Action Refinement in Process Algebra and Security Issues*

Annalisa Bossi[1], Carla Piazza[2], and Sabina Rossi[1]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
{bossi,srossi}@dsi.unive.it
[2] Dipartimento di Matematica e Infomatica, Università di Udine, Italy
carla.piazza@dimi.uniud.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*. We study the relationships between action refinement and security properties within the Security Process Algebra (SPA). First we formalize the concept of action refinement as a structural inductive transformation. Then we prove several compositional results which can be exploited in the stepwise development of processes. Finally, we consider information flow security properties for SPA processes and define a decidable class of secure processes which is closed under refinement.

1 Introduction

In the development of a complex system it is common practice first to describe it succinctly as a simple abstract specification and then to stepwise refine it towards 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 (e.g., [28]).

In the context of process algebra, the refinement methodology amounts to defining a mechanism for replacing abstract actions with more concrete terms. We adopt the terminology *action refinement* [16] to refer to this stepwise development of systems specified as terms of a process algebra. In the literature, action refinement is also referred to as *vertical refinement* as opposed to *horizontal refinement* indicating any transformation of a system making it more nearly executable, for instance more deterministic, without adding new actions or expanding sub-computations. The latter is usually expressed in terms of pre-orders such as trace inclusion or simulation. We studied the relationships between this second form of refinement and information flow security in [3]. However, we cannot use the results obtained in [3] to deal with vertical refinement since the two forms of refinement provide orthogonal mechanisms for program development.

* Supported by the MIUR projects 2005015785 "Fondamenti Logici dei Sistemi Distribuiti e Codice Mobile" and 2005015491 "Vincoli per la programmazione con insiemi, l'analisi di sistemi con automi, il ragionamento su intervalli e la bioinformatica".

In process algebra, action refinement is usually defined in languages including a sequential composition operator ";" that allows one to syntactically substitute a process for an action. So, for instance, the refinement of r in the process $a;r;b;\mathbf{0}$ with the process F can be defined as the process $a;F;b;\mathbf{0}$. This is the most followed approach (see, e.g., [1, 15]). However, many process algebras, e.g., CCS, do not include the sequential composition operator. Thus in order to support action refinement, action-prefixing is usually replaced by sequential composition. As noticed in [1] this modification requires to introduce a suitable notion of termination and to consequently adapt the semantic equivalences.

Here we follow a different approach and instead of modifying our language, we define action refinement as a structural inductive transformation. 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, denoted by $Ref(r, E, F)$, is intended to be obtained from E by expanding each occurrence of r in E through F. We assume that the process F indicates its termination by a distinguished label $\overline{\texttt{done}}$, i.e., following Milner's terminology (see [20]), F is *well-terminating*. The refined process is obtained by applying a structural inductive transformation based on the *Before* operator defined in [20] as:

$$Before[F, E] \stackrel{\text{def}}{=} (F[\bar{f}/\overline{\texttt{done}}]|f.E) \setminus \{f\}.$$

For instance, if E is the process $r.a.\mathbf{0}$ where r is the action we intend to refine by the process $F \stackrel{\text{def}}{=} b_1.b_2.\overline{\texttt{done}}.\mathbf{0}$, the refined process, denoted by $Ref(r, E, F)$, will be the process $Before[F, E] \stackrel{\text{def}}{=} (b_1.b_2.\bar{f}.\mathbf{0}|f.a.\mathbf{0}) \setminus \{f\}$ which corresponds to the sequential composition of processes F and $a.\mathbf{0}$, and hence it models the substitution of the action r in E with F. In practice we follow the static syntactic approach to action refinement (see, e.g., [22]).

The main motivation behind our approach is that of studying the relationships between action refinement and security. Indeed, in system development, it is important to consider security related issues from the very beginning. Considering security only at the final step could lead to a poor protection, or, even worse, could make it necessary to restart the development from scratch. On the other hand, taking into account security from the abstract specification level, better integrates it in the whole development process, possibly driving some implementation choices. 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 [11, 14, 18, 23], i.e., properties that allow one to express constraints on how information should flow among different groups of entities. These properties are 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. In [2] we studied persistent information flow security properties for the *Security Process Algebra* (SPA) introduced in [11]. These properties are obtained

as instances of a *generalized unwinding condition* which requires that each high level action is "simulated" in such a way that it is impossible for the low level user to infer which high level actions have been performed. This general framework allows us to uniformly deal with some decidable subclasses of the well-known *NDC* and *BNDC* properties for SPA processes defined in [11]. The fact that we do not modify our language to introduce action refinement allows us to reason on the relationships between action refinement and the security properties of SPA processes. In particular, we study the conditions under which our notions of security are preserved under action refinement.

The paper is organized as follows. Section 2 introduces the SPA language. In Section 3 we formalize the notion of action refinement and provide some compositionality results. In Section 4 we introduce our information flow security properties and define decidable classes of secure processes which are closed under action refinement. Finally, in Section 5 we discuss some related work. The proofs of the results presented in this paper are reported in [5].

2 The SPA Language

The *Security Process Algebra* (SPA) language [11] is a variation of Milner's CCS [20] where the set of visible actions is partitioned into two security levels, high and low, in order to specify multilevel systems. The SPA syntax is based on: 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 and low security actions such that $\overline{H} = H$ and $\overline{L} = L$. The syntax of SPA *terms* is as follows[3]:

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

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 an occurrence of Z is *free* in a term T if it 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 $E, F, G, \ldots$ We introduce also a notion of *bound* and *free* actions. We say that an action a is *bound* in a term T if it belongs to a restriction, i.e., $\setminus v$ occurs in T and $a \in v$, or is used in a relabelling operator, i.e., f occurs in T and $f(a) \neq a$ or $f(b) = a$ for $b \neq a$. We identify SPA terms up to α-conversion, thus we can assume that a bound action can occur only in a restriction or a relabelling operator or in their scopes. Hence, the set of actions occurring in a term T can be split into two disjoint sets: the set $bound(T)$ of actions which are bound in T and the set $free(T)$ of actions which are not bound in T.

[3] Actually in [11] 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 $a, \bar{a} \notin v$
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 semantics of SPA terms.

The operational semantics of SPA processes is given in terms of *Labelled Transition Systems* (LTS). 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. The operational semantics for an agent E is the subpart of the SPA LTS reachable from the initial state E. 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 Z is a free variable in T, 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 $T[Z]$.

We will use the following notations. If $t = t_1 \cdots t_n \in Act^*$ and $E \xrightarrow{t_1} \cdots \xrightarrow{t_n} E'$, then we write $E \xrightarrow{t} E'$ and we say that E' is *reachable* from E, also denoted by $E \rightsquigarrow E'$. We denote by $Reach(E)$ the set of all processes reachable from E. We also write $E \stackrel{t}{\Longrightarrow} E'$ if $E(\xrightarrow{\tau})^* \xrightarrow{t_1} (\xrightarrow{\tau})^* \cdots (\xrightarrow{\tau})^* \xrightarrow{t_n} (\xrightarrow{\tau})^* E'$ where $(\xrightarrow{\tau})^*$ denotes a (possibly empty) sequence of τ labelled transitions. If $t \in Act^*$, then $\hat{t} \in \mathcal{L}^*$ is the sequence gained by deleting all occurrences of τ from t. As a consequence,

$E \stackrel{\hat{a}}{\Longrightarrow} E'$ stands for $E \stackrel{a}{\Longrightarrow} E'$ if $a \in \mathcal{L}$, and for $E(\stackrel{\tau}{\rightarrow})^* E'$ if $a = \tau$ (note that $\stackrel{\tau}{\Longrightarrow}$ requires at least one τ transition while $\stackrel{\hat{\tau}}{\Longrightarrow}$ means zero or more τ transitions).

The concept of *behavioral equivalence* is used to establish equalities among processes and it is based on the idea that two processes have the same semantics if and only if their behavior cannot be distinguished by an external observer. We recall here the definition of *strong bisimulation* [20], which equates two processes when they are able to mutually simulate their behavior step by step.

Definition 1. (Strong Bisimulation) *A symmetric binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ over processes is a* strong bisimulation *if $(E, F) \in \mathcal{R}$ implies, for all $a \in Act$, if $E \stackrel{a}{\rightarrow} E'$, then there exists F' such that $F \stackrel{a}{\rightarrow} F'$ and $(E', F') \in \mathcal{R}$.*

Two processes E and F are strongly bisimilar*, denoted by $E \sim F$, if there exists a strong bisimulation $\mathcal{R}$ containing the pair (E, F).*

A SPA term with free variables is called *context*[4]. If $C[Y_1, \ldots, Y_n]$ is a context with free variables $Y_1, \ldots, Y_n$, then we denote by $C[T_1, \ldots, T_n]$ the term obtained from $C[Y_1, \ldots, Y_n]$ by simultaneously replacing all the occurrences of $Y_1, \ldots, Y_n$ with the terms $T_1, \ldots, T_n$, respectively. For instance, if $C[X] \stackrel{\text{def}}{=} h.\mathbf{0}|(l.X + \tau.\mathbf{0})$ and $D[X, Y] \stackrel{\text{def}}{=} (l.X + \tau.\mathbf{0})|Y$ are contexts, then 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 $(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})|\bar{l}.\mathbf{0}$.

Finally, observe that our calculus does not provide a sequential composition operator. However, following Milner [20], we can define it by introducing the convention that processes indicate their termination by a distinguished label $\overline{\texttt{done}}$.

Definition 2. (Strongly Well-terminating process) *Let F be a SPA process. F is* strongly well-terminating *if for every $F' \in Reach(F)$ it holds:*

(1) *$F' \stackrel{\texttt{done}}{\rightarrow}$ is impossible;*
(2) *if $F' \stackrel{\alpha}{\rightarrow} \mathbf{0}$ then $F' \sim \overline{\texttt{done}}.\mathbf{0}$;*
(3) *if $F' \stackrel{\overline{\texttt{done}}}{\rightarrow}$ then $F' \sim \overline{\texttt{done}}.\mathbf{0}$.*

Our definition is a slight variation of Milner's notion of well-termination. The latter simply consists of points (1) and (3) above (point (2) is omitted) and thus it models the class of processes which *may* indicate their termination but they may also not indicate it. Although the theory developed in this paper holds also for Milner's definition, we prefer to adopt the strong notion of well-termination since it leads to a more meaningful notion of refinement.

When F is strongly well-terminating, the sequential composition of processes F and E can be defined through the operator *Before* introduced by Milner in [20].

Definition 3. (Before operator) *Let E be a SPA term and F be a SPA process such that F is strongly well-terminating.*

$$Before[F, E] \stackrel{\text{def}}{=} (F[\bar{f}/\overline{\texttt{done}}]|f.E) \setminus \{f\}$$

where $\bar{f}/\overline{\texttt{done}}$ denotes the relabelling function replacing $\overline{\texttt{done}}$ with a new name $\bar{f}$.

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

3 Action Refinement

It is standard practice in software development to obtain the final program by first defining an abstract, possibly not executable, specification and then refining it until one arrives to a concrete specification that can directly be implemented. Abstract operations are replaced by more detailed programs which can possibly be further refined. In the context of process algebra, this stepwise development amounts to interpreting actions on a higher level of abstraction by more complex processes on a lower level. This is obtained by introducing a mechanism to transform actions into processes. There are several ways to do this. Here we follow a syntactic approach defining the refinement as a syntactic process transformation.

3.1 Action Refinement for SPA Processes

To define action refinement we need to specify (1) which are the processes F that can be used to refine a process E and (2) which are the actions r refinable in E. A process F can be used to refine a process E only if the free actions of E do not occur bound in F, and vice-versa. Notice that this condition is not restrictive since, by α-conversion, we can always assume that the two processes do not share bound actions. Moreover, we require that F is different from $\mathbf{0}$ and that it is strongly well terminating. In this case we say that F is *pluggable* in E.

Definition 4. (Pluggable terms) *Let E be a SPA term and F be a SPA process. F is* pluggable *in E if*

(*a*) $bound(E) \cap free(F) = bound(F) \cap free(E) = \emptyset$;
(*b*) *F is not the process $\mathbf{0}$;*
(*c*) *F is strongly well-terminating.*

Notice that in the above definition E is a SPA term, i.e., it may have free variables. This is necessary to allow us to define the notion of refinement by structural induction on E.

If F is pluggable in E, then an abstract action r occurring in E is refinable with F if r is not bound in E and it does not occur in F otherwise we would enter into an infinite loop of refinements. All these requirements are formalized in the following notion of refinability.

Definition 5. (Refinable actions) *Let E be a SPA term, F be a SPA process, and $r \in \mathcal{L}$. The action r is said to be* refinable in E with F *if:*

(*a*) *F is pluggable in E;*
(*b*) *$r \notin bound(E)$;*
(*c*) *r does not occur in F.*

Example 1. Consider the processe $E \stackrel{\text{def}}{=} (r.a.\mathbf{0} | \bar{a}.b.\mathbf{0}) \setminus \{a, \bar{a}\}$ and the process $F \stackrel{\text{def}}{=} c.d.\overline{\texttt{done}}.\mathbf{0}$. In this case the action r is refinable in E with F.

Consider now the process E as above and $F_1 \stackrel{\text{def}}{=} (b.\overline{\texttt{done}}.\mathbf{0} + c.d.\overline{\texttt{done}}.\mathbf{0}) \setminus \{b\}$. In this case condition (a) 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. However, it is immediate to see that we can exploit α-conversion and transform F_1 into $F_2 \stackrel{\text{def}}{=} (e.\overline{\texttt{done}}.\mathbf{0} + c.d.\overline{\texttt{done}}.\mathbf{0}) \setminus \{e\}$. Now, r is refinable in E with F_2. □

The intended meaning of the refinement of an abstract action r in a process E with a refining process F is that of expanding each occurrence of r in E by F. In order to support action refinement, in the literature the prefixing operator is usually replaced by sequential composition ";" [1, 15]. Here we follow a different approach and model sequential composition by using a construction based on *well-terminating* processes and the *Before* operator as suggested in [20].

Let r be an action refinable in E with F. To define the refinement of E with F we replace each occurrence of r in E through the *Before* operator having F as first argument and the subprocess of E which follows r as second argument. Thus, for instance the refinement of r in $E \stackrel{\text{def}}{=} a.r.b.\mathbf{0}$ with $F \stackrel{\text{def}}{=} c.d.\overline{\texttt{done}}.\mathbf{0}$ is obtained by replacing $r.b.\mathbf{0}$ with $Before[F, b.\mathbf{0}]$, i.e., it is $a.Before[c.d.\overline{\texttt{done}}.\mathbf{0}, b.\mathbf{0}]$ that is exactly $a.(c.d.\overline{\texttt{done}}.\mathbf{0}[\bar{f}/\overline{\texttt{done}}] | f.b.\mathbf{0}) \setminus \{f\}$.

The notion of *action refinement* is defined by structural induction on the term to be refined as follows:

Definition 6. (Action Refinement) *Let E be a SPA term and F be a SPA process such that r is an action refinable in E with F. The* refinement of r in E with F *is the term $Ref(r, E, F)$ inductively defined as follows:*

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

Point (3) of the above definition deals with the basic case in which we replace an occurrence of r with the refining process F. If $E \stackrel{\text{def}}{=} r.E_1$ and r is the only occurrence of r in E, then $Ref(r, E, F) \stackrel{\text{def}}{=} Before[F, E_1] \stackrel{\text{def}}{=} (F[\bar{f}/\overline{\texttt{done}}] | f.E_1) \setminus \{f\}$ 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: indeed, condition (a) of Definition 4 ensures that undesired bindings of actions will never occur, while condition (b) of Definition 5 guarantees that we never refine restricted or relabelled actions. Point (c) of Definition 5 is useful to prevent infinite loops of refinements.

Point (b) of Definition 4 requires that F is not the empty process. This choice is motivated by the fact that in the literature there is no general agreement on what an empty refinement, i.e., the refinement of an action into the empty process, should be. In [25] actions refined into the empty process are simply erased (*forgetful refinements*), while in [10] those actions are deadlocked since the empty refinement is interpreted as an erroneous step in the top down development procedure. In many other works the empty refinement is simply ignored in order to avoid technical problems (see [1]). Here we follow this approach and assume that the refining process is always non empty.

Finally, point (c) of Definition 4 requires that the refining process F is strongly well-terminating. This allows us to define the sequential composition of SPA processes in the spirit of [20]. In the literature, the sequential composition operator ";" is just added to the language in order to allow, for instance, the refinement of $E \stackrel{\text{def}}{=} r.a.\mathbf{0}$ with $F \stackrel{\text{def}}{=} b.\mathbf{0}|c.\mathbf{0}$ obtaining the refined process $(b.\mathbf{0}|c.\mathbf{0});a.\mathbf{0}$. Using our definition, F is not pluggable in E since it is not well-terminating. Notice that we cannot simply replace the $\mathbf{0}$'s of F with $\overline{\mathtt{done}}.\mathbf{0}$, since the resulting process would not be well-terminating. However, following Milner [20], we can define the *strongly well-terminating* parallel composition operator:

$$P \; Par \; Q \stackrel{\text{def}}{=} (P[\bar{f}_1/\overline{\mathtt{done}}] \mid Q[\bar{f}_2/\overline{\mathtt{done}}] \mid (f_1.f_2.\overline{\mathtt{done}}.\mathbf{0} + f_2.f_1.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{f_1, f_2\}$$

The process $P \; Par \; Q$ is strongly well-terminating and performs an action $\overline{\mathtt{done}}$ when and only when both component agents have terminated. Thus, in the above example, we can use the well-terminating process $b.\overline{\mathtt{done}}.\mathbf{0} \; Par \; c.\overline{\mathtt{done}}.\mathbf{0}$ to refine the action r in $E \stackrel{\text{def}}{=} r.a.\mathbf{0}$.

Example 2. Let $E \stackrel{\text{def}}{=} r.a.\mathbf{0} + b.\mathbf{0}$ and $F \stackrel{\text{def}}{=} c.\overline{\mathtt{done}}.\mathbf{0} + d.\overline{\mathtt{done}}.\mathbf{0}$. It is immediate to observe that r is refinable in E with F. By applying Definition 6 we get:

$$\begin{aligned}
\mathit{Ref}(r, E, F) &\stackrel{\text{def}}{=} \mathit{Ref}(r, r.a.\mathbf{0}, F) + \mathit{Ref}(r, b.\mathbf{0}, F) \\
&\stackrel{\text{def}}{=} \mathit{Before}[F, \mathit{Ref}(r, a.\mathbf{0}, F)] + b.\mathbf{0} \\
&\stackrel{\text{def}}{=} \mathit{Before}[F, a.\mathbf{0}] + b.\mathbf{0} \\
&\stackrel{\text{def}}{=} (c.\overline{\mathtt{done}}.\mathbf{0} + d.\overline{\mathtt{done}}.\mathbf{0}[\bar{f}/\overline{\mathtt{done}}] | f.a.\mathbf{0}) \setminus \{f\} + b.\mathbf{0} \\
&\sim c.\tau.a.\mathbf{0} + d.\tau.a.\mathbf{0} + b.\mathbf{0}.
\end{aligned}$$

□

Example 3. Let $E \stackrel{\text{def}}{=} (a.r.b.\mathbf{0}) \setminus \{b\}$ and $F \stackrel{\text{def}}{=} c.d.\overline{\mathtt{done}}.\mathbf{0}$. Since $\mathit{bound}(E) = \{b\}$ and b does not occur in F, we have that r is refinable in E with F. Indeed:

$$\begin{aligned}
\mathit{Ref}(r, E, F) &\stackrel{\text{def}}{=} (\mathit{Ref}(r, a.r.b.\mathbf{0}, F)) \setminus \{b\} \\
&\stackrel{\text{def}}{=} (a.\mathit{Ref}(r, r.b.\mathbf{0}, F)) \setminus \{b\} \\
&\stackrel{\text{def}}{=} (a.\mathit{Before}[F, \mathit{Ref}(r, b.\mathbf{0}, F)]) \setminus \{b\} \\
&\stackrel{\text{def}}{=} (a.\mathit{Before}[F, b.\mathbf{0}]) \setminus \{b\} \\
&\stackrel{\text{def}}{=} (a.(c.d.\overline{\mathtt{done}}.\mathbf{0}[\bar{f}/\overline{\mathtt{done}}] | f.b.\mathbf{0}) \setminus \{f\}) \setminus \{b\} \\
&\sim (a.c.d.\tau.b.\mathbf{0}) \setminus \{b\}.
\end{aligned}$$

Notice that, our notion of refinability does not allow us to directly refine r in E with $F_1 \stackrel{\text{def}}{=} b.d.\overline{\texttt{done}}.\mathbf{0}$. However, we can first apply an α-conversion transforming E into the equivalent process $E_1 \stackrel{\text{def}}{=} (a.r.e.\mathbf{0})\setminus\{e\}$ and then refine r in E_1 with F_1 getting, as expected, the refined process which behaves as $(a.b.d.\tau.e.\mathbf{0})\setminus\{e\}$. □

Example 4. Let $E \stackrel{\text{def}}{=} a.r.b.\mathbf{0}|r.c.\mathbf{0}$ and $F \stackrel{\text{def}}{=} c.d.\overline{\texttt{done}}.\mathbf{0}$. By applying our definition and by the example above we get:

$$\begin{aligned} \mathit{Ref}(r,E,F) &\stackrel{\text{def}}{=} \mathit{Ref}(r,a.r.b.\mathbf{0},F)|\mathit{Ref}(r,r.c.\mathbf{0},F) \\ &\stackrel{\text{def}}{=} (a.(c.d.\overline{\texttt{done}}.\mathbf{0}[\bar{f}/\overline{\texttt{done}}]|f.b.\mathbf{0})\setminus\{f\}) \mid \\ &\qquad\qquad (c.d.\overline{\texttt{done}}.\mathbf{0}[\bar{f}/\overline{\texttt{done}}]|f.c.\mathbf{0})\setminus\{f\} \\ &\sim a.c.d.\tau.b.\mathbf{0}|c.d.\tau.c.\mathbf{0}. \end{aligned}$$

As expected, the two occurrences of r in E are replaced by two copies of F. □

From now on when we write $\mathit{Ref}(r,E,F)$ we tacitly assume that r is refinable in E with F. Notice that if r is refinable in E with F and E is strongly well-terminating then also $\mathit{Ref}(r,E,F)$ is strongly well-terminating.

3.2 Compositionality

At any fixed level of abstraction 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 do not care about the ordering in which the refinements occur.

First we show that our refinement can locally be applied to the subcomponents in which the actions to be refined occur.

Lemma 1. *Let $C[Z_1,\ldots,Z_n]$ be a SPA context, $E_1,\ldots,E_n$ be SPA terms, F be a SPA process, and $r \in \mathcal{L}$ be refinable in $C[E_1,\ldots,E_n]$ with F. Then*

$$\mathit{Ref}(r,C[E_1,\ldots,E_n],F) \stackrel{\text{def}}{=} \mathit{Ref}(r,C,F)[\mathit{Ref}(r,E_1,F),\ldots,\mathit{Ref}(r,E_n,F)].$$

In particular, if C is a context with no occurrences of r, the above lemma ensures that $\mathit{Ref}(r,C[E_1,\ldots,E_n],F) \stackrel{\text{def}}{=} C[\mathit{Ref}(r,E_1,F),\ldots,\mathit{Ref}(r,E_n,F)]$. Therefore, if we consider a process E of the form $E_1|E_2|\ldots|E_n$ and an action r occurring only in E_i for some i, then it is sufficient to apply the refinement to E_i to obtain $\mathit{Ref}(r,E,F) \stackrel{\text{def}}{=} E_1|\ldots|\mathit{Ref}(r,E_i,F)|\ldots|E_n$.

If we need to refine two actions in a term E, they can be swapped in the following sense.

Lemma 2. *Let E be a SPA term, F_1,F_2 be SPA processes, r_1 and r_2 be actions refinable in E with F_1 and F_2, respectively. If r_1 does not occur in F_2, then*

$$\mathit{Ref}(r_2,\mathit{Ref}(r_1,E,F_1),F_2) \stackrel{\text{def}}{=} \mathit{Ref}(r_1,\mathit{Ref}(r_2,E,F_2),\mathit{Ref}(r_2,F_1,F_2)).$$

In particular, if also r_2 does not occur in F_1, then

$$\mathit{Ref}(r_2,\mathit{Ref}(r_1,E,F_1),F_2) \stackrel{\text{def}}{=} \mathit{Ref}(r_1,\mathit{Ref}(r_2,E,F_2),F_1).$$

4 Preserving Security Properties under Refinement

In this section we first present some information flow security properties for SPA processes. Then we investigate conditions under which our notions of security are preserved under action refinement.

4.1 Security Properties

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 [13, 19], even in the presence of any possible malicious process (attacker).

In [11] Focardi and Gorrieri introduce the properties *Non-Deducibility on Compositions* (*NDC*) and *Bisimulation-based Non-Deducibility on Compositions* (*BNDC*) in order to capture every possible information flow from a *classified* (*high*) level of confidentiality to an *untrusted* (*low*) one. The definitions of *NDC* and *BNDC* are based on the basic idea of Non-Interference [14]: "No information flow is possible from high to low if what is done at the high level *cannot interfere* in any way with the low level". More precisely, a system E is secure if what a low level user sees of the system is not modified by composing any high process Π to E. The concept of *low observation* is expressed in terms of an *equivalence relation on low level actions* between processes. The idea is that two systems cannot be distinguished by a low level observer if and only if they are equated by an equivalence relation considering low level actions only. The two properties *NDC* and *BNDC* differ only on the low level observation equivalence they consider. *NDC* is based on *trace equivalence on low actions*, denoted by $\approx^l_T$, while *BNDC* considers the notion of *weak bisimilarity on low actions*, denoted by $\approx^l_B$.

The definition of *weak bisimilarity on low actions* (*trace equivalence on low actions*) is the same as the definition of weak bisimilarity [20] (trace equivalence) except that low and silent actions only (belonging to the set $L \cup \{\tau\}$), instead of all actions (belonging to the set Act), are considered.

Weak bisimilarity on low actions equates two processes if they are able to mutually simulate their low level behavior step by step. Moreover, it does not care about internal τ actions.

Definition 7. (Weak Bisimulation on Low Actions) *A symmetric binary relation $\mathcal{R}$ over processes is a* weak bisimulation on low actions *if $(E, F) \in \mathcal{R}$ implies, for all $a \in L \cup \{\tau\}$, 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 on low actions*, denoted by $E \approx^l_B F$, if there exists a weak bisimulation on low actions $\mathcal{R}$ containing (E, F).*

Trace equivalence on low actions equates two processes if they have the same sets of low traces, again, without considering the τ actions.

Definition 8. (Trace Equivalence on Low Actions) *The set of traces $T^l(E)$ associated with a process E is defined by: $T^l(E) = \{t \in (L \cup \{\tau\})^* \mid \exists E' : E \stackrel{t}{\Longrightarrow}$*

$E'\}$. *Two processes* E, F *are* trace equivalent on low actions*, denoted by* $E \approx_T^l F$*, if* $T^l(E) = T^l(F)$.

Trace equivalence on low actions is less demanding than weak bisimilarity on low actions, hence if two processes are weakly bisimilar on low actions, then they are also trace equivalent on low actions.

Properties *BNDC* and *NDC* are thus formally defined as follows:

$$E \in BNDC \text{ if for all high processes } \Pi,\ \ E \approx_B^l (E|\Pi)$$
$$E \in NDC \text{ if for all high processes } \Pi,\ \ E \approx_T^l (E|\Pi).$$

Since weak bisimilarity on low actions is stronger than trace equivalence on low actions, it holds that *BNDC* implies *NDC*.

Properties *NDC* and *BNDC* are difficult to use in practice: *NDC* is not decidable in polynomial time, while the decidability of *BNDC* is still an open problem. In [12], Focardi and Rossi introduce the property *Persistent BNDC* (*P_BNDC*) which is a natural persistent extension of *BNDC* (i.e., a system E is *P_BNDC* if every state E' reachable from E is *BNDC*) and it is a sufficient condition for *BNDC*. They show the decidability of *P_BNDC* by exploiting a bisimulation based characterization. Other persistent security properties have been later introduced, e.g., the properties *Persistent NDC* (*P_NDC*) in [4] and *Compositional P_BNDC* (*CP_BNDC*) in [2].

All the *persistent* properties mentioned above can be defined as instances of a *generalized unwinding condition* [2] which requires that each high level action is "simulated" in such a way that it is impossible for the low level user to infer which high level actions have been performed. The generalized unwinding condition is parametric with respect to two binary relations on processes: an equivalence relation on low actions, $\backsim^l$, which represents the low level view, and a transition relation, $\dashrightarrow$, which characterizes a local connectivity.

Definition 9. (Generalized Unwinding) *Let* $\sim^l$ *be an equivalence relation on low actions and* $\dashrightarrow$ *be a binary relation on processes. The* unwinding class $\mathcal{W}(\backsim^l, \dashrightarrow)$ *is defined as*

$$\mathcal{W}(\backsim^l, \dashrightarrow) \stackrel{\text{def}}{=} \{E \in \mathcal{E} \mid \forall\ F, G \in Reach(E)$$
$$\text{if } F \stackrel{h}{\rightarrow} G \text{ then } \exists G' \text{ such that } F \dashrightarrow G' \text{ and } G \backsim^l G'\}.$$

It holds that *P_NDC* coincides with $\mathcal{W}(\approx_T^l, \stackrel{\hat{\tau}}{\Longrightarrow})$ [4], *P_BNDC* coincides with $\mathcal{W}(\approx_B^l, \stackrel{\hat{\tau}}{\Longrightarrow})$ and *CP_BNDC* coincides with $\mathcal{W}(\approx_B^l, \stackrel{\tau}{\Longrightarrow})$ [2]. Moreover, *P_NDC* $\subseteq$ *NDC* and *P_BNDC*, *CP_BNDC* $\subseteq$ *BNDC*.

Example 5. 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 6. Let us consider a distributed data base (adapted from [16]) 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

$$E \stackrel{\text{def}}{=} recZ.(qry_1.Z + upd_1.Z + \tau.Z + upd_2.recW.(qry_2.W + upd_2.W + \tau.W + upd_1.Z)).$$

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. □

4.2 Classes of Secure Processes closed under action refinement

In this section we investigate conditions under which our notions of security are preserved under action refinement. In particular, we are interested in the definition of classes of processes satisfying an instance of $\mathcal{W}(\sim^l, \dashrightarrow)$ and closed under action refinement.

We first introduce the concept of $(\mathcal{P}, r)$-refinable contexts, where $\mathcal{P}$ is a process property and r is an action. Intuitively, a class of contexts is $(\mathcal{P}, r)$-refinable if all processes in it satisfy $\mathcal{P}$ and it is closed under refinement of the action r.

Definition 10 ($(\mathcal{P}, r)$-refinable contexts). *Let $\mathcal{P}$ be a class of processes and r be an action. A class $\mathcal{C}$ of contexts is said to be* $(\mathcal{P}, r)$-refinable *if:*

- *if $E \in \mathcal{C}$ and E is a process, then $E \in \mathcal{P}$;*
- *if $E, F \in \mathcal{C}$ and r is refinable in E with F then $Ref(r, E, F) \in \mathcal{C}$.*

We introduce a parametric definition of classes of contexts. Given a sequence $s = s_1, s_2, \ldots, s_n$ of actions, we denote by $s.E$ the process $s_1.s_2.\ldots.s_n.E$. Moreover, given a set v of actions we denote by $s \cap v$ the set of actions occurring both in s and in v, while, given a relabelling f we denote by $f[s]$ the set $\{s_i \mid f(s_i) \neq s_i\}$. A relation $\circ$ over terms is a *congruence* if $C_1[Z] \circ C_2[Z]$ and $E_1 \circ E_2$ imply $C_1[E_1] \circ C_2[E_2]$.

Definition 11. **($\mathcal{C}(\asymp, s)$)** *Let $\asymp$ be a reflexive congruence over SPA terms and s be a sequence of actions. $\mathcal{C}(\asymp, s)$ is the class of contexts containing: the process $\mathbf{0}$; Z, where Z is a variable; $l.C_1$, $h.C_1 + s.C_1'$, $C_1 \setminus v$, $C_1[f]$, $C_1 + C_2$, $C_1|C_2$, and $recZ.C_1$, with $l \in L \cup \{\tau\}$, $h \in H$, $C_1 \asymp C_1'$, $s \cap v = \emptyset$, $f[s] = \emptyset$, and $C_1, C_1', C_2 \in \mathcal{C}(\asymp, s)$.*

Let $\dashrightarrow$ be a binary relation on processes, we say that s *entails* $\dashrightarrow$ if $E \stackrel{s}{\rightarrow} E'$ implies $E \dashrightarrow E'$. Let $\asymp$ be a binary relation on terms, we say that $\asymp$ is *preserved under refinement* of action r if $E \asymp E'$ implies $Ref(r, E, F) \asymp Ref(r, E', F)$.

The following theorem provides sufficient conditions to ensure that all the processes in the class $\mathcal{C}(\asymp, s)$ are secure and the class itself is closed under refinement of r.

Theorem 1. *Let $\mathcal{W}(\sim^l, \dashrightarrow)$ be an unwinding condition. If $s \in (L \cup \{\tau\})^*$ is a sequence of low and silent actions which entails $\dashrightarrow$, r is an action which does not occur in s, $\asymp$ is a reflexive congruence preserved under refinement of r, and $\asymp \cap (\mathcal{E} \times \mathcal{E}) \subseteq \sim^l$, then the class $\mathcal{C}(\asymp, s)$ is $(\mathcal{W}(\sim^l, \dashrightarrow), r)$-refinable.*

Moreover, under the above conditions, if E and F are two processes such that $E, F \in \mathcal{C}(\asymp, s)$ and r is a refinable action in E with F, then $Ref(r, E, F) \in \mathcal{W}(\sim^l, \dashrightarrow)$ and $Ref(r, E, F) \in \mathcal{C}(\asymp, s)$.

Let $\equiv$ denote the syntactic equality between SPA terms. It is immediate to see that $\equiv$ is a reflexive congruence preserved under refinement and it is included in $\sim^l$ for each binary relation $\sim^l$ over SPA terms. We can then instantiate the above theorem with $\equiv$ as the relation $\asymp$, obtaining the following corollary.

Corollary 1. *Let $\mathcal{W}(\sim^l, \dashrightarrow)$ be an unwinding condition. If $s \in (L \cup \{\tau\})^*$ is a sequence of low and silent actions which entails $\dashrightarrow$, r is an action which does not occur in s, then the class $\mathcal{C}(\equiv, s)$ is $(\mathcal{W}(\sim^l, \dashrightarrow), r)$-refinable.*

Moreover, under the above conditions, if E and F are two processes such that $E, F \in \mathcal{C}(\equiv, s)$ and r is a refinable action in E with F, then $Ref(r, E, F) \in \mathcal{W}(\sim^l, \dashrightarrow)$ and $Ref(r, E, F) \in \mathcal{C}(\equiv, s)$.

Example 7. Consider again the abstract specification of the distributed data base represented through the SPA process E of Example 6. The process E belongs to the class $\mathcal{C}(\equiv, \tau)$ of Definition 11. In fact, $C_1 \stackrel{\text{def}}{=} qry_2.W + upd_2.W + \tau.W + upd_1.Z \in \mathcal{C}(\equiv, \tau)$, and then $C_2 \stackrel{\text{def}}{=} recW.C_1 \in \mathcal{C}(\equiv, \tau)$. Hence, $C_3 \stackrel{\text{def}}{=} qry_1.Z + upd_1.Z + \tau.Z + upd_2.C_2 \in \mathcal{C}(\equiv, \tau)$. Therefore $E \stackrel{\text{def}}{=} recZ.C_3 \in \mathcal{C}(\equiv, \tau)$.

We can refine the update actions by requiring that each update is requested and confirmed, i.e., we refine upd_1 with $F_1 \stackrel{\text{def}}{=} req_1.cnf_1.\overline{\texttt{done}}.\mathbf{0}$ and upd_2 with $F_2 \stackrel{\text{def}}{=} req_2.cnf_2.\overline{\texttt{done}}.\mathbf{0}$, where $req_1, cnf_1, req_2, cnf_2$ are low security level actions. We obtain:

$$
\begin{array}{l}
Ref(upd_2, Ref(upd_1, E, F_1), F_2) \stackrel{\text{def}}{=} \\
Ref(upd_2, Ref(upd_1, recZ.(qry_1.Z + upd_1.Z + \tau.Z + \\
\qquad upd_2.recW.(qry_2.W + upd_2.W + \tau.W + upd_1.Z)), F_1), F_2) \stackrel{\text{def}}{=} \\
recZ.(qry_1.Z + (req_1.cnf_1.\overline{\texttt{done}}.\mathbf{0}[\bar{f}/\overline{\texttt{done}}]|f.Z) \setminus \{f\} + \tau.Z + \\
\qquad (req_2.cnf_2.\overline{\texttt{done}}.\mathbf{0}[\bar{f}/\overline{\texttt{done}}]|f.(recW.(qry_2.W + \\
\qquad (req_2.cnf_2.\overline{\texttt{done}}.\mathbf{0}[\bar{f}/\overline{\texttt{done}}]|f.W) \setminus \{f\} + \\
\qquad \tau.W + (req_1.cnf_1.\overline{\texttt{done}}.\mathbf{0}[\bar{f}/\overline{\texttt{done}}]|f.Z) \setminus \{f\})) \setminus \{f\} \sim \\
recZ.(qry_1.Z + req_1.cnf_1.\tau.Z + \tau.Z + \\
\qquad req_2.cnf_2.\tau.recW.(qry_2.W + req_2.cnf_2.\tau.W + \tau.W + req_1.cnf_1.\tau.Z)).
\end{array}
$$

Since F_1 and F_2 are in $\mathcal{C}(\equiv, \tau)$ and τ entails $\stackrel{\hat{\tau}}{\Longrightarrow}$, by applying Corollary 1 we have that the process $Ref(upd_2, Ref(upd_1, E, F_1), F_2)$ is in $\mathcal{W}(\approx^l_B, \stackrel{\hat{\tau}}{\Longrightarrow})$, i.e., it is *P_BNDC*. □

Another binary relation over SPA terms, which can be used to find sufficient and decidable conditions for proving both *P_NDC* and *P_BNDC* is the relation $\eqsim$ defined as follows: $E_1 \eqsim E_2$ if and only if $E_1 \equiv E_2$ or $E_1 \sim E_2$ and E_1, E_2

are high level processes (i.e., they have neither variables nor low level actions) or $E_1 \equiv D_1[E'_1]$ and $E_2 \equiv D_2[E'_2]$, with $D_1[Z] \eqsim D_2[Z]$ and $E'_1 \eqsim E'_2$.

By instantiating $\asymp$ with $\eqsim$ and s with τ we obtain the class $\mathcal{C}(\eqsim, \tau)$ which is decidable with the proof system presented in the Appendix. By exploiting Theorem 1, we can prove that the class of secure processes $\mathcal{C}(\eqsim, \tau)$ is closed under refinement of low level actions, as stated by the following corollary.

Corollary 2. *The class $\mathcal{C}(\eqsim, \tau)$ is both (P_NDC, r)-refinable and (P_BNDC, r)-refinable, for all low level actions $r \in L$.*

The proof follows from the fact that P_NDC coincides with $\mathcal{W}(\approx^l_T, \stackrel{\hat{\tau}}{\Longrightarrow})$, P_BNDC coincides with $\mathcal{W}(\approx^l_B, \stackrel{\hat{\tau}}{\Longrightarrow})$, and τ entails $\stackrel{\hat{\tau}}{\Longrightarrow}$. In order to prove that $\eqsim$ is preserved under refinement we exploit Lemma 1.

Notice that we can obtain the same result by replacing $\sim$ (strong bisimulation) in the definition of $\eqsim$ with any congruence included in $\approx^l_B$. For instance we can use the weak progressing bisimulation defined in [21].

5 Conclusions and Related Works

In this paper we study the relationships between action refinement and information flow security within the Security Process Algebra (SPA).

Action refinement has been extensively studied in the literature. There are essentially two interpretations of action refinement: *semantic* and *syntactic* (see [15]). 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 the 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. The use of the *Before* operator to realize the refinement allows us to keep the original SPA language without introducing a sequential composition operator for processes.

In [1] Aceto and Hennessy introduce a static syntactic notion of action refinement on a variation of CCS in which action-prefixing is replaced by sequential composition and neither recursion nor relabellings are allowed. The semantics of this language is expressed as a strong bisimilarity extended with a condition on the termination of processes. Instead of extending the language, we follow Milner's approach and implement sequential composition by context operations. This allows us to consider the full language with recursion and relabelling.

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.,[9, 6]). As an example, consider the processes $E \stackrel{\text{def}}{=} r.\mathbf{0}|b.\mathbf{0}$ and $F \stackrel{\text{def}}{=} a_1.a_2.\overline{\texttt{done}}.\mathbf{0}$. The atomic refinement of r in E with F should be a process where r is replaced by F and the execution of $a_1.a_2$ 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, 10, 26]). Unfortunately, the standard behavioral equivalences of CCS, such as strong and weak bisimulation and trace equivalence, are not preserved under non-atomic refinements. In the literature different equivalences based on non-interleaving semantics which are preserved under refinement have been studied (see, e.g., [7, 27]). In this paper we follow the *non-atomic* approach. Actually, this approach is on the whole more popular than the former.

Recently in [24], Seehusen and Stølen addressed the problem of preserving trace-based security properties under transformations from an abstract specification to a concrete one. The particular transformations they deal with may be understood as a special case of action refinement where the concrete specification is generated automatically from the abstract specification. The information flow security framework presented in the paper is inspired by [18] and is based on the composition of basic security predicates. This approach is quite simple and allows one to capture many trace-based properties expressed over event systems. Following Jacob's observations [17], the authors notice that information flow properties are in general not preserved by the standard notions of refinement. As argued by Jacob, the problem originates from the inability of most specification languages to distinguish between the two sources of nondeterminism, named, underspecification and unpredictability. The authors then propose to refine the notion of refinement and that of secure information flow such that this distinction is taken into consideration. Based on this approach they propose quite *ad hoc* conditions under which transformations maintain security.

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 [8]. The relations between this form of refinement and information flow security have been studied in [3]. Although both action refinement and the refinement considered in [3] aim at transforming a system specification into a more executable one, the principles behind the two kinds of transformations are completely different, and thus a comparison is not meaningful.

6 Acknowledgments

We would like to acknowledge the anonymous referees of the preliminary version of the paper for their useful comments and suggestions.

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, D. Macedonio, C. Piazza, and S. Rossi. Unwinding in Information Flow Security. *Electronic Notes in Theoretical Computer Science*, 99:127–154, 2004.
3. 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 Computer Society Press, 2003.
4. A. Bossi, C. Piazza, and S. Rossi. Modelling Downgrading in Information Flow Security. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 187–201. IEEE Computer Society Press, 2004.
5. A. Bossi, C. Piazza, and S. Rossi. Action Refinement in Process Algebra and Security Issues. Technical Report CS-2007-8, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy, 2007.
6. G. Boudol. Atomic Actions. *Bulletin of the EATCS*, 38:136–144, 1989.
7. M. Bravetti and R. Gorrieri. Deciding and axiomatizing weak ST bisimulation for a process algebra with recursion and action refinement. *ACM Transaction on Computational Logic*, 3(4):465–520, 2002.
8. 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.
9. 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.
10. P. Degano and R. Gorrieri. A Causal Operational Semantics of Action Refinement. *Information and Computation*, 122(1):97–119, 1995.
11. 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.
12. R. Focardi and S. Rossi. Information Flow Security in Dynamic Contexts. *Journal of Computer Security*, 14(1):65–110, 2006.
13. 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 Computer Society Press, 1987.
14. 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 Computer Society Press, 1982.
15. U. Goltz, R. Gorrieri, and A. Rensink. Comparing Syntactic and Semantic Action Refinement. *Information and Computation*, 125(2):118–143, 1996.
16. R. Gorrieri and A. Rensink. Action Refinement. Technical Report UBLCS-99-09, University of Bologna (Italy), 1999.
17. J. Jacob. On the Derivation of Secure Components. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'89)*, pages 242–247. IEEE Computer Society Press, 1989.
18. H. Mantel. Possibilistic Definitions of Security - An Assembly Kit -. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 185–199. IEEE Computer Society Press, 2000.

19. J. McLean. Security Models and Information Flow. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'90)*, pages 180–187. IEEE Computer Society Press, 1990.
20. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
21. U. Montanari and V. Sassone. CCS Dynamic Bisimulation is Progressing. In *Proc. of the Int. Symposium on Mathematical Foundations of Computer Science (MFCS'91)*, volume 520 of *LNCS*, pages 346–356. Springer-Verlag, 1991.
22. 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.
23. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.
24. F. Seehusen and K. Stølen. Maintaining Information Flow Security Under Refinement and Transformation. In *Proceedings of the 4th International Workshop on Formal Aspects in Security and Trust (FAST'06)*, volume 4691 of *LNCS*, pages 143–157. Springer-Verlag, 2007.
25. R. J. van Glabbeek and U. Goltz. Equivalence Notions for Concurrent Systems and Refinement of Actions. In A. Kreczmar and G. Mirkowska, editors, *Proc. of Mathematical Foundations of Computer Science (MFCS'89)*, volume 379 of *LNCS*, pages 237–248. Springer-Verlag, 1989.
26. R. J. van Glabbeek and U. Goltz. Refinement of Actions and Equivalence Notions for Concurrent Systems. *Acta Informatica*, 37(4/5):229–327, 2001.
27. R. J. van Glabbeek and Frits W. Vaandrager. The Difference between Splitting in n and n+1. *Information and Computation*, 136(2):109–142, 1997.
28. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.