Preserving (Security) Properties under Action Refinement*

Annalisa Bossi, Carla Piazza, and Sabina Rossi

Dipartimento di Informatica, Università Ca' Foscari di Venezia
via Torino 155, 30172 Venezia, Italy
{bossi,piazza,srossi}@dsi.unive.it

Abstract. In the design process of distributed systems we may have to replace abstract specifications of components by more concrete specifications, thus providing more detailed design information. In the context of process algebra this well-known approach is often referred to as *action refinement*. In this paper we study the relationships between action refinement, compositionality, and (security) process properties within the Security Process Algebra (SPA). We formalize the concept of action refinement both as a structural inductive definition and in terms of subsequent context compositions. We study compositional properties of our notion of refinement and provide conditions under which general process properties are preserved through it. Finally, we consider information flow security properties and define decidable classes of secure terms which are closed under action refinement.

1 Introduction

In the development of complex systems it is common practice to first describe it succinctly as a simple abstract specification and then refine it stepwise to a more concrete implementation. This hierarchical specification approach has been successfully developed for sequential systems where abstract-level instructions are expanded until a concrete implementation is reached (see, e.g., [21]).

In the context of process algebra, this refinement methodology amounts to defining a mechanism for replacing abstract actions with more concrete processes. We adopt the terminology *action refinement* to refer to this stepwise development of systems specified as terms of a process algebra. We refer to [14] for a survey on the state of the art of action refinement in process algebra.

Action refinement in process algebras is usually defined by extending the syntax with some compositional operator [1, 13]. Here we follow a different approach and instead of extending the language, we use a construction based on context composition. This allows us to reason on the relationships between action refinement and the security properties of SPA processes that we have deeply studied in, e.g., [3]. In the last part of the paper we prove that our definition of action refinement is indeed equivalent to the one presented in [1].

* This work has been partially supported by the EU Contract IST-2001-32617 "Models and Types for Security in Mobile Distributed Systems" (MyThS).

In this paper we model action refinement as a ternary function *Ref* taking as arguments an action r to be refined, a system description E on a given level of abstraction and an interpretation of the action r on this level by a more concrete process F on a lower abstraction level. The refined process can be obtained either by applying a structural inductive definition or through a more complex context composition as described by the following simple example.

Let E be the process $a.r.b.\mathbf{0} + c.\mathbf{0}$ and r be the action we intend to refine by the process $F \equiv d_1.d_2.\mathbf{0}$. The refined process, denoted by $Ref(r, E, F)$, will be the process $a.d_1.d_2.b.\mathbf{0} + c.\mathbf{0}$ which can be obtained in two equivalent ways: (1) we can either apply a structural inductive definition as follows: $Ref(r, E, F) = a.Ref(r, r.b.\mathbf{0}, F) + c.\mathbf{0} = a.F'[b.\mathbf{0}] + c.\mathbf{0}$ where $F'[Y]$ is the context $d_1.d_2.Y$ and $F'[b.\mathbf{0}]$ is the process $d_1.d_2.b.\mathbf{0}$; (2) or we can compute the refinement by a single context composition as $E'[F'[b.\mathbf{0}]]$ where $E'[X]$ is the context $a.X + c.\mathbf{0}$ while $F'[Y]$ is as above the context $d_1.d_2.Y$.

Our definitions follow the static syntactic approach to action refinement (see, e.g., [19]). We prove several compositional properties of our notion of refinement. Indeed, compositional properties are fundamental in the stepwise development of complex systems. They allow us to refine sub-components of the system, while guaranteeing that the final result does not depend on the order in which the refinements are applied. We also provide conditions under which our notion of refinement preserves general properties of processes and, in particular, we focus on security properties.

In system development, it is important to consider security related issues from the very beginning. Indeed, considering security only at the final step could lead to a poor protection or, even worst, could make it necessary to restart the development from scratch. A security-aware stepwise development requires that the security properties of interest are either preserved or gained during the development steps, until a concrete (i.e., implementable) specification is obtained.

In this paper we consider *information flow security* properties (see, e.g., [12, 9, 15]), i.e., properties that allow one to express constraints on how information should flow among different groups of entities. These properties are usually formalized by considering two groups of entities labelled with two security levels: *high* (H) and *low* (L). The only constraint is that no information should flow from H to L. For example, to guarantee confidentiality in a system, it is sufficient to label every confidential (i.e., secret) information with H and then partition each system user as H and L, depending on whether such a user is or is not authorized to access confidential information. The constraint of no information flow from H to L guarantees that no access to confidential information is possible by L-labelled users. We consider the bisimulation-based security property named *Persistent Bisimulation-based non Deducibility on Compositions* (P_BNDC, for short) [10]. Property P_BNDC is based on the idea of Non-Interference [12] and requires that every state which is reachable by the system still satisfies a basic Non-Interference property. We show how to both instantiate and extend the results obtained for general process properties in order to provide decidable conditions ensuring that P_BNDC is preserved under action refinement.

The paper is organized as follows. In Section 2 we recall some basic notions of the SPA language. In Section 3 we formalize our notion of action refinement as a structural inductive definition. We also study its compositional properties. In Section 4 we reformulate action refinement in terms of context composition and we state conditions under which general process properties are preserved through action refinement. In Section 5 we consider the security property *P_BNDC* and define decidable classes of *P_BNDC* processes which are closed under action refinement. Finally, in Section 6 we discuss some related works.

2 Basic Notions

The *Security Process Algebra* (SPA) [9] is a variation of Milner's CCS [18] where the set of visible actions is partitioned into two security levels, high and low, in order to specify multilevel systems. SPA syntax is based on the same elements as CCS, i.e.: a set $\mathcal{L} = I \cup O$ of *visible* actions where $I = \{a, b, \ldots\}$ is a set of *input* actions and $O = \{\bar{a}, \bar{b}, \ldots\}$ is a set of *output* actions; a special action τ which models internal computations, not visible outside the system; a function $\bar{\cdot} : \mathcal{L} \rightarrow \mathcal{L}$, such that $\bar{\bar{a}} = a$, for all $a \in \mathcal{L}$. $Act = \mathcal{L} \cup \{\tau\}$ is the set of all *actions*. The set of visible actions is partitioned into two sets, H and L, of high security actions and low security actions such that $\overline{H} = H$ and $\overline{L} = L$, where $\overline{H}$ and $\overline{L}$ are obtained by applying function $\bar{\cdot}$ to all the elements in H and L, respectively.

The syntax of SPA *terms* is as follows[1]:

$$T ::= \mathbf{0} \mid Z \mid a.T \mid T + T \mid T|T \mid T \setminus v \mid T[f] \mid recZ.T$$

where Z is a variable, $a \in Act$, $v \subseteq \mathcal{L}$, $f : Act \rightarrow Act$ is such that $f(\bar{l}) = \overline{f(l)}$ for $l \in \mathcal{L}$, $f(\tau) = \tau$, $f(H) \subseteq H \cup \{\tau\}$, and $f(L) \subseteq L \cup \{\tau\}$. We apply the standard notions of *free* and *bound* (occurrences of) variables in a SPA term. More precisely, all the occurrences of the variable Z in $recZ.T$ are *bound*; while Z is *free* in a term T if there is an occurrence of Z in T which is not bound.

A SPA *process* is a SPA term without free variables. We denote by $\mathcal{E}$ the set of all SPA processes, ranged over by $E, F, \ldots$.

The operational semantics of SPA processes is given in terms of *Labelled Transition Systems* (LTS, for short). In particular, the LTS $(\mathcal{E}, Act, \rightarrow)$, whose states are processes, is defined by structural induction as the least relation generated by the axioms and inference rules reported in Figure 1.

Intuitively, $\mathbf{0}$ is the empty process that does nothing; $a.E$ is a process that can perform an action a and then behaves as E; $E_1 + E_2$ represents the nondeterministic choice between the two processes E_1 and E_2; $E_1|E_2$ is the parallel composition of E_1 and E_2, where executions are interleaved, possibly synchronized on complementary input/output actions, producing the silent action τ; $E \setminus v$ is a process E prevented from performing actions in v; $E[f]$ is the process E whose actions are renamed *via* the relabelling function f; if in T there is

[1] Actually in [9] recursion is introduced through constant definitions instead of the *rec* operator.

Prefix	$\dfrac{-}{a.E \xrightarrow{a} E}$
Sum	$\dfrac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \quad \dfrac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$
Parallel	$\dfrac{E_1 \xrightarrow{a} E_1'}{E_1 \mid E_2 \xrightarrow{a} E_1' \mid E_2} \quad \dfrac{E_2 \xrightarrow{a} E_2'}{E_1 \mid E_2 \xrightarrow{a} E_1 \mid E_2'} \quad \dfrac{E_1 \xrightarrow{l} E_1' \quad E_2 \xrightarrow{\bar{l}} E_2'}{E_1 \mid E_2 \xrightarrow{\tau} E_1' \mid E_2'}$
Restriction	$\dfrac{E \xrightarrow{a} E'}{E \setminus v \xrightarrow{a} E' \setminus v}$ if $a \notin v$
Relabelling	$\dfrac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$
Recursion	$\dfrac{T[recZ.T[Z]] \xrightarrow{a} E'}{recZ.T[Z] \xrightarrow{a} E'}$

with $a \in Act$ and $l \in \mathcal{L}$.

Fig. 1. The operational rules for SPA

at most the free variable Z, then $recZ.T[Z]$ is the recursive process which can perform all the actions of the process obtained by substituting $recZ.T[Z]$ to the place-holder Z in the term $T[Z]$.

The concept of *observation equivalence* is used to establish equalities among processes and it is based on the idea that two systems have the same semantics if and only if they cannot be distinguished by an external observer. This is obtained by defining an equivalence relation over $\mathcal{E}$ equating two processes when they are indistinguishable. In this paper we consider the relations named *weak bisimulation*, $\approx$, and *strong bisimulation*, $\sim$, defined by Milner for CCS [18]. They equate two processes if they are able to mutually simulate their behavior step by step.

We use the following notations: $E \xrightarrow{a} E'$ to denote the transition labelled by a from E to E', $E \stackrel{a}{\Longrightarrow} E'$ to denote any sequence of transitions $E(\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^* E'$ where $(\xrightarrow{\tau})^*$ denotes a (possibly empty) sequence of τ labelled transitions, and $E \stackrel{\hat{a}}{\Longrightarrow} E'$ which stands for $E \stackrel{a}{\Longrightarrow} E'$ if $a \in \mathcal{L}$, and for $E(\xrightarrow{\tau})^* E'$ if $a = \tau$. We say that E' is reachable from E if there exist $a_1, \ldots, a_n \in Act$ such that $E \xrightarrow{a_1} \ldots \xrightarrow{a_n} E'$.

Weak bisimulation does not care about internal τ actions while strong bisimulation does.

Definition 1 (Weak and Strong Bisimulation). *A symmetric binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ over processes is a* weak bisimulation *if $(E, F) \in \mathcal{R}$ implies, for all $a \in Act$, if $E \xrightarrow{a} E'$, then there exists F' such that $F \stackrel{\hat{a}}{\Longrightarrow} F'$ and $(E', F') \in \mathcal{R}$. Two processes $E, F \in \mathcal{E}$ are* weakly bisimilar*, denoted by $E \approx F$, if there exists a weak bisimulation $\mathcal{R}$ containing the pair (E, F).*

The definition of strong bisimulation *is obtained by replacing $\stackrel{\hat{a}}{\Longrightarrow}$ with $\xrightarrow{a}$ in the sentence above. Two processes $E, F \in \mathcal{E}$ are* strongly bisimilar*, denoted by $E \sim F$, if there exists a strong bisimulation $\mathcal{R}$ containing the pair (E, F).*

The relation $\approx$ ($\sim$) is the largest weak (strong) bisimulation and it is an equivalence relation.

A SPA term with free variables can be seen as an environment with holes (the free occurrences of its variables) in which other SPA terms can be inserted. The result of this substitution is still a SPA term, which could be a process. For instance, in the term $h.\mathbf{0}|(l.X + \tau.\mathbf{0})$ we can replace the variable X with the process $\bar{h}.\mathbf{0}$ obtaining the process $h.\mathbf{0}|(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})$; or we can replace X by the term $a.Y$ obtaining the term $h.\mathbf{0}|(l.a.Y + \tau.\mathbf{0})$. When we consider a SPA term as an environment we call it *context*[2], i.e., a SPA context is a SPA term in which free variables may occur.

Given a context C, we use the notation $C[Y_1, \ldots, Y_n]$ to emphasize the free variables $Y_1, \ldots, Y_n$ occurring in C. The term $C[T_1, \ldots, T_n]$ is obtained from $C[Y_1, \ldots, Y_n]$ by simultaneously replacing all the free occurrences of $Y_1, \ldots, Y_n$ with the terms $T_1, \ldots, T_n$, respectively. For instance, given the contexts $C[X] \equiv h.\mathbf{0}|(l.X + \tau.\mathbf{0})$ and $D[X, Y] \equiv (l.X + \tau.\mathbf{0})|Y$, the notation $C[\bar{h}.\mathbf{0}]$ stands for $h.\mathbf{0}|(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})$, while the notation $D[\bar{h}.\mathbf{0}, \bar{l}.\mathbf{0}]$ stands for $(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})|\bar{l}.\mathbf{0}$.

Following [18] we extend binary relations on processes to contexts as follows.

Definition 2 (Relations on Contexts). *Let $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ be a relation over processes. Let C, D be two contexts with free variables $Y_1, \ldots, Y_n$. We say that $C[Y_1, \ldots, Y_n]\ \mathcal{R}\ D[Y_1, \ldots, Y_n]$ if $C[E_1, \ldots, E_n]\ \mathcal{R}\ D[E_1, \ldots, E_n]$ for any choice of $E_1, \ldots, E_n \in \mathcal{E}$. We also use $C\ \mathcal{R}\ D$ to denote $C[Y_1, \ldots, Y_n]\ \mathcal{R}\ D[Y_1, \ldots, Y_n]$.*

As an example, the contexts $C[X, Y] \equiv a.X + \tau.Y$ and $D[X, Y] \equiv a.\tau.X + \tau.Y$ are weakly bisimilar since for all $E, F \in \mathcal{E}$ it holds $a.E + \tau.F \approx a.\tau.E + \tau.F$.

Strong bisimulation is a congruence, i.e., if $C[X] \sim D[X]$ and $E \sim F$, then $C[E] \sim D[F]$. Weak bisimulation is not a congruence, i.e., if two contexts $C[X]$ and $D[X]$ are weakly bisimilar, and two processes E and F are weakly bisimilar, then $C[E]$ and $D[F]$ are not necessarily weakly bisimilar. However, weak bisimulation is a congruence over the *guarded* SPA language whose terms are defined by replacing the production $T + T$ with $a.T + a.T$ in the SPA syntax.

3 Action Refinement

It is standard practice in software development to obtain the final program starting from an abstract, possibly not executable, specification by successive refinement steps. Abstract operations are replaced by more detailed programs which

[2] Notice that a SPA term denotes either a process or a context.

can be further refined, until a level is reached where no more abstractions occur. In the context of process algebra, this stepwise development amounts to interpreting actions on a higher level of abstraction by more complicated processes on a lower level. This is obtained by introducing a mechanism to replace actions by processes. There are several ways to do this. We adopt the syntactic approach and define the refinement step as a syntactic process transformation.

We need to introduce some notation. Given a process F and a variable Y, we denote by $F^0[Y]$ the context obtained by replacing each occurrence of $\mathbf{0}$ in F with the variable Y. As an example, consider the process $F \equiv recZ.(a.Z + b.\mathbf{0})$. Then $F^0[Y] \equiv recZ.(a.Z + b.Y)$.

To introduce our notion of action refinement we also need to define which are the *refinable* actions of a process. This concept is based on the following notions of *bound* and *free* actions.

Definition 3. (Bound and Free actions) *Let T be a SPA term. The set of* bound *actions of T, denoted by $bound(T)$, is inductively defined as follows:*

$$
\begin{array}{l}
bound(\mathbf{0}) = \emptyset \\
bound(Z) = \emptyset \quad \textit{where } Z \textit{ is a variable} \\
bound(a.T) = bound(T) \\
bound(T_1 + T_2) = bound(T_1) \cup bound(T_2) \\
bound(T_1 | T_2) = bound(T_1) \cup bound(T_2) \\
bound(T \setminus v) = bound(T) \cup v \\
bound(T[f]) = bound(T) \cup \{a, f(a) \mid f(a) \neq a\} \\
bound(recZ.T) = bound(T)
\end{array}
$$

An action occurring in T is said to be free *if it is not bound. We denote by $free(T)$ the set of free actions of T.*

In practice, an action is bound in a term T if either it is restricted in a subterm of T or it belongs to the domain or the codomain of a relabelling function f occurring in T. For instance, the actions a and $\bar{a}$ occur bound in the process $E \equiv a.\mathbf{0} + recZ.((\bar{a}.Z + b.a.\mathbf{0}) \setminus \{a, \bar{a}\})$.

An abstract action r occurring in a process E is refinable if r is not bound in E and, in order to avoid problems with synchronizations, $\bar{r}$ does not occur in E. We also require that the process F which is intended to refine r is different from $\mathbf{0}$ and that it does not contain the parallel operator. Moreover, r and $\bar{r}$ should not occur in F otherwise we would enter into an infinite loop of refinements. Finally we impose that the free actions of F are not bound in E and vice-versa, to avoid undesired bindings of actions in the refined process. All these requirements are formalized in the following notion of *refinability*. We will discuss them in the next subsection.

Definition 4. (Refinability) *Let E, F be SPA processes and $r \in \mathcal{L}$. The action r is said to be* refinable *in E with F if:*

(*a*) *F is not the process $\mathbf{0}$;*
(*b*) *F does not contain any occurrence of the parallel operator;*

(c) $r \notin bound(E)$ *and* $\bar{r}$ *does not occur in* E*;*
(d) r *and* $\bar{r}$ *do not occur in* F*;*
(e) *for all subterm* E' *of* E, $(bound(E') \cap free(F)) \cup (bound(F) \cap free(E')) = \emptyset$

Example 1. Consider the processes $E \equiv (r.a.\mathbf{0}|\bar{a}.b.\mathbf{0}) \setminus \{a, \bar{a}\}$ and $F \equiv c.d.\mathbf{0}$. In this case action r is refinable in E with F.

Consider now the processes E as above and $F_1 \equiv b.\mathbf{0}+(c.d.\mathbf{0})\setminus\{b\}$. In this case condition (e) of Definition 4 is not satisfied since $bound(F_1) \cap free(E) = \{b\} \neq \emptyset$. Hence r is not refinable in E with F_1. □

The refinement of an abstract action r in a process E with a refining process F is obtained by replacing each occurrence of r in E with F. In order to support action refinement, in the literature the prefixing operator is usually replaced by sequential composition ";" (see [1, 13]). Here we follow a different approach and instead of extending the language, we use a construction based on context composition. Thus, for instance the refinement of the action r in the process $E \equiv a.r.b.\mathbf{0}$ with the process $F \equiv c.d.\mathbf{0}$ is obtained by substituting $b.\mathbf{0}$ for Y in $a.F^0[Y] \equiv a.c.d.Y$, i.e., it is the process $a.F^0[b.\mathbf{0}]$. The conditions on the refinable actions and the fact that F does not contain the parallel operator, ensure that our notion of action refinement is comparable with more classical ones like, e.g., [1] (see Section 6). Moreover, the fact that we do not modify our language, allows us to directly apply our security notions for SPA processes also when reasoning on action refinement.

Our notion of *action refinement* is defined by structural induction on the process to be refined.

Definition 5. (Action Refinement) *Let* E, F *be SPA processes such that* r *is an action refinable in* E *with* F*. The* refinement of r in E with F *is the process* $Ref(r, E, F)$ *inductively defined on the structure of* E *as follows:*

(1) $Ref(r, \mathbf{0}, F) \equiv \mathbf{0}$
(2) $Ref(r, Z, F) \equiv Z$
(3) $Ref(r, r.E_1, F) \equiv F^0[Ref(r, E_1, F)]$
(4) $Ref(r, a.E_1, F) \equiv a.Ref(r, E_1, F)$, *if* $a \neq r$
(5) $Ref(r, E_1[f], F) \equiv Ref(r, E_1, F)[f]$
(6) $Ref(r, E_1 \setminus v, F) \equiv Ref(r, E_1, F) \setminus v$
(7) $Ref(r, E_1 + E_2, F) \equiv Ref(r, E_1, F) + Ref(r, E_2, F)$
(8) $Ref(r, E_1|E_2, F) \equiv Ref(r, E_1, F)|Ref(r, E_2, F)$
(9) $Ref(r, recZ.E_1, F) \equiv recZ.Ref(r, E_1, F)$

Point (3) of definition above deals with the basic case in which we replace an occurrence of r with the refining process F. If $E \equiv r.E_1$ and r is the only occurrence of r in E, then $Ref(r, E, F) \equiv F^0[Ref(r, E_1, F)] \equiv F^0[E_1]$ representing the process which first behaves as F and then, when the execution of F is terminated, proceeds as E_1. In all the other cases the refinement process enters inside the components of E. This is correct also when restriction or relabelling operators are involved since conditions (c), (d) and (e) of Definition 4 guarantee

that we never refine restricted or relabelled actions and that undesired bindings of actions will never occur.

Example 2. Let $E \equiv r.a.\mathbf{0} + b.\mathbf{0}$ and $F \equiv c.\mathbf{0} + d.\mathbf{0}$. It is immediate to observe that r is refinable in E with F. By applying points 7. and 3. of Definition 5 we get $Ref(r, E, F) \equiv c.a.\mathbf{0} + d.a.\mathbf{0} + b.\mathbf{0}$.

Let $E \equiv (a.r.b.\mathbf{0}) \setminus \{b\}$ and $F \equiv c.d.\mathbf{0}$. Since $bound(E) = \{b\}$ and b does not occur in F, r is refinable in E with F. By applying points 6., 4. and 3. of our definition of action refinement we get $Ref(r, E, F) \equiv (a.c.d.b.\mathbf{0}) \setminus \{b\}$. Notice that, our notion of refinability does not allow us to refine r in E with $F_1 \equiv b.d.\mathbf{0}$. However, as done in [1], we can first apply an α conversion mapping E into the equivalent process $E_1 \equiv (a.r.e.\mathbf{0}) \setminus \{e\}$ and then refine r in E_1 with F_1 getting the expected process $(a.b.d.e.\mathbf{0}) \setminus \{e\}$.

Let $E \equiv a.r.b.\mathbf{0}|r.c.\mathbf{0}$ and $F \equiv c.d.\mathbf{0}$. Applying our definition we get that $Ref(r, E, F) \equiv a.c.d.b.\mathbf{0}|c.d.c.\mathbf{0}$. As expected, since in E there are two occurrences of r we replace them with two copies of F. In this way it is possible that new synchronizations are generated. □

From now on when we write $Ref(r, E, F)$ we tacitly assume that r is refinable in E with F.

Notice that we do not allow the use of the parallel composition in the process F. In fact, if $E \equiv r.a.\mathbf{0}$ and $F \equiv b.\mathbf{0}|c.\mathbf{0}$, by applying our notion of refinement we would obtain the process $b.a.\mathbf{0}|c.a.\mathbf{0}$, i.e., we would duplicate part of E. Usually this undesired behavior is avoided by exploiting the concatenation operator ";" in the definition of action refinement (obtaining the process $(b.\mathbf{0}|c.\mathbf{0}); a.\mathbf{0}$). Here instead, we prefer to impose restrictions on F. This assumption is only mildly restrictive since, if F is a finite state[3] process, then there always exists a process F_1, strongly bisimilar to F, which does not contain any occurrence of the parallel operator (see [17]). For instance, in the previous example it is sufficient to consider $F_1 \equiv b.c.\mathbf{0} + c.b.\mathbf{0}$ in order to get the expected $Ref(r, E, F_1) \equiv b.c.a.\mathbf{0} + c.b.a.\mathbf{0}$.

At any fixed abstraction level during the top-down development of a program, it is unrealistic to think that there is just one action to be refined at that level. Compositional properties of the refinement operation allow us to discard the ordering in which the refinements occur.

First we show that our refinement is local to the components in which the action to be refined occurs. This is a consequence of the following theorem.

Theorem 1. *Let $E_1, \ldots, E_n$ and F be terms. Let $C[Z_1, \ldots, Z_n]$ be a context with no occurrences of r and $\bar{r}$. It holds*

$$Ref(r, C[E_1, \ldots, E_n], F) \equiv C[Ref(r, E_1, F), \ldots, Ref(r, E_n, F)].$$

Hence, if we have a term G which is of the form $E_1|E_2|\ldots|E_n$ and the action r occurs only in E_i it is sufficient to apply the refinement to E_i to obtain $Ref(r, G, F) \equiv E_1|E_2|\ldots|Ref(r, E_i, F)|\ldots|E_n$.

[3] A process is finite state if it reaches only a finite number of different processes. Notice that a finite state process can be recursive.

Example 3. Consider the process $G \equiv recV.(a.V + recW.(a.W + r.W))$. We can decompose it into $C[Z] \equiv recV(a.V + Z)$ and $E \equiv recW.(a.W + r.W)$ and apply the refinement to E. If $F \equiv b.c.\mathbf{0}$ we get that $Ref(r, E, F) \equiv recW.(a.W + b.c.W)$. Hence, $Ref(r, G, F) \equiv recV.(a.V + recW.(a.W + b.c.W))$. □

If we need to refine two actions in a process E, then the order in which we apply the refinements is irrelevant.

Theorem 2. *Let E be a term. Let F_1 and F_2 be two terms with no occurrences of r_1, r_2, $\bar{r}_1$, and $\bar{r}_2$.*

$$Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv Ref(r_1, Ref(r_2, E, F_2), F_1).$$

Example 4. Let $E \equiv r_1.a.\mathbf{0} + r_2.b.r_2.\mathbf{0}$, $F_1 \equiv b.\mathbf{0}$ and $F_2 \equiv c.\mathbf{0}$. We have that $Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv b.a.\mathbf{0} + c.b.c.\mathbf{0} \equiv Ref(r_1, Ref(r_2, E, F_2), F_1)$. □

Moreover, we can refine r_1 in E using F_1 and r_2 in F_1 using F_2 independently from the order in which the refinements are applied as stated by the following theorem.

Theorem 3. *Let E, F_1, F_2 be terms such that r_1 and $\bar{r}_1$ do not occur in F_2.*

$$Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv Ref(r_1, Ref(r_2, E, F_2), Ref(r_2, F_1, F_2)).$$

Example 5. Let $E \equiv r_1.a.\mathbf{0} + a.r_2.\mathbf{0}$, $F_1 \equiv b.r_2\mathbf{0}$ and $F_2 \equiv c.\mathbf{0}$. We have $Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv Ref(r_2, b.r_2.a.\mathbf{0} + a.r_2.\mathbf{0}, F_2) \equiv b.c.a.\mathbf{0} + a.c.\mathbf{0} \equiv Ref(r_1, r_1.a.\mathbf{0} + a.c.\mathbf{0}, b.c.\mathbf{0}) \equiv Ref(r_1, Ref(r_2, E, F_2), Ref(r_2, F_1, F_2))$. □

4 Preserving Process Properties under Refinement

A process property $\mathcal{P}$ is nothing but a class of processes, i.e., the class of processes which satisfy $\mathcal{P}$. In particular, we are interested in classes of processes expressing security notions, i.e., classes of processes which are all secure (with respect to a particular notion of security). We intend to investigate conditions under which notions of security are preserved under action refinement. This correspond to analyze conditions under which classes of processes are closed with respect to action refinement.

We start by characterizing our notion of refinement uniquely in terms of context composition, spelling out the recursion on the structure of E. To do this we introduce a suitable operation which realizes the necessary links from the parts of E which precede an occurrence of r and the parts of E which follow that occurrence. In other words we have to hook F to E, whenever an action r occurs.

We define the set $E@r$ of the parts of E which syntactically follow the outermost occurrences of an action r, and the context $E\{r\}$ which represents the part of E before the outermost occurrences of r.

Definition 6 ($E@r$ and $E\{r\}$). *Let E be a SPA term and r be a refinable action in E. The set of terms $E@r$ is inductively defined as follows:*

$$\begin{array}{ll}
\mathbf{0}@r = \emptyset; & Z@r = \emptyset; \\
(r.T)@r = \{T\}; & (a.T)@r = T@r, \ \textit{if } a \neq r; \\
(T_1 + T_2)@r = T_1@r \cup T_2@r; & (T_1|T_2)@r = T_1@r \cup T_2@r; \\
(T \setminus v)@r = T@r; & (T[f])@r = T@r; \\
(recZ.T)@r = T@r. &
\end{array}$$

Let $E@r = \{T_1, \ldots, T_n\}$ and $\{X_{T_1}, \ldots, X_{T_n}\}$ be a set of distinct variables indexed in the elements of $E\{r\}$. The context $E\{r\}$ is inductively defined as follows

$$\begin{array}{ll}
\mathbf{0}\{r\} = \mathbf{0}; & Z\{r\} = Z; \\
(r.T)\{r\} = X_T; & (a.T)\{r\} = a.(T\{r\}), \ \textit{if } a \neq r; \\
(T_1 + T_2)\{r\} = T_1\{r\} + T_2\{r\}; & (T_1|T_2)\{r\} = T_1\{r\}|T_2\{r\}; \\
(T \setminus v)\{r\} = (T\{r\}) \setminus v; & (T[f])\{r\} = (T\{r\})[f]; \\
(recZ.T)\{r\} = recZ.(T\{r\}). &
\end{array}$$

Notice that if $E@r = \{T_1, \ldots, T_n\}$, then $\{X_{T_1}, \ldots, X_{T_n}\}$ is the set of free variables of $E\{r\}$. In the following we will write $E\{r\}$ to denote the context $E\{r\}[X_{T_1}, \ldots, X_{T_n}]$. Thus $E\{r\}[S_1, \ldots, S_n]$ represents the term obtained from $E\{r\}[X_{T_1}, \ldots, X_{T_n}]$ by simultaneously replacing all the free occurrences of the variables $X_{T_1}, \ldots, X_{T_n}$ with the terms $S_1, \ldots, S_n$, respectively.

Example 6.

- Let $E \equiv r.\mathbf{0}|a.\mathbf{0}$. We have that $E@r$ is $\{\mathbf{0}\}$ and $E\{r\}$ is $X_{\mathbf{0}}|a.\mathbf{0}$.
- Let $E \equiv (a.r.\mathbf{0} + b.r.c.r.a.\mathbf{0}) \mid r.\mathbf{0}$. The set $E@r$ contains two processes and is equal to $\{\mathbf{0}, c.r.a.\mathbf{0}\}$. Note that the term $c.r.a.\mathbf{0}$ in $E@r$ contains an occurrence of r. The context $E\{r\}$ is $(a.X_{\mathbf{0}} + b.X_{c.r.a.\mathbf{0}}) \mid X_{\mathbf{0}}$. The set of the free variables of $E\{r\}$ is exactly $\{X_T \mid T \in E@r\}$.
- Let $E \equiv recZ.(a.Z + r.Z)$. We have that $E@r$ is $\{Z\}$ and $E\{r\}$ is $recZ.(a.Z + X_Z)$. In this case $E@r$ has only one element which is not a process. □

The refinement of an action r in E with F can be equivalently obtained by successive context compositions as follows.

Definition 7 (Partial Refinement). *Let E and F be terms, and $r \in Act$ be an action refinable in E with F. Let Y be a variable which does not occur neither in E nor in $E\{r\}$ nor in F. Let $E@r = \{T_1, \ldots, T_n\}$. The* partial refinement *$ParRef(r, E, F)$* of r in E with F *is defined as*

$$ParRef(r, E, F) \equiv E\{r\}[F^0[T_1], \ldots, F^0[T_n]].$$

The following theorem provides an alternative characterization of our notion of refinement.

Theorem 4. *The refinement $Ref(r, E, F)$ of r in E with F satisfies*

- *$Ref(r, E, F) \equiv ParRef^0(r, E, F) \equiv E$, if r does not occur in E;*

– $Ref(r, E, F) \equiv ParRef^{n+1}(r, E, F) \equiv ParRef(r, ParRef^{n}(r, E, F), F)$, *if* r *occurs* $n + 1$ *times in* E.

Intuitively $E@r$ are the parts of E which syntactically follow the occurrences of the action r, while $E\{r\}$ is the part of E which precedes the r's. The holes X_T's in $E\{r\}$ serve to hook the refinement F. Similarly, the free variable Y of $F^0[Y]$ serves to hook the elements of $E@r$ after the execution of F. The partial refinement $ParRef(r, E, F)$ replaces in E as many occurrences as possible of r with F. In the case of nested occurrences of r (e.g., $r.a.r.\mathbf{0}$) the partial refinement replaces only the first occurrence. Hence in order to replace all the occurrences in the worst case it is necessary to compute the partial refinement n times, where n is the number of occurrences of r in E. We would obtain the same result by arbitrarily choosing at each step one occurrence of r replacing it with F, and going on until there are no more occurrences of the refineble action r.

Example 7. We consider again the second process of Example 6, i.e., let $E \equiv (a.r.\mathbf{0} + b.r.c.r.a.\mathbf{0}) \mid r.0$ and $F \equiv e.f.\mathbf{0}$. The partial refinement $ParRef(r, E, F)$ is the process $E' \equiv (a.e.f.\mathbf{0} + b.e.f.c.r.a.\mathbf{0}) \mid e.f.\mathbf{0}$. The context $E'\{r\}$ coincides with $(a.e.f.\mathbf{0} + b.e.f.c.X_{a.\mathbf{0}}) \mid e.f.\mathbf{0}$. Hence $Ref(r, E, F) = ParRef(r, E', F) = (a.e.f.\mathbf{0} + b.e.f.c.e.f.a.\mathbf{0}) \mid e.f.\mathbf{0}$. □

Let $\mathcal{P}$ be a generic process property. We are now ready to introduce some conditions which imply that $\mathcal{P}$ is preserved under action refinement.

Definition 8 ($\mathcal{P}$-refinable contexts). *Let $\mathcal{P}$ be a class of processes. A class $\mathcal{C}$ of contexts is said to be a class of* $\mathcal{P}$-refinable contexts *if:*

– *if $C \in \mathcal{C}$ and C is a process, then $C \in \mathcal{P}$;*
– *if $C, D \in \mathcal{C}$, then $C[D] \in \mathcal{C}$;*
– *if $C \in \mathcal{C}$ and r is refinable in C, then $C@r \cup \{C\{r\}\} \subseteq \mathcal{C}$.*

Theorem 5. *Let $\mathcal{P}$ be a class of processes and $\mathcal{C}$ be a class of $\mathcal{P}$-refinable contexts. Let E and F be processes. If $E, F^0[Y] \in \mathcal{C}$, then $Ref(r, E, F)$ is a process in $\mathcal{P}$ and it is a $\mathcal{P}$-refinable context in $\mathcal{C}$.*

In order to apply Theorem 5 we need to be able to characterize classes of $\mathcal{P}$-refinable contexts. In the following section we analyze one of the security property considered in [3], namely P_BNDC, and we show how to apply Theorem 5.

5 Action Refinement and Information Flow Security

Information flow security in a multilevel system aims at guaranteeing that no high level (confidential) information is revealed to users running at low security levels [11, 9, 16], even in the presence of any possible malicious process (attacker). *Persistent Bisimulation-based Non Deducibility on Composition* (P_BNDC, for short) [10] is an information flow security property suitable to analyze processes in completely dynamic hostile environments, i.e., environments which can be

dynamically reconfigured at run-time. The notion of P_BNDC is based on the idea of Non-Interference [12] and requires that every state which is reachable by the system still satisfies a basic Non-Interference property. If this holds, one is assured that even if the environment changes during the execution no malicious attacker will be able to compromise the system, as every possible reachable state is guaranteed to be secure. In this paper we present P_BNDC through its unwinding characterization (see [3]).

The definition of P_BNDC in terms of unwinding condition points out that all the high level actions of a P_BNDC process can be locally simulated by a sequence of τ actions.

Definition 9 (P_BNDC). *A process E is P_BNDC if for all E' reachable from E and for all $h \in H$, if $E' \xrightarrow{h} E''$, then $E' \stackrel{\hat{\tau}}{\Longrightarrow} E'''$ with $E'' \setminus H \approx E''' \setminus H$.*

Example 8. Let $l \in L$ and $h \in H$. The process $h.l.h.\mathbf{0} + \tau.l.\mathbf{0}$ is P_BNDC. The process $h.l.\mathbf{0}$ is not P_BNDC. □

Example 9. Let us consider a distributed data base (adapted from [14]) which can take two values and which can be both queried and updated. In particular, the high level user can query it through the high level actions qry_1 and qry_2, while the low level user can only update it through the low level actions upd_1 and upd_2. Hence $qry_1, qry_2 \in H$ and $upd_1, upd_2 \in L$. We can model the data base with the SPA process E defined as

$$E \equiv recZ.(qry_1.Z + upd_1.Z + \tau.Z + \\ upd_2.recW.(qry_2.W + upd_2.W + \tau.W + upd_1.Z)).$$

The process E is P_BNDC. Indeed, whenever a high level user queries the data base with a high level action moving the system to a state X then a τ action moving the system to the same state X may be performed, thus masking the high level interactions with the system to low level users. □

The decidability of P_BNDC has been proved in [10] and an efficient (polynomial) algorithm has been presented in [3]. A proof system which allows us to incrementally build P_BNDC processes has been obtained by exploiting both the unwinding characterization of P_BNDC (Definition 9) and the compositional properties of P_BNDC with respect to most of the operators of the SPA language. Here we exploit the same compositionality properties to define classes of P_BNDC and P_BNDC-refinable contexts.

Definition 10 (The classes $\mathcal{C}_{rec}$ and $\mathcal{C}_{par}$).

- *$\mathcal{C}_{rec}$ is the class of contexts containing: the process $\mathbf{0}$; Z, where Z is a variable; $l.C$, with $l \in L \cup \{\tau\}$, $h.C + \tau.C$, with $h \in H$, $C \setminus v$, $C[f]$, $C_1 + C_2$, and $recZ.C$, with $C, C_1, C_2 \in \mathcal{C}_{rec}$.*
- *$\mathcal{C}_{par}$ is the class of contexts containing: the process $\mathbf{0}$; Z, where Z is a variable; $l.C$, with $l \in L \cup \{\tau\}$, $h.C + \tau.C$, with $h \in H$, $C \setminus v$, $C[f]$, $C_1 + C_2$, and $C_1 | C_2$, with $C, C_1, C_2 \in \mathcal{C}_{par}$.*

Theorem 6. *The classes $\mathcal{C}_{rec}$ and $\mathcal{C}_{par}$ are* P_BNDC*-refinable.*

Next corollary is an immediate consequence of Theorems 5 and 6.

Corollary 1. *Let E and F be processes. If $E, F^0[Y] \in \mathcal{C}_{rec}$ (resp. $\in \mathcal{C}_{par}$), then $Ref(r, E, F)$ is a P_BNDC process and it is in $\mathcal{C}_{rec}$ (resp. $\in \mathcal{C}_{par}$).*

Example 10. Consider again the abstract specification of the distributed data base represented through the SPA process E of Example 9. The process E belongs to the class $\mathcal{C}_{rec}$ of Definition 10. In fact, $C_1 \equiv qry_2.W + upd_2.W + \tau.W + upd_1.Z \in \mathcal{C}_{rec}$, then $C_2 \equiv recW.C_1 \in \mathcal{C}_{rec}$. Hence, $C_3 \equiv qry_1.Z + upd_1.Z + \tau.Z + upd_2.C_2 \in \mathcal{C}_{rec}$. Thus $E \equiv recZ.C_3 \in \mathcal{C}_{rec}$.

We can refine the update actions by requiring that each update is requested and confirmed, i.e., we refine upd_1 using $F_1 \equiv req_1.cnf_1.\mathbf{0}$ and upd_2 using $F_2 \equiv req_2.cnf_2.\mathbf{0}$, where $req_1, cnf_1, req_2, cnf_2$ are low security level actions. We obtain that the process $Ref(upd_2, Ref(upd_1, E, F_1), F_2)$ is

$$recZ.(qry_1.Z + req_1.cnf_1.Z + \tau.Z + req_2.cnf_2.recW.(qry_2.W + req_2.cnf_2.W + \tau.W + req_1.cnf_1.Z)).$$

Since $F_1^0[Y]$ and $F_2^0[Y]$ are in C_{rec}, by applying Theorem 1 we have that the process $Ref(upd_2, Ref(upd_1, E, F_1), F_2)$ is *P_BNDC*. □

By exploiting the compositionality of *P_BNDC* with respect to ! (see [4]) we obtain that the above results hold also if we extend the class $\mathcal{C}_{par}$ by including all the contexts of the form $!C$ with $C \in \mathcal{C}_{par}$.

We conclude this section observing that it is immediate to prove Theorem 1 also for the properties *Compositional P_BNDC* (*CP_BNDC*, for short) and *Progressing P_BNDC* (*PP_BNDC*, for short) presented in [3].

6 Related Work

Action refinement has been extensively studied in the literature. There are essentially two interpretations of action refinement: *semantic* and *syntactic* (see [13]). In the semantic interpretation an explicit refinement operator, written $E[r \rightarrow F]$, is introduced in the semantic domain used to interpret the terms of the algebra. The semantics of $E[r \rightarrow F]$ models the fact that r is an action of E to be refined by process F. In the syntactic approach, the same situation is modelled by syntactically replacing r by F in E. The replacement can be *static*, i.e., before execution, or *dynamic*, i.e., r is replaced as soon as it occurs while executing E. In order to correctly formalize the replacement, the process algebra is usually equipped with an operation of sequential composition (rather than the more standard action prefix), as, e.g., in ACP, since otherwise it would not be closed under the necessary syntactic substitution. Our approach to action refinement follows the static, syntactic interpretation. However, the use of context composition to realize the refinement allows us to keep the original SPA language without introducing a sequential composition operator for processes.

Our definition of action refinement is equivalent, in most cases, to the classical static syntactic approaches presented in the literature. We show this by comparing our definition with the one proposed by Aceto and Hennessy in [1] to model action refinement for CCS processes. First, observe that the language considered in [1] is a variation of CCS with the sequential operator ; but without recursion and renaming. Moreover, their semantics is expressed as a strong bisimilarity extended with a condition on the termination of processes, here denoted by $\sim_{\surd}$. In [1] a refinement is nothing but a function $\rho : \mathcal{L} \to \mathcal{E}$ which maps each action a into its refinement. Given a process E its refinement $E\rho$ is obtained by syntactically replacing each action a occurring in E with $\rho(a)$. Since by Definition 4 we can avoid the parallel operator in the refining process F, the following theorem holds.

Theorem 7. *Let E and F be two processes without recursion and renaming. Consider the function $\rho : \mathcal{L} \to \mathcal{E}$ defined as*

$$\rho(a) = \begin{cases} F & \text{if } a = r \\ a & \text{otherwise} \end{cases}$$

Let $E\rho$ be the refinement of E with ρ as defined in [1]. If F is a guarded process, then

$$E\rho \sim_{\surd} \mathit{Ref}(r, E, F).$$

Action refinement is also classified as *atomic* or *non-atomic*. Atomic refinement is based on the assumption that actions are atomic and their refinements should in some sense preserve this atomicity (see, e.g.,[7, 5]). As an example, consider the processes $E \equiv r.\mathbf{0}|b.\mathbf{0}$ and $F \equiv a_1.a_2.\mathbf{0}$. The refinement of r in E with F is a process $(a_1.a_2).\mathbf{0}|b.\mathbf{0}$ where the execution of $a_1.a_2.\mathbf{0}$ is non-interruptible, i.e., action b cannot be executed in between the execution of a_1 and a_2. On the other hand, non-atomic refinement is based on the view that atomicity is always relative to the current level of abstraction and may, in a sense, be destroyed by the refinement (see, e.g., [1, 8, 20]). In this paper we follow the *non-atomic* approach. Actually, this approach is on the whole more popular then the former.

In the literature the term *refinement* is also used to indicate any transformation of a system that can be justified because the transformed system implements the original one on the *same* abstraction level, by being more nearly executable, for instance more deterministic. The implementation relation is expressed in terms of pre-orders such as trace inclusion or various kinds of simulation. Many papers in this tradition can be found in [6]. The relations between this form of refinement and information flow security have been studied in [2].

References

1. L. Aceto and M. Hennessy. Adding Action Refinement to a Finite Process Algebra. *Information and Computation*, 115(2):179–247, 1994.
2. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Refinement Operators and Information Flow Security. In *Proc. of the 1st IEEE Int. Conference on Software Engineering and Formal Methods (SEFM'03)*, pages 44–53. IEEE, 2003.

3. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Verifying Persistent Security Properties. *Computer Languages, Systems and Structures*, 2004. To appear. Available at `http://www.dsi.unive.it/~srossi/cl04.ps`.
4. A. Bossi, D. Macedonio, C. Piazza, and S. Rossi. Information Flow Security and Recursive Systems. In *Proc. of the Italian Conference on Theoretical Computer Science (ICTCS'03)*, volume 2841 of *LNCS*, pages 369–382. Springer-Verlag, 2003.
5. G. Boudol. Atomic Actions. *Bulletin of the EATCS*, 38:136–144, 1989.
6. J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings*, volume 430 of *Lecture Notes in Computer Science*. Springer, 1990.
7. J. W. de Bakker and E. P. de Vink. Bisimulation Semantics for Concurrency with Atomicity and Action Refinement. *Fundamenta Informaticae*, 20(1):3–34, 1994.
8. P. Degano and R. Gorrieri. A Causal Operational Semantics of Action Refinement. *Information and Computation*, 122(1):97–119, 1995.
9. R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In R. Focardi and R. Gorrieri, editors, *Proc. of Foundations of Security Analysis and Design (FOSAD'01)*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.
10. R. Focardi and S. Rossi. Information Flow Security in Dynamic Contexts. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 307–319. IEEE, 2002.
11. S. N. Foley. A Universal Theory of Information Flow. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'87)*, pages 116–122. IEEE, 1987.
12. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'82)*, pages 11–20. IEEE, 1982.
13. U. Goltz, R. Gorrieri, and A. Rensink. Comparing Syntactic and Semantic Action Refinement. *Information and Computation*, 125(2):118–143, 1996.
14. R. Gorrieri and A. Rensink. Action Refinement. Technical Report UBLCS-99-09, University of Bologna (Italy), 1999.
15. H. Mantel. Possibilistic Definitions of Security - An Assembly Kit -. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 185–199. IEEE, 2000.
16. J. McLean. Security Models and Information Flow. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'90)*, pages 180–187. IEEE, 1990.
17. J. K. Millen. Unwinding Forward Correctability. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'94)*, pages 2–10. IEEE, 1994.
18. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
19. M. Nielsen, U. Engberg, and K. S. Larsen. Fully Abstract Models for a Process Language with Refinement. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 523–548. Springer-Verlag, 1989.
20. R. J. van Glabbeek and U. Goltz. Refinement of Actions and Equivalence Notions for Concurrent Systems. *Acta Informatica*, 37(4/5):229–327, 2001.
21. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.