Formal enforcement of security policies on concurrent systems

Mahjoub Langar\textsuperscript{a,1}, Mohamed Mejri\textsuperscript{a}, Kamel Adi\textsuperscript{b}

\textsuperscript{a} LSFM Group, Computer Science Department, Laval University, Quebec, QC, Canada
\textsuperscript{b} LIRIS Group, Computer Science Department, University of Quebec in Outaouais, Gatineau, QC, Canada

\textbf{A R T I C L E  I N F O}

\textbf{Article history:}
Received 30 April 2010
Accepted 16 March 2011
Available online 11 May 2011

\textbf{Keywords:}
Execution monitoring
Security policies
Language based security
Concurrent systems
Process algebra

\textbf{A B S T R A C T}

This paper introduces a formal and modular technique allowing to automatically enforce a security policy on a given concurrent system. Given a concurrent program $P$ and a security policy $\phi$, we automatically generate another program $P'$ that satisfies $\phi$ and behaves like $P$, except that it stops when $P$ tries to violate the security policy $\phi$. We use extended version of process algebra ACP (Algebra of Communicating Process) and BPA (Basic Process Algebra) as formal languages to specify both concurrent system and security policy.

© 2011 Elsevier Ltd. All rights reserved.

\section{1. Introduction}

One of the important goals of the software development process is to prove that the produced systems always meet their requirements. However, making this kind of proofs requires high qualified persons and they are almost very subtle and complex. Since it is also a very expensive operation, it is generally omitted and we try to test the system as best as we can to reduce the risk of errors. For some critical system however, we have no choice and we pay the price to avoid the catastrophic consequences of errors. Therefore, tools like theorem prover and those used for automatic generation of software are helpful to significantly reduce the cost of proof. We hope that this tool will be powerful and simple enough so that even small companies and individuals with little budget and theoretical skills can benefit from it to produce high quality software.

This paper proposes an algebraic and automatic approach that could generate from a given program, and a security policy, a new version of this program that respects the requested security...
policy. More precisely, we define an operator \( \otimes \) that takes as input a process \( P \) and a security policy \( \Phi \) and generates \( P' = P \otimes \Phi \), a new process that respects the following conditions:

- \( P' \models \Phi \), i.e., \( P' \) “satisfies” the security policy \( \Phi \).
- \( P' \subseteq P \), i.e., behaviors of \( P \otimes \Phi \) are also behaviors of \( P \).
- \( \forall Q : ((Q \models \phi) \land (Q \subseteq P)) \Rightarrow Q \subseteq P' \), i.e., all good behaviors of \( P \) are also behaviors \( P \otimes \Phi \).

Amongst the desired properties, the traces of \( P' \) should be those accepted by both of \( P \) and \( \Phi \) (i.e. \( P \cap \Phi \)). Resolving this kind of problem is however not possible for any kind of security policies. In fact, at run time and some time even statically, we cannot guarantee that something good will happen (liveness properties) which may lead to an execution trace that does not respect the security policy if the process has not be stopped from the beginning. To deal with this problem, we can propose different kind of attitudes:

- **Conservative enforcement:** The program should be terminated as soon as its violates the security policy even if the current run could be completed by another suffix so that the security property will be respected. With this attitude, only safety properties could be enforced and liveness properties will block the program from the early beginning.
- **Liberal enforcement:** Do not abort the execution of the process if it could be completed by a suffix so that the security policy will be satisfied. With this attitude, we enforce more properties than the conservative one, but the program may terminate without satisfying the security policy.

To sum up, as shown by Fig. 1, the conservative enforcement will generate fault negative while the liberal enforcement will generate fault positive and no one of them reach the desired result.

In the rest of this paper we develop the liberal enforcement, but the idea can be extended to handle the conservative approach. For the specification of concurrent systems, we use an extended version of ACP “Algebra for Communicating Process” (Baeten, 2005) denoted by \( ACP^\Phi \). The language used for the specification of security policies, in the other side, is the “Basic Process Algebra” denoted by \( BPA^\delta_1 \). The language \( ACP^\Phi \) is principally \( ACP \) enhanced with and enforcement operator so that no extra effort
Table 1
Syntax of $BPA^*_δ,1$.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1, P_2 ::= δ \mid 1 \mid a \mid P_1.P_2 \mid P_1 + P_2 \mid P_1.\ast P_2$</td>
<td>$P_1, P_2$ are processes.</td>
</tr>
</tbody>
</table>

is needed to produce a process that respects a given security policy. We prove also that $ACP^φ$ and $ACP$ are two equivalent languages by defining some functions allowing to automatically translate process written in $ACP^φ$ to their equivalent forms in $ACP$. These results provide an elegant technique allowing to automatically enforce security policies on systems specified using $ACP$. Indeed, we need simply to write the enforced system using the embedded enforcement operator of $ACP^φ$ and to translate it to its equivalent in $ACP$.

This paper is structured as follows. We start by the related work in Section 2. Section 3 presents the language used for specifying security policies. Section 4 describes the syntax and the semantics of our calculus used for specifying concurrent programs. In Section 5, we present a formal framework allowing to enforce security policies on concurrent programs. Section 6 gives the main theorem stating the correctness of our method. Section 7 illustrates the approach by an example. Finally, we provide some concluding remark in Section 8.

2. Related work

Many promising formal frameworks for automatic enforcement of security policies in programs have been proposed during the last years. Their goal is to ensure that a program respects a given security policy which generally specifies acceptable executions of the program and can be expressed in terms of access control problems, information flow, availability of resources, confidentiality, etc. (Schneider, 2000). The literature records various techniques for enforcing security policies belonging to mainly two principal classes: static approaches including typing theory (Morisset et al., 1999), Proof Carrying Code (Necula, 1997), and dynamic approaches including reference monitors (Bauer et al., 2002; Ligatti et al., 2005; Martinell and Matteucci, 2007), Java stack inspection (Erlingsson and Schneider, 2000). Static analysis aims at enforcing properties before program execution. In dynamic analysis, however, the enforcement takes place at run time by intercepting critical events during the program execution and halting the latter whenever an action is attempting to violate the property being enforced. Recently, several researchers have explored rewriting techniques (Hamlen et al., 2003) in order to gather advantages of both static and dynamic methods. The idea consists in modifying a program statically, so that the produced version respects the requested requirements. The rewritten program is generated from the original one by adding, when necessary, some tests at some critical points to obtain the desired behaviors.

The first significant effort in the study of Execution Monitoring (EM) security enforcement is presented by Schneider (2000). The important contribution of Schneider (2000) is the characterization of EM-enforceable security policies by security automata. The EMs considered by Schneider are enforcement mechanisms that work by monitoring the execution of untrusted programs and interrupt them whenever execution is about to violate the security policy being enforced. Starting from the work of Schneider described above, Ligatti et al. (2005) have defined another version of security automata. They introduced edit automata which are transformers on the program action stream, rather than simple recognizers.

Examples of formal works related to the rewriting of sequential programs are numerous and some of them can be found in Langar and Mejri (2005), Langar et al. (2010), Ligatti et al. (2005), Mejri and Fujita (2008) and Hakima et al. (2009). However, only few attempts have targeted concurrent programs. This is due to the complexity added by the parallelism operator. Even simple systems become widely complicated when they are executed in parallel (Fokkink, 2000).

3. $BPA^*_δ,1$: the specification language of security policy

We start by giving the syntax and the semantics of $BPA^*_δ,1$, the language that will be used to specify security policies. Since in the proposed enforcement approach we transform the security policy to a
Table 2
Semantics of $BPA_{\delta,1}^*$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(R_a^a)\quad \frac{a \rightarrow 1}{\square}</td>
<td>\frac{a \rightarrow p'}{p}</td>
</tr>
<tr>
<td>$(R_1)\quad \frac{a \rightarrow p'}{p + q}</td>
<td>\frac{a \rightarrow q'}{p + q}</td>
</tr>
<tr>
<td>$(R_2)\quad \frac{a \rightarrow p'}{p^* + q}</td>
<td>\frac{a \rightarrow q'}{p^* + q}</td>
</tr>
<tr>
<td>$(R_3)\quad \frac{a \rightarrow p'}{P_1 \cdot (P_2^* Q)}</td>
<td>\frac{a \rightarrow q'}{P_1 \cdot (P_2^* Q)}</td>
</tr>
</tbody>
</table>

monitor, we reach immediately this goal by choosing $BPA_{\delta,1}^*$ since it is a subset of $ACP^\phi$ the language used to specify programs.

3.1. Syntax of $BPA_{\delta,1}^*$

The syntax of $BPA_{\delta,1}^*$ is presented by the BNF grammar in Table 1, where $a$ is an action in a given finite set $A$, 1 and $\delta$ are two constants representing respectively the successful termination and the deadlock. Furthermore, the operator “$\cdot$” represents the sequential composition: $P_1 \cdot P_2$ is the process that first executes $P_1$ until it terminates, and then $P_2$ starts. The operator “$+$” represents the alternative composition: $P_1 + P_2$ is the process that either executes $P_1$ or $P_2$ but not both of them. The operator “$^*$” represents the iteration. It is a binary version of the Kleene star operator (Bergstra and Ponse, 2001): $P_1^* P_2$ is the process that behaves like $P_1 \cdot (P_1^* P_2) + P_2$. In what follows, we denote by $\mathcal{P}_\phi$ the set of processes generated by $BPA_{\delta,1}^*$.

3.2. Semantics of $BPA_{\delta,1}^*$

The operational semantics of $BPA_{\delta,1}^*$ is defined by the transition relation $\rightarrow \in \mathcal{P}_\phi \times A \times \mathcal{P}_\phi$ shown in Table 2.

3.3. Shortcuts

For the sake of simplicity, we use in what follows the following shortcuts:

$\begin{align*}
A & \doteq \sum_{a \in A} a \\
\neg A & \doteq A - A \\
- a & \doteq A - \{a\} \\
\phi^\omega & \doteq \phi^\delta
\end{align*}$

where $A$ is a subset of $A$, $a \in A$ and “$\doteq$” is the abbreviation symbol. It should be clear from the context when $A$ is a set of atomic actions and $A$ is a process.

3.4. Example

Hereafter, we specify various security properties using $BPA_{\delta,1}^*$.

- $(\neg\text{send})^\omega$: this property states that a program should never executes action send.
- $(\neg\text{read})^\omega ((\text{read.}(\neg\text{send})^\omega))$: this process represents a security property which prohibits a send operation after a read has been executed.

4. Program specification

In this section, we present the formal language that we use to specify concurrent programs. It is a modified version of $ACP$ (Algebra of Communicating Processes) (Baeten, 2005). Process algebra is a
4.1. Syntax

The syntax of $ACP^\phi$ is presented by the BNF grammar given in Table 3. Note that the merge operator $\|_\gamma$ and the communication operator $\_\_\gamma$ are parameterized by a communication function $\gamma$ defined as shown hereafter:

**Definition 4.1 (Communication Function).** A communication function is any commutative and associative function form $A \times A$ to $A$, i.e.: $\gamma : A \times A \to A$ is a communication function if:

1. $\forall a, b \in A.: \gamma(a, b) = \gamma(b, a)$, and
2. $\forall a, b, c \in A.: \gamma(\gamma(a, b), c) = \gamma(a, \gamma(b, c))$.

Constants 1 and $\delta$, atomic actions, operators “+”, “*”, “*" have the same semantic as shown for $L_\psi$. The merge operator “$\|_\gamma$” represents the parallel composition: $P_1\|_\gamma P_2$ is the process that executes $P_1$ and $P_2$ in parallel with the possibility of synchronization according to the function $\gamma$. Notice that the function $\gamma$ can change from one composition to another. For instance, $(P_1\|_1 P_2)\|_2 P_3$ is a valid process, where $\gamma_1 \neq \gamma_2$. The left merge operator “$\|_\gamma$” has the same meaning as the merge operator, but with the restriction that the first step must come from the left process: $P_1\|_\gamma P_2$ is the process that first executes an action in $P_1$ and then run the remaining part of $P_1$ in parallel with $P_2$. The communication merge operator “$\_\gamma$” represents a synchronized composition (communication between processes). Thus, $P_1\_\gamma P_2$ represents the merge of two processes $P_1$ and $P_2$ with the restriction that the first step is a communication between $P_1$ and $P_2$. The unary operator “$\_H$” represents a restriction operator, where $H \subseteq A$: the process $\_H(P)$ can evolve only by executing actions that are not in $H$. The unary operator “$\_\tau$” represents the abstraction operator, where $I$ is any set of atomic actions called internal actions: it abstracts all output action in $I$ by the silent action $\tau$. Finally, the operator “$\_P_\psi$”, where $P_\psi$ is a $BPA^+_s$ process, represents our enforcement operator: $\_P_\psi(P)$ is the processes that can evolve only if $P$ can evolve by an action that does not immediately lead to the violation of the security policy $P_\psi$. In what follows, we denote by $\\mathcal{P}$ the set of processes generated by $ACP^\phi$.

4.2. Semantics

The operational semantics of $ACP^\phi$ is defined by the transition relation $---\in \mathcal{P} \times A \times \mathcal{P}$ shown in Table 5, where the relation “$\equiv$” is defined in Table 4.
5. Formal enforcement of security policies

The principal goal of this research is to define a formal framework allowing to enforce security policies on concurrent programs. To achieve this goal, we used Basic Process Algebra which is suitable for the specification of our targeted class of security policies. In addition, we defined a new version of ACP calculus enhanced with an enforcement operator. Given a program \( P \) and a security policy \( P_\psi \), the desired enforcement can be achieved using the enforcement operator \( \partial_{P_\psi} (P) \). Indeed, the operational semantics of \( \partial_{P_\psi} \) is defined in such a way that \( P \) executes only actions that are allowed by \( P_\psi \). In other words, we have:

\[
P \otimes \psi = \partial_{P_\psi} (P).
\]

Some of the important features of the enforcement operator is that it allows us to enforce locally a given security policy e.g.: \( P, \partial_{P_\psi} (Q), P \mid \partial_{P_\psi} (Q) \), etc. This allows to reduce the overhead induced by the monitoring when the untrusted part of the system are known. Besides, the enforcement operator allows us to enforce different policies in different parts of the system, e.g. \( \partial_{P_\psi} (P), \partial_{P_\psi} (Q), \partial_{P_\psi} (P) \mid \partial_{P_\psi} (Q) \), etc.

5.1. ACP\( ^\phi \) expressivity

The principal difference between the introduced process algebra (ACP\( ^\phi \)) and the original one (ACP), is the introduction of the enforcement operator \( \partial_{P_\psi} \). In this section we prove that the enforcement
Given a process 

Given an integer 

Given an atomic action 

The function 

Given an integer 

Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>Synchronization actions</td>
</tr>
</tbody>
</table>

The idea of transforming a security policy to a process that monitors the system is achieved via the introduction of what we call synchronization actions (commonly used in synchronization logic). The original version of the process is rewritten so that its actions will be controlled and approved by an execution monitor extracted from the security policy that we want to enforce. Let us clarify the idea by a simple example. Suppose that the process is \( a + b \) and the security policy is \( P_\psi = a \) which means that only the action \( a \) is allowed. To monitor the process, the security policy will be transformed to a process containing only synchronization actions where each action \( x \) is replaced by a sequence of two actions \( \tau d, \tau f \) used to capture the start and the end of the action \( x \). Therefore, the security policy \( a \) will be transformed to the monitor \( \tau d, \tau f \). The process, on the other side, will be modified so that it will ask for permission before executing any action. More precisely, any action \( x \) of the process will be substituted by \( x d, x f \) so that it can be executed only when the control action \( x d \) could synchronize with \( \tau d \) of the monitor (i.e., the monitor allows the action). For example, the process \( a + b \) will be transformed to \( a d, a f + b d, b f \). Now, when the two processes are executed in parallel \( (a d, a f + b d, b f) |_{\gamma} \tau d, \tau f \) then they can communicate (synchronize) on their synchronization actions if \( \gamma \) allows it. In order to really enforce the security policy, we need to force the synchronization using the \( \partial_H \) i.e.: \( \partial_H((a d, a f + b d, b f) |_{\gamma} \tau d, \tau f) \), where \( H = \{a d, a f, b d, b f, \tau d, \tau f\} \). Finally, we clean the output stream of the process by abstracting the communication on synchronization actions by the silent action \( \tau \) as follows: \( \partial_H(\tau_i((a d, a f + b d, b f) |_{\gamma} \tau d, \tau f)) \), where \( I = \{\gamma(a d, \tau d), \gamma(a f, \tau f)\} \). To sum up, \( \partial_H(\tau_i((a d, a f + b d, b f) |_{\gamma} \tau d, \tau f)) \) is the enforced version of the process \( a + b \) by the security policy \( a \).

To formalize, this idea, we need to introduce the following notations:

- Given a set of actions \( A \), its corresponding synchronization set, denoted by \( C(A) \) is:

\[
C(A) = \bigcup_{a \in A} \{ad, af, \tau d, \tau f\}.
\]

- Given a process \( P \), we denote by \( C(P) \) the set of its synchronization actions.

- Given an atomic action \( a \in A \), we denote by \( a^\times \) the set:

\[
\bigcup_{a \in A \setminus \{a\}} a.
\]

- Given an integer \( i \) and a set of action \( A \), we denote by \( A^i \) the set containing the actions of \( A \) indexed by \( i \).

- Given an integer \( i \) and a set of action \( A \), we denote by \( I_i \) the set:

\[
\bigcup_{a \in A \setminus \{a\}} \{a | a^\times\}.
\]

- The function \( \gamma_0 \) will be used to denote the communication function defined as follows:

\[
\gamma_0(a, \tau) = \begin{cases} a | \tau & \text{if } a \in A \cup C(A) \\ \delta & \text{else.} \end{cases}
\]

5.1.2. Creating the monitor

Synchronization actions are introduced in a controller process by the function \( \| - \| \) defined in Table 6.
Indeed, we will prove in the next section that the process
\[ (\tau_i, 0) \]
will not be controlled since they are internal actions (they cannot be seen from outside).

5.1.4. Enforced version

Synchronization actions are added in the controlled process by the function \([-\cdot]^H\) defined in Table 7 where:

- \( H \) is a set of trusted function in \( A \) representing trusted actions. Amongst others, the parameter \( H \)
  is introduced to avoid the control of synchronization actions. This situation appear when we have
  more then one security policy. For example, suppose that we want to enforce both \( P_{\psi_1} = a \) and
  \( P_{\psi_2} = a.c \) on \( P = a.b \parallel \gamma.c.d \). To this end, we should execute the process \( \partial_{\psi_1} (\partial_{\psi_2} (P)) \), which
  is transformed to the process \( \partial_{\psi_2} ([\partial_{\psi_1} (P)]_i) \) where \( H_1 \) is the set of synchronization actions
  added to \( P \) after the first transformation, i.e. \( H_1 = C ([\partial_{\psi_1} (P)]_i) \) where \( [\cdot] \) is a shortcut of \([-\cdot]^H\).

5.1.4. Enforced version \( \partial_{\psi_1} (P) \)

Now, we are ready to express the enforcement operator using the standard ACP operators. Indeed, we will prove in the next section that the process \( \partial_{\psi_1} (P) \) is “equivalent” to the process
\[ \partial_{H_1} (\tau_i ([P]_i \parallel [P_{\psi}]_i)) \] for any integer \( i \).

Table 6

\[ \begin{array}{ll}
  \| - \| : & BPA^{\ast,1}_1 \times N \rightarrow ACP^\phi \\
  \| 1 \|_i = 1 & \\
  \| \delta \|_i = \delta & \\
  \| a \|_i = a_i & \\
  \| P_{\psi_1}, P_{\psi_2} \|_i = \| P_{\psi_1} \|_i \parallel \| P_{\psi_2} \|_i & \\
  \| P_{\psi_1} + P_{\psi_2} \|_i = \| P_{\psi_1} \|_i + \| P_{\psi_2} \|_i & \\
  \| P_{\psi_1}^* P_{\psi_2} \|_i = \| P_{\psi_1} \|_i ^* \parallel \| P_{\psi_2} \|_i & \\
\end{array} \]

Table 7

\[ \begin{array}{ll}
  [-\cdot] : & ACP^\phi \times N \times 2^A \rightarrow ACP^\phi \\
  \| 1 \|_i = 1 & \\
  \| \delta \|_i = \delta & \\
  \| a \|_i = \begin{cases}
    a & \text{if } a \in H \cup \{ \tau \} \\
    a_i \cdot a_i & \text{else}
  \end{cases} & \\
  \| P_{\psi_1}, P_{\psi_2} \|_i = \| P_{\psi_1} \|_i \cdot \| P_{\psi_2} \|_i & \\
  \| P_{\psi_1} + P_{\psi_2} \|_i = \| P_{\psi_1} \|_i + \| P_{\psi_2} \|_i & \\
  \| P_{\psi_1}^* P_{\psi_2} \|_i = \| P_{\psi_1} \|_i ^* \parallel \| P_{\psi_2} \|_i & \\
\end{array} \]
6. Main result

The purpose of this section is to prove that enforcement process can be specified in ACP. The initial version of the process in $\text{ACP}^\phi$ and its corresponding version in $\text{ACP}$ are equivalent with respect to the $\tau$-bissimulation defined hereafter.

**Definition 6.1 ($\tau$-bissimulation).** A binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ over processes is a $\tau$-bissimulation, if for all $(P, Q)$ in $S$ we have:

(i) If $P \xrightarrow{a} P'$ then $Q \xrightarrow{a} Q'$ and $(P', Q') \in S$, and

(ii) If $Q \xrightarrow{a} Q'$ then $P \xrightarrow{a} P'$ and $(Q', P') \in S$

where $\xrightarrow{a} = (\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^*$.

**Definition 6.2 ($\leftrightarrow_\tau$).** We define $\leftrightarrow_\tau$ as the biggest $\tau$-bissimulation:

$$\leftrightarrow_\tau = \bigcup \{ S : S \text{ is a } \tau \text{-bissimulation} \}$$

Finally, the following theorem proves the equivalence between the enforcement operator and its transformed form.

**Theorem 6.3 (Soundness).** $\forall P \in \text{ACP}^\phi, \forall P_\psi \in \text{BPA}_{\psi,1}^\ast$, we have:

$$\partial P_\psi(P) \leftrightarrow_\tau \partial H_i(\tau_i(\| P \|_i) \parallel \| P_\psi \|_i))$$

for any $i \in \mathbb{N}$.

6.1. Proof of soundness

**Definition 6.4 (Natural Order).** Let be $P$ and $Q$ two processes in $\mathcal{P}$. We say that $P$ is naturally smaller then $Q$, denoted $P \sqsubseteq_N Q$ if the following condition is satisfied:

$$P + Q = Q.$$

**Proposition 6.5.** Let be $P$ and $Q$ two processes in $\mathcal{P}_\phi$. The $\text{BPA}_{\psi,1}^\ast$ process translation function preserves the natural order over processes:

$$P_\psi \sqsubseteq_N P_\psi' \iff \| P_\psi \|_i \sqsubseteq_N \| P_\psi' \|_i,$$

for any $i \in \mathbb{N}$.

**Proof.**

$$P_\psi \sqsubseteq_N P_\psi'$$

$\iff$ $\| P_\psi \|_i \sqsubseteq_N \| P_\psi' \|_i$, $\square$

$\iff$ $\| P_\psi + P_\psi' \|_i = \| P_\psi' \|_i$ needed

$\iff$ $\| P_\psi + P_\psi' \|_i = \| P_\psi \|_i + \| P_\psi' \|_i$. $\square$

$\iff$ $\| P_\psi \|_i \sqsubseteq_N \| P_\psi' \|_i$. $\square$
Proposition 6.6. Let be \( P \) and \( Q \) two processes in \( \mathcal{P} \). The ACP\(^\phi \) process translation function preserves the natural order over processes:

\[
P \sqsubseteq_N Q \iff [P]_i \sqsubseteq_N [Q]_i,
\]
for any \( i \in \mathbb{N} \).

**Proof.**

\begin{align*}
P \sqsubseteq_N Q & \iff \text{Definition of } \sqsubseteq_N \\ & \iff P + Q = Q \\ & \iff \text{Application of transformation function } \lceil \rceil_i \\ & \iff \text{Table 7: } [P + Q]_i = [P]_i + [Q]_i \\ & \iff [P]_i + [Q]_i = [Q]_i \\ & \iff \text{Definition of } \sqsubseteq_N \\ & \iff [P]_i \sqsubseteq_N [Q]_i. \quad \square
\end{align*}

**Definition 6.7 (Notation).** For \( P \in \mathcal{P} \) and \( \forall a \in \mathcal{A}, \) we define \( P \Downarrow a \) by:

\[
P \Downarrow a \iff \exists Q \mid P \overset{a}{\rightarrow} Q.
\]

We extend the definition of \( \Downarrow \) over a sequence \( \sigma \in \mathcal{A}^* \) as follow:

\[
P \Downarrow \sigma \iff \exists Q \mid P \overset{\sigma}{\rightarrow} Q.
\]

**Proposition 6.8.** For \( P, P' \in \mathcal{P} \) and \( \forall a \in \mathcal{A}, \) we have:

\[
P \overset{a}{\rightarrow} P' \iff [P]_i \overset{a_i \cdot a_{d_j}}{\rightarrow} [P']_i
\]
for any \( i \in \mathbb{N} \).

**Proof.**

\begin{align*}
P \overset{a}{\rightarrow} P' & \iff \text{Definition of } \sqsubseteq_N \\ & \iff a.P' \sqsubseteq_N P \\ & \iff \text{Proposition 6.6} \\ & \iff [a.P']_i \sqsubseteq_N [P]_i \\ & \iff \text{Table 7: } [P.Q]_i = [P]_i.[Q]_i \\ & \iff [a]_i.[P']_i \sqsubseteq_N [P]_i \\ & \iff \text{Table 7: rule } [a]_i \\ & \iff a_{d_j} \cdot a_{d_j} \cdot [P']_i \sqsubseteq_N [P]_i \\ & \iff \text{Definition of } \sqsubseteq_N \\ & \iff [P]_i \overset{a_{d_j} \cdot a_{d_j}}{\rightarrow} [P']_i. \quad \square
\end{align*}
Proposition 6.9. \( \forall P, P' \in \mathcal{P}_\phi \) and \( \forall a \in A \) we have:

\[
P \xrightarrow{a} P' \iff \prod_{i} P_{\psi} \xrightarrow{\tilde{a}_{\psi} \tilde{a}'_{\psi}} \prod_{i} P'_{\psi}
\]
for any \( i \in \mathbb{N} \).

Proof.

\[
P \xrightarrow{a} P' \iff \prod_{i} \text{Definition of } \subseteq \text{ since } \mathcal{P}_\phi \subseteq \mathcal{P}\]

\[
a.P'_{\psi} \subseteq_{N} P_{\psi}
\]

\[
\prod_{i} a.P'_{\psi} \subseteq_{N} \prod_{i} P_{\psi}
\]

\[
\prod_{i} \text{Table 6: } \prod_{i} P.Q \equiv \prod_{i} P \iff Q \prod_{i}
\]

\[
\prod_{i} a.P'_{\psi} \subseteq_{N} \prod_{i} P_{\psi}
\]

\[
\prod_{i} \text{Table 6: rule } \prod_{i} a \prod_{i}
\]

\[
\prod_{i} \text{Definition of } \subseteq_{N}
\]

\[
\prod_{i} \tilde{a}_{\psi} \tilde{a}'_{\psi} \prod_{i} P_{\psi} \subseteq_{N} \prod_{i} P_{\psi}
\]

\[
\prod_{i} \text{Definition of } \subseteq_{N}
\]

\[
\prod_{i} \tilde{a}_{\psi} \tilde{a}'_{\psi} \prod_{i} P'_{\psi} \subseteq_{N} \prod_{i} P'_{\psi}
\]

\[
\prod_{i} \text{Definition of } \subseteq_{N}
\]

Proposition 6.10. \( \forall P, P' \in \mathcal{P} \) we have:

\[
P \xrightarrow{\tau} P' \iff [P] \xrightarrow{\tau} [P']
\]
for any \( i \in \mathbb{N} \).

Proof.

\[
P \xrightarrow{\tau} P' \iff \text{Definition of } \subseteq_{N}
\]

\[
\tau.P' \subseteq_{N} P
\]

\[
\tau.P' \subseteq_{N} [P]
\]

\[
\text{Table 7: } [P.Q] = [P].[Q]
\]

\[
\tau.P' \subseteq_{N} [P]
\]

\[
\text{Table 7: } [\tau] = \tau
\]

\[
\tau.P' \subseteq_{N} [P]
\]

\[
\text{Definition of } \subseteq_{N}
\]

\[
[P] \xrightarrow{\tau} [P']
\]
\[\square\]
Proposition 6.11. \( \forall P, P' \in \mathcal{P} \) and \( \forall P_\psi, P'_\psi \in \mathcal{P}_\psi \) we have:

\[
\partial P_\psi (P) \xrightarrow{a} \partial P_\psi (P') \iff \begin{cases} P \xrightarrow{a} P' \\ P_\psi \xrightarrow{a} P'_\psi. \end{cases}
\]

Proof.

\[
\partial P_\psi (P) \xrightarrow{a} \partial P_\psi (P')
\]

\[\iff \quad \{ \text{Rule } R_{\partial_\psi} \text{ of Table } 5 \} \]

\[
\begin{aligned}
P \xrightarrow{a} P' \\
P_\psi \xrightarrow{a} P'_\psi. \quad \Box
\end{aligned}
\]

Proposition 6.12. \( \forall P, P' \in \mathcal{P} \) and \( \forall P_\psi, P'_\psi \in \mathcal{P}_\psi \) we have:

\[
\partial P_\psi (P) \xrightarrow{a} \partial P_\psi (P') \iff \begin{cases} [P]_i \xrightarrow{a'_i.a'_j} [P']_i \\ \|P_\psi\|_i \xrightarrow{a'_i.a'_j} \|P'_\psi\|_i. \end{cases}
\]

for any \( i \in \mathbb{N} \).

Proof.

\[
\partial P_\psi (P) \xrightarrow{a} \partial P_\psi (P')
\]

\[\iff \quad \{ \text{Proposition } 6.11 \} \]

\[
\begin{aligned}
P \xrightarrow{a} P' \\
P_\psi \xrightarrow{a} P'_\psi.
\end{aligned}
\]

\[\iff \quad \{ \text{Propositions } 6.8 \text{ and } 6.9 \} \]

\[
\begin{aligned}
[P]_i \xrightarrow{a'_i.a'_j} [P']_i \\
\|P_\psi\|_i \xrightarrow{a'_i.a'_j} \|P'_\psi\|_i. \quad \Box
\end{aligned}
\]

Proposition 6.13. \( \forall P, P' \in \mathcal{P} \) and \( \forall P_\psi \in \mathcal{P}_\psi \) we have:

\[
\partial P_\psi (P) \xrightarrow{\tau^*} \partial P_\psi (P') \iff P \xrightarrow{\tau^*} P'.
\]

Proof.

\[
P \xrightarrow{\tau^*} P'
\]

\[\iff \quad \{ \text{Definition of } \xrightarrow{\tau^*} \} \]

\[
P \xrightarrow{\tau} P'' \land P'' \xrightarrow{\tau^*} P'
\]

\[\iff \quad \{ \text{Rule } R_{\partial_\psi}^5 \text{ of Table } 5 \} \]

\[
\partial P_\psi (P) \xrightarrow{\tau} \partial P_\psi (P'') \land P'' \xrightarrow{\tau^*} P'
\]

\[\iff \quad \{ \text{Repeat the above steps and Definition of } \xrightarrow{\tau^*} \} \]

\[
\partial P_\psi (P) \xrightarrow{\tau^*} \partial P_\psi (P'). \quad \Box
\]
Proposition 6.14. \( \forall P, P' \in \mathcal{P} \) and \( \forall P_\psi, P'_\psi \in \mathcal{P}_\psi \) we have:

\[
\partial_{P_\psi}(P) \overset{a}{\rightarrow} \partial_{P'_\psi}(P') \iff \left\{ \begin{array}{l}
\frac{[P]_i}{\partial_{P_\psi}(P)} = \frac{[P']_i}{\partial_{P'_\psi}(P')}, \\
\|P_\psi\|_i = \|P'_\psi\|_i
\end{array} \right.
\]

for any \( i \in \mathbb{N} \).

Proof.

\[
\partial_{P_\psi}(P) \overset{a}{\rightarrow} \partial_{P'_\psi}(P')
\]

\[
\iff \quad \|\text{Definition of } \overset{a}{\rightarrow}\|
\]

\[
\partial_{P_\psi}(P) \overset{\tau^*}{\rightarrow} \partial_{P_\psi}(P_1) \overset{a}{\rightarrow} \partial_{P'_\psi}(P_2) \overset{\tau^*}{\rightarrow} \partial_{P'_\psi}(P')
\]

\[
\iff \quad \|\text{Proposition 6.13}\|
\]

\[
\partial_{P_\psi}(P) \overset{\tau^*}{\rightarrow} \partial_{P_\psi}(P_1) \overset{a}{\rightarrow} \partial_{P'_\psi}(P_2) \overset{\tau^*}{\rightarrow} \partial_{P'_\psi}(P')
\]

\[
P \overset{\tau^*}{\rightarrow} P_1
\]

\[
P_2 \overset{\tau^*}{\rightarrow} P'
\]

\[
\iff \quad \|\text{Proposition 6.10}\|
\]

\[
\partial_{P_\psi}(P) \overset{\tau^*}{\rightarrow} \partial_{P_\psi}(P_1) \overset{a}{\rightarrow} \partial_{P'_\psi}(P_2) \overset{\tau^*}{\rightarrow} \partial_{P'_\psi}(P')
\]

\[
\iff \quad \|\text{Proposition 6.12}\|
\]

\[
\frac{[P]_i}{\partial_{P_\psi}(P)} = \frac{[P]_i}{\partial_{P'_\psi}(P')},
\frac{[P]_i}{\partial_{P_\psi}(P)} = \frac{[P]_i}{\partial_{P'_\psi}(P')},
\frac{\|P_\psi\|_i}{\partial_{P_\psi}(P)} = \frac{\|P'_\psi\|_i}{\partial_{P'_\psi}(P')},
\frac{\|P_\psi\|_i}{\partial_{P_\psi}(P)} = \frac{\|P'_\psi\|_i}{\partial_{P'_\psi}(P')}
\]

\[
\iff \quad \|\text{Definition of } \overset{a}{\rightarrow}\|
\]

\[
\frac{[P]_i}{\partial_{P_\psi}(P)} = \frac{[P]_i}{\partial_{P'_\psi}(P')},
\frac{\|P_\psi\|_i}{\partial_{P_\psi}(P)} = \frac{\|P'_\psi\|_i}{\partial_{P'_\psi}(P')}
\]

\[
\Box
\]

Proposition 6.15. \( \forall P, P' \in \mathcal{P} \) and \( \forall P_\psi, P'_\psi \in \mathcal{P}_\psi \).

If \( \partial_{H_i}(\tau_i([P]_i \|_{\geq 0} \| P_\psi \|_i)) \overset{a}{\rightarrow} \partial_{H_i}(\tau_i([P']_i \|_{\geq 0} \| P'_\psi \|_i)) \), then we have:

1. \( S \downarrow \tau^* \alpha^* \) and \( a \) is not a control action: \( a \notin \mathcal{C}(A) \)
2. \( [P]_i \downarrow \sigma, a \)
3. \( [P]_i \downarrow \sigma^*, d_j \)
4. \( \|P_\psi\|_i \downarrow d_j \|P'_\psi\|_i \)
5. if \( S \) perform a \( \tau \) step, then:
   5.1 \( [P]_i \) perform a \( \tau \) step, or
   5.2 \( [P]_i \|_{\geq 0} \| P_\psi \|_i \) will be synchronized executing an action from \( l_i \)
6. \( [P]_i \downarrow \sigma^*, d_j \quad a \quad d_j \)

for any \( i \in \mathbb{N} \).
Proof. Case 1.

\[
\frac{\partial H_i(\tau_i([P_i]|_{\gamma_0} [P_{\varphi_i} |])) \xrightarrow{a} \partial H_i(\tau_i([P'_i]|_{\gamma_0} [P'_{\varphi_i} |]))}{S \xrightarrow{\rightarrow} T}
\]

\[S \downarrow \tau^* a \tau^*\]

\[
\Rightarrow \quad \downarrow \text{Form of } S: \text{definition of } \partial H_i \text{ and set } H_i \]

\[S \downarrow \tau^* a \tau^* \text{ and } a \not\in C(A).\]

Case 2.

\[
\frac{\partial H_i(\tau_i([P_i]|_{\gamma_0} [P_{\varphi_i} |])) \xrightarrow{a} \partial H_i(\tau_i([P'_i]|_{\gamma_0} [P'_{\varphi_i} |]))}{S \xrightarrow{\rightarrow} T}
\]

\[\Rightarrow \quad \downarrow \text{Case 1}\]

\[S \downarrow \tau^* a \text{ and } a \not\in C(A)\]

\[
\Rightarrow \quad \downarrow \text{Form of } S\]

\[
\begin{align*}
\text{Case 1} & : [P_i] \downarrow \sigma . a \\
\text{Case 2} & : [P_{\varphi_i}] \downarrow \sigma . a \\
\text{Case 3} & : [P_i]|_{\gamma_0} [P_{\varphi_i}] \downarrow a
\end{align*}
\]

\[
\Rightarrow \quad \downarrow \text{Definition of } \sqcap_i : \text{[P_{\varphi_i}] is formed only by control actions}\]

\[
\begin{align*}
\text{Case 1} & : [P_i] \downarrow \sigma . a \\
\text{Case 2} & : \text{Impossible: [P_{\varphi_i}] cannot execute } a \not\in C(A) \\
\text{Case 3} & : [P_i]|_{\gamma_0} [P_{\varphi_i}] \downarrow a
\end{align*}
\]

\[
\Rightarrow \quad \downarrow \text{[P_i]|_{\gamma_0} [P_{\varphi_i}] is under the scope of } \tau_i \text{ operator and definitions of } \gamma_0, I_i\]

\[
\begin{align*}
\text{Case 1} & : [P_i] \downarrow \sigma . a \\
\text{Case 3} & : \text{Impossible: every synchronization action, } \gamma_0(a_p, a_{P_{\varphi_i}}) \text{ between } [P_i] \text{ and } [P_{\varphi_i}] \text{ belongs to } I_i
\end{align*}
\]

\[
\Rightarrow \quad \downarrow \text{Conclusion}\]

\[\Rightarrow \quad [P_i] \downarrow \sigma . a.\]

Case 3.

\[
\frac{\partial H_i(\tau_i([P_i]|_{\gamma_0} [P_{\varphi_i} |])) \xrightarrow{a} \partial H_i(\tau_i([P'_i]|_{\gamma_0} [P'_{\varphi_i} |]))}{S \xrightarrow{\rightarrow} T}
\]

\[\Rightarrow \quad \downarrow \text{Case 2}\]

\[\Rightarrow \quad [P_i] \downarrow \sigma . a\]

\[\Rightarrow \quad \downarrow \text{Definition of } \sqcap_i : \text{Actions of } P \text{ transformed to } a_d^i . a . a_d^i\]

\[\Rightarrow \quad [P_i] \downarrow \sigma . a_d^i . a.\]

Case 4.

\[
\frac{\partial H_i(\tau_i([P_i]|_{\gamma_0} [P_{\varphi_i} |])) \xrightarrow{a} \partial H_i(\tau_i([P'_i]|_{\gamma_0} [P'_{\varphi_i} |]))}{S \xrightarrow{\rightarrow} T}
\]

\[\Rightarrow \quad \downarrow \text{Case 3}\]
\[ [P]_i \downarrow \sigma'.a_d^i.a \]
\[ \Rightarrow \quad \text{Case 2 and } [P]_i \text{ is under the scope of } \partial_{\Pi_l} \text{ operator } \]
\[ \left\{ \begin{array}{l}
[P]_i \downarrow \sigma'.a_d^i.a \\
[P]_i \text{ should synchronize with another process to be able to execute } a_d^i
\end{array} \right. \]
\[ \Rightarrow \quad \text{∥ Form of } S \text{ ∥} \]
\[ \left\{ \begin{array}{l}
[P]_i \downarrow \sigma'.a_d^i.a \\
[P]_i \text{ should synchronize with } [P_{\psi}]_i \text{ to be able to execute } a_d^i
\end{array} \right. \]
\[ \Rightarrow \quad \text{∥ Definition of } \| \|_i \text{ ∥} \]
\[ [P_{\psi}]_i \xrightarrow{\sigma_d^i} \sigma \cdot [P_{\psi}]_i. \]

**Case 5.**

\[ \partial_{\Pi_l}(\tau_0([P]_i||y_0 [P_{\psi}]_i)) \xrightarrow{\tau} Q \]
\[ \Rightarrow \quad \text{∥ Definition of } I_l \text{ and operators } \tau_0 \text{ and } \partial_{\Pi_l} \text{ ∥} \]
\[ \left\{ \begin{array}{l}
\text{Case 1 } [P]_i \text{ synchronizes internally and follow a } \tau \text{ step} \\
\text{Case 2 } [P_{\psi}]_i \text{ follow a } \tau \text{ step} \\
\text{Case 3 } [P]_i||y_0 [P_{\psi}]_i \text{ synchronizes by executing an action from } I_l
\end{array} \right. \]
\[ \Rightarrow \quad \text{∥ Definition of } \| \|_i; \text{ Case 2 impossible } ∥ \]
\[ \left\{ \begin{array}{l}
\text{5.1 } [P]_i \text{ synchronizes internally and follow a } \tau \text{ step} \\
\text{5.2 } [P]_i||y_0 [P_{\psi}]_i \text{ synchronizes by executing an action from } I_l.
\end{array} \right. \]

**Case 6.**

\[ \partial_{\Pi_l}(\tau_0([P]_i||y_0 [P_{\psi}]_i)) \xrightarrow{a} \partial_{\Pi_l}(\tau_0([P']_i||y_0 [P_{\psi}]_i)) \]
\[ \Rightarrow \quad \text{∥ Case 3 } ∥ \]
\[ [P]_i \downarrow \sigma'.a_d^i.a \]
\[ \Rightarrow \quad \text{∥ Actions of } P \text{ transformed to } a_d^i.a_d^j, \text{ Case 4 and } S \xrightarrow{\ast \ast} T \text{ ∥} \]
\[ [P]_i \downarrow \sigma'.a_d^i.a_d^j. \quad \Box \]

**Lemma 6.16.** \( \forall P, P' \in \mathcal{P} \text{ and } \forall P_{\psi}, P_{\psi}' \in \mathcal{P}_{\psi} \text{ we have:} \)
\[ \partial_{P_{\psi}}(P) \xrightarrow{a} \partial_{P_{\psi}'}(P') \Leftrightarrow \partial_{\Pi_l}(\tau_0([P']_i||y_0 [P_{\psi}]_i)) \xrightarrow{a} \partial_{\Pi_l}(\tau_0([P]_i||y_0 [P_{\psi}]_i)) \]
for any \( i \in \mathbb{N}. \)

**Proof.**

\[ \Rightarrow \quad \text{∥ Proposition 6.14 } ∥ \]
\[ \left\{ \begin{array}{l}
\partial_{P_{\psi}}(P) \xrightarrow{a} \partial_{P_{\psi}'}(P') \\
[P]_i \xrightarrow{a_d^i.a_d^j} [P']_i \\
[P_{\psi}]_i \xrightarrow{a_d^i.a_d^j} [P_{\psi}']_i
\end{array} \right. \]
$$\Rightarrow \quad \text{Definition of } \rightarrow \bullet$$

\[
\begin{align*}
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{1|0}^\psi \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\partial|1}^\psi \text{ where } a_0 | a_1' \in I, \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\partial|0}^\psi \text{ where } \tau \notin H, \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&P_1 \xrightarrow{a} P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\partial|0}^\psi \text{ where } a \notin I, \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\partial|0}^\psi \text{ and } a \notin H, \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\psi|0}^\psi \text{ where } a_0 | a_1' \in I, \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\psi|0}^\psi \text{ where } a_0 | a_1' \in I, \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\psi|0}^\psi \text{ where } a_0 | a_1' \in I, \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\psi|0}^\psi \text{ where } a_0 | a_1' \in I, \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\psi|0}^\psi \text{ where } a_0 | a_1' \in I, \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\psi|0}^\psi \text{ where } a_0 | a_1' \in I, \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&P_2 \xrightarrow{a'} [P']^i \\
&\text{Rule } R_{\psi|0}^\psi \text{ where } a_0 | a_1' \in I.
\end{align*}
\]
\[
\begin{align*}
\{ & \partial H_i (\tau_i (\textit{P}_i |_\textit{y}_0 \parallel \textit{P}_\psi |_i)) \xrightarrow{r} \partial H_i (\tau_i (\textit{P}_i |_\textit{y}_0 \parallel \textit{P}_\psi |_i)) \\
& \partial H_i (\tau_i (\textit{P}_1 |_\textit{y}_0 \parallel \textit{P}_i ^* , \parallel \textit{P}_\psi ^* |_i)) \xrightarrow{a} \partial H_i (\tau_i (\textit{P}_2 |_\textit{y}_0 \parallel \textit{P}_i ^* , \parallel \textit{P}_\psi ^* |_i)) \\
& \tau_i (\textit{P}_2 |_\textit{y}_0 \parallel \textit{P}_i ^* , \parallel \textit{P}_\psi ^* |_i) \xrightarrow{r} \tau_i (\textit{P}_1 |_\textit{y}_0 \parallel \textit{P}_\psi ^* |_i) \\
\} \quad \text{\small Rule \textit{R}_\partial H_i \text{, where } \tau \notin H_i \} \\
\Rightarrow \quad \{ & \partial H_i (\tau_i (\textit{P}_i |_\textit{y}_0 \parallel \textit{P}_\psi |_i)) \xrightarrow{r} \partial H_i (\tau_i (\textit{P}_1 |_\textit{y}_0 \parallel \textit{P}_\psi |_i)) \\
& \partial H_i (\tau_i (\textit{P}_1 |_\textit{y}_0 \parallel \textit{P}_i ^* , \parallel \textit{P}_\psi ^* |_i)) \xrightarrow{a} \partial H_i (\tau_i (\textit{P}_2 |_\textit{y}_0 \parallel \textit{P}_i ^* , \parallel \textit{P}_\psi ^* |_i)) \\
& \partial H_i (\tau_i (\textit{P}_2 |_\textit{y}_0 \parallel \textit{P}_i ^* , \parallel \textit{P}_\psi ^* |_i)) \xrightarrow{r} \partial H_i (\tau_i (\textit{P}_1 |_\textit{y}_0 \parallel \textit{P}_\psi ^* |_i)) \\
\} \quad \text{\small Definition of } \xrightarrow{r} \} \\
\Rightarrow \quad \partial H_i (\tau_i (\textit{P}_i |_\textit{y}_0 \parallel \textit{P}_\psi |_i)) \xrightarrow{a} \partial H_i (\tau_i (\textit{P}_1 |_\textit{y}_0 \parallel \textit{P}_\psi |_i)) \\
\quad \text{\small Proposition 6.15, Case 6} \\
\exists P' \in \mathcal{P} : [\textit{P}]_i \xrightarrow{\sigma.a_i.a_i^j} [\textit{P'}]_i \\
\Rightarrow \quad \text{\small Proposition 6.15, Case 5.1} \\
\exists P', Q \in \mathcal{P} : [\textit{P}]_i \xrightarrow{\tau^*} [\textit{Q}]_i \xrightarrow{d_i.a_i.a_i^j} [\textit{P'}]_i \\
\Rightarrow \quad \text{\small Propositions 6.10 and 6.13} \\
\begin{align*}
[\textit{P}]_i \xrightarrow{\tau^*} [\textit{Q}]_i \xrightarrow{d_i.a_i.a_i^j} [\textit{P'}]_i \\
\Rightarrow & \partial P_i (\textit{P}) \xrightarrow{\tau^*} \partial P_i (\textit{Q}) \\
\Rightarrow \quad \text{\small Proposition 6.8} \\
[\textit{P}]_i \xrightarrow{\tau^*} [\textit{Q}]_i \xrightarrow{d_i.a_i.a_i^j} [\textit{P'}]_i \\
\Rightarrow & \text{\small Proposition 6.15, Case 4} \\
\begin{align*}
\{ & \textit{Q} \xrightarrow{a} \textit{P'} \\
& \partial P_i (\textit{P}) \xrightarrow{\tau^*} \partial P_i (\textit{Q}) \\
\} \quad \text{\small Proposition 6.9} \\
\{ & \textit{Q} \xrightarrow{a} \textit{P'} \\
& \textit{P}_i \xrightarrow{a} \textit{P'} \\
& \partial P_i (\textit{P}) \xrightarrow{\tau^*} \partial P_i (\textit{Q}) \\
\} \quad \text{\small Rule \textit{R}_\partial P_i } \\
\Rightarrow \quad \{ & \partial P_i (\textit{Q}) \xrightarrow{a} \partial P_i (\textit{P'}) \\
& \partial P_i (\textit{P}) \xrightarrow{\tau^*} \partial P_i (\textit{Q}) \\
\} \\
\end{align*}
\end{align*}
\]
\[ \Rightarrow \quad \{ \text{Definition of } \xrightarrow{a} \} \]
\[ \partial_{p_\psi}(P) \xrightarrow{a} \partial_{p_\psi}(P'). \quad \Box \]

6.1.1. Proof of Theorem 6.3

We will prove that the set of all couple of the form \((\partial_{p_\psi}(P), \partial_{H}(\tau_{i}(\tau_{l}(\tau_{p})(\psi)))))\) is a \(\tau\)-bissimulation, i.e.,

\[
\bigcup_{p \in P, \psi \in P_\psi} \{ \partial_{p_\psi}(P), \partial_{H}(\tau_{i}(\tau_{l}(\tau_{p})(\psi))) \} \subseteq \leftrightarrow_\tau.
\]

We have just to prove that any couple of this set, always evolves to another couple in the same set.

Case i:
\[
\partial_{p_\psi}(P) \xrightarrow{a} \partial_{p_\psi}(P')
\]
\[ \Rightarrow \quad \{ \text{Lemma 6.16} \} \]
\[ \partial_{H}(\tau_{i}(\tau_{l}(\tau_{p})(\psi))) \xrightarrow{a} \partial_{H}(\tau_{i}(\tau_{l}(\tau_{p})(\psi))) \]

Case ii:
\[
\partial_{H}(\tau_{i}(\tau_{l}(\tau_{p})(\psi))) \xrightarrow{a} \partial_{H}(\tau_{i}(\tau_{l}(\tau_{p})(\psi)))
\]
\[ \Rightarrow \quad \{ \text{Lemma 6.16} \} \]
\[ \partial_{p_\psi}(P) \xrightarrow{a} \partial_{p_\psi}(P'). \quad \Box \]

7. Example

Hereafter we show how our techniques works on a simple example. Consider the following program:

\[ P = \text{read.copy} || \text{y0.open.transform} || \text{y0.write.send}. \]

Which is composed of two concurrent processes, and the following security property:

\[ P_\psi : (\neg \text{read})^*(\text{read.}(\neg \text{send})^\omega). \]

In order to enforce \(P_\psi\) on the program \(P\), we should execute the process:

\[ \partial_{p_\psi}(\text{read.copy} || \text{y0.write.send}). \]

Which is equivalent to execute the process:

\[ \partial_{H_0}(\tau_0(\text{read.copy} || \text{y0.write.send})) \]

where \(H_0 = C(A)^1\) and \(I_0 = \bigcup_{\alpha \in H_0} \{ \alpha | \bar{\alpha} \}\), where \(A = \{ \text{read, copy, write, send} \}\).

In order to simplify the presentation, the letter \(c\) denotes \(\text{copy}\), the letter \(r\) denotes \(\text{read}\), the letter \(w\) denotes \(\text{write}\) and the letter \(s\) denotes \(\text{send}\).

Firstly, we should calculate \(\text{read.copy} || \text{y0.w.s} \) and \((\neg \text{read})^*(\text{read.}(\neg \text{send})^\omega)):\n
\[
\text{read.copy} || \text{y0.w.s} = \text{read.} \cdot \text{r.} \cdot \text{c.} \cdot \text{e.} \cdot \text{f.} || \text{y0.} \cdot \text{w.} \cdot \text{w.} \cdot \text{f.} \cdot \text{d.} \cdot \text{d.} \cdot \text{f.} \cdot \text{f.} \cdot \text{f.} \cdot \text{f.} \\
(\neg \text{read})^*(\text{read.}(\neg \text{send})^\omega) = (\text{f.} \cdot \text{f.} \cdot \text{f.})^* (\text{f.} \cdot \text{f.} \cdot \text{f.} \cdot \text{f.} \cdot \text{f.} \cdot \text{f.}).
\]

We obtain the following process:

\[
\partial_{H_0}(\tau_0(\text{read.} \cdot \text{r.} \cdot \text{c.} \cdot \text{e.} \cdot \text{f.} || \text{y0.} \cdot \text{w.} \cdot \text{w.} \cdot \text{f.} \cdot \text{d.} \cdot \text{d.} \cdot \text{f.} \cdot \text{f.} \cdot \text{f.} \cdot \text{f.})).
\]

\[ \phi^c \]
For example, developing the following sequence of actions: read, write, send. Note that this sequence violates the property $P_\varphi$, and the program should be blocked before executing the action send.

$$\partial_{\phi_0}(\tau_0((r_f . r_f . c_d . c_f || w_0 . w_f . w_f . s_d . s_f || w_0 (F_{c_f} . F_{f_f} . (\delta_{d_{c}} . \delta_{f_{c}})_{\omega}))))
\xrightarrow{\tau} \{ \text{Rules } R_{\phi}^c, R_{\varphi}^c \text{ and } R_{\omega}, \text{ where } \gamma_0(r_f, F_{\varphi}) = r_f, F_{\varphi} \in I_0 \}
$$

$$\partial_{\phi_0}(\tau_0((r_f . r_f . c_d . c_f || w_0 . w_f . s_d . s_f || w_0 (F_{c_f} . F_{f_f} . (\delta_{d_{c}} . \delta_{f_{c}})_{\omega}))))
\xrightarrow{\tau} \{ \text{Rules } R_{\phi}^c, R_{\varphi}^c \text{ and } R_{\omega}, \text{ where } r \not\in I_0 \}
$$

$$\partial_{\phi_0}(\tau_0((c_d . c_f || w_0 . w_f . s_d . s_f || w_0 (F_{c_f} . F_{f_f} . (\delta_{d_{c}} . \delta_{f_{c}})_{\omega}))))
\xrightarrow{\tau} \{ \text{Rules } R_{\phi}^c, R_{\varphi}^c \text{ and } R_{\omega}, \text{ where } \gamma_0(w_f, \delta_{d_{c}}) = w_f, \delta_{d_{c}} \in I_0 \}
$$

$$\partial_{\phi_0}(\tau_0((c_d . c_f || w_0 . w_f . s_d . s_f || w_0 (F_{c_f} . F_{f_f} . (\delta_{d_{c}} . \delta_{f_{c}})_{\omega}))))
\xrightarrow{\tau} \{ \text{Rules } R_{\phi}^c, R_{\varphi}^c \text{ and } R_{\omega}, \text{ where } r \not\in I_0 \}
$$

$$\partial_{\phi_0}(\tau_0((c_d . c_f || w_0 . w_f . s_d . s_f || w_0 (F_{c_f} . F_{f_f} . (\delta_{d_{c}} . \delta_{f_{c}})_{\omega}))))
\xrightarrow{\tau} \{ \text{Rules } R_{\phi}^c, R_{\varphi}^c \text{ and } R_{\omega}, \text{ where } \gamma_0(w_f, \delta_{d_{c}}) = w_f, \delta_{d_{c}} \in I_0 \}
$$

As we can see the subprocess $(c_d . c_f || w_0 . s_d . s_f)$ cannot execute the action send, because it should firstly synchronize with another process to execute the action $s_d$, which is impossible.

8. Conclusion and future work

This paper presents a formal automatic approach for the enforcement of security policies in concurrent programs. We used a variant of BPA to specify security policy and we have enhanced ACP with a special enforcement operator $\partial_{\varphi}$. We proved that ACP* and ACP are equivalent by showing that $\forall \Phi \in ACP^\Phi$, $\forall \Phi \in BPA_{S,1}$, we have:

$$P \otimes \Phi = \partial_{\varphi}(P) \leftrightarrow_{\text{def}} \partial_{\varphi}(\tau_k(\Phi || \forall \Phi || \Phi)).$$

This results provided an elegant technique allowing to automatically enforce security policies on systems specified using ACP. Indeed, we need simply to write the enforced system using the embedded enforcement operator of ACP* and to take it to its equivalent in ACP given by the previous formula. This result allows us to apply the approach to real language such as Java.

As future work, we want to extend $BPA_{S,1}$ to handle exceptions i.e.: we give the end user the possibility to specify the actions to be executed when the security policy is about to be violated instead of simply halting the program. We want also to optimize the enforced program so that we reduce as much as we can the treatment added by the enforcement operator.

References


