state \( s \)) of paths (all the paths \( \pi \) such that \( \pi[0] = s \)).

### 3.3. Transformation theorems

The verification of parameterized systems is known to be in general undecidable [8]. Nevertheless, under certain conditions, the verification of a system with an arbitrary number of instances can be reduced to the verification of a system with a pre-defined number of instances. This number is called the cutoff of the system. The approach presented in this paper is based on the work by Emerson and Kahlon [10] where a Cutoff Theorem that holds for systems that comply with Definition 3 is presented. For the sake of self-containedness of the paper, such a theorem has been reported here.

In the following, the notation \( (n_1, \ldots, n_k) \preceq (m_1, \ldots, m_k) \) denotes the pair-wise less-than \( (\preceq) \) operator.

**Theorem 2 (Emerson and Kahlon’s Single Cutoff Lemma [10]).** Let \( \phi \) be \( \bigwedge F A (i) \) or \( \bigwedge E F (i) \) where \( i = (i_1, \ldots, i_n) \) is a set of indices such that for each \( i_k \in i_1, 1 \leq i_k \leq n_k \); where \( f \) is an \( \text{LTL} \setminus \text{x} \). Then

\[
\forall (n_1, \ldots, n_k) \preceq (1, \ldots, 1), (U_1^{(n_1)}, \ldots, U_k^{(n_k)}) \models \phi \quad \iff \quad (U_1^{(m_1)}, \ldots, U_k^{(m_k)}) \models \phi
\]

where the cutoff \( (c_1, \ldots, c_k) \) is given by \( c_i = |U_i| + 2 \) if \( i \in i \), \( c_i = |U_i| + 1 \) otherwise.

Furthermore, Clarkson and Schneider [13] proved another result that is used in this work.

**Theorem 3 (k-Hypersafety Transformation Theorem [13]).** Let \( P \) be a system. For each \( k \)-hypersafety \( k \text{SH} \), there exists a 1-hypersafety \( 1 \text{SH} \) such that:

\[
P = k \text{SH} \quad \iff \quad P^{(k)} = 1 \text{SH}
\]

Intuitively, this means that the product can be used to transform the problem of verifying a \( k \)-hyperproperty in the problem of verifying a simple safety property.

### 4. Grid system verification: a theoretical framework

A grid system may have some regularities that can be exploited in order to make its verification efficient. Therefore, this section reports some novel theoretical results that can be useful for formal verification.

The first result regards the transformation of safety hyperproperties in simple safety properties:

**Theorem 4 (Hypersafety Transformation Theorem).** Let \( U \) be an LTS definition. Let \( \phi \) be a safety hyperproperty. Let \( \phi \) be a safety property.

\[
\begin{align*}
\forall (U \models \phi) & \quad \iff \quad \forall k, (U^{(k)}) \models \phi.
\end{align*}
\]

**Proof.** \( (U \models \phi) \iff (U^{(k)}) \models \neg \phi \) (according to Clarkson and Schneider [13]), where \( k \)-\( \phi \) is a \( k \)-safety hyperproperty.

\[
\forall k, (U \models \neg \phi) \quad \iff \quad \forall k, (U^{(k)}) \models \phi
\]

Intuitively, this means that parameterized systems can fully cover hypersafety, which are an important part of hyperproperties. The way in which \( \phi \) is built from \( \phi \) is reported in the work of Clarkson and Schneider [13].

The second result regards concurrent systems with multiple process instances, called clients, requesting single instances, called servers, for services. This situation can be analyzed with a corollary of the Emerson and Kahlon’s Reduction Theorem (Theorem 2).

**Corollary 5 (Client–Server Single Cutoff Lemma).** Let \( (U_1^{(n_1)}, \ldots, U_k^{(n_k)}) \) be an Emerson and Kahlon’s System (Definition 3).

\[
\forall (n_1, \ldots, n_k) \preceq (1, \ldots, 1), (U_1^{(n_1)}, \ldots, U_k^{(n_k)}) \models \phi \quad \iff \quad (U_1^{(m_1)}, \ldots, U_k^{(m_k)}) \models \phi
\]

where the cutoff \( (c_1, \ldots, c_k) \) is given by \( c_i = |U_i| + 2 \) if \( i \in i \), \( c_i = |U_i| + 1 \) otherwise.

**Proof.** By the Truncation Lemma [10], we have that for all \( n_1, \ldots, n_k \) and \( p \):

\[
(U_1^{(n_1)}, \ldots, U_k^{(n_k)}) \models E h(1_p) \quad \iff \quad (U_1^{(m_1)}, \ldots, U_k^{(m_k)}) \models E h(1_p)
\]

where \( n_p = \min(n_p, |U_p| + 2) \) and for \( p \neq q, n_p = \min(n_p, |U_q| + 1) \).

This clearly holds when some \( n_k = 1 \). Therefore, the above result together with Theorem 2, symmetry and duality between \( A \) and \( E \) operators prove the corollary. \( \square \)

The third result regards concurrent systems with some independent processes. Let us introduce the notion of dependency/independency between definitions as follows:

\[
\begin{align*}
\Phi \models \Psi & \quad \iff \quad \Phi \models \Psi \quad \text{and} \quad \Psi \models \Phi.
\end{align*}
\]
Definition 9 (Dependency). Let \((U^{(n_1)}_1, \ldots, U^{(n_k)}_k)\) be a Concurrent System with Disjunctive Guards (Definition 3). Let \(U_i\) be a LTS definition such that: \(\Sigma_i = \{g \mid g \equiv \bigvee_{r \in \Pi_i} \Gamma^i_r \lor \bigvee_{j \leq l < n_i} \Gamma^i_j\}\).

Then,

- **(Self-dependency)** \(U_i\) depends on itself (denoted by \(U_i \leadsto U_i\)) if and only if there exists a guard \(g \in \Sigma_i\) where \(g \equiv \bigvee_{r \in \Pi_i} \Gamma^i_r \lor \bigvee_{j \leq l < n_i} \Gamma^i_j\) and such that \(\Gamma^i_g \neq T^i_l\);

- **(Outer-dependency)** \(U_i\) depends on \(U_j\) (denoted by \(U_i \leadsto U_j\)) if and only if either
  - there exists \(g \in \Sigma_i\) where \(g \equiv \bigvee_{r \in \Pi_i} \Gamma^i_r \lor \bigvee_{j \leq l < n_i} \Gamma^i_j\) and such that \(\Gamma^i_g \neq T^i_j\);
  - or there exists \(U_j\), such that \(U_i \leadsto U_i\) and \(U_i \leadsto U_j\) (transitivity).

Intuitively, \(U_i\) depends on itself (resp. \(U_i\) depends on \(U_j\)) when for each \(i\), there is \(r\) (resp. \(k\)) such that the behavior of \(U_j^i\) depends on the behavior of \(U^j\) (resp. of \(U^i\)). In fact, when \(\Gamma^i_g \neq T^i_j\) (\(\Gamma^j_g \neq T^i_j\) currently) affects the behavior of \(U_j\) (which can be either allowed or not). It should be noted that \(U_i \leadsto U_i\) does not usually imply that \(U_i \leadsto U_j\).

Independence (denoted by \(U_i \not\leadsto U_j\) and \(U_i \not\leadsto U_i\), respectively) means that \(U_i\) does not depend on \(U_j\) or \(U_i\). Notice that it may be still the case that \(U_i \leadsto U_j\).

Theorem 6 (Independent Process Theorem). Let \((U^{(n_1)}_1, \ldots, U^{(n_k)}_k)\) be a Concurrent System with Disjunctive Guards (Definition 3). Let \(U_i \not\leadsto U_j\) for each \(h < k\). Let \(\phi(i)\) be \(Af(i)\) or \(Ef(i)\) where \(1 \leq l < k\) and \(f\) is an \(\text{LTL} \land \chi\) path formula. Then

\[
\forall(n_1, \ldots, n_k), \quad \phi(i) \iff (U^{(n_1)}_1, \ldots, U^{(n_k)}_k) = Af(i).
\]

Proof. (Case \(\Rightarrow\))

\[(U^{(n_1)}_1, \ldots, U^{(n_k)}_k) = Af(i) \implies \forall \pi, \pi' = f(i) \text{ (by Definition 8)}\]

Every time a transition such as Transition (1) with \(h = k\) occurs in path \(\pi\), transition (1) can be substituted by state

\[(s^1_1, \ldots, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1})\]

obtaining the path

\[(\ldots (s^1_1, \ldots, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1})).\]

As a consequence, applying the two rules above to reach each transition in \(\pi\), it is possible to obtain a path \(\pi'\) at this point, using the block bisimulation introduced by Emerson and Namjoshi [9] (for this reason \(f\) must be a \(\text{LTL} \land \chi\) formula) and taking into account that \(f\) does not contain states of \(U_h\), it is possible to prove that:

\[
\forall \pi, \pi' = f(i) \implies \forall \pi'' = f(i) \text{ (by Definition 8)}\]

\[
\forall \pi, \pi' = \pi'' \implies Af(i).
\]

\[
\forall \pi, \pi', \pi'' = (\ldots (s^1_1, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1})), (s^1_1, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1} - 1, \ldots) = f(i).
\]

In this case also, each path \(\pi'\) contains transitions as the following:

\[
(s^1_1, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1} - 1),
\]

\[
\overline{\pi} (s^1_1, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1} - 1).
\]

(2)

Transition (2) means that there exists \(u, h\) such that \(s^u_h \overline{\pi} s^u_h\), \(s^u_h = s^u_h\) (for each \(v \neq u\) and \(j \neq h\)), and \((s^1_1, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1} - 1) \equiv f(i).

Taking into account that \(g\) does not depend on \(c_1, \ldots, c_k\), it is possible to substitute transition (2) with the following transition:

\[
(s^1_1, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1} - 1, c_1, \ldots, c_k).
\]

Obtaining the path

\[
(\ldots (s^1_1, s^1_{h-1}, s^k_{h-1} - 1, \ldots, s^k_{h-1} - 1), (s^1_1, s^1_{h-1}, s^k_{h-1} - 1, s^k_{h-1} - 1, \ldots), \ldots).
\]

As a consequence, applying the above rule to each transition in \(\pi'\), it is possible to obtain a path \(\pi\). At this point, using the block bisimulation and taking into account that \(f\) does not contain states of \(U_h\), it is possible to prove that:

\[
\forall \pi, \pi' = f(i) \implies \forall \pi'' = f(i) \text{ (by Definition 8)}\]

\[
\forall \pi, \pi' = \pi'' \implies Af(i).
\]

The case for \(Ef(i)\) can be derived by the equivalence

\[
Ef(i) \equiv \neg A \overline{f}(i).
\]

Furthermore, it can be useful, in certain circumstances, to group in a different way LTS instances. This is possible thanks to the following result:

Theorem 7 (Splitting Theorem). Let \((U^{(n_1)}_1, \ldots, U^{(n_k)}_k)\) be a Concurrent System with Disjunctive Guards (Definition 3). Then

\[
\forall(n_1, \ldots, n_k), \quad \phi(i) \iff (U^{(n_1)}_1, \ldots, U^{(n_k)}_k) = \phi.
\]

\[
\forall(n_1, \ldots, n_k) \geq (1, \ldots, 1), (U^{(n_1)}_1, \ldots, U^{(n_k)}_k) = \phi \iff \forall\phi(p, n_1, \ldots, n_k) \geq (1, \ldots, 1), (U^{(n_1)}_1, U^{(n_1-p)}_1, \ldots, U^{(n_k)}_k) = \phi.
\]

\[\text{A sequence of states that do not change is equivalent to a single state.}\]
indexes. Thanks to any bijective pairing function LTS definitions must be adapted, coherently. can be rewritten with a single index. Indeed, let \[ U \equiv \left\{ \begin{array}{ll} s_1, \ldots, s_p, \ldots, s_i, \ldots, s_{n_i}, \ldots, s_{n_1}, \ldots, s_1 \end{array} \right\}. \]

This means that instance states can be also grouped as:

\[ \left( \begin{array}{c} s_1, \ldots, s_p, \ldots, s_i, \ldots, s_{n_i}, \ldots, s_{n_1}, \ldots, s_1 \end{array} \right) \]

The fact that guards are only in disjunctive form guarantees that a different grouping does not change the truth of \( \phi \). □

Intuitively, this means that a set of instances \( U_1, \ldots, U_p, U_1^{p+1}, \ldots, U_{n_1}^{n+1} \), belonging to the same definition, can be split in two sets: \( U_1, \ldots, U_p \) and \( U_1^{p+1}, \ldots, U_{n_1}^{n+1} \). Notice that indexes of guards in all LTS definitions must be adapted, coherently.

Moreover, LTS definitions for grid systems may have multiple indexes. Thanks to any bijective pairing function, such definitions can be rewritten with a single index. Indeed, let \( C_{n,m} : [1 \cdots n] \times [1 \cdots m] \to [1 \cdots n \cdot m] \) be: \( C_{n,m}(i,j) = (i - 1)m + j \), then

\[ U^i_j \equiv \left( U^{i,J,j} \right). \tag{3} \]

The proof is trivial and is omitted. The result can be extended to a generic tuple of indexes, such as, \( U^i_j^{l,k} \equiv \left( U^{i,J,j,k} \right) \) and so on.

5. Grid system verification: a methodological framework

The theoretical results obtained in Section 4 together with the results obtained by other authors and briefly reported in Section 3 allow for the definition of a methodology in order to verify, under certain conditions, grid system security. These are the following conditions: (1) the grid system can be modeled as a Concurrent System with Disjunctive Guards with the presence of possible multiple indexes; (2) the security specification can be denoted by an indexed \( \text{cct}^* \) formula. The resulting methodology is reported in Fig. 4 and consists of six steps.

5.1. Modeling

The first step consists in modeling the grid system as a Concurrent System with Disjunctive Guards with the presence of possible multiple indexes. Indeed, a grid system is composed of a certain number of actor types and, for each actor type there may be several instances. The presence of more instances for the same actor is a consequence of job migration (when the actor is a grid job), more jobs to manage (when the actor is a computing element), or multiple requests for the same resource (when the actor is a resource manager). Furthermore, more instances may be the consequence of multiple parameters. In all the above cases, different instances may need to be linked among them and this can be done by means of multiple indexes. Provided that it is possible to find bijective pairing functions (e.g. \( C(i,j) \)), then multiplex indexes are not a problem.

Furthermore, it is useful to model a grid job that has a malicious or defective behavior. This allows the verification of security requirements even when the worst conditions occur. This approach is borrowed from a formal verification of security protocols, where it is a standard practice and is based on the so-called Dolev–Yao model of intruder [41].

From all the above considerations, a grid system must be modeled as a set of honest actors and an intruder.

Honest Actor. Modeling an honest actor means that the model must be able to represent all the legal actions the actor can perform in the right order. Such information can be obtained by analyzing system technical specifications and/or inspecting its code. Furthermore, how many instances of a given actor type are running is not known a priori. As a consequence, modeling an honest actor can be considered as a parameterized problem. Before going into the details of what an honest actor is, let us introduce two auxiliary concepts: Data Labeled Transition Systems (Data LTS or D-LTS) and Action Labeled Transition Systems (Action LTS or A-LTS).

Definition 10 (Data LTS). Let \( D = \{d_1, \ldots, d_n\} \) be a finite set representing the domain of some datatype. Then, the Data LTS (D-LTS, for short) \( D \) built upon \( D \) is an Emerson and Kahlon’s LTS (Definition 3) such that:

- \( S_D = \{s_0\} \) where \( s_0 \) is the initial state;
- \( S_D = \{s_d \mid d \in D\} \cup S_0 \).

Intuitively, the aim of D-LTS is to represent the data stored in the actor’s memory as an Emerson and Kahlon’s System. In other words, each state of a given D-LTS represents a possible value of the given datatype.

Definition 11 (Action LTS). Let \( L = \{l_1, \ldots, l_m\} \) be a finite set of local actions. Let \( M = \{m_1, \ldots, m_l\} \) be a finite set of messages. Then, the Action LTS (A-LTS, for short) \( A \) built upon \( L \) and \( M \) is an Emerson and Kahlon’s LTS (Definition 3) such that:

- \( S_A = \{s_0\} \) where \( s_0 \) is the initial state;
- \( S_A = \{s_l \mid l \in L\} \cup \{s_m, s_m \mid m \in M\} \cup S_0 \).

Intuitively, the aim of an Action LTS is to represent the behavior of a given actor that can execute actions enumerated in \( L \) and exchange messages enumerated in \( M \). The order in which actions are executed is represented by the transition relation \( \rightarrow \). Notice that, actor synchronization is therefore obtained by means of message passing. \( !m \) denotes the action of sending \( m \), whereas \( ?m \) denotes the action of receiving a message \( m \).

Now, it is possible to define an honest actor as the composition of its behavior (an A-LTS) and its data (a D-LTS).

Definition 12 (Honest Actor). Let \( L = \{l_1, \ldots, l_p\} \) be a finite set of local actions. Let \( M = \{m_1, \ldots, m_q\} \) be a finite set of messages. Let \( D = \{d_1, \ldots, d_k\} \) be a finite domain. Finally, let \( A \) be the A-LTS defined over \( L \) and \( M \) and let \( D \) be the D-LTS defined over \( D \). Then, an Honest Actor is the Emerson and Kahlon’s LTS:

\[ H = (S_H, \Sigma_H, \rightarrow_H, S_{H0}) \]
where $S_0 = \{(S_0, S_0)\}$. Furthermore, $S_0$, $\Sigma_H$, and $\rightarrow_H$ are the smallest sets satisfying the following constraints:

**local** For each local action $(l, q)$, where $l \in \mathcal{L}$, $p \in \mathcal{D}$ is its input parameter, and $q \in \mathcal{D}$ is its output parameter:

- $S_H \supseteq \{(s_1, s_p), (s_1, s_q) \mid s_1 \in S_A, s_p, s_q \in S_D\}$
- If $p \neq q$, then $S_H \supseteq \{T\}$
- If $p = q$, then $\rightarrow_H \supseteq \{(s_1, s_p) \rightarrow (s_1, s_q) \mid s_1 \in S_A, s_p, s_q \in S_D\}$

**send** For each action of sending a message $m \in \mathcal{M}$ (denoted by $\text{snd}(p, q)$ or $\text{snd}(p)$ for short) to another actor $U$, where $p \in \mathcal{D}$ is its input parameter. Let $j$ be an index ranging over all the instances of $U$.

- $S_H \supseteq \{(s_{m_0}, s_p), (s_{m_0}, s_q), (s_{m_0}, s_p) \mid s_{m_0} \in S_A, s_p \in S_D\}$
- $\Sigma_H \supseteq \{T, \sqrt[1]{(s_{m_0}, s_q)}\}$
- $\rightarrow_H \supseteq \{(s_{m_0}, s_p) \rightarrow (s_{m_0}, s_q), (s_{m_0}, s_p) \rightarrow \sqrt[1]{(s_{m_0}, s_q)} \mid s_{m_0} \in S_A, s_p, s_q \in S_D, (s_{m_0}, s_q) \in S_U\}$

**receive** For each action of receiving a message $m \in \mathcal{M}$ (denoted by $\text{rcv}(p, q)$) from another actor $U$, where $p \in \mathcal{D}$ is its input parameter, and $q \in \mathcal{D}$ is its output parameter. Let $j$ be an index ranging over all the instances of $U$.

- $S_H \supseteq \{(s_{m_0}, s_p), (s_{m_0}, s_q), (s_{m_0}, s_p) \mid s_{m_0} \in S_A, s_p \in S_D\}$
- $\Sigma_H \supseteq \{T, \sqrt[1]{(s_{m_0}, s_q)}\}$
- $\rightarrow_H \supseteq \{(s_{m_0}, s_p) \rightarrow (s_{m_0}, s_q), (s_{m_0}, s_p) \rightarrow \sqrt[1]{(s_{m_0}, s_q)} \mid s_{m_0} \in S_A, s_p, s_q \in S_D, (s_{m_0}, s_q) \in S_U\}$

**initial** For each initial action $a(p, q)$ admitted in the system (i.e., $S_A \subseteq S \rightarrow_a$), where $p$ represents the input for $a$:

- $\rightarrow_H \supseteq \{(s_{a_1}, s_q) \rightarrow (s_{a_2}, s_p) \mid (s_{a_1}, s_q) \in S_H\}$

When $a = !m$ then $(s_{a_1}, s_p) = (s_{m_0}, s_p)$. The case for $?m$ is similar.

**sequence** For each sequence of actions $a_1(p, d); a_2(d, q)$ admitted in the system (i.e., $S_A \supseteq S \rightarrow_{a_1} \rightarrow_{a_2} \rightarrow_a$), where $d$ represents the output of $a_1$ that must be used as input for $a_2$:

- $\rightarrow_H \supseteq \{(s_{a_1}, s_q) \rightarrow (s_{a_2}, s_q) \mid (s_{a_1}, s_q) \in S_H\}$

When $a_1 = !m$ (resp. $a_2 = ?m$), then $(s_{a_1}, s_q) = (s_{m_0}, s_q)$ (resp. $(s_{a_2}, s_q) = (s_{m_0}, s_q)$). The case for $?m$ is similar.

Intuitively, an input parameter denotes the data that an actor needs in order to achieve the related action. An output parameter denotes the data that an actor obtains after the execution of the related action. Therefore, the fragment of automation for an action $a(p, q)$ represents the fact that when an instance of parameter $p$ is available (represented by the state $(s_1, s_p)$ for a local action, the state $(s_{m_0}, s_p)$ for a sending action, the state $(s_{m_0}, s_q)$ for a receiving action), the honest actor starts the execution of action $a$ that ends by returning an instance of parameter $q$ (represented by the state $(s_1, s_q)$ for a local action, the state $(s_{m_0}, s_q)$ for a sending action, the state $(s_{m_0}, s_q)$ for a receiving action). It should be noted that the action of sending a message does not change the data of the sender actor, i.e., it does not have any output parameters. Furthermore, the action of sending a message for an actor $H$ must be synchronized with the action of receiving a message of a given actor $U$. This is represented by the guard $\sqrt[1]{(s_{m_0}, s_q)}$ for the sender and the guard $\sqrt[1]{(s_{m_0}, s_q)}$ for the receiver. Finally, when a local action does not produce any output (i.e. $p = q$), no transition is inserted in $\rightarrow_H$.

For example, let us consider the chirp_fetch process in the Condor System [42]. This process receives access file requests from other processes, checks who the owner is, and eventually performs the request. As a consequence, the domain consists of all the users, and the D-LTS can be built according to Definition 10, formally:

$$\mathcal{D} = \{\text{condor}, \text{user}_1, \ldots, \text{user}_n\}.$$  \hspace{1cm} (4)

$$\mathcal{S}_U = \{\text{condor}, \text{user}_1, \ldots, \text{user}_n\} \cup \{S_0\},$$

$$\rightarrow_D = \mathcal{D} \times \{T\} \times \mathcal{D}.$$  \hspace{1cm} (5)

Furthermore, the set of actions consists of opening, reading, and closing a file; the set of messages consists of the related requests. As a consequence, the A-LTS can be built according to Definition 11, as follows:

$$\mathcal{E} = \{\text{open}, \text{read}, \text{close}\},$$

$$\mathcal{M} = \{\text{open} \_ \text{req}, \text{read} \_ \text{req}, \text{close} \_ \text{req}\},$$

$$\mathcal{S}_A = \{\text{open}, \text{read}, \text{close}, \text{open} \_ \text{req}, \text{read} \_ \text{req}, \text{close} \_ \text{req}\} \cup \{S_0\}.$$  \hspace{1cm} (6)

The transition relation $\rightarrow_a$ is reported in Fig. 5.

The honest actor can thus be obtained by applying Definition 12, and the result (limited to two users) is reported in Fig. 6. It should be noted that all the actions used in the example do not produce any output, i.e. all the actions are of the form $a(p, p)$ (condition $p = q$ holds).

**Intruder.** The security of a system should be checked against all possible hostile processes. Usually, this problem is overcome by analyzing the case where only the most powerful intruder is considered [41]. The Dolev–Yao model represents one of the most general attacker models for protocols, but there is not a unique way to generalize it to systems with a more complex behavior than message delivery [40]. In the methodology proposed in this paper, an intruder must be able to (1) perform any operation the system allows it, (2) eavesdrop any message, (3) store and reuse any information it can access/eavesdrop/encrypt or decrypt with the proper key. In other words, it can neither perform an operation nor access a resource if it does not have the authorization, and it cannot decrypt a piece of data without the proper decryption key (the so-called perfect encryption assumption). The above assumptions represent an adaptation of the Dolev–Yao assumptions [41], adopted in security protocol verification [43], to the case of Grid computing. This means that a formal model of intruder must be able to choose, at each step, the next action to perform in a nondeterministic way in order to try all possible combinations. For
Fig. 6. chirp_fetch honest actor model.

reasons similar to those for honest actors, the intruder model can be considered as a parameterized model, as well. As a consequence, an intruder can be modeled as the composition of its behavior (an A-LTS) and its data (a D-LTS).

**Definition 13** (Intruder). Let $\mathcal{L}$, $\mathcal{M}$, $\mathcal{D}$ be a set of local actions, exchanged messages, and a domain, respectively, as in **Definition 12**. Let $A$ and $D$ be an A-LTS and a D-LTS, respectively, defined over them. Then, an Intruder is an Emerson and Kahlion’s LTS:

$$I = (S_I, \Sigma_I, \rightarrow_I, S_{I_0})$$

where $S_{I_0} = \{(s_{A_0}, s_{D_0})\}$. Furthermore, $S_I$, $\Sigma_I$, and $\rightarrow_I$ are the smallest sets satisfying the following constraints:

(local) For each local action $l(p, q)$, where $l \in \mathcal{L}$, $p \in \mathcal{D}$ is its input parameter, and $q \in \mathcal{D}$ is its output parameter:

- $S_I \supseteq \{(s_{A_0}, s_{D_0}, s_A, s_D) \mid s_A \in S_A, s_D \in S_D\}$
- $\Sigma_I \supseteq \{I\}$
- if $p \neq q$, then
  - $\rightarrow_I \supseteq \{(s_{A_0}, s_D) \mid (s_{A_0}, s_D) \rightarrow (s_{A_0}, s_D) \mid (s_{A_0}, s_D), (s_{A_0}, s_D) \in S_I\}$
- else
  - $\rightarrow_I \supseteq \{(s_{A_0}, s_D) \rightarrow (s_{A_0}, s_D) \mid (s_{A_0}, s_D), (s_{A_0}, s_D) \in S_I\}$

(send) For each action of sending a message $m \in \mathcal{M}$ (denoted by $!m(p, p)$ or $!m(p)$ for short) to another actor $U$, where $p \in \mathcal{D}$ is its input parameter. Let $j$ be an index ranging over all the instances of $U$. Then

- $S_I \supseteq \{(s_{A_0}, s_D), (s_{m_0}, s_D), (s_{m_0}, s_D) \mid s_{m_0} \in S_A, s_D \in S_D\}$
- $\Sigma_I \supseteq \{I, \forall j(x_{d_j}, x_{d_j}'_j)\}$
- $\rightarrow_I \supseteq \{(s_{A_0}, s_D) \rightarrow (s_{m_0}, s_D), (s_{m_0}, s_D) \rightarrow (s_{m_0}, s_D) \mid s_m \in S_A, s_D \in S_D, (s_{m_0}, s_D) \in S_I\}$

(receive) For each action of receiving a message $m \in \mathcal{M}$ (denoted by $?m(p, q)$ from another actor $U$, where $p \in \mathcal{D}$ is its input parameter, and $q \in \mathcal{D}$ is its output parameter. Let $j$ be an index ranging over all the instances of $U$. Then

- $S_I \supseteq \{(s_{A_0}, s_D), (s_{m_0}, s_D), (s_{m_0}, s_D) \mid s_{m_0} \in S_A, s_D \in S_D\}$
- $\Sigma_I \supseteq \{I, \forall j(x_{d_j}, x_{d_j}'_j)\}$
- $\rightarrow_I \supseteq \{(s_{A_0}, s_D) \rightarrow (s_{m_0}, s_D), (s_{m_0}, s_D) \rightarrow (s_{m_0}, s_D) \mid s_m \in S_A, s_D \in S_D, (s_{m_0}, s_D) \in S_I\}$

(initial) For each $d \in \mathcal{D}$:

- $S_I \supseteq \{(s_{A_0}, d) \mid d \in \mathcal{D}\}$
- $\rightarrow_I \supseteq \{(s_{A_0}, s_D) \rightarrow (s_{A_0}, s_D) \mid d \in \mathcal{D}\}$

Basically, the main difference between the honest actor model and the intruder model consists in a transition toward one of the states of the form $(s_{A_0}, s_D)$ after the execution of each action. This allows the intruder to perform actions in no predefined order.

For example, if one wanted to define an intruder based on the chirp_fetch process described above (see Eqs. (4) and (5)), the resulting intruder model would be the one as reported in Fig. 7.

5.2. Specifying

The second step consists in specifying those requirements that the grid system must satisfy in order to be considered safe and secure. According to Clarkson and Schneider [13] and Schneider [40], most of the requirements can be classified into safety hyperproperties and liveness hyperproperties. For this work, indexed $\text{ctlt}^x$ has been chosen, which is able to capture a large set of hyperproperties. In fact, all the hyperproperties that contain a set of traces starting from the same initial state. The kind of LTSs used in this work have only one initial state. Therefore, the kind of hyperproperties modeled by indexed $\text{ctlt}^x$ is as wide as possible. Furthermore, as a consequence of **Theorem 4**, all the safety hyperproperties can be verified by means of a parameterized model, and several important security requirements can be modeled as hypersafety. The consequence of all the above considerations is that the methodology proposed in this paper allows for the verification of several interesting requirements such as access control [44], Goguen and Meseguer’s noninterference [45], observational determinism [46], and secret sharing [47].
For the convenience of the reader, two popular examples of safety and liveness hyperproperties are reported below.

**Safety:**  \( \bigwedge_i \text{AG} \neg \phi(i) \)

where \( \phi(i) \) represents a “bad” state.

**Liveness:**  \( \bigwedge_i \text{A}(\phi(i) \rightarrow F \psi(i)) \)

where \( \phi(i) \) must have as a consequence that \( \psi(i) \) will eventually occur.

### 5.3. Splitting

The third step consists in applying Theorem 7. This means looking for LTS definitions so that a subset of their instances does not depend on the rest of the instances. If such a definition exists, its instances can be split into two groups (the first one independent of the second one).

### 5.4. Removing

The fourth step consists in applying the Independent Processes Theorem 6. This means looking for LTS definitions on which no other definition depends. If such a definition exists, all its instances can be removed, simplifying the verification problem. Notice that this step together with the previous one allow for the removal of a part of the instances of a given definition.

### 5.5. Reducing

The fifth step consists in applying the Reduction Theorem 2 and Corollary 5. This means computing the cutoff of the system and, thus, reducing the original problem to the problem of verifying the behavior of a small set of LTS instances.

### 5.6. Model checking

The sixth step consists in verifying the system as an outcome of the previous steps. Notice that each step is sound and complete according to the theorems that have been applied. Therefore, the verification of the resulting system is equivalent to the verification of the original system. The verification technique used in the proposed methodology is model checking [48].

### 6. Case study

This section presents the application of the methodology presented in Section 5 to Condor [49], a job manager for Grid system. This choice was inspired by a known vulnerability [50] that involved Condor and the protocol used to access files on remote hosts. This vulnerability leads to a privilege escalation, i.e., to a host security violation.

#### 6.1. The Condor system

Condor [51] is a software suite to manage system workload that has been developed at the University of Wisconsin. In particular, Condor manages the jobs for a Grid computing system [49]. Actually, Condor incorporates many of the emerging Grid-based methodologies and protocols [11]. For instance, Condor-G [52] is fully interoperable with resources managed by Globus [53,54].

In Condor [42], a job is a program to be executed along with a description of how to start the program, including command line arguments, files and the program environment. Jobs are prepared and submitted to Condor, which takes care of finding the correct machine type and running the job. It places the job in a queue until the required resources are available. Users do not submit to global queues because Condor has a decentralized model where users submit to a local queue on their computer and, Condor processes running on that computer, interact with the Condor matchmaker. Finally, Condor starts the submitted job on an available grid machine. It also takes care of switching the job to another machine if necessary and sharing available resources with other jobs in the queue. When the job completes, Condor sends a notification to the users.

In order to run a job submitted to Condor, there are interactions between three components, as shown in Fig. 8: the Submission machine (S), the Execution machine (E) and the Central Manager (CM). Each component runs some processes, a brief description of the individual processes is presented.

In the Submission machine, the schedd represents a job queue. This daemon records a job event and manages all of its data file during its lifetime. Finally, it starts the shadow process to manage the job execution. Indeed, the shadow represents a job on the Submission machine that has been matched with an Execution
machine. This process transfers files, handles forwarded system calls, and updates the job status.

In the Execution machine, the `startd` process records the host status and sets the `starter` process to manage the job execution. Indeed, the `starter` represents a job on the Execution machine that is running or that has to be started by this machine.

The Central Manager runs two daemon processes: the `collector` and the `negotiator`. The former is a repository of machine contexts (ClassAd) from all the servers in the system. These data are used to get the status of the system and to find idle execution hosts. The latter performs matchmaking between jobs and execution hosts; if a match is found, it informs the `schedd` containing the job and the `startd` so the letter can run the job.

The interaction protocol can be split in three phases: matchmaking, connecting and starting.

Matching requires four steps. In the first step, the `schedd` process in the Submission sends the job context to the Central Manager. In the second step, the `startd` process sends the machine context to the Central Manager. These two steps, that are labeled 1a and 1b in the Fig. 8, can be executed in any order or in parallel. In the third step, a matchmaker scans the known ClassAd (the job and machine context) and creates pairs that satisfy each other constraints and preferences. In the final step, the matchmaker sends to the `schedd` an identification of a machine where the job can be executed.

In the connecting phase, the `schedd` forks a `shadow`. This process contacts the `startd` in the Execution machine and sends the job requirement. If the `startd` finds that the job requirements match its machine context and the Execution machine is still idle, it sends an OK message containing an identification of the Execution machine to the `shadow`. If one of these conditions is not satisfied, it sends a not-OK message and the matchmaking and connecting phase are repeated. This is useful because the matchmaker operates on information that may be out of date. The `startd` checks to make sure that the job and machine ClassAd still match. For instance, the owner of the Execution machine may have reclaimed the machine for his/her own use after that the matchmaker performed that match, and therefore no jobs are able to run on the machine.

In the last phase, the `startd` process forks a `starter`. The `shadow` sends the job (executable code) directly to the `starter` and it begins its execution (the `starter` forks the job).

6.2. Verification

First of all, let us make some remarks. For the sake of clarity, a simplified version of honest actors and intruders will be used, with respect to the description given in Section 5.1. Namely, the processes representing all possible inputs that the operation can receive will be abstracted away and coded into the structure of the process itself.

From a graphical point of view, the rounded rectangles denote the states of each process, and the related labels are their names. The state with an incoming arrow from the special dot must be considered the initial state for the given process (see state `init` in Fig. 9).

It should also be noted that state names could have special symbols like `!` and `?` as prefixes, as already explained in Section 5.1.

Finally, guards on transitions will be represented either as $\bigvee_{i \in \text{proc}_1} \phi(i)$ or just as $\bigvee \phi(i)$, where $\phi$ is an indexed boolean formula involving states of $\text{proc}_1$ or other definitions in the system. The former denotes that the transition considers only instances of the process whose name is $\text{proc}_i$. The latter notation is just a shortening for a longer guard, namely

$$\bigvee_{i \in \text{proc}_1} \phi(i) \lor \cdots \lor \bigvee_{i \in \text{proc}_n} \phi(i),$$

provided that in the system instances of processes $\text{proc}_1 \ldots \text{proc}_n$ exist and their states are mentioned in $\phi$. Intuitively, the guard $\bigvee_{i \in \text{proc}_1} s^i$ is considered satisfied any time instance $i$ of definition $\text{proc}_1$ is in state $s$.

**MODELING.** The first process, named `submit`, is depicted in Fig. 9. It has a fairly simple logic, it is invoked by the user and receives the job to be executed together with some metadata. After the
matchmaking phase submit triggers the starter process on the (possibly remote) execute host and after that triggers the shadow process on the (local) submit host.

The starter process is responsible for launching the actual job and is represented in Fig. 10. After launching the job, starter forwards to the underlying operating system any request it receives for pausing or restarting the job. Finally, it returns as soon as the grid job returns.

In Fig. 11, the shadow process is depicted. In this model the focus is put on how the set_job_attr Owner request is handled, following the Condor implementation of the Chirp protocol. As it can be seen, such requests are forwarded via the chirp_set_owner process toward the local schedd process that takes care of them.

The process chirp_set_owner is represented in Fig. 12. In particular, it describes the behavior of the command condor_chirp when it is invoked with parameter set_job_attr Owner as in:

```
$ condor_chirp set_job_attr Owner foo
```

where the attribute owner of the job is set to value foo. As in the case of submit, also the logic of chirp_set_owner is rather simple: after having forwarded the request together with its arguments to the schedd process and waited for a reply, it ends its execution.

Figs. 13–15 together represent the schedd process. More precisely, the state trusted,sock-A (resp. untrusted,sock-A) in Fig. 14 (resp. Fig. 15) is the same as the state trusted,sock-A (resp. untrusted,sock-A) in Fig. 13. The whole picture, then, is obtained by merging these states in one big process. Similarly, from state trusted,sock-B (resp. untrusted,sock-B), a subprocess dual to the one depicted in Fig. 14 (resp. Fig. 15) begins, which is not represented for the sake of space.

The process depicted in the aforementioned figures has been extracted from the method SetAttribute in src/condor_schedd. V6/qmgmt.cpp of Condor 7.2.4 that actually handles the Chirp message “set_job_attr Owner”.

The process handles an internal queue of processes that are sent from the local host to the grid. Basically, it can be seen that after reading the name received as input, it is compared against the sock owner and queue superuser. In case the configuration states each user should be trusted (cfr. states trusted,sock-A and trusted,sock-B), whatever name is received, it is set to be the new owner of the job (i.e., the state !chirp_op_end is reached). On the contrary, if no user should be trusted (cfr. states untrusted,sock-A and untrusted,sock-B), the job owner is reset only if the received name is the same as the sock owner, or the username is that of another user who submitted a job into the system. In all the other cases the “set_job_attr Owner” operation fails.

In Fig. 16, a simple process is shown representing a data structure which stores the owner attribute for a given job during its execution. Since it is not possible to represent all the infinite usernames that the owner attribute can assume, in this work an abstract model has been considered. In this abstraction, the attribute can be either set to A or B. Those attribute values are described by means of the states ?owner_A and ?owner_B, respectively. This abstraction naturally describes the evolution of the above data structure:

- the non-determinism among the initial transitions expresses the fact that it is not possible to foresee which user this job will belong to;
- at any write operation, the value of the attribute can change or remain the same.

Finally, in Fig. 17, the process chirp_fetch is showed. It models the invocation of command:

```
$ condor_chirp fetch rfoo lfoo
```

that copies the file rfoo from the (possibly remote) submit host onto the file lfoo on the local Execute machine. The model has been inferred inspecting the code of method chirp_get_one_file contained in file src/condor_chirp/condor_chirp.cpp of Condor 7.2.4.
Fig. 12. chirp_set_owner process.

Fig. 13. schedd process (part 1).

Fig. 14. schedd process (part 2).
This model follows the previous abstraction that a resource, in this case a file, can be either owned by user A or by user B. The set of states of this model is obtained as the product of the sets:

\[\{\text{init}\} \cup (\{\text{?chirp\_close, ?chirp\_open, ?chirp\_read}\} \times \{A, B\})\]

The structure of the intruder process is called intr and is modeled in Fig. 18. intr follows the generic structure explained in Section 5.1, and Table 1 summarizes the messages that can be "exchanged" by the system processes. If \(\mu\) is a message named in Table 1, then \(\text{intr}[\mu]\) is the name of one process that composes the intruder. There may exist an unknown number of instances of this process \(\text{intr}[\mu]\).

The messages \(\mu\) admitted by the system are enumerated in Table 1. The intruder job is obtained by combining the many copies of process \(\text{intr}[\mu]\), and it is named job in order to communicate with process starter, already described in Fig. 10. In symbols, it is possible to say that:

\[U^{(n)}_{\text{job}} = (U^{(n_1)}_{\text{intr[?chirp\_close]}}, \ldots, U^{(n_k)}_{\text{intr[?submit\_start]}})\]

for some \(n_1, \ldots, n_k\), where \(n = n_1 + \cdots + n_k\).

Considering \(m\) the number of grid hosts, \(p\) the number of grid jobs, \(r\) the number of available resources, the system that is modeled and verified is then the following:

\[(U^{(m-p)}_{\text{sched}}, U^{(p)}_{\text{submit}}, U^{(p)}_{\text{shadow}}, U^{(p)}_{\text{start}}, U^{(p)}_{\text{chirp\_set\_owner}},\]
\[U^{(p)}_{\text{owner}}, U^{(p \cdot r)}_{\text{chirp\_fetch}}, U^{(m \cdot p)}_{\text{intr}})\]

Let us suppose that \(U^{(m \cdot p)}_{\text{intr}}\) is just a shortening for the longer composition: \(U^{(m \cdot n_1)}_{\text{intr[?chirp\_close]}}, \ldots, U^{(m \cdot n_k)}_{\text{intr[?submit\_start]}}, \) for some \(n_1, \ldots, n_k\), i.e., it represents an unbounded number of operations executed by the intruder using one of the messages of Table 1.

Every process is started by an instance of submit that launches its own copies of starter and shadow, the latter is associated to its own attribute owner and may launch a copy of chirp\_set\_owner. As a consequence, every such process exists in \(p\) copies in the modeled system. Process sched runs in every executing host, but it is possible for any process to execute it (since it is a regular executable file present in the system) and thus it exists in \(m - p\) copies. Finally, every job \(i\) that accesses a resource \(j\) runs a different copy of process chirp\_fetch, and thus there will be \(p \cdot r\) copies of it.

Specifying. The host protection requirement that has been analyzed with the current example is the correct authorization before accessing data available in the system. In particular, the desideratum is to guarantee that if the owner of a job \(j\) does not correspond to the owner of resource \(r\), then \(j\) should not be allowed to access \(r\)'s content.
In a more formal fashion, this correct authorization requirement can be stated as the following hypersafety properties:

\[
\bigwedge_{j,r} \neg(\text{owner}_j \land F(\text{chirp_read}, B)_{\text{chirp_fetch}}) \tag{7}
\]

\[
\bigwedge_{j,r} \neg(\text{owner}_j' \land F(\text{chirp_read}, A)_{\text{chirp_fetch}}) \tag{8}
\]

Splitting. The grid system (6) can be rewritten by isolating one of the Execution machines and assuming that \(p_1\), the \(p\) processes (where \(p_1 < p\)) are executing on that machine. In symbols, the set of instances \(U_{\text{schedd}}^{(m+p)}\) is divided into two sets, namely \(U_{\text{schedd}}^{(1+p_1)}\) and \(U_{\text{schedd}}^{(m-1+p-p_1)}\). It should be underlined that the sets of instances induced by the remaining definitions are either directly dependent on process schedd, (as the instances of chirp_set_owner) or indirectly dependent on it (as the instances of all the other processes). Such sets must be split as well, and their guards must be rewritten accordingly. For example, once \(U_{\text{schedd}}^{(p)}\) is split into \(U_{\text{chirp_set_owner}}^{(p)}\) and \(U_{\text{chirp_set_owner}}^{(m-p)}\), it might be the case that instances of the former group refer to instances of \(U_{\text{chirp_set_owner}}^{(1+p_1)}\), while instances of the latter group might refer to instances of \(U_{\text{chirp_set_owner}}^{(m-p)}\).

After the splitting operation justified by Theorem 7, the grid system becomes:

\[
(U_{\text{chirp_set_owner}}^{(1+p_1)}, U_{\text{chirp_set_owner}}^{(m-p)}, U_{\text{chirp_fetch}}^{(m-1)}, U_{\text{shadow}}^{(m)}, U_{\text{owner}}^{(m)}, U_{\text{chirp_fetch}}^{(m)}, U_{\text{start}}^{(1)}, U_{\text{intra}}^{(1)}, U_{\text{submit}}^{(1)}, U_{\text{shadow}}^{(1)}, U_{\text{owner}}^{(1)}, U_{\text{chirp_fetch}}^{(1)}, U_{\text{intr}}^{(1)}, U_{\text{submit}}^{(1)}, U_{\text{shadow}}^{(1)}, U_{\text{owner}}^{(1)}, U_{\text{chirp_fetch}}^{(1)}, U_{\text{intr}}^{(1)}).
\]

Removing. Observing System (9), it is possible to establish the following properties: \(U_{\text{chirp_set_owner}}^{(1+p_1)}\) (resp. \(U_{\text{chirp_set_owner}}^{(p)}, U_{\text{chirp_fetch}}^{(m-1)}, U_{\text{shadow}}^{(m)}, U_{\text{owner}}^{(m)}, U_{\text{chirp_fetch}}^{(m)}\)) does not depend on instances \(U_{\text{chirp_set_owner}}^{(1+p_1)}, U_{\text{chirp_fetch}}^{(m-1)}, U_{\text{shadow}}^{(m)}, U_{\text{owner}}^{(m)}, U_{\text{chirp_fetch}}^{(m)}\). As a consequence, applying Theorem 6 to System (9), it is possible to obtain the following equivalent system:

\[
(U_{\text{chirp_set_owner}}^{(1+p_1)}, U_{\text{chirp_fetch}}^{(m-1)}, U_{\text{shadow}}^{(m)}, U_{\text{owner}}^{(m)}, U_{\text{chirp_fetch}}^{(m)}).
\]

With a similar reasoning, it is possible to iterate the splitting and removing phases isolating one of the \(p_1\) processes. Since the instances of actors modeling processes are independent from their siblings, the obtained system is:

\[
(U_{\text{chirp_set_owner}}^{(2)}, U_{\text{chirp_fetch}}^{(1)}, U_{\text{shadow}}^{(1)}, U_{\text{owner}}^{(1)}, U_{\text{chirp_fetch}}^{(1)}).
\]

Notice that Theorem 6 cannot be applied to the original System (6). In fact, it is the splitting phase that makes independency relationships among instances evident, and thus, their removal as well.

Reducing. With Theorem 2 and Corollary 5 allow to compute the amount of instances needed for the verification, summarized in Table 2. Notice that in order to avoid the state explosion problem, it is possible to reduce the system even more. As an example, in process schedd, it is possible to avoid the state trusted adding transitions:

\(\neg \text{schedd \_start} \rightarrow \text{trusted, sock} \rightarrow \text{A}\)

\(\neg \text{schedd \_start} \rightarrow \text{trusted, sock} \rightarrow \text{B}\).

Applying this transformation recursively, it is possible to avoid the states that do not represent a synchronization with other processes. In other words, it is possible to remove the state whose name does not begin with the special characters ! or ?. Starting from the depicted model, with 60 states, it is possible to reduce it to a system with 46 states. Such transformation is actually an abstraction of the system that does not lose information for the property considered. In Table 2, the number of states in the reduced systems are specified between parentheses, and the cutoffs are computed on the reduced system.

It has been shown how the initial problem of verifying Specification (7) in a parameterized system can be reduced to the problem of verifying the same specification on a finite system:

\[
(U_{\text{chirp_set_owner}}^{(2)}, U_{\text{chirp_fetch}}^{(1)}, U_{\text{shadow}}^{(1)}, U_{\text{owner}}^{(1)}) \models \bigwedge_{j,r} \neg(\text{owner}_j \land F(\text{chirp_read}, B)_{\text{chirp_fetch}}) \tag{12}
\]

iff

\[
(U_{\text{chirp_set_owner}}^{(2)}, U_{\text{chirp_fetch}}^{(1)}, U_{\text{shadow}}^{(1)}, U_{\text{owner}}^{(1)}) \models \bigwedge_{j,r} \neg(\text{owner}_j \land F(\text{chirp_read}, B)_{\text{chirp_fetch}}). \tag{13}
\]

An analogue reduction can be obtained for Specification (8).

Notice that the reduced system has a relatively small size. Indeed, model checking can still be efficiently applied to many more large-sized systems [48].

Model checking. For this experiment, NuSMV [55] was used, a well-known state-of-the-art model checker. All NuSMV models used in this example are available at: http://www.pagliarecci.it/condor-example.zip Using NuSMV, it was possible to verify that Specification (13) does not hold. Indeed, NuSMV generates

<table>
<thead>
<tr>
<th>Table 1</th>
<th>The messages exchanged in the system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chirp_close</td>
<td>chirp_op_end</td>
</tr>
<tr>
<td>chirp_read</td>
<td>chirp_set_owner_A</td>
</tr>
<tr>
<td>job_restart</td>
<td>job_return</td>
</tr>
<tr>
<td>owner_A</td>
<td>owner_B</td>
</tr>
<tr>
<td>schedd_start</td>
<td>shadow_return</td>
</tr>
<tr>
<td>starter_start</td>
<td>submit_return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>The instances needed for the verification.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>States</td>
</tr>
<tr>
<td>owner</td>
<td>3</td>
</tr>
<tr>
<td>intr</td>
<td>3</td>
</tr>
<tr>
<td>schedd</td>
<td>60 (46)</td>
</tr>
<tr>
<td>starter</td>
<td>10 (8)</td>
</tr>
</tbody>
</table>
a counterexample where a job from user B is first scheduled by `schedd` process (state B-seen), then a job owned by A executes `chirp_set_owner` by specifying username B as parameter. After that, the `sCHEDd` allows to modify it and consequently, the `chirp_fetch` process is allowed to read the file. This is an example of privilege escalation attack.

One solution to this problem is to modify the implementation to include a `safe` version of process owner where the attribute is set only once per grid job. This solution has already been discussed in the vulnerability description [50] and is depicted in Fig. 19.

A second solution would be to allow instances of process `chirp_set_owner` to be called only from instances of `shadow` since the operation of setting the owner of a job may be considered a `system operation`. This requires to substitute the first transitions of Fig. 12 with the followings:

\[
\begin{align*}
\text{init} & \rightarrow ?\text{chirp_set_owner}_A \\
\text{init} & \rightarrow ?\text{chirp_set_owner}_B
\end{align*}
\]

7. Conclusions

The basic idea presented in this paper consists in exploiting formal verification techniques, for instance model checking, in order to verify the security requirements of grid systems. This idea came up from the observation that model checking has been recognized as a very useful technique to verify security requirements in different domains as, for instance, the security properties of security protocols.

Grid systems result from combining of many copies of a finite amount of processes. Indeed, they are also intrinsically dynamic systems since resources and users can, by definition, join and leave the network at any moment. Both aspects make them a natural example of parameterized systems, where the number of instances is not known a priori. Unfortunately, parameterized systems are, in general, undecidable. Furthermore, security requirements have also been recognized to be very complex to model. As a matter of fact, they can be modeled as hyperproperties that are considered, in general, undecidable, as well. Nevertheless, there are some results on both parameterized systems and hyperproperties to make decidable verification when there are some regularities.

It is in this particular line of research that this work has been inserted. The results presented in this paper are twofold: (1) a methodology has been presented to verify grid system security; (2) a set of original theoretical results has been presented that can be exploited for verification.

Regarding the methodology, this work shows how to model a grid system. This is actually a novel contribution of the present work since the related work mainly focuses on verifying the security of grid protocols only. In this work, formal verification has been applied to a different part of a grid system. According to the proposed methodology, it is possible to model grid processes on the basis of technical specifications and code inspection. A general schema of the model is reported in Section 5.1 whereas a concrete example, the Condor system, is reported in Section 6. In order to verify security requirements, a model of intruder is also needed which should be as powerful as possible. In the present work, the underlying idea of the Dolev–Yao model of intruder [41] has been adapted in order to verify host protection requirements (see Section 5.1). As far as we know, this is a novel contribution in the research area of grid security. Whether the proposed intruder model suffices for security verification or a more powerful model is needed is still an open question, and it will be the goal of another work in the future. Finally, security requirements can be modeled, in general, as hyperproperties. This work proposes to model security requirements as Indexed $\mathit{ctl}^\ast\neg x$ and shows how this is enough for hypersafety. Which fragment of hyperliveness is covered by Indexed $\mathit{ctl}^\ast\neg x$ is another open question to be investigated in the future. Notice that, the proposed models and specifications have been enough to reproduce a well-known security problem in Condor (see Section 6.2).

Regarding the theoretical results, they cover two complementary aspects that are exploited in the proposed methodology. First of all, it has been proved that parameterized systems can be used for hypersafety verification problems. This is an extension of the Clarkson and Schneider’s result stating that a self-composed system can be used for $k$-hypersafety verification. Furthermore, it has been proved that parameterized systems can be decomposed, when some regularities hold, in independent sub-systems and that all the independent sub-systems, except for the one to be verified, can be dropped from the verification process.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

References


