REWRITING-BASED SECURITY ENFORCEMENT OF CONCURRENT SYSTEMS
A Formal Approach

Mahjoub Langar, Mohamed Mejri
LSFM Research Group, Computer Science Department, Laval University, Québec, Québec, Canada

Kamel Adi
LRSI Research Group, Computer Science Department, University of Quebec in Outaouais, Gatineau, Québec, Canada

Keywords: Language based security, Runtime verification, Concurrent systems, Process algebra, Formal verification.

Abstract: Program security enforcement is designed to ensure that a program respects a given security policy, which generally specifies the acceptable executions of that. In general, the enforcement is achieved by adding some controls (tests) inside the target program or process. The major drawback of existing techniques is either their lack of precision or their inefficiency, especially those dedicated for concurrent languages. This paper proposes an efficient algebraic and fully automatic approach for security program enforcement: given a concurrent program P and a security policy ϕ, it automatically generates another program P' that satisfies ϕ and behaves like P, except that it stops when P tries to violate the security policy ϕ.

1 INTRODUCTION

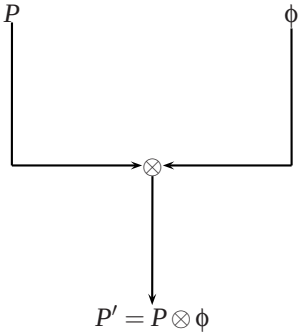
Until today, fully secure computer systems are still a distant dream, mainly due to the subtleness and the complexity of the problem. However, one of the current and promising avenues of research for securing computer systems is the development of formal frameworks for automatic enforcement of security policies in programs. The goal of those approaches is to ensure that a program respects a given security policy which generally specifies acceptable executions of the program. Security policies can be expressed in terms of access control problems, information flow, availability of resources, confidentiality, etc. (Schneider, 2000) and the literature records various techniques for enforcing security policies. Thus, we mainly distinguish two principal classes: static approaches including typing theory (Morrisett 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 runtime 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 (K. Hamlen and Schneider, 2003) in order to gather advantages of both static and dynamic methods. The idea consists in modifying statically a program, 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 behavior.

The literature record many formal works (Langar et al., 2007; Langar and Mejri, 2005; Ligatti et al., 2005; Mejri and Fujita, 2008; Ould-Slimane et al., 2009) related to the rewriting of sequential programs, 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). However, the research community knows that this challenge cannot be ignored for a long time due to the urgent need for securing concurrent systems which are pervasive in every sphere of human activity.

Langar M., Mejri M. and Adi K. (2010).
REWRITING-BASED SECURITY ENFORCEMENT OF CONCURRENT SYSTEMS - A Formal Approach.
In *Proceedings of the International Conference on Security and Cryptography*, pages 66-74
DOI: 10.5220/0002996100660074

This paper proposes an algebraic and fully 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, this paper formally defines a syntactic operator $\otimes$ that takes as input a process P and a security policy ϕ and generates $P' = P \otimes \phi$, a new process that respects the following conditions:

- $P' \mid\sim \phi$, i.e., P' "satisfies" the security policy ϕ.
- $P' \sqsubseteq P$, i.e., behaviours of $P \otimes \phi$ are subset of the behaviours of P.
- $\forall\, Q : ((Q \mid\sim \phi) \wedge (Q \sqsubseteq P)) \Rightarrow Q \sqsubseteq P'$, i.e., all good behaviours of P are in $P \otimes \phi$.



For the specification of concurrent systems, the suggested model uses an extended version of ACP "Algebra for Communicating Process" (Baeten, 2005) denoted by ACP^{ϕ}. Moreover, the logic used for the specification of security policies is an extended version of regular languages denoted by L_{φ}. The language ACP^{ϕ} is defined so that the enforcement operator is embedded in it. Furthermore, we define a set of translation functions that allow to express our enforcement operator in terms of standard operator of the process algebra *ACP*.

This paper is structured as follows. Section 2 presents a logic for specifying security policies. Section 3 describes the syntax and the semantics of our calculus used for specifying concurrent programs. In Section 4 we present a formal framework based on the introduced logic and process algebra for security policies enforcement on programs. Section 5 gives the main theorem stating the correctness of our method. Section 6 illustrates our method by An example. Finally, Section 7 provides concluding remarks.

2 SECURITY POLICY SPECIFICATION

The purpose of this section is to define an adequate logic for specifying the required security policies. The important requested features of this logic are:

- the ability for specifying linear and temporal properties. We are interested by properties that could not be always statistically verified. Therefore we expect that the program will be monitored at runtime so that the violation of the requested security property could be forbidden. To that end, and since at runtime, we can see only one branch of the program at a given time, then the expected logic needs to be linear.
- suitable for safety properties. Safety properties are the class of properties that can be dynamically enforced. By safety we mean that something bad will never happen during the execution of a program.
- the ability for specifying ω-properties (infinite properties). An execution of a program could generate an infinite sequence of actions and it is important to be able to monitor these behaviours.

For the above stated reasons and others that will be clarified later, the adopted logic is based on the extended regular expressions (Sen and Rosu, 2003), enhanced with the ability for specifying infinite behaviours.

2.1 Syntax

The syntax of the proposed logic is presented by the BNF grammar in Table 1, where a is an action in a given finite set $\mathcal{A}$, tt and ff denote the Boolean constants and 1 denotes an empty sequence of actions. Furthermore, the logic contains standard propositional connectives ($\neg, \wedge$ and $\vee$) and a temporal operator (.) denoting the operation sequencing. In the sequel, we note L_{φ} the set of formulas that can be specified in our logic.

For reasons that will be detailed below, we consider a deterministic Kleene operator, i.e.: only formulas φ_1 and φ_2 such that $\upsilon(\varphi_1) \cap \upsilon(\varphi_2) = \emptyset$ are allowed in the form $\varphi_1^* \varphi_2$. Intuitively, $\upsilon(\varphi)$ is the set of the first actions allowed by the formula φ. For example, $\upsilon(a.b.c) = \{a\}$, $\upsilon((a \vee b).c) = \{a, b\}$ and $\upsilon(a.(b \vee c)) = \{a\}$. More formally, the function υ is defined as follows:

$\upsilon : L_{\varphi} \rightarrow 2^{\mathcal{A}}$		
$\upsilon(tt)$	=	$\mathcal{A}$
$\upsilon(ff)$	=	$\emptyset$
$\upsilon(1)$	=	$\emptyset$
$\upsilon(a)$	=	$\{a\}$
$\upsilon(\varphi_1^* \varphi_2)$	=	$\upsilon(\varphi_1) \cup \upsilon(\varphi_2)$
$\upsilon(\varphi_1 \vee \varphi_2)$	=	$\upsilon(\varphi_1) \cup \upsilon(\varphi_2)$
$\upsilon(\varphi_1 \wedge \varphi_2)$	=	$\upsilon(\varphi_1) \cup \upsilon(\varphi_2)$
$\upsilon(\neg\varphi)$	=	$\mathcal{A} \setminus \upsilon(\varphi)$
$\upsilon(\varphi_1.\varphi_2)$	=	$\upsilon(\varphi_1) \cup (o(\varphi_1) \ominus \upsilon(\varphi_2))$

where the function $o(\varphi)$ allows to know whether the language accepted by the formula contain the empty sequence and it is defined as follows :

$o : L_\varphi \to \{0,1\}$		
$o(tt)$	$=$	1
$o(ff)$	$=$	0
$o(1)$	$=$	1
$o(a)$	$=$	0
$o(\varphi_1.\varphi_2)$	$=$	$o(\varphi_1) \times o(\varphi_2)$
$o(\varphi_1 \vee \varphi_2)$	$=$	$max(o(\varphi_1), o(\varphi_2))$
$o(\varphi_1^* \varphi_2)$	$=$	$o(\varphi_2)$
$o(\neg\varphi)$	$=$	$(o(\varphi)+1)\ mod\ (2)$

and the function $\ominus$ is defined as follows:

$$\begin{array}{rccc} \ominus : & \{0,1\} \times 2^{\mathcal{A}} & \longrightarrow & 2^{\mathcal{A}} \\ & (0,S) & \mapsto & \emptyset \\ & (1,S) & \mapsto & S \end{array}$$

Table 1: Syntax of L_φ.

$$\varphi_1, \varphi_2 \quad ::= \quad tt \mid ff \mid 1 \mid a \mid \varphi_1.\varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \varphi_1^* \varphi_2$$

2.2 Semantics

Let $\mathcal{T}$ be the monoid $(\mathcal{A}, ., \varepsilon)$, the semantics of L_φ is defined by the function $[\![\ ]\!] : L_\varphi \to \mathcal{T}$ as shown in Table 2. Intuitively, any trace respects the formula *true*. No trace respects the formula *false* and only the empty trace (ε) respects the formula 1. Only traces that have prefixes respecting φ_1 and suffixes respecting φ_2 respect $\varphi_1.\varphi_2$. Only traces that respect φ_1 or φ_2 respect $\varphi_1 \vee \varphi_2$. Only traces that respect both φ_1 and φ_2 respect $\varphi_1 \wedge \varphi_2$. A trace respects $\varphi_1^* \varphi_2$ if either it respects φ_2 or it has a prefix that respects φ_1 and a suffix that respect $\varphi_1^* \varphi_2$. Finally, a trace respects $\neg\varphi$, if it does not respect φ.

2.3 Shortcuts

For the sake of simplicity, we define the following shortcuts where A is a subset of $\mathcal{A}$, $a \in \mathcal{A}$ and "$\doteq$" is the abbreviation symbol:

$$\begin{array}{rcl} A & \doteq & \bigvee_{a \in A} a \\ -A & \doteq & \mathcal{A} - A \\ -a & \doteq & \mathcal{A} - \{a\} \\ - & \doteq & \mathcal{A} - \emptyset \\ \varphi^\omega & \doteq & \varphi^* ff \end{array}$$

2.4 Example

Hereafter, we specify various properties using the logic L_φ.

- $(-send)^\omega$: this formula is satisfied by infinite traces that do not contain the action *send*.
- $(-read)^*(read.(-send)^\omega)$: this formula represents a security property which prohibits a *send* operation after a *read* has been executed.

2.5 Derivative

In this subsection, we adapt the notion of Brzozowski derivatives or "residuals" (see (Brzozowski, 1964; Owens et al., 2009; Sen and Rosu, 2003) for more detail) to our logic. The notion of derivatives expresses the idea of "program evolution", in the sense that it gives the remaining part of a program after the execution of a given action. Similarly, the derivative of a formulae φ with respect to an atomic action a, denoted by $[\varphi]_a$, is the remaining part of φ that needs to be respected by an arbitrary trace t so that $a.t$ respects φ. More formally, the definition of the derivative is as shown in Table 3.

Hereafter, we extend the definition of derivative to an arbitrary trace as follows:

Definition 1. *Let φ be a formula in L_φ, ξ a trace in $\mathcal{T}$, and a an action in $\mathcal{A}$. The derivative of φ with respect to ξ denoted by $[\varphi]_\xi$ is defined as follows:*

- $[\varphi]_\varepsilon = \varphi$
- $[\varphi]_{a.\xi} = [[\varphi]_a]_\xi$

Definition 2. *Let φ be a formula in L_φ, ξ a trace in $\mathcal{T}$, and a an action in $\mathcal{A}$.*

- *We say that a trace ξ satisfies the formula φ, denoted by $\xi \vDash \varphi$, if $\xi \in [\![\varphi]\!]$.*
- *We say that a trace ξ prefix-satisfies a formulae φ, denoted by $\xi|\sim \varphi$, if there exists a trace ξ' such that $\xi.\xi' \vDash \varphi$.*

Proposition 2.1. *Let φ be a formula in L_φ and ξ a trace in $\mathcal{T}$. The following notations are equivalent:*

- $\xi|\sim \varphi$
- $[\varphi]_\xi \neq ff$
- $\exists x \mid \xi.x \vDash \varphi$

Proof. The proof follows directly from the definition of $|\sim$ and $\vDash$. □

Table 2: Semantics of L_φ formulae.

$$
\begin{array}{rcl}
[\![tt]\!] & = & \mathcal{T} \\
[\![ff]\!] & = & \emptyset \\
[\![1]\!] & = & \{\varepsilon\} \\
[\![a]\!] & = & \{a\} \\
[\![\varphi_1.\varphi_2]\!] & = & \{\xi_1.\xi_2 | \xi_1 \in [\![\varphi_1]\!] \text{ and } \xi_2 \in [\![\varphi_2]\!]\} \\
[\![\varphi_1 \vee \varphi_2]\!] & = & [\![\varphi_1]\!] \cup [\![\varphi_2]\!] \\
[\![\varphi_1 \wedge \varphi_2]\!] & = & [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![\varphi_1^* \varphi_2]\!] & = & \begin{cases} [\![\varphi_1]\!]^* \cup \{\xi_1.\xi_2 | \xi_1 \in [\![\varphi_1]\!]^* \text{ and } \xi_2 \in [\![\varphi_2]\!]\} \text{ If } [\![\varphi_2]\!] \neq \emptyset \\ [\![\varphi_1]\!]^\omega \text{ elsewhere} \end{cases} \\
[\![\neg\varphi]\!] & = & \mathcal{T} \setminus [\![\varphi]\!]
\end{array}
$$

Table 3: Derivative of a formula.

$$
\begin{array}{rcl}
\multicolumn{3}{c}{[-]_- : L_\varphi \times \mathcal{A} \rightarrow L_\varphi} \\
\hline
[tt]_a & = & tt \\
[ff]_a & = & ff \\
[1]_a & = & ff \\
[a]_a & = & 1 \\
[b]_a & = & ff \quad \text{Where } a \neq b \\
[\varphi_1.\varphi_2]_a & = & \begin{cases} [\varphi_1]_a.\varphi_2 \vee [\varphi_2]_a \; if \; o(\varphi_1) = 1 \\ [\varphi_1]_a.\varphi_2 \qquad if \; o(\varphi_1) = 0 \end{cases} \\
[\varphi_1 \vee \varphi_2]_a & = & [\varphi_1]_a \vee [\varphi_2]_a \\
[\varphi_1^* \varphi_2]_a & = & [\varphi_1]_a.\varphi_1^* \varphi_2 \vee [\varphi_2]_a \\
[\neg\varphi]_a & = & \neg[\varphi]_a
\end{array}
$$

3 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) which is developed by Jan Bergstra and Jan Willem Klop in 1982 (Baeten, 2005). This new algebra, denoted ACP^ϕ, has the particularity of explicitly handling the monitoring concept through its operator "∂_ϕ^ξ". For instance, the process $\partial_\phi^\xi(P)$ can execute only actions that prefix-satisfies the security policy ϕ. This process algebra is a powerful language for specifying and studying concurrent systems. It provides a modular tool for the high-level description of interactions, communications and synchronizations between a collection of processes.

3.1 Syntax

The syntax of ACP^ϕ is presented by the BNF grammar in Table 4. Note that the merge operator $||_\gamma$ and the communication operator $|_\gamma$ are parameterized by the *communication function* γ as defined hereafter:

Definition 3 (Communication Function). *A communication function is any commutative and associative function form $\mathcal{A} \times \mathcal{A}$ to $\mathcal{A}$, i.e.: $\gamma : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ is a communication function if:*

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

Table 4: Syntax of ACP^ϕ.

$$
\begin{array}{rcl}
P & ::= & 1 \mid \delta \mid a \mid P_1.P_2 \mid P_1 + P_2 \mid \\
 & & P_1 ||_\gamma P_2 \mid P_1 \lfloor\!\lfloor_\gamma P_2 \mid P_1 |_\gamma P_2 \mid \\
 & & P_1^* P_2 \mid \partial_H(P) \mid \tau_I(P) \mid \partial_\phi^\xi(P)
\end{array}
$$

Essentially, a process is either an atomic action or a combination of others processes according to some well defined operators. Constants 1 and δ represent the successful termination and the deadlock respectively. Constants $a, b, c, \ldots$ are called atomic actions. The operator "." represents the sequential composition: $P_1.P_2$ is the process that first executes P_1 until it terminates, and then P_2 starts. The operator "+" represents the alternative composition: $P_1 + P_2$ represents the process that either executes P_1 or P_2 but not both of them. 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 γ. Notice that the function γ can change from one composition to another. For instance, $(P_1|_{\gamma_1} P_2)|_{\gamma_2} P_3$ is a valid process, where $\gamma_1 \neq \gamma_2$. The left merge operator "$\lfloor\!\lfloor_\gamma$" has the same meaning as the merge operator, but with the restriction that the first step must come from the left process: $P_1 \lfloor\!\lfloor_\gamma P_2$ 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 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 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 $P_1.(P_1^* P_2) + P_2$. The unary operator "∂_H" represents a restriction operator, where $H \subseteq \mathcal{A}$: the process $\partial_H(P)$ can evolve only by executing actions that are not in

H. The unary operator "τ_I" 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 τ. Finally, the operator "∂_ϕ^ξ", where ϕ is a L_φ formula and ξ is a trace from $\mathcal{T}$, represents our enforcement operator: $\partial_\phi^\xi(P)$ is the processes that can evolve only if P can evolve by actions that do not lead to the violation of the security policy ϕ. In the sequel, we denote by $\mathcal{P}$ the set of processes generated by ACP^ϕ.

3.2 Semantics

The operational semantics of ACP^ϕ is defined by the transition relation $\longrightarrow \in \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ shown in table 6, where the relation "$\equiv$" is defined in Table 5.

Table 5: Axiom of ACP^ϕ.

$P+Q$	$\equiv$	$Q+P$	$P\|\|_\gamma Q$	$\equiv$	$Q\|\|_\gamma P$
$P\|_\gamma Q$	$\equiv$	$Q\|_\gamma P$	$P+\delta$	$\equiv$	P
$\delta.P$	$\equiv$	δ	$1.P$	$\equiv$	P

In the following we define an ordering relation, noted $\sqsubseteq$, on processes.

Definition 4 (Order Relation). *Let P and Q be two processes in $\mathcal{P}$. We say that P is smaller tan Q, denoted by $P \sqsubseteq Q$, if the following condition is satisfied:*

$$P \xrightarrow{a} P' \text{ then } Q \xrightarrow{a} Q' \text{ and } P' \sqsubseteq Q'.$$

4 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 defined a logic which is suitable for to the specification of our targeted class of security policies and we used a version of *ACP* calculus enhanced with an enforcement operator ∂_ϕ^ξ for the specification of concurrent programs. The following theorem states that the desired enforcement can be achieved by the operator ∂_ϕ^ξ.

Theorem 4.1. *Let P be a process in ACP^Φ and ϕ a formula in L_φ. Let $\otimes : ACP^\Phi \times L_\varphi \to ACP^\Phi$ be defined by $P \otimes \phi = \partial_\phi^\varepsilon(P)$. The following three properties hold:*

(i) $P \otimes \phi \mid\sim \phi$,
(ii) $P \otimes \phi \sqsubseteq P$ *and*
(iii) $\forall P' : ((P' \mid\sim \phi) \wedge (P' \sqsubseteq P)) \Rightarrow P' \sqsubseteq P \otimes \phi$.

Proof.

(i) $\partial_\phi^\varepsilon(P) \mid\sim \phi$: This follows directly from the semantics of ACP^ϕ which was defined in such a way that $\partial_\phi^\varepsilon(P)$ can evolve only when the security policy is respected rule $(R_{\partial_\phi^\xi})$ of table 6.

(ii) $\partial_\phi^\varepsilon(P) \sqsubseteq P$: This also follows directly from the rule $(R_{\partial_\phi^\xi})$ of table 6.

(iii) Let P' be a process such that : $P' \mid\sim \phi \wedge P' \sqsubseteq P$ and suppose that $P' \xrightarrow{a} P'_1$. Since $P' \sqsubseteq P$, it follows from the definition of $\sqsubseteq$ that $P \xrightarrow{a} P_1$. Since $P' \mid\sim \phi$ it follows, from the definition of $\mid\sim$, that $a \mid\sim \phi$. Finally, since implies that $P \xrightarrow{a} P_1$ and $a \mid\sim \phi$ it follows from the the rule $(R_{\partial_\phi^\xi})$ in table 6 that $\partial_\phi^\varepsilon(P) \xrightarrow{a} \partial_\phi^a(P_1)$ and we conclude that $P' \sqsubseteq \partial_\phi^\varepsilon(P) = P \otimes \phi$. □

Some of the important features of the enforcement operator is that it allows us to enforce locally a given security policy e.g.: $P.\partial_\phi^\xi(Q)$, $P \mid \partial_\phi^\xi(Q)$, 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. $\partial_{\phi'}^{\xi'}(P).\partial_\phi^\xi(Q)$, $\partial_{\phi'}^{\xi'}(P) \mid \partial_\phi^\xi(Q)$, etc. However, the inconvenience of this operator is that its implementation could be heavy and the result could be not efficient. In fact, as shown by the rule $(R_{\partial_\phi^\xi})$ of Table 6, we need to save the history of the execution of a process and we need to check that any new requested action cannot violate the security policy when added to the history. Another future interesting point is to compare the expressiveness of the ACP^Φ with *ACP*.

In the rest of this paper we prove that the enforcement operator $\partial_\phi^\varepsilon(P)$ does not bring any additional expressivity to the language since it can be expressed as a combination of the rest of the operators in ACP^Φ. In addition to the importance of this result from the theoretical point of view, it gives us an efficient and elegant way to implement our enforcement operator. Basically, the idea consists in transforming the security policy as a process that runs in parallel with the system. The controlled system will be modified by adding some synchronization actions so that it can evolve only when the requested action is allowed by the security policy.

Normal Form of L_φ Formulas. In the sequel, we show how to transform a formula in L_φ, specifying a given security policy, as a process in ACP^Φ so that it could follow the analyzed system step by step and forbid it whenever it tries to violate the security policy.

Table 6: Operational semantics of ACP^{ϕ}.

$(R_{\equiv})\ \frac{P \equiv P_1 \quad P_1 \xrightarrow{a} P_2 \quad P_2 \equiv Q}{P \xrightarrow{a} Q}$	$(R^{a})\frac{\square}{a \xrightarrow{a} 1}$
$(R_{.})\frac{P \xrightarrow{a} P'}{P.Q \xrightarrow{a} P'.Q}$	$(R_{+})\frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'}$
$(R_{*})\frac{P \xrightarrow{a} P'}{P^*Q \xrightarrow{a} P'.(P^*Q)}$	$(R^{d}_{*})\frac{Q \xrightarrow{a} Q'}{P^*Q \xrightarrow{a} Q'}$
$(R_{\between_\gamma})\frac{P \xrightarrow{a} P'}{P \between_\gamma Q \xrightarrow{a} P'\|\|_\gamma Q}$	$(R_{\|\|_\gamma})\frac{P \xrightarrow{a} P'}{P\|\|_\gamma Q \xrightarrow{a} P'_1\|\|_\gamma P_2}$
$(R^{C}_{\|\|_\gamma})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P\|\|_\gamma Q \xrightarrow{\gamma(a,b)} P'\|\|_\gamma Q'}\ \gamma(a,b) \neq \delta$	$(R_{\|_\gamma})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P\|_\gamma Q \xrightarrow{\gamma(a,b)} P'\|\|_\gamma Q'}\ \gamma(a,b) \neq \delta$
$(R^{\phi}_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{\tau} \tau_I(P')}\ a \in I$	$(R_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{a} \tau_I(P)}\ a \notin I$
$(R_{\partial_H})\frac{P \xrightarrow{a} P'}{\partial_H(P) \xrightarrow{a} \partial_H(P')}\ a \notin H$	$(R_{\partial^{\xi}_{\phi}})\frac{P \xrightarrow{a} P'}{\partial^{\xi}_{\phi}(P) \xrightarrow{a} \partial^{\xi.a}_{\phi}(P')}\ \xi.a \mid\sim \phi$

Of course, the process reflecting the formula, must, amongst others, produce the same traces given by the semantic of the formula. Also, we should be able to find an equivalence representation of any operator of the logic in ACP^{Φ}. Operators like "." and "$*$" of the logic will be easily transformed into the same operators in ACP^{Φ}. However, the situation is different when we want to translate operators that are in L_φ and not in ACP^{Φ} such as the conjunction or the negation. Some operators have their corresponding one in ACP^{Φ}, like the disjunction that can be translated to a choice, but they have different algebraic properties. In fact, as processes, $a.(b+c)$ and $a.b+a.c$ do not have the same behaviours but as a formulae, $a.(b \vee c)$ and $a.b \vee a.c$ has the same semantics. Therefore, one can ask the question what is the appropriate translation of the formula $a.b \vee a.c$? Is it $a.(b+c)$ or $a.b+a.c$? In fact, the first choice is more appropriate since it allows monitoring the analyzed system according to the following idea: the system cannot evolve only if the process capturing the formula can evolve. So if our system is $a.(b+c)$ and our formula is $a.b \vee a.c$ and we capture the formula by the process $a.b+a.c$, there is a risk that the system will be blocked even if it is trivial that all its traces respect the formula.

We define a transformation function that rewrites a logical formula to an equivalent one, called normal form formula, so that the translation from the formula to the process could be easily achieved. This normalization involves the following three kind of transformation:

Conjunctive Normal Form. A formula is in CNF (Conjunctive Normal Form) if it is a conjunction of terms, where a term is a disjunction of literals. More formally, it is formulae of the form $\bigwedge_{i\in 1..n} \varphi_i$ where each φ_i does not contain the conjunction operation. We will show later how we can enforce a formula in CNF.

Elimination of the Form $\neg\varphi$. Since the negation operator in L_φ has not any "equivalent" one in ACP^{Φ}, then we need to remove it as following: first, we transform a formula of the form $\neg\varphi$ in such a way that the scope of the negation operator will be limited to atomic actions. For instance, the formula $\neg(ab)$ will be transformed to $\neg a \vee a.\neg b$. After that the negation of an atomic action will be eliminated using its complimentary part as it will be shown later.

Deterministic Formula. To resolve the problem of the disjunction operator, we need to find its equivalent deterministic form which always exists since we restricted our logic L_φ so that the star operator is always deterministic. We can prove that this deterministic form is the normal form of the rewriting system composed by the rule: $a.\varphi_1 \vee a.\varphi_2 \rightarrow a.(\varphi_1 \vee \varphi_2)$.

In the rest of the document, we will consider only formula in L_φ that are in their normal form "$\bigwedge_{i\in 1..n} \varphi_i$". This set is denoted by $L_{N(\varphi)}$. Moreover, the set of terms (φ_i) that does not contain the conjunction operator will be denoted by $L^{d}_{N(\varphi)}$.

Synchronization Actions. The idea of transforming a formula to a process that monitors the system is achieved via the introduction of what we call synchronization actions (commonly used in synchronization logic). Let us clarify the idea by a simple example. Suppose that the process is $a+b$ and the security policy is $\phi = a$ which means that only the action a is allowed. To monitor the process, the security policy will be transformed as a process containing only synchronization actions where each action a is replaced by a sequence of two actions $\overline{a_d}.\overline{a_f}$ used to capture the start and the end of the action a. Therefore, the formula a will be captured by the monitor $P_\phi = \overline{a_d}.\overline{a_f}$. The process, on the other hand, will be modified to include the complimentary part of these synchronizations actions in such a way that each action a will be replaced by $a_d.a.a_f$. For example, the process $a+b$ will be transformed to $a_d.a.a_f + b_d.b.b_f$. Now, when the two processes are executed in parallel $(a_d.a.a_f + b_d.b.b_f) \mid_\gamma \overline{a_d}.\overline{a_f}$ they can communicate (synchronize) on their synchronization actions if γ allows it. Now, to really enforce, the security policy we need to force the synchronization using the ∂_H. i.e.: $\partial_H((a_d.a.a_f + b_d.b.b_f) \mid_\gamma \overline{a_d}.\overline{a_f})$, where $H = \{a_d, a_f, b_d, b_f, \overline{a_d}, \overline{a_f}\}$. Finally, we clean the output stream of the process by abstracting the communication on synchronization actions by the silent action τ as follows: $\tau_I(\partial_H((a_d.a.a_f + b_d.b.b_f) \mid_\gamma \overline{a_d}.\overline{a_f}))$, where $I = \{\gamma(a_d, \overline{a_d})\}$. The final version is the enforced version of the system with the security policy ϕ that behaves as expected.

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

- Given a set of actions A, the corresponding synchronization set, denoted by A_C is:

$$A_C = \bigcup_{a \in A} \{a_d, a_f, \overline{a_d}, \overline{a_f}\}$$

  So $\mathcal{A}_C$ will denote the set of synchronization action associated with $\mathcal{A}$. Moreover, for any process P, $\mathcal{A}_C(P)$ returns the set of synchronization actions in P.

- The process a^c where a is in $\mathcal{A}_C$ is:

$$\sum_{\alpha \in \mathcal{A}_C \setminus \{a\}} \alpha$$

- For any integer i, the set ${\mathcal{A}_C}^i$ is the version of $\mathcal{A}_C$ indexed by an integer i :

$${\mathcal{A}_C}^i = \bigcup_{a \in \mathcal{A}_C} \{a^i\}$$

- The set H_i will be used to denote the set ${\mathcal{A}_C}^i$.
- The set I_i will be used to denote the set $\bigcup_{\alpha \in {\mathcal{A}_C}^i} \{\alpha | \overline{\alpha}\}$.
- The function γ_0 will be used to denote the communication function defined as follows:

$$\gamma_0(a, \overline{a}) = \begin{cases} a|\overline{a} & \text{if} \quad a \in \mathcal{A} \cup \mathcal{A}_C \\ \delta & \text{else} \end{cases}$$

From Formula to Process: Translation Function. The translation function is defined as shown in Table 7.

Table 7: L_φ formulae translation function.

$\lceil\lvert - \rvert\rceil : L^d_{N(\varphi)} \times \mathbb{N} \to ACP$
$\lceil\lvert tt \rvert\rceil_i = (\sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f)^* \sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f + 1$
$\lceil\lvert ff \rvert\rceil_i = \delta$
$\lceil\lvert 1 \rvert\rceil_i = 1$
$\lceil\lvert \delta \rvert\rceil_i = \delta$
$\lceil\lvert a \rvert\rceil_i = \overline{a}^i_d.\overline{a}^i_f$
$\lceil\lvert \phi_1.\phi_2 \rvert\rceil_i = \lceil\lvert \phi_1 \rvert\rceil_i.\lceil\lvert \phi_2 \rvert\rceil_i$
$\lceil\lvert \phi_1 \vee \phi_2 \rvert\rceil_i = \lceil\lvert \phi_1 \rvert\rceil_i + \lceil\lvert \phi_2 \rvert\rceil_i$
$\lceil\lvert \phi_1^* \phi_2 \rvert\rceil_i = \lceil\lvert \phi_1 \rvert\rceil_i{}^* \lceil\lvert \phi_2 \rvert\rceil_i$
$\lceil\lvert \neg a \rvert\rceil_i = \overline{a}^{i\,c}_d.\overline{a}^{i\,c}_f.((\sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f)^* \sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f + 1)$

Adding Synchronization Actions to Process. Synchronization actions are added in a process by the function $\lceil - \rceil$ defined in Table 8 where the function $\mathcal{A}_C(P)$ returns the set of synchronization actions in P.

Table 8: ACP^ϕ Processes translation function.

$\lceil - \rceil : ACP^\Phi \times \mathbb{N} \times 2^\mathcal{A} \to ACP^\Phi$		
$\lceil 1 \rceil^H_i$	$=$	1
$\lceil \delta \rceil^H_i$	$=$	δ
$\lceil a \rceil^H_i$	$=$	$\begin{cases} a & \text{If } a \in H \cup \{\tau\} \\ a^i_d.a.a^i_f & \text{Else} \end{cases}$
$\lceil P_1.P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i.\lceil P_2 \rceil^H_i$
$\lceil P_1 + P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i + \lceil P_2 \rceil^H_i$
$\lceil P_1^* P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^{H^*}_i \lceil P_2 \rceil^H_i$
$\lceil P_1 \lvert\lvert_{\gamma_0} P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i \lvert\lvert_{\gamma_0} \lceil P_2 \rceil^H_i$
$\lceil P_1 \lfloor\lfloor P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i \lfloor\lfloor \lceil P_2 \rceil^H_i$
$\lceil P_1 \lvert P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i \lvert \lceil P_2 \rceil^H_i$
$\lceil \partial_{H'}(P) \rceil^H_i$	$=$	$\partial_{H'}(\lceil P \rceil^{H \cup H'}_i)$
$\lceil \tau_I(P) \rceil^H_i$	$=$	$\tau_I(\lceil P \rceil^{H \cup I}_i)$
$\lceil \partial^\xi_{\bigwedge_{j \in 1..n} \varphi_j}(P) \rceil^H_i$	$=$	$\lceil \partial^\xi_{\bigwedge_{j \in 2..n} \varphi_j} \lceil \partial^\xi_{\varphi_1}(P) \rceil^H_i \rceil^{H \cup H_1}_{i+1}$
$\lceil \partial^\xi_\varphi(P) \rceil^H_i$	$=$	$\partial_{H_i}(\tau_{I_i}(\lceil P \rceil^H_i \lvert\lvert_{\gamma_0} \lceil\lvert [\varphi]_\xi \rvert\rceil_i))$
where $H_1 = \mathcal{A}_C(\lceil \partial^\xi_{\varphi_1}(P) \rceil^H_i)$		

Enforced Version $\partial^{\xi}_{\varphi}(P)$. Now, we are ready to express the enforcement operator using the standard *ACP* operators. Indeed, we'll prove in the next section that the process $\partial^{\xi}_{\varphi}(P)$ is "equivalent" to the process $\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\lceil[\varphi]_{\xi}\rceil_i))$ for any integer i.

5 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 ACP^{ϕ} and its corresponding version in *ACP* are equivalent with respect to the τ-bissimulation defined hereafter:

Definition 5 (τ-bissimulation). *A binary relation $S \subseteq \mathcal{P}\times\mathcal{P}$ over processes is a τ-bissimulation, if $(P,Q)\in S$ implies:*

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

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

where $\overset{a}{\twoheadrightarrow} = (\overset{\tau}{\rightarrow})^* \overset{a}{\rightarrow} (\overset{\tau}{\rightarrow})^*$

Definition 6 ($\underline{\leftrightarrow}_{\tau}$). *We define $\underline{\leftrightarrow}_{\tau}$ as the biggest τ-bissimulation:*

$$\underline{\leftrightarrow}_{\tau} = \bigcup\{S : S \text{ is a } \tau\text{-bissimulation }\}$$

Finally, The following theorem states the equivalence between the enforcement operator and its transformation.

Theorem 5.1 (Main Theorem). *$\forall P \in ACP^{\phi}$, $\forall\varphi\in L^{d}_{N(\varphi)}$, and $\forall\xi\in\mathcal{T}$, we have :*

$$\partial^{\xi}_{\varphi}(P)\underline{\leftrightarrow}_{\tau}\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\lceil[\varphi]_{\xi}\rceil_i))$$

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

6 EXAMPLE

Hereafter we show how our techniques works on a simple example. We consider the program $P = read.copy||_{\gamma_0}write.send$, which is composed of two concurrent processes, and the property given by the following formulae $\varphi : (-read)^*(read.(-send)^{\omega})$. In order to enforce φ on the program P, we execute the process : $\partial^{\varepsilon}_{\varphi}(read.copy||_{\gamma_0}write.send)$. Which is equivalent to execute the process :

$$\partial_{H_1}(\tau_{I_1}(\lceil read.copy||_{\gamma_0}write.send\rceil_1||_{\gamma_0}\lceil[(-read)^*(read.(-send)^{\omega})]_{\varepsilon}\rceil_1)).$$

In order to simplify the presentation, the letter c denotes *copy*, the letter r denotes *read*, the letter w denotes *write* and the letter s denotes *send*.

Firstly, we should calculate $\lceil r.c||_{\gamma_0}w.s\rceil_1$ and $\lceil[(-r)^*(r.(-s)^{\omega})]_{\varepsilon}\rceil_1$:

$$\lceil r.c||_{\gamma_0}w.s\rceil_1 = r^1_d.r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f$$

$$\lceil[(-r)^*(r.(-s)^{\omega})]_{\varepsilon}\rceil_1 = (\overline{r}^{1^c}_d.\overline{r}^{1^c}_f)^*(\overline{r^1_d}.\overline{r^1_f}.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega})$$

We obtain the following process :
$\partial_{H_1}(\tau_{I_1}(\underbrace{(r^1_d.r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}(\overline{r}^{1^c}_d.\overline{r}^{1^c}_f)^*(\overline{r^1_d}.\overline{r^1_f}.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega})}_{\phi'})))$, where

$H_1 = \mathcal{A}_C{}^1$ and $I_1 = \bigcup_{\alpha\in\mathcal{A}_C{}^1}\{\alpha|\overline{\alpha}\}$

For example developing the following sequence of actions : *read.write.send*. Note that this sequence violates the property ϕ, and the program should be blocked before executing the action *send*.

$\partial_{H_1}(\tau_{I_1}((r^1_d.r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}(\overline{r}^{1^c}_d.\overline{r}^{1^c}_f)^*(\overline{r^1_d}.\overline{r^1_f}.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega})))$

$\overset{\tau}{\longrightarrow}$ {| Rules $R^{C}_{||_{\gamma}}, R^{\phi}_{\tau}$ and R_{∂_H} where $\gamma_0(r^1_d,\overline{r}^1_d) = r^1_d|\overline{r}^1_d \in I_1$ |}

$\partial_{H_1}(\tau_{I_1}((r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}\overline{r^1_f}.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega}))$

$\overset{r}{\longrightarrow}$ {| Rules $R_{||_{\gamma}}, R_{\tau}$ and R_{∂_H} where $r\notin I_1$ |}

$\partial_{H_1}(\tau_{I_1}((r^1_f.c^1_d.c.c^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}\overline{r^1_f}.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega}))$

$\overset{\tau}{\longrightarrow}$ {| Rules $R^{C}_{||_{\gamma}}, R^{\phi}_{\tau}$ and R_{∂_H} where $\gamma_0(r^1_f,\overline{r}^1_f) = r^1_f|\overline{r}^1_f \in I_1$ |}

$\partial_{H_1}(\tau_{I_1}((c^1_d.c.c^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}\overline{s}^{1^c}_d.\overline{s}^{1^c}_f.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega}))$

$\overset{\tau}{\longrightarrow}$ {| Rules $R^{C}_{||_{\gamma}}, R^{\phi}_{\tau}$ and R_{∂_H} where $\gamma_0(w^1_d,\overline{s}^{1^c}_d) = w^1_d|\overline{s}^{1^c}_d \in I_1$ |}

$\partial_{H_1}(\tau_{I_1}((c^1_d.c.c^1_f||_{\gamma_0}w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}\overline{s}^{1^c}_f.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega}))$

$\overset{w}{\longrightarrow}$ {| Rules $R_{||_{\gamma}}, R_{\tau}$ and R_{∂_H} where $r\notin I_1$ |}

$\partial_{H_1}(\tau_{I_1}((c^1_d.c.c^1_f||_{\gamma_0}w^1_f.s^1_d.s.s^1_f)||_{\gamma_0} > \overline{s}^{1^c}_f.(\overline{s}^{1^c}_d.\overline{s}^{1^c}_f)^{\omega}))$

$\overset{\tau}{\longrightarrow}$ {| Rules $R^{C}_{||_{\gamma}}, R^{\phi}_{\tau}$ and R_{∂_H} where $\gamma_0(w^1_f,\overline{s}^{1^c}_f) = w^1_f|\overline{s}^{1^c}_f \in I_1$ |}

$$\partial_{H_1}(\tau_{I_1}((c_d^1.c.c_f^1||_{\gamma_0}s_d^1.s.s_f^1)||_{\gamma_0}(\bar{s}_d^{1^c}.\bar{s}_f^{1^c})^\omega))$$

As we can see the subprocess $(c_d^1.c.c_f^1||_{\gamma_0}s_d^1.s.s_f^1)$ cannot execute the action *send*, because it should firstly synchronize with another process to execute the action s_d^1, which is impossible.

7 CONCLUSIONS

This paper presents an original and innovative contribution for the enforcement of security policies on parallel programs. We first defined a dedicated algebraic calculus for the specification of parallel programs and a dedicated logic for the specification of security policies. The originality of the presented calculus is that it implements a special enforcement operator ∂_ϕ^ξ. Thus, this research project has formally defined the syntax and the semantics of the specification languages and showed how these could be used to provide mechanisms for automatic enforcement of security requirements on parallel programs. Subsequently, this paper demonstrated important results of soundness and completeness of the suggested technique. In a second step, this project was interested in the practical aspects of the presented method and showed how the enforcement operator could, in fact, be defined by standard *ACP* operators. These results are, in fact, very important both from a theoretical and practical perspectives and allow us to consider the application of our method on real languages like C or Java. Consequently, we are currently implementing a software prototype of our method that operates on the Java language. As a future work, we plan to extend the logic L_φ to 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 execution of the program.

REFERENCES

Baeten, J. C. M. (2005). A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146.

Bauer, L., Ligatti, J., and Walker, D. (2002). More enforceable security policies. In *In Foundations of Computer Security*.

Bergstra, W. F. J. A. and Ponse, A. (2001). *Handbook Of Process Algebra*, chapter chapter 5 : Process Algebra with Recursive Operations, pages 333–389. Elsevier.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *J. ACM*, 11(4):481–494.

Erlingsson, U. and Schneider, F. B. (2000). Irm enforcement of java stack inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA. IEEE Computer Society.

Fokkink, W. (2000). *Introduction to Process Algebra*. Springer-Verlag, Berlin.

K. Hamlen, G. M. and Schneider, F. (2003). Computability classes for enforcement mechanisms. Technical Report TR2003-1908, Cornell University.

Langar, M. and Mejri, M. (2005). Formal and efficient enforcement of security policies. In *FCS*, pages 143–149.

Langar, M., Mejri, M., and Adi, K. (2007). A formal approach for security policy enforcement in concurrent programs. In *Security and Management*, pages 165–171.

Ligatti, J., Bauer, L., and Walker, D. (2005). Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16.

Martinell, F. and Matteucci, I. (2007). Through modeling to synthesis of security automata. *Electron. Notes Theor. Comput. Sci.*, 179:31–46.

Mejri, M. and Fujita, H. (2008). Enforcing security policies using algebraic approach. In *SoMeT*, pages 84–98.

Morrisett, G., Walker, D., Crary, K., and Glew, N. (1999). From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568.

Necula, G. C. (1997). Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA. ACM.

Ould-Slimane, H., Mejri, M., and Adi, K. (2009). Using edit automata for rewriting-based security enforcement. In *DBSec*, pages 175–190.

Owens, S., Reppy, J., and Turon, A. (2009). Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190.

Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50.

Sen, K. and Rosu, G. (2003). Generating optimal monitors for extended regular expressions. In *In Proceedings of the 3rd Workshop on Runtime Verification (RV03). Elsevier Science.*