Consiglio Nazionale delle Ricerche

Action Refinement for Security Properties Enforcement

F. Martinelli, I. Matteucci

IIT TR-11/2010

Technical report

Febbraio 2010

Istituto di Informatica e Telematica

Action Refinement for Security Properties Enforcement [*]

Fabio Martinelli, Ilaria Matteucci
Fabio.Martinelli@iit.cnr.it
Ilaria.Matteucci@iit.cnr.it
Istituto di Informatica e Telematica - C.N.R., Via Moruzzi 1, 56124 Pisa, Italy

February 2, 2010

Abstract

In this paper we propose an application of the *action refinement* theory for enforcing security policies at different levels of abstraction by using process algebra *controller operators*.

Let us consider a system that cooperates with a possible untrusted component managed by a programmable controller operator in such a way that the considered composed system is secure, *i.e.*, the composed system works as expected. Firstly, the considered system is specified at a high level of abstraction. Successively, we refine it by applying a refinement function in such a way that we pass through different abstraction levels.

Here we investigate on the set of features a refinement function needs to have for guaranteeing that a considered system, which is secure at high level, once refined is still secure regardless the behaviour of the implementation of the untrusted component. Indeed, by applying an action refinement function, it is possible to refine the system, the controller program and the possible untrusted component as if they were three independent entities, in such a way that their implementation does not depend on each other. Hence the capability of the controller operator to make the system secure regardless the behaviour of the untrusted component at high level, is also preserved at a lower level.

Keyword: Action refinement function , controller operator, enforcing security property.

1 Introduction

In the development of software components, it is quite often required to relate systems belonging to different abstraction levels. In particular, a lot of work has been done

[*]Work partially supported by EU-funded project "Software Engineering for Service-Oriented Overlay Computers"(SENSORIA), by EU-funded project FP7-214859 CONSEQUENCE (*Context-aware data-centric information sharing*) and by EU-funded project FP7-231167 "CONNECT" (Emergent Connectors for Eternal Software Intensive Networked Systems).

to study the transition from the specification (*high level* of abstraction) to the implementation (*low level* of abstraction) of a component. Such a transition is referred to as *refinement* procedure.

The basic idea is that, given two processes P and $r(P)$, $r(P)$ is the *refinement* of P if it is described at a deeper level of detail than P. In this paper we refer to the *action refinement* theory developed in [1, 2, 3]. Indeed, in combination with our framework for enforcing secure systems through the usage of process algebra controller operators (see, *e.g.*, [4, 5]), we show how and when it is possible to enforce security property through different levels of abstraction. As a matter of fact, here we investigate the features that an action refinement function needs to have in order to be suitable for guaranteeing that a system, secured at high level of abstraction, results secure also once refined.

Referring to [4, 5], we enforce security properties at high level of abstraction by exploiting a process algebra controller operator denoted by $\triangleright_T$. It works by monitoring a *target system*, the system we want to check, and terminating any execution that is about to violate the security policy being enforced. Using our formalism, let S be the system we want to make secure. Let X be a target system, that interacts with S. We monitor X through the usage of a controller program Y according to the semantics definition of $\triangleright_T$ ($Y \triangleright_T X$). Thus the specification of the whole system at high level of abstraction is $S\|_{\mathbf{A}}(Y \triangleright_T X)$, where $\mathbf{A}$ is a set of all possible malicious actions that can be performed by an attacker. S and $Y \triangleright_T X$ synchronize their behaviour on $\mathbf{A}$ according to the semantics definition of the CSP parallel operator $\|_{\mathbf{A}}$ (*e.g.*, [6]).

Let P be a security policy represented by a process that describes the "correct" behaviour of the system, *i.e.*, the expected behaviour that the considered system must have. We consider that a system is secure whenever its behaviour is *compliant* w.r.t. the behaviour described by P. For our purpose, the notion of compliance coincides with the notion of *weak simulation* relation [7], denoted by $\preceq$, that guarantees that all actions executed by the system are also executed by the policies. Speaking in terms of security and recalling the results presented in [8, 9], if the system is weakly similar to the policy P this means that it does not perform actions that are not allowed by the policy. This guarantees that the system is secure.

Hence, at a high level of abstraction the problem of enforcing security policy by $\triangleright_T$ is specified as follows:

$$\forall X \in \mathcal{E}_{\mathbf{A}} \quad S\|_{\mathbf{A}}(Y \triangleright_T X) \preceq P \tag{1}$$

where $\mathcal{E}_{\mathbf{A}}$ represents the set of malicious high level processes, *i.e.*, processes that perform actions in $\mathbf{A}$.

As mentioned above, in this paper we investigate on the set of features a refinement function [1, 2, 3] needs to have in order to be suitable for guaranteeing that a secure system at high level, once refined is still secure. Indeed, starting from Statement (1) in which we have a secure system, we prove that, once we apply an action refinement function with certain features, the system at a lower level of abstraction is still secure, regardless of the behaviour of the implementation of the possible malicious component. To do this, we first have to find which are the assumptions we have to guarantee about the refinement function r in order to be sure that it is suitable for preserving security properties. Secondly, we have to prove the following results:

- the refinement function is distributive with respect to the parallel operator and the controller operator $\triangleright_T$. In such a way that we are able to conclude that the refinement of the system $S\|_{\mathbf{A}}(Y \triangleright_T X)$, $r(S\|_{\mathbf{A}}(Y \triangleright_T X))$, is $r(S)\|_{\tilde{r}(\mathbf{A})}r(Y) \triangleright_T r(X)$ where $r(S)$, $r(Y)$ and $r(X)$ are the refinement of S, Y and X respectively and where $\tilde{r}(\mathbf{A}) \subseteq Act_C$ represents the alphabet of the processes obtained during the refinement procedure of the action in $\mathbf{A}$ and Act_C is the set of concrete actions.

- the refinement function preserves the weak simulation. This permits to conclude that:
$$\forall r(X) \quad r(S)\|_{\tilde{r}(\mathbf{A})}(r(Y) \triangleright_T r(X)) \preceq r(P)$$
  More generally we are able to conclude that:
$$\forall X \in \mathcal{E}_{\tilde{r}(\mathbf{A})} \quad r(S)\|_{\tilde{r}(\mathbf{A})}(r(Y) \triangleright_T X) \preceq r(P)$$
  where $\mathcal{E}_{Act_C}$ represents the set of all possible malicious processes described at high level. This statement means that, since the implementation of each component S, Y and X of the system is independent from each other, the refined systems can be made secure by the refinement of the controller operator $r(Y)$ regardless of the behaviour of the implementation of the target, *i.e.*, for all possible $r(X)$. Hence, provided that the system is secure at the specification level, it remains secure also at implementation level.

An advantage of this result is that, once we have proved that a system is secure at the specification level, we are able to implement it in such a way that it is secure also at lower level regardless of the implementation of the possible untrusted component. The only requirement is that the implementation of such possible untrusted component is described at the same abstraction level of the program controller. Moreover, our approach permits to control only the possible untrusted part of the system at whatever level of abstraction we are talking about.

This paper is organized as follows: Section 2 recalls some background notions about process algebra and action refinement techniques. Section 3 presents our approach to the application of the refinement theory for enforcing security policies at different levels of abstraction. Section 4 describes a simple example in which the application of the refinement procedure guarantees the enforcement of a security policy along the TCP/IP stack. Section 5 compares our work with what already exists in literature and Section 6 concludes the paper.

2 Background

In this section we are going to recall some notions about process algebra operators and refinement functions.

2.1 A process algebra

The universe of interest is modelled by exploiting *process algebra* [6, 10] formalisms in which *processes* autonomously and concurrently can proceed in their computation but they have also the possibility to communicate and synchronize among themselves. They can perform actions which may represent computation steps.

Let Act be a set of action names, ranged over by $a, b, c \ldots$ and an invisible action τ that models the internal, non observable action. Furthermore, let $\checkmark$ be a termination predicate and $\mathcal{E}$ be a set of processes ranged over by $P, Q, E, F \ldots$.

The syntax of the considered process algebra is the following:

$$P ::= \mathbf{0} \mid a.P \mid P;P \mid P+P \mid P\|_A P \mid P[f] \mid P/A$$

where $A \subseteq Act$ and the relabelling function $f : Act \mapsto Act$ must be such that $f(\tau) = \tau$.

Informally, the meaning of these operators is the following: $\mathbf{0}$ is the term that does nothing; a (closed) term $a.P$ represents a process that performs an action a and then behaves as P. The term $P;P$ describes the sequences of two components. At the beginning it executes all actions of the first component, then starts to execute the second one. The term $P+P$ represents the non-deterministic choice: choosing the action of one of the two components means dropping the other. The term $P\|_A P$ is the *synchronous parallel operator* on the set of action A. Any action in A is performed when both the components of the term perform it. On the other hand, all actions not included in A are performed whenever one of the two components performs it. The process P/A is the *hiding* operator and behaves like P but the actions in A are replaced by τ; the process $P[f]$ behaves like P, though its actions are renamed through relabelling function f.

The operational semantics of the presented process algebra is described by a labelled transition system $(\mathcal{E}, Act, \rightarrow)$, where $\mathcal{E}$ is the set of all terms and $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a transition relation defined by structural induction as the least relation generated by the set of the structural operational semantics rules of Table 1. The transition relation $\rightarrow$ defines the usual concept of derivation in one step. As a matter of fact $P \xrightarrow{a} P'$ means that process P evolves in one step into process P' by executing action $a \in Act$. The transitive and reflexive closure of $\bigcup_{a \in Act} \xrightarrow{a}$ is written $\rightarrow^*$. In particular we use the notation $P \xrightarrow{\tau}^* P'$ ($P \overset{\epsilon}{\Longrightarrow} P'$ or $P \overset{\tau}{\Longrightarrow} P'$), in order to denote that P and P' belong to the reflexive and transitive closure of $\xrightarrow{\tau}$. Also, $P \overset{a}{\Rightarrow} P'$ if $P \xrightarrow{\tau}^* P_\tau \xrightarrow{a} P'_\tau \xrightarrow{\tau}^* P'$ where P_τ and P'_τ denote intermediate states[1].

Moreover, the basic one-step transitions are extended to τ-abstracting transitions in the usual way:

$$P \overset{\sigma}{\Longrightarrow} P' \Leftrightarrow P \overset{a_1 \ldots a_n}{\Longrightarrow} P' \Leftrightarrow P \xrightarrow{\tau}^* \xrightarrow{a_1} \xrightarrow{\tau}^* \ldots \xrightarrow{\tau}^* \xrightarrow{a_n} \xrightarrow{\tau}^* P'$$

where Act^* is the set of sequences of actions and $\sigma \in Act^*$.

Given a process P, $Der(P) = \{P' | P \rightarrow^* P'\}$ is the set of its derivatives. A process P is said *finite state* if $Der(P)$ is finite. $Sort(P)$ is the set of names of actions that syntactically appear in process P.

[1]We can use the short notation $P \xrightarrow{\tau}^* \xrightarrow{a} \xrightarrow{\tau}^* P'$ when the intermediate states are not relevant.

$$\frac{}{0\checkmark} \quad \frac{P\checkmark \quad Q\checkmark}{(P+Q)\checkmark} \quad \frac{P\checkmark \quad Q\checkmark}{(P;Q)\checkmark} \quad \frac{P\checkmark \quad Q\checkmark}{(P\|_A Q)\checkmark} \quad \frac{P\checkmark}{(P/A)\checkmark} \quad \frac{P\checkmark}{(P[f])\checkmark}$$

$$\frac{}{a.P \xrightarrow{a} P} \quad \frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P;Q} \quad \frac{P\checkmark \quad Q \xrightarrow{a} Q'}{P;Q \xrightarrow{a} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'} \quad \frac{Q \xrightarrow{a} Q'}{P+Q \xrightarrow{a} Q'}$$

$$\frac{P \xrightarrow{a} P' a \notin A}{P\|_A Q \xrightarrow{a} P'\|_A Q} \quad \frac{Q \xrightarrow{a} Q' a \notin A}{P\|_A Q \xrightarrow{a} P\|_A Q'} \quad \frac{P \xrightarrow{a} P' \; Q \xrightarrow{a} Q' a \in A}{P\|_A Q \xrightarrow{a} P'\|_A Q'}$$

$$\frac{P \xrightarrow{a} P' a \notin A}{P/A \xrightarrow{a} P'/A} \quad \frac{P \xrightarrow{a} P' a \in A}{P/A \xrightarrow{\tau} P'/A}$$

Table 1: *SOS* system for process algebra.

2.1.1 A process algebra controller operator for enforcement.

The notion of enforcement mechanism we refer to was introduced in [11]. An enforcement mechanism works by monitoring a *target system* and terminating any execution that is about to violate the security policy being enforced.

[11] presents a specification of enforcement mechanisms as *security automata* defined as a triple $(\mathcal{Q}, q_0, \delta)$ where $\mathcal{Q}$ is a set of states, q_0 is the initial one and $\delta : Act \times \mathcal{Q} \rightarrow 2^{\mathcal{Q}}$, where Act is a set of actions, is the transition function.

An enforcement mechanism works by processing a sequence $a_1 a_2 \dots$ of actions. At each step only one action is considered and, for each action, we calculate the *current state set* Q' that is the set of the possible states reachable after performing the current action, *i.e.*, if the automaton is checking the action a_i then $Q' = \bigcup_{q \in Q} \delta(a_i, q)$. If the automaton can make a transition on a given action, *i.e.*, Q' is not empty, then the target is allowed to perform that step. The state of the automaton changes according to the transition rules. Otherwise, the target execution is terminated. Thus, at every step, it verifies if the action is in the set of the possible actions or not.

We follow the approach given in [12] to describe the behaviour of security automata by using the process algebra notation. We use σ to denote a sequence of actions, $\cdot$ for the empty sequence and τ^2 to represent an internal action.

We denote with E the controller program and with F the target. We work, without loss of generality, under the additional assumption that E and F never perform the internal action τ since automata do not consider internal action. We define the controller

[2] In [12] internal actions are denoted by $\cdot$. According to the standard notation of process algebras, we use τ to denote an internal action.

operators $\triangleright_T$ as follows:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright_T F \xrightarrow{a} E' \triangleright_T F'}$$

This operator models Schneider's automaton (when considering only deterministic automata) [4, 5]. Its semantics rule states that if E and F perform the same action, such action is allowed and the controlled process $E \triangleright_T F$ performs it, otherwise it halts. It is important to note that this operator is similar to the parallel operator $\|_A$ defined above when we consider A as the whole set of actions Act.

2.1.2 Behavioural equivalence: Simulation and Bisimulation.

Behavioural equivalences allow to compare the behaviour of different processes. We recall the notion of *strong* and *weak simulation* and also *strong* and *weak bisimulation*.

Let us start with the strong version of the relation.

Definition 2.1 *Let $(\mathcal{E}, Act, \rightarrow)$ be an LTS of concurrent processes over the set of actions Act, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* strong simulation*, denoted by $\prec$, over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

$$\textit{if } P \xrightarrow{a} P' \textit{ then } \exists\, Q' \textit{ s.t. } Q \xrightarrow{a} Q' \textit{ and } (P', Q') \in \mathcal{R}$$

Recalling that the converse $\mathcal{R}^{-1}$ of any binary relation $\mathcal{R}$ is the set of pairs (Q, P) such that (P, Q) are in $\mathcal{R}$, we give the following definition.

Definition 2.2 *A* strong bisimulation *is a relation $\mathcal{R}$ s.t. both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong simulations. We represent with $\sim$ the union of all the strong bisimulations.*

In the strong simulation and bisimulation we haven't assumed a distinguished role for the τ action. In order to abstract from those actions when comparing two systems, we recall the weak version of simulation and bisimulation as follows.

Definition 2.3 *Let $(\mathcal{E}, Act, \rightarrow)$ be an LTS of concurrent processes over the set of actions Act, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* weak simulation*, denoted by $\preceq$, over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

$$\textit{if } P \xrightarrow{a} P' \textit{ then } \exists\, Q' \textit{ such that } Q \overset{a}{\Longrightarrow} Q' \textit{ and } (P', Q') \in \mathcal{R}$$

Recalling that the converse $\mathcal{R}^{-1}$ of any binary relation $\mathcal{R}$ is the set of pairs (Q, P) such that (P, Q) are in $\mathcal{R}$, we give the following definition.

Definition 2.4 *A* weak bisimulation *is a relation $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations.*

Two processes P and Q are weakly bisimilar if there exists a bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. The maximal weak bisimulation is $\approx$ which is the union of every weak bisimulation. It is easy to check that this relation is still a weak bisimulation and moreover is reflexive, symmetric and transitive.

$r^*(0)$	$:=$	0	
$r^*(a.P)$	$:=$	$r(a); r(P)$	
$r^*(P+Q)$	$:=$	$r^*(P) + r^*(Q)$	
$r^*(P;Q)$	$:=$	$r^*(P); r^*(Q)$	
$r^*(P\|_{\mathbf{A}}Q)$	$:=$	$r^*(P)\|_{\tilde{r}(\mathbf{A})}r^*(Q)$	if r is distinct on $\mathbf{A}$
$r^*(P[f])$	$:=$	$r^*(P)[f]$	if $f\lceil\{a \mid r(a) \neq a\}$ is the identity function
$r^*(P/\mathbf{A})$	$:=$	$r^*(P)/\tilde{r}(\mathbf{A})$	if r preserves $\mathbf{A}$

Table 2: Syntactic Refinement.

2.2 Action Refinement

Here we recall the notion of *action refinement* by following the theory developed in [1, 2, 3].

Let Act_{Ab} be the set of *abstract* actions and Act_C be the set of *concrete* actions. Moreover, let $\mathcal{E}_{Act_{Ab}}$ and $\mathcal{E}_{Act_C}$ be the set of high and low level processes, whose actions are in Act_{Ab} and Act_C, respectively. A *refinement function* r maps abstract actions to concrete processes, where the notion of *abstract* and *concrete* are accompanied by a change of alphabet, the implementation of a specification S is given by the *syntactic substitution* of a concrete process $r(a)$ for actions a in S.

For our purpose, in order to avoid unnecessary complications, we single out the fragment of refinement terms, **R**, that can be used as the refinement of abstract actions as follows:

$$P ::= \mathbf{0} \mid a.P \mid P + P$$

According to [3], the terms in **R** have to satisfy some specific criteria. Indeed, if a process P is a refinement of an atomic action, then it is required that P is:

- non-empty, *i.e.*, a visible abstract action cannot simply disappear during refinement.
- eventually terminating, *i.e.*, the refinement of a given action cannot "get stuck" during execution.

Here we consider a *syntactic action refinement*. For a given refinement function $r : Act_{Ab} \to \mathbf{R}_C$, syntactic substitution can be formalized as a partial function $r^* : \mathcal{E}_{Act_{Ab}} \rightharpoonup \mathcal{E}_{Act_C}$, defined according to the set of rules in Table 2.

For each action $a \in Act_{Ab}$, let $\tilde{r}(a)$ be the alphabet of a refinement term $r(a)$, *i.e.*, $\tilde{r}(a) = Sort(r(a))$. Now we recall the following definition, largely used in the rest of the paper.

Definition 2.5 ([3]) *Let $r : Act_{Ab} \to \mathbf{R}_C$ be a refinement function and $A \subseteq Act_{Ab}$ a subset of actions.*

- *r is said to* preserve *A if $\tilde{r}(a) \cap \tilde{r}(b) = \emptyset$ for all $a \in A$ and $b \in Act_{Ab} \backslash A$;*
- *r is said to be* distinct *on A if the following conditions hold:*

– *for all distinct $a \in A$ and $b \in Act_{Ab}$, $\tilde{r}(a) \cap \tilde{r}(b) = \emptyset$;*

– *for all $a \in A$ and all sub processes $P+Q$ of $r(a)$, $Sort(P) \cap Sort(Q) = \emptyset$.*

In other words, a refinement function r *preserves* a certain set $A \subseteq Act_{Ab}$ if there is no overlap between the actions occurring in the refinement of (the elements of) A and of (the elements of) $Act_{Ab} \backslash A$. On the other hand, r is *distinct* on A if also the refinement of different actions in A have disjoint alphabets, and the images of individual actions in A contain no more than a single instance of any action. This implies that a distinct refinement function is also deterministic and a deterministic refinement on A means also that it preserves A.

3 Application of Refinement to enforcement

In this section we show how and when the action refinement theory, recalled in the previous section, combined with our framework on controller operators, is suitable for enforcing security properties at lower level of abstraction.

Let P be a policy. According to [8, 9], the simulation relation guarantees that all actions executed by the system are also executed by the policies, *i.e.*, the system does not perform actions that are not allowed by the policy. This guarantees that the system is secure.

Let S be the system whose behaviour must be compliant with policy P. We restrict ourselves to consider a subset of processes made of sequences of actions, $+$ and $\|_A$ where $A \subseteq Act_{Ab}$ is the set of all possible malicious actions in Act_{Ab} and for every $Q = Q_1 \|_A Q_2$, $Sort(Q_2) \subseteq A$. We name this subset of processes as $\mathcal{E}_{\subset}$.

At specification level, we ensure that the following statement:

$$\forall X \in \mathcal{E}_{\mathbf{A}} \quad S \|_{\mathbf{A}} (Y \triangleright_T X) \preceq P \qquad (2)$$

holds by controlling the behaviour of the possible malicious component X through the usage of a controller program Y that monitors X according to the semantics of $\triangleright_T$.

The aim of this section is to provide a proof of the following statement: if a system is secure at high specification level then the same system, once refined through a refinement function r, will still be secure at a lower level of abstraction, regardless the behaviour of a possible malicious component. Such a malicious component has to be described at the same level of abstraction of the rest of the refined system. In order to guarantee this, we assume that r respects the following features:

- the refinement function r represents a syntactic refinement, *i.e.*, hereafter r and r^* coincide;
- the refinement function r is distinct on Act;
- the refinement function r is the identity function w.r.t. the internal action τ;

Let r be an action refinement function with the features listed above. We prove the following theorem.

Theorem 3.1 *Let $S\|_{\boldsymbol{A}}(Y \triangleright_T X)$ be a secure system,* i.e.*, it is weakly similar to P. Let $r(S)$, $r(Y)$, $r(X)$ and $r(P)$ be the syntactic refinement of S, Y, X and P w.r.t. r respectively. The following relation holds:*

$$\forall r(X) \in \mathcal{E}_{\tilde{r}(\boldsymbol{A})} \quad r(S)\|_{\tilde{r}(\boldsymbol{A})}(r(Y) \triangleright_T r(X)) \preceq r(P)$$

This means that, given a secure system, once this is refined by applying a refinement function r to each of its components, it is still secure also at a lower level.

To prove Theorem 3.1 we have to show that refinement function r preserves the weak simulation.

Let us start by proving the following lemma.

Lemma 3.1 *Let P, Q and E be processes in $\mathcal{E}_{\subset}$ then*

$$P\|_A Q \preceq E \;\Rightarrow\; r(P)\|_{\tilde{r}(A)} r(Q) \preceq r(E)$$

According to the semantics definition of $\triangleright_T$, $Y \triangleright_T X \in \mathcal{E}_{\subset}$ because both Y and X perform only actions in A and $Y \triangleright_T X$ performs action performed by both Y and X. Hence $Sort(Y \triangleright_T X) \subseteq Sort(X)$. Moreover, referring to the definition of syntactic refinement, $r(Y \triangleright_T X) = r(Y) \triangleright_T r(X)$. Hence, the proof of Theorem 3.1 follows directly from Lemma 3.1.

This result permits us to guarantee that the implementation by the refinement function r of the controller program Y does not depend on the behaviour of the implementation of the target X. Hence $r(Y)$ makes the system secure for all possible $r(X)$.

Hence, we have proven that whenever a given system is secure at specification level, this remains secure after a syntactic refinement also at implementation level regardless of the implementation of the possible malicious component.

Moreover, we are able to guarantee that the system is still secure at implementation level regardless the behaviour of the untrusted component under the additional assumption that the behaviour of the malicious agent is described at the same level of abstraction as the rest of the system. Hence the following Corollary of Theorem 3.1 holds.

Corollary 3.1 *In the same hypotheses of the previous theorem we can also prove that:*

$$\forall X \in \mathcal{E}_{\tilde{r}(\boldsymbol{A})} \quad r(S)\|_{\tilde{r}(\boldsymbol{A})}(r(Y) \triangleright_T X) \preceq r(P)$$

where $\mathcal{E}_{\tilde{r}(\boldsymbol{A})}$ is the set of all possible malicious processes described at a concrete level.

Proof: It follows by noticing that $(r(Y) \triangleright_T X) \preceq r(Y)$ for any $X \in \mathcal{E}_{\tilde{r}(\mathbf{A})}$ and from the fact the $\preceq$ is a pre-congruence w.r.t. to the operators of the calculus.

□

4 An example: Refinement through the TCP/IP levels

Referring to [13], we propose a simple example of a possible application of the framework we have described in the previous sections.

The TCP/IP protocol suite implements the standard connection model of computer network with four layers: application, transport, network and link. Here we consider the refinement between the *application* and the *transport* level:

Application layer is the interface between the applications and the network (client-side protocol). The network concepts involved at this level are *Daemons* and *Terminal servers*.

Transportation layer manages the host-to-host communications. The network concepts involved at this level are *Hosts* and *ports*.

Let us consider a network composed by two terminal servers, H_1 and H_2, that are the only agents of the considered system. Moreover, let us suppose that daemons are `send` and `receive`, where these two actions denote the daemons of sending or receiving mails. Hence the set of abstract actions is $Act_A = \{\texttt{send},\ \texttt{receive}\}$. The expected behaviour of H_1 and H_2 is the following:

$$\begin{array}{lcl} H_1 & = & \texttt{send}.H_1 + \texttt{receive}.H_1 \\ H_2 & = & \texttt{receive}.H_2 \end{array}$$

The policy of the systems, denoted by P, says that H_1 is allowed to send and receive mail, instead H_2 is only allowed to receive when H_1 receives them. It cannot send mail. Hence we have that P can be expressed in process algebra at application level as follows:

$$P = H_1 \|_{\{\texttt{receive}\}} H_2$$

Assuming we know that H_1 is a trusted component, *i.e.*, it performs only allowed actions, but we are not sure that H_2 behaves correctly. Hence we control this components, in order to make the system secure. Let $Y = \texttt{receive}.Y$ be a controller operator, then it is easy to see that

$$H_1 \|_{\{\texttt{receive}\}} (Y \triangleright_T H_2) \preceq P$$

Now, let us move to the transportation level, in which the ports must be specified. Hence, the action refinement function r applied to the considered daemons is the following:

$$\begin{array}{lcl} r(\texttt{send}) & = & \texttt{socket}_{25}.\texttt{sendmail} \\ r(\texttt{receive}) & = & \texttt{socket}_{993}.\texttt{receivemail} \end{array}$$

where 25 is the port of SMTP daemons that allows to send mails, and 993 is the port of IMAP daemons that permits us to receive mails. It is not difficult to see that r is allowable and distinct. Indeed it is non-empty, there is no τ action. Each action is mapped in a finite sequence of actions, distinct from one another.

Applying the refinement to our system we have that H_1 and H_2 are peers of the transportation layer, hence their communications take place on specified ports. Thus they are refined as:

$$\begin{array}{lcl} r(H_1) & = & \texttt{socket}_{25}.\texttt{sendmail}.r(H_1) + \texttt{socket}_{993}.\texttt{receivemail}.r(H_1) \\ r(H_2) & = & \texttt{socket}_{993}.\texttt{receivemail}.r(H_2) \end{array}$$

Moreover the refinement of the policy P is

$$r(P) = r(H_1) \|_{\{\texttt{socket}_{993},\texttt{receivemail}\}} r(H_2)$$

Through the same refinement, we have $r(Y) = \texttt{socket}_{993}.\texttt{receivemail}.r(Y)$. Hence it is not difficult to verify that

$$r(H_1) \|_{\{\texttt{socket}_{993},\texttt{receivemail}\}} (r(Y) \triangleright_T r(H_2)) \preceq r(P)$$

Moreover we have:

$$\forall H_2^{\{\texttt{socket}_{993},\texttt{receivemail}\}}$$

$$r(H_1) \|_{\{\texttt{socket}_{993},\texttt{receivemail}\}} (r(Y) \triangleright_T H_2^{\{\texttt{socket}_{993},\texttt{receivemail}\}}) \preceq r(P)$$

where $H_2^{\{\texttt{socket}_{993},\texttt{receivemail}\}}$ is a concrete process. As a matter of fact, whatever the behaviour of the second component of the parallel composition is, the refinement of the controller program $r(Y)$, guarantees that the system is correct.

5 Related work

Much work has been done in developing process refinement theories, although not much work has been done dealing with refinement in security.

In [14] the authors use action refinement for security issues. They give a definition of the refinement function on process algebra terms, similar to the one given in [1] that we have used in this paper. They are focused on the analysis of security issues, in particular the analysis of information flow properties. Our approach, instead, is focused on enforcing security properties through different levels of abstraction.

In [13] the authors present a case study on analysis of refinement. Indeed, using Event-B they refine a controller for a security property across all the different network layers of the TCP/IP stack and prove that such refinement is valid. On the other hand, here we present a general framework for security properties that are yet satisfied in the transition through different abstraction levels by refinement procedure.

Referring to the framework of policies refinement, in [15] KAOS is introduced. This is a goal oriented methodology to analyze and refine requirements, especially, security requirements. KAOS is based on several models which give different perspectives to the situation analysed. A policy template can be used with KAOS in order to refine policies. The template is filled in for each security requirement and refined according to the goal refinements until they reach an implementation detail level. KAOS is suitable for the refinement process because of its library of refinement patterns that have been proved and can be reused in concrete situations. Refinement is formally defined in KAOS and can be easily checked using model checking tools like Spin, KAOS etc. Properties like completeness, correctness and minimality that a refinement must verify are clearly defined and have been used to verify the refinement patterns. In addition to it refinement suit, KAOS offers other interesting mechanisms to reason on security requirement. An advantage of our work w.r.t. KAOS, is that we are able to refine enforcement mechanisms. In KAOS the enforcement is not treated specifically.

Also in [16] the authors refer to the policy refinement problem as the passage from the high specification level of a system to the implementation of the system itself guaranteeing that the goals are achieved. There are no considerations neither investigation about security aspects.

In [16] the authors refer to the policy refinement problem as the passage from the high specification level of a system to the implementation of the system itself guaranteeing that the goals are achieved. As a matter of fact, in this paper, the authors have presented an approach where a formal representation of a system, based on the Event Calculus, can be used in conjunction with adductive reasoning techniques to derive the sequence of operations that will allow a given system to achieve a desired goal. There are no considerations neither investigation performed on security aspects. On the other hand, here we present how it is possible to enforce safety properties across different levels of abstraction.

[17] presents a slightly different notion of the refinement of policies: Policy refinement is the process that decomposes the high level policy relevant to a composite system into a set of policies that are executed in its constituent parts to implement the behaviour indented by the high level policy. This means that, in [17], a high level policy describes the global behaviour of the whole considered system, and, on the other hand, a low level policy describes the behaviour of a single component of the system in such a way that, when it is composed with the other components, the global behaviour of the system satisfies the high level policy. In other words, the refinement procedure described in [17] is a projection of a global policy into local ones whose composition gives the global one again.

6 Conclusion and Future work

This work extends our previous one based on process algebra and logic for the *specification* (*e.g.*, see [18]), *verification* (*e.g.*, see [19]) and automated *synthesis* of enforcing mechanisms for secure systems (*e.g.*, see [5]), with also *refinement* theory.

Indeed, in this paper we have presented a way for applying action *refinement* theory for the enforcement of security properties at different specification levels. By starting from our theory on process algebra operators (see [4, 5]) we have considered the process algebra controller operator $\triangleright_T$ as enforcement mechanism for checking the behaviour of possible untrusted components. Referring to this operator, we have shown how it is possible to refine a system, including the existing controller program, by guaranteeing that the resulting one is yet secure (w.r.t. the refined security policy).

As future work, we aim to investigate the application of action refinement function to other controller operators. Indeed, starting from the definition of security automata given in [12, 20], in [4, 21] we have defined other three controller operators, $\triangleright_S$, $\triangleright_I$ and $\triangleright_E$. Our goal will be the study of the necessary and sufficient conditions for the validity of Theorem 3.1also for suppression, insertion and edit automata

References

[1] R. Gorrieri, A. Rensink, Action refinement, in: Handbook of Process Algebra, Elsevier, 2001, pp. 1047–1147.

[2] A. Rensink, R. Gorrieri, Action refinement as an implementation relation, in: TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, Springer-Verlag, London, UK, 1997, pp. 772–786.

[3] A. Rensink, R. Gorrieri, Vertical implementation, Inf. Comput. 170 (1) (2001) 95–133.

[4] F. Martinelli, I. Matteucci, Partial model checking, process algebra operators and satisfiability procedures for (automatically) enforcing security properties, Technical report, IIT-CNR, presented at the International Workshop on Foundations of Computer Security (FCS05) (2005).

[5] F. Martinelli, I. Matteucci, An approach for the specification, verification and synthesis of secure systems, Electr. Notes Theor. Comput. Sci. 168 (2007) 29–43.

[6] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[7] R. Milner, Communicating and mobile systems: the π-calculus, Cambridge University Press, 1999.

[8] N. Dragoni, F. Massacci, K. Naliuka, I. Siahaan, T. Quillinan, I. Matteucci, C. Schaefer, Deliverable 2.1.4-Methodologies and tools for contract matching-S3MS European Project (2007).

[9] P. Greci, F. Martinelli, I. Matteucci, A framework for contract-policy matching based on symbolic simulations for securing mobile device application, in: T. Margaria, B. Steffen (Eds.), ISoLA, Vol. 17 of Communications in Computer and Information Science, Springer, 2008, pp. 221–236.

[10] M. Hennessy, Algebraic Theory of Processes, The MIT Press, Cambridge, Mass., 1988.

[11] F. B. Schneider, Enforceable security policies, ACM Transactions on Information and System Security 3 (1) (2000) 30–50.

[12] L. Bauer, J. Ligatti, D. Walker, More enforceable security policies, in: I. Cervesato (Ed.), Foundations of Computer Security: proceedings of the FLoC'02 workshop on Foundations of Computer Security, DIKU Technical Report, Copenhagen, Denmark, 2002, pp. 95–104.

[13] N. Stouls, M. L. Potet, Security policy enforcement through refinement process., in: J. Julliand, O. Kouchnarenko (Eds.), The 7th International B Conference, Vol. 4355 of Lecture Notes in Computer Science, Springer, 2007, pp. 216–231.

[14] A. Bossi, C. Piazza, S. Rossi, Action refinement in process algebra and security issues, in: A. King (Ed.), LOPSTR, Vol. 4915 of Lecture Notes in Computer Science, Springer, 2007, pp. 201–217.

[15] A. Dardenne, A. V. Lamsweerde, S. Fickas, Goal-directed requirements acquisition, in: Science of Computer Programming, 1993, pp. 3–50.

[16] A. K. Bandara, E. Lupu, J. D. Moffett, A. Russo, A goal-based approach to policy refinement, in: POLICY, IEEE Computer Society, 2004, pp. 229–239.

[17] V. W. Kevin Carey, David Lewis, Automated policy-refinement for managing composite services, in: M-Zones White Paper June 04, whitepaper 06/04, Ireland, 2004.

[18] R. Focardi, F. Martinelli, A uniform approach for the definition of security properties, in: FM '99: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems-Volume I, Springer-Verlag, London, UK, 1999, pp. 794–813.

[19] F. Martinelli, Analysis of security protocols as *open* systems, Theoretical Computer Science 290 (1) (2003) 1057–1106.

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

[21] F. Martinelli, I. Matteucci, Through modeling to synthesis of security automata, Electr. Notes Theor. Comput. Sci. 179 (2007) 31–46.

A Technical Proofs

To prove Lemma 3.1 we have to demonstrate the following results.

Lemma A.1 *Let r be a syntactic action refinement. Let r be distinct on Act, deterministic and the identity on τ. The following holds*

$$\forall a \in Sort(P)\ P \xrightarrow{a} P' \text{ and } r(a) \xrightarrow{\gamma} R \Rightarrow r(P) \xrightarrow{\gamma} V \text{ where } V \sim R; r(P')$$

where P is a process in $\mathcal{E}_{\subset}$.

Proof: Let us proceed by induction on the structure of P.

$P = 0$ in this case the thesis holds trivially.

$P = a.P'$ According to the set of rules in Table 2, we have that

$$r(P) = r(a.P') = r(a); r(P')$$

If $r(a) \xrightarrow{\gamma} R$, according to the semantics definition of the sequences of processes, since $a \neq \surd$, then $r(a); r(P') \xrightarrow{\gamma} R; r(P')$. Hence $r(P) \xrightarrow{\gamma} V$ where V is $R; r(P')$.

$P = P_1 + P_2$ According to the set of rules in Table 2, we have that

$$r(P) = r(P_1 + P_2) = r(P_1) + r(P_2)$$

According to the hypothesis $P \xrightarrow{a} P$. We have two possibilities:

- $P_1 \xrightarrow{a} P_1'$. In this case P behaves as P_1, structurally simpler than P, so $P \xrightarrow{a} P_1'$. Hence $r(P_1) \xrightarrow{\gamma} R; r(P_1')$. Since $r(P) = r(P_1) + r(P_2)$ then, in this case $r(P)$ behaves as $r(P_1)$, so $r(P) \xrightarrow{\gamma} V$ where V is $R; r(P_1')$. for the semantic definition of the choice operator $+$.
- $P_2 \xrightarrow{a} P_2'$. In this case P behaves as P_2, structurally simpler than P, so $P \xrightarrow{a} P_2'$. Hence $r(P_2) \xrightarrow{\gamma} R; r(P_2')$ Since $r(P) = r(P_1) + r(P_2)$ then, in this case $r(P)$ behaves as $r(P_2)$, so $r(P) \xrightarrow{\gamma} V$ where V is $R; r(P_1')$. for the semantic definition of the choice operator $+$.

$P = P_1 \|_A P_2$ According to the set of rules in Table 2, we have that

$$r(P) = r(P_1 \|_A P_2) = r(P_1) \|_{\tilde{r}(A)} r(P_2)$$

According to the hypothesis that r is distinct on Act we can infer that r preserves each $A \subseteq Act$. Hence, if $Sort(P_2) \subseteq A$ then $Sort(r(P_2) \subseteq \tilde{r}(A)$.

For hypothesis $P \xrightarrow{a} P$. We have two possibilities:

- $P_1 \xrightarrow{a} P_1'$ and $a \notin A$. In this case $P \xrightarrow{a} P_1' \|_A P_2$. For inductive hypothesis, $r(P_1) \xrightarrow{\gamma} R; r(P_1')$. Hence, since $Sort(r(a)) \cap \tilde{r}(A) = \emptyset$ according to the fact r is distinct on Act, then $r(P) \xrightarrow{\gamma} R; r(P_1') \|_{\tilde{r}(A)} r(P_2)$.

  To conclude this part of proof, we have to note that each action R and $r(P_2)$ cannot synchronize on each other because R is residual of a a that is not in A neither in $Sort(P_2)$. This leads to conclude that the process $R; r(P_1') \|_{\tilde{r}(A)} r(P_2)$ behaves as a process $R; Y$ where $Y = r(P_1') \|_{\tilde{r}(A)} r(P_2)$. Let $\sim$ be a bisimulation equivalence defined as follows:

  $$\sim = \{(R; r(P_1') \|_{\tilde{r}(A)} r(P_2), R; (r(P_1') \|_{\tilde{r}(A)} r(P_2)))| \ R \text{ is residual of an action } a \notin A \text{ and } r \text{ is deterministic } \}$$

  if $R; r(P_1') \|_{\tilde{r}(A)} r(P_2) \xrightarrow{\delta} X$**, then** $\exists Z$ **s.t.** $R; Y \xrightarrow{\delta} Z$ **and** $X \sim Z$ Let us consider that $R; r(P_1') \|_{\tilde{r}(A)} r(P_2) \xrightarrow{\delta} X$. According to the semantics definition of $\|_{\tilde{r}(A)}$ jointly to the fact that R and $r(P)$ cannot synchronize on each other, we have that $R; r(P_1') \xrightarrow{\delta} R'; r(P_1')$, that, according to the semantics definition of sequence operator, means that $R \xrightarrow{\delta} R'$. In this case X is $R'; r(P_1') \|_{\tilde{r}(A)} r(P_2)$.

  Since $R \xrightarrow{\delta} R'$, then $R; Y \xrightarrow{\delta} R'; Y$. Hence Z exists and it is $R'; Y$ and $X \sim Z$

if $R;(r(P_1')\|_{\tilde{r}(A)}r(P_2)) \xrightarrow{\delta} X$**, then** $\exists Z$ **s.t.** $R;r(P_1')\|_{\tilde{r}(A)}r(P_2) \xrightarrow{\delta} Z$ **and** $X \sim Z$

According to the semantics definition of sequence operator

$$R;(r(P_1')\|_{\tilde{r}(A)}r(P_2)) \xrightarrow{\delta} X$$

means that $R \xrightarrow{\delta} R'$. Hence X is $R';(r(P_1')\|_{\tilde{r}(A)}r(P_2))$. Since R and $r(P)$ cannot synchronize on each other, $\delta \notin \tilde{r}(A)$,

$$R;r(P_1')\|_{\tilde{r}(A)}r(P_2) \xrightarrow{\delta} R';r(P_1')\|_{\tilde{r}(A)}r(P_2)$$

Hence Z is $R';r(P_1')\|_{\tilde{r}(A)}r(P_2)$ and $X \sim Z$.

- $P_1 \xrightarrow{a} P_1'$ and $P_2 \xrightarrow{a} P_2'$. In this case $P \xrightarrow{a} P_1'\|_A P_2'$. For inductive hypothesis, $r(P_1) \xrightarrow{\gamma} R;r(P_1')$ and $r(P_2) \xrightarrow{\gamma} R;r(P_2')$ where R is the same because r is deterministic. Hence $r(P) \xrightarrow{\gamma} R;r(P_1')\|_{\tilde{r}(A)}R;r(P_2')$. To conclude we have to prove that $R;r(P_1')\|_{\tilde{r}(A)}R;r(P_2') \sim R;Y$ where $Y = r(P_1')\|_{\tilde{r}(A)}r(P_2')$.

  Let $\sim$ be a bisimulation equivalence defined as follows:

$$\sim = \{(R;r(P_1')\|_{\tilde{r}(A)}R;r(P_2'), R;(r(P_1')\|_{\tilde{r}(A)}r(P_2'))| \ R \text{ is residual of an action } a \in A \text{ and } r \text{ is deterministic } \}$$

if $R;r(P_1')\|_{\tilde{r}(A)}R;r(P_2') \xrightarrow{\delta} X$**, then** $\exists Z$ **s.t.** $R;Y \xrightarrow{\delta} Z$ **and** $X \sim Z$

Let us consider that $R;r(P_1')\|_{\tilde{r}(A)}R;r(P_2') \xrightarrow{\delta} X$. According to the semantics definition of $\|_{\tilde{r}(A)}$ jointly to the fact that each action performed by R is in $\tilde{r}(A)$ and r is deterministic, we have that $R;r(P_1') \xrightarrow{\delta} R';r(P_1')$ and $R;r(P_2') \xrightarrow{\delta} R';r(P_2')$, that, according to the semantics definition of sequence operator, means that $R \xrightarrow{\delta} R'$. In this case X is $R';r(P_1')\|_{\tilde{r}(A)}R';r(P_2')$.

Since $R \xrightarrow{\delta} R'$, then $R;Y \xrightarrow{\delta} R';Y$. Hence Z exists and it is $R';Y$ and $X \sim Z$

if $R;(r(P_1')\|_{\tilde{r}(A)}r(P_2')) \xrightarrow{\delta} X$**, then** $\exists Z$ **s.t.** $R;r(P_1')\|_{\tilde{r}(A)}R;r(P_2') \xrightarrow{\delta} Z$ **and** $X \sim Z$

According to the semantics definition of sequence operator,

$$R;(r(P_1')\|_{\tilde{r}(A)}r(P_2)) \xrightarrow{\delta} X$$

means that $R \xrightarrow{\delta} R'$. Hence X is $R';(r(P_1')\|_{\tilde{r}(A)}r(P_2'))$.
Since $\delta \in \tilde{r}(A)$,

$$R;r(P_1')\|_{\tilde{r}(A)}R;r(P_2') \xrightarrow{\delta} R';r(P_1')\|_{\tilde{r}(A)}R';r(P_2)$$

Hence Z is $R';r(P_1')\|_{\tilde{r}(A)}R';r(P_2)$ and $X \sim Z$.

This concludes the proof.

We also need the following lemmata

Lemma A.2 *Let E be a process. If $E \stackrel{\hat{\tau}}{\Longrightarrow} E'$ then $E' \preceq E$.*

Proof: Let us consider the following relation definition:

$$S = \{(E', E) | E \stackrel{\hat{\tau}}{\Longrightarrow} E'\}$$

We prove that this is a weak simulation:

$$E' \xrightarrow{a} E'' \quad \exists E_1 \text{ s.t. } E \stackrel{a}{\Longrightarrow} E_1 \ \wedge (E'', E_1) \in S$$

$a = \tau$ In this case $E \stackrel{\hat{\tau}}{\Longrightarrow} E' \xrightarrow{\tau} E''$, hence $E_1 = E''$ and (E'', E'') is obviously in S.

$a \neq \tau$ In this case $E \stackrel{\hat{\tau}}{\Longrightarrow} E' \xrightarrow{a} E''$. Hence, by considering that $\stackrel{\hat{\tau}}{\Longrightarrow}$ means a sequence of zero or more τ actions, we can also write,$E \stackrel{a}{\Longrightarrow} E''$. Hence $E'' = E_1$ and $(E'', E'') \in S$.

Under this assumption we are able to prove the following result.

Lemma Let P, Q and E be processes in $\mathcal{E}_{\mathbb{C}}$ then

$$P \|_A Q \preceq E \ \Rightarrow \ r(P) \|_{\tilde{r}(A)} r(Q) \preceq r(E)$$

Proof: Let us consider the following relation definition:

$$\begin{aligned} S \quad = \quad & \{(r(P)\|_{\tilde{r}(A)} r(Q), r(E)) | P \|_A Q \preceq E \wedge Sort(Q) \subseteq A\} \cup \\ & \{(R; r(P)\|_{\tilde{r}(A)} r(Q), R; r(E)) | a \notin A \text{ s.t. } r(a) \xrightarrow{\gamma} R \not\xrightarrow{\checkmark} \wedge P \|_A Q \preceq E \\ & \quad \wedge Sort(Q) \subseteq A\} \cup \\ & \{(R; r(P)\|_{\tilde{r}(A)} R; r(Q), R; r(E)) | a \in A \text{ s.t. } r(a) \xrightarrow{\gamma} R \not\xrightarrow{\checkmark} \wedge P \|_A Q \preceq E \\ & \quad \wedge Sort(Q) \subseteq A\} \end{aligned}$$

- Let us consider that $r(P)\|_{\tilde{r}(A)} r(Q) \xrightarrow{\gamma} X$. We have to prove that there exists Y s.t. $r(E) \stackrel{\gamma}{\Longrightarrow} Y$ and $(X, Y) \in S$.

  $\gamma = \tau$ There are two possibilities:

  - The transition is due to $r(P) \xrightarrow{\tau} X_1$. Then, it is possible to prove that X_1 is equivalent to $r(P')$ for some P' such that $P \xrightarrow{\tau} P'$. This follows from the kind of processes we consider and the fact that $r(\tau) = \tau$. In this case $X = r(P')\|_{\tilde{r}(A)} r(Q)$. Furthermore, we have also that $P \|_A Q \xrightarrow{\tau} P' \|_A Q$. According to the hypothesis of the theorem, if $P \|_A Q \preceq E$, then there exists E' s.t. $E \stackrel{\hat{\tau}}{\Longrightarrow} E' \ \wedge \ P' \|_A Q \preceq E'$. Since r is the identity on τ, $r(E) \stackrel{\hat{\tau}}{\Longrightarrow} r(E')$. Then Y exists and it is $r(E')$ and $(X, Y) = (r(P')\|_{\tilde{r}(A)} r(Q), r(E')) \in S$.

- The transition is due to $r(Q) \xrightarrow{\tau} X_1$. The reasoning proceed as in the previous case.

$\gamma \neq \tau$ Since r is distinct on Act, we can conclude that, if $\gamma \neq \tau$ then $\forall a \in Act$ s.t. $r(a) \xrightarrow{\gamma} R$, $\tau \notin Sort(R)$. There are two cases to consider:

$r(a) \xrightarrow{\gamma} R$ **where** $a \notin A$**.** In this case $P \xrightarrow{a} P'$. Since r is distinct on Act and deterministic, this implies that $r(P) \xrightarrow{\gamma} R; r(P')$. Hence $X = R; r(P') \|_{\tilde{r}(A)} r(Q)$. Furthermore, $P \|_A Q \xrightarrow{a} P' \|_A Q$. Since $P \|_A Q \preceq E$, there exists E' s.t. $E \overset{a}{\Longrightarrow} E'$ and $P' \|_A Q \preceq E'$. Now, let us consider that $r(E)$: $E \overset{a}{\Longrightarrow} E'$ means $E \overset{\hat{\tau}}{\Longrightarrow} E_\tau \xrightarrow{a} E'_\tau \overset{\hat{\tau}}{\Longrightarrow} E'$. Hence $r(E) \overset{\hat{\tau}}{\Longrightarrow} E_\tau \xrightarrow{\gamma} R; E'_\tau$. Let us consider $Y = R; E'_\tau$. Since, $P' \|_A Q \preceq E'$ and, according to Lemma A.2, $E' \preceq E'_\tau$, then $P' \|_A Q \preceq E'_\tau$. This guarantees that $(X, Y) = (R; r(P') \|_{\tilde{r}(A)} r(Q), R; E'_\tau) \in S$.

$r(a) \xrightarrow{\gamma} R$ **where** $a \in A$**.** In this case $P \xrightarrow{a} P'$ and $Q \xrightarrow{a} Q'$. Since r is distinct on Act and deterministic, this implies that $r(P) \xrightarrow{\gamma} R; r(P')$ and $r(Q) \xrightarrow{\gamma} R; r(Q')$. Hence $X = R; r(P') \|_{\tilde{r}(A)} R; r(Q')$. Furthermore, $P \|_A Q \xrightarrow{a} P' \|_A Q'$. Since $P \|_A Q \preceq E$, there exists E' s.t. $E \overset{a}{\Longrightarrow} E'$ and $P' \|_A Q' \preceq E'$. Following the same reasoning made before, we have that $Y = R; E'_\tau$ and

$$(X, Y) = (R; r(P') \|_{\tilde{r}(A)} R; r(Q'), R; E'_\tau) \in S$$

- Let us consider that $R; r(P) \|_{\tilde{r}(A)} r(Q) \xrightarrow{\gamma} X$. We have to prove that there exists Y s.t. $R; r(E) \overset{\gamma}{\Longrightarrow} Y$ and $(X, Y) \in S$. According to the assumption that r is distinct on Act, it is possible to conclude that $Sort(R) \cap \tilde{r}(A) = \emptyset$ because R is the residual of the refinement of an action $a \in Sort(P) \backslash A$. Hence, the process $R; r(P) \|_{\tilde{r}(A)} r(Q)$ performs γ iff $R \xrightarrow{\gamma} R'$. This implies that $X = R'; r(P) \|_{\tilde{r}(A)} r(Q)$. Furthermore, according to the assumption that r is deterministic, this implies also that $R; r(E) \xrightarrow{\gamma} R'; r(E)$. Hence, let Y be $R'; r(E)$ then the couple $(X, Y) = (R'; r(P) \|_{\tilde{r}(A)} r(Q), R'; r(E)) \in S$ because we are already in the hypothesis that $P \|_A Q \preceq E$.

- Let us consider that $R; r(P) \|_{\tilde{r}(A)} R; r(Q) \xrightarrow{\gamma} X$. We have to prove that there exists Y s.t. $R; r(E) \overset{\gamma}{\Longrightarrow} Y$ and $(X, Y) \in S$. According to the assumption that r is distinct on Act, it is possible to conclude directly from the definition that $\forall A \subseteq Act$ r preserves A. This implies that $Sort(R) \subseteq \tilde{r}(A)$ because R is the residual of the refinement of an action $a \in A$. Hence, the process $R; r(P) \|_{\tilde{r}(A)} R; r(Q)$ performs γ iff $R \xrightarrow{\gamma} R'$. Furthermore $X = R'; r(P) \|_{\tilde{r}(A)} R'; r(Q)$. The conclusion is similar to the previous one. Indeed, according to the assumption that r is deterministic, $R; r(E) \xrightarrow{\gamma} R'; r(E)$. Hence, let Y be $R'; r(E)$ then the couple $(X, Y) = (R'; r(P) \|_{\tilde{r}(A)} R'; r(Q), R'; r(E)) \in S$ because we are already in the hypothesis that $P \|_A Q \preceq E$.