Static Detection of Logic Flaws in Service-Oriented Applications *

Chiara Bodei[1], Linda Brodo[2], and Roberto Bruni[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
{chiara,bruni}@di.unipi.it
[2] Dipartimento di Scienze dei Linguaggi, Università di Sassari, Italy
brodo@uniss.it

Abstract. Application or business logic, used in the development of services, has to do with the operations that define the application functionalities and not with the platform ones. Often security problems can be found at this level, because circumventing or misusing the required operations can lead to unexpected behaviour or to attacks, called *application logic attacks*. We investigate this issue, by using the CaSPiS calculus to model services, and by providing a Control Flow Analysis able to detect and prevent some possible misuses.

1 Introduction

More and more web surfers use applications based on web service technology for their transactions, such as bank operations or e-commerce purchases. The increasing availability of information exchange over e-services comes at the price of new security threats: new clever forms of attacks can come out at a rate that is growing with growth in usage.

Among the many different kinds of attacks that a malicious hacker can launch against web services applications, we here focus on the so-called *application logic attacks*, which are tailored to exploit the vulnerabilities of the specific functionalities of the application rather than the more general ones provided by the used platform, i.e. they violate application or business logic (see e.g., [19, 11]). This logic represents the functions or the services that a particular site provides, in terms of the steps required to finalise a business goal, e.g. in an e-shop, the application logic can establish that customers' personal data necessary to complete an order must be provided only after the shopping basket is completed. Logic bugs in the application design may open dangerous loopholes that allow a user to do something that isn't allowed by the business, just by abusing or misusing the functions of the application, even without modifying them. Unfortunately, often security is not considered from the very beginning of the application development: the focus is on what the user is expected and allowed to do and not on the possible pathological usage scenarios that a goofy user may encounter

* Research supported by the EU FET-GC2 Project IST-2005-016004 Sensoria, by the Italian PRIN Project "SOFT" and by the Italian FIRB Project Tocai.it.

or a malicious user may exploit. For example, take a conference management system handling blind peer reviews: a malicious author could insert the names of "unfriendly" Program Committee members as co-authors of her/his paper to make them in conflict and exclude their opinions from the discussion phase. The fictitious co-authoring could then be removed in case the paper got accepted, without compromising the overall consistency of the conference management system and review process. Of course to prevent this misuse it could suffice to notify all authors about the submission as part of the application logic.

Logical vulnerabilities are subtle, application specific, and therefore difficult to detect. As usual, in the development of complex systems resorting to formal methods can be helpful. In particular, we try to transfer and adapt some techniques used in the field of network security (see e.g.,[10, 8]). We develop a Control Flow Analysis for analysing the `close`-free fragment of CaSPiS [13, 15], a process calculus recently introduced for modelling service oriented computing. The key features of CaSPiS are a disciplined, built-in management of long-running dyadic (and possibly nested) sessions between services and their clients, together with data-flow and orchestration primitives such as the pipeline operator and pattern-matching primitives that are suited, e.g., to deal with XML-like data typical of web service scenarios. The analysis statically approximates the behaviour of CaSPiS processes, in terms of the possible service and communication synchronisations. More precisely, what the analysis predicts encompasses everything that may happen, while what the analysis *does not* predict corresponds to something that cannot happen. The session mechanism is particularly valuable for the kind of analysis we use, because it guarantees that sibling sessions established between different instances of the same service and the corresponding clients do not interfere one with the other by leaking information, with two main consequences: first, our analysis can focus on each client-server conversation separately and second, we can focus on the application logic, neither having to commit on any specific implementation of sessioning over a certain platform nor worrying about the analysis of such a realisation.

The analysis we propose borrows some ideas from [20, 7], because it exploits the similarity between the nesting hierarchies introduced by session primitives and the nesting hierarchies used in Ambients-like calculi. Furthermore to take care of malicious users, we modify the classical notion of Dolev-Yao attacker [17]: the attacker we are interested to model, that we call *malicious customer*, is an insider or, more precisely, an accredited user of a service that has no control of the communication channels, but that does not follow the intended rules of the application protocol, e.g., (s)he can cheat or introduce inconsistent data.

We apply our framework to an example inspired by a known logic-flawed application for e-commerce, where an abuse of functionality is possible, in which the attacker unexpectedly alters data, therefore modifying the application behaviour. The CyberOffice shopping cart [23] could be attacked by modifying the hidden field used for the price, within the web order form. The web page could be indeed downloaded normally, edited unexpectedly outside the browser and then submitted regularly with the prices set to any desired value, included zero or even a negative value. If no data-consistency control was performed by the server when the form was returned then the attack could be successful. Weak forms of validation could lead to similar problems, like when checking all item prices but not the total, or when checking goods price, but not the expedition costs. Intuitively, the information exchange between the customer C and the e-shop service S (and the data base DBI storing item prices) can be represented by the following informal protocol narration (steps from 1. to 4.), that we borrow from network security literature.

1. $C \rightarrow S: \quad ItemA$
2. $S \rightarrow DBI: Code, ItemA$
3. $DBI \rightarrow C: OrderForm(Code, ItemA, PriceA)$
4. $C \rightarrow S: \quad PaymentForm(Code, ItemA, PriceA, Name, Cc)$

...

Narration of the Protocol between Customer and E-shop Service

$4'. \quad C \rightarrow S: \quad PaymentForm'(Code, ItemA, FakedPriceA, Name, Cc)$

Attack on Fourth Step

The customer chooses an item $ItemA$, receives its price $PriceA$ inside an $OrderForm$ and can finalise the order by filling in a $PaymentForm$ with personal data like $Name$ and credit card information Cc. In the same form are reported: the transaction $Code$, the item and its price (for simplicity, we assume expedition expenses included). In case of pathological usage (step $4'$), the required information is added on a forged copy of the payment form, where the attacker has altered the price field, using the forged price $FakedPriceA$, instead of the one received from DBI within the $OrderForm$.

When modelled in CaSPiS, our analysis of the e-shop service is able to detect the possible price modification in harshly designed processes. The attack relies on the fact that S does not check that the third field of the received form has the correct value. This is because the application logic relies on step 4 to acquire personal data and credit card information of the customer that are considered as good enough credentials for establishing the "circle of trust" over the pending commercial transaction and it does not expect that misuses may still arise. Furthermore, to save on the number of exchanged messages, it delegates the DBI service (likely running on a separate, dedicated server) to communicate the price directly to the customer, so that S cannot perform any validation over it when the form is returned. A possible fix to the problem would consists in having the price information sent to S first and then redirected to the customer, so that it could be easy for the server to match the price included in $PaymentForm$ against the one received by DBI, as shown below.

$2''. \; S \rightarrow DBI: Code, ItemA$
$3''. \; DBI \rightarrow S: Code, ItemA, PriceA$
$4''. \quad S \rightarrow C: \quad OrderForm(Code, ItemA, PriceA)$
$5''. \quad C \rightarrow S: \quad PaymentForm(Code, ItemA, PriceA, Name, Cc)$
...

Alternative Narration

Related Work. Recently some works have faced the issue of security in the composition and verification of web applications. In [5] the authors propose security libraries, automatically analysed with ProVerif [6], using the WS-Security policies of [22]. WS ReliableMessaging is instead analysed in [24] with the AVISPA [3] verification toolkit, initially developed to check properties of cryptographic protocols. Recently, in [12], security properties have been considered in an extension of the π-calculus with session types and correspondence assertions [25]. In [1], behavioral types are used to statically approximate the behavior of processes. Building on [2, 15], the type system [18] for CaSPiS is instead introduced to address the access control properties related to security levels. Safety properties of service behaviour is faced in [4]. Services can enforce security policies locally and can invoke other services respecting given security contracts. Formal reasoning about systems is obtained by means of dynamic and static semantics.

The focus of our interest is in formalising the logic used to develop an application and discovering its intrinsic weaknesses, due to a design practice that does not consider security as a first-class concern.

Plan of the Paper. In Section 2, we present the calculus. In Section 3, we introduce the Control Flow Analysis and the analysis of the malicious customer or attacker. In Section 4, we apply our framework to the above shown example of logic-flawed application for e-commerce. Section 5 concludes the paper. Proofs of theorems and lemmata presented throughout the paper are reported in [9].

2 The Calculus

We introduce here the fragment of CaSPiS that is sufficient to handle our modelling needs (i.e., without the constructs for handling session termination): it is essentially the one considered in [15]. Due to space limitation, we refer the interested reader to [13, 14] for the motivation around CaSPiS design choices, its detailed description and many examples that witness its flexibility.

Syntax. Let $\mathcal{N} \ni n,\ n', ...$ be a countable set of names, that includes the set $\mathcal{N}_{srv} \ni s,\ s', ...$ of *service names* and the set $\mathcal{N}_{sess} \ni r,\ r', ...$ of *session names*, with $\mathcal{N}_{srv} \cap \mathcal{N}_{sess} = \emptyset$. We also let x range over variables (for service names and data). We distinguish here between *definition* occurrences and *use* occurrences of variables. A definition variable occurrence $?x$ is when x gets its binding value, while a use occurrence x is when the value has been bound. We assume that a variable cannot occur in both forms inside the same input, as in $(..., ?x, ..., x, ...)$. For the sake of simplicity, we let v range over values, i.e. names and use variables and $\tilde{v}$ on tuples. In the second part of the paper, we shall use a more general kind of input, including pattern matching. The syntax of CaSPiS is presented in Fig. 1, where the operators occur in decreasing order of precedence.

As usual, the empty summation is the nil process $\mathbf{0}$ (whose trailing is mostly omitted), parallel composition is denoted by $P|Q$ and restriction by $(\nu n)P$. The construct $r^p \triangleright P$ indicates a generic session side with polarity p (taking values

$$
\begin{array}{lll}
P, Q ::= & & \textit{processes} \\
 & \mid\ s.P & \text{service definition} \\
 & \mid\ \overline{v}.P & \text{service invocation} \\
 & \mid\ \Sigma_{i \in I} \pi_i P_i & \text{guarded sum} \\
 & \mid\ r^p \triangleright P & \text{session (considered as run-time syntax)} \\
 & \mid\ P > (?\tilde{x})Q & \text{pipeline (written } P > \tilde{x} > Q \text{ in the literature)} \\
 & \mid\ (\nu n)P & \text{restriction} \\
 & \mid\ P|Q & \text{parallel composition} \\
 & \mid\ !P & \text{replication} \\
 & & \\
p, q\ \ ::= +|- & & \textit{polarities} \\
 & & \\
\pi, \pi' ::= & & \textit{action prefixes} \\
 & \mid\ (?\tilde{x}) & \text{input} \\
 & \mid\ \langle \tilde{v} \rangle & \text{output} \\
 & \mid\ \langle \tilde{v} \rangle^{\uparrow} & \text{return}
\end{array}
$$

Fig. 1. Syntax of CaSPiS

in $\{+, -\}$). Sessions are mostly intended as run-time syntax. In fact, differently from other languages that provide primitives for explicit session naming and creation, here all sessions are transparent to programmers as they can be built automatically, resulting in a more elegant and disciplined style of writing processes. A fresh session name r and two polarised session ends $r^- \triangleright P$ and $r^+ \triangleright Q$ are generated (on client and service sides, resp.) upon each service invocation $\overline{s}.P$ of the service $s.Q$. We say $r^- \triangleright P$ is the dual session side of $r^+ \triangleright Q$ and vice versa. As P and Q share a session, their I/O communications are directed toward the dual session side. We let p, q range over polarities and let $\overline{p}$ denote the opposite polarity of p, where $\overline{+} = -$ and $\overline{-} = +$. The prefix $\langle \tilde{v} \rangle^{\uparrow}$ is used to output values to the enclosing parent session and the pipe $P > (?\tilde{x})Q$ is a construct that spawns a fresh instance $Q[\tilde{v}/\tilde{x}]$ of Q on any value $\tilde{v}$ produced by P. Note that we use a slight modification of the pipeline $P > Q$ introduced in [13], similar to the variation considered in [15] and closer to Orc's *sequencing* [16]: the variables $\tilde{x}$ to be bound after pipeline synchronisation are included in a special input $(?\tilde{x})$, called *pipeline input* preceding the right branch process.

We assume processes are designed according to some typical well-formedness criteria: (i) the containment relation between session identifiers is acyclic (i.e. we cannot have processes like $r^p \triangleright (P|r^p \triangleright Q)$); (ii) for each session identifier r, each r^+ and r^- occurs once in the process and never in the scope of a dynamic operator; (iii) in any summation $\Sigma_i \pi_i$, all prefixes π_i are of one and the same kind (all outputs or all inputs or all returns).

Semantics. The *reduction semantics* of CaSPiS exploits a rather standard *structural congruence* $\equiv$ on processes, defined as the least congruence satisfying the clauses in Fig. 2. The binders for the calculus are $(\tilde{x}).P$ for $\tilde{x}$ in P and $(\nu n)P$ for n in P, with standard notions of free names (fn) and bound names (bn) of

– $(\mathcal{P}/_\equiv, |, \mathbf{0})$ is a commutative monoid;
– $!P \equiv P|\,!P$;
– $(\nu n)\mathbf{0} \equiv \mathbf{0}$, $(\nu n)(\nu n')P \equiv (\nu n')(\nu n)P$, $(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin \mathsf{fn}(P)$;
– $r^p \rhd (\nu n)P \equiv (\nu n)(r^p \rhd P)$ if $r \neq n$;
– $((\nu n)P) > (?x)Q \equiv (\nu n)(P > (?x)Q)$ if $n \notin \mathsf{fn}(Q)$;

Fig. 2. Structural Congruence Laws

a process. Processes are considered equivalent up to the α-renaming of bound names. We present the operational semantics of the calculus by means of reduction contexts. The one-hole context $\mathbb{C}[\![\cdot]\!]$ is useful to insert a process P, the result being denoted $\mathbb{C}[\![P]\!]$ (process P replaces the hole inside the context), into an arbitrary nesting of operators. Before describing the reduction rules, we need to fix some terminology. Let us call *dynamic operators* any service definition $s.[\![\cdot]\!]$, service invocation $\overline{s}.[\![\cdot]\!]$, prefix $\pi[\![\cdot]\!]$, left-sided pipeline $P > (?x)[\![\cdot]\!]$ and replication $![\![\cdot]\!]$. The remaining operators are called *static*. The contexts we are interested in are called *static*, and characterised by the fact that the hole occurs in an actively running position and it is ready to interact (e.g. it is not under a prefix): formally, we say that a context is *static* if its holes do not occur in the scope of a dynamic operator. Moreover, we say that a context is *session-immune* if its hole does not occur under a session operator, and *pipeline-immune* if its hole does not occur under a right-sided pipeline operator. In the following we let $\mathbb{C}[\![\cdot]\!]$ range over static contexts, $\mathbb{S}[\![\cdot]\!]$ over static session-immune contexts, and $\mathbb{P}[\![\cdot]\!]$ over contexts that are static, session-immune and pipeline-immune. Roughly, a static session-immune context $\mathbb{S}[\![\cdot]\!]$ can "intercept" concretion prefixes but not abstraction and return prefixes, while a static, session-immune and pipeline-immune context $\mathbb{P}[\![\cdot]\!]$ cannot "intercept" any prefix. Analogous definitions apply to the case of two-holes contexts $\mathbb{C}[\![\cdot,\cdot]\!]$. Below we let $\mathbb{C}_r[\![\cdot,\cdot]\!]$ be a context of the form $\mathbb{C}[\![r^p \rhd \mathbb{P}[\![\cdot]\!], r^{\overline{p}} \rhd \mathbb{S}[\![\cdot]\!]]\!]$ (for some $\mathbb{P}[\![\cdot]\!]$ and $\mathbb{S}[\![\cdot]\!]$), which helps us to characterise the most general situation in which intra-session communication can happen, and we write $\mathbb{S}_{r^p}[\![\cdot]\!]$ for a context of the form $r^p \rhd \mathbb{S}[\![\cdot]\!]$

The reduction rules of CaSPiS are given in Fig. 3, where we assume that r is fresh in Sync and that $|\tilde{x}| = |\tilde{v}|$. It can be shown that reductions preserve all well-formedness criteria mentioned above, in the sense that well-formed processes always reduce to well-formed processes.

Naming conventions. To distinguish among different occurrences of the same service, we assume to annotate each of them with a different index, as in $s_{@k}$. As a consequence, we can uniquely identify the fresh name used in case of synchronisation on the same service, e.g., when the synchronisation happens on the service s, on the occurrences $s_{@k}$ and $\overline{s}_{@m}$, then the session name is $r^p_{s@m:k}$. When unambiguous, we simply use s, $\overline{s}$ and r^p_s.

To distinguish among different pipeline constructs, we annotate each pipeline operator with a different label $l \in \mathcal{L}$, as in $>_l$. Also, we identify the left branch with a label l_0 and the right branch with l_1. The same annotation l_1 enriches

$$
\begin{array}{ll}
\text{Sync} & \mathbb{C}[\![\, \overline{s}.P, s.Q \,]\!] \;\rightarrow\; (\nu r)\mathbb{C}[\![\, r^- \triangleright P, r^+ \triangleright Q \,]\!] \\
\text{S Sync} & \mathbb{C}_r[\![\, \langle\tilde{v}\rangle P + \sum_i \pi_i P_i, (?\tilde{x})Q + \sum_j \pi_j Q_j \,]\!] \;\rightarrow\; \mathbb{C}_r[\![\, P, Q[\tilde{v}/\tilde{x}] \,]\!] \\
\text{S Sync Ret} & \mathbb{C}_r[\![\, \mathbb{S}'_{r'^p}[\![\, \langle\tilde{v}\rangle^\uparrow P + \sum_i \pi_i P_i \,]\!], (?\tilde{x})Q + \sum_j \pi_j Q_j \,]\!] \;\rightarrow \\
 & \mathbb{C}_r[\![\, \mathbb{S}'_{r'^p}[\![\, P \,]\!], Q[\tilde{v}/\tilde{x}] \,]\!] \\
\text{P Sync} & \mathbb{C}[\![\, \mathbb{P}[\![\, \langle\tilde{v}\rangle P + \sum_i \pi_i P_i \,]\!] > (?\tilde{x})Q \,]\!] \;\rightarrow\; \mathbb{C}[\![\, Q[\tilde{v}/\tilde{x}] | (\mathbb{P}[\![\, P \,]\!] > (?\tilde{x})Q) \,]\!] \\
\text{P Sync Ret} & \mathbb{C}[\![\, \mathbb{P}[\![\, \mathbb{S}_{r^p}[\![\, \langle\tilde{v}\rangle P + \sum_i \pi_i P_i \,]\!] \,]\!] > (?\tilde{x})Q \,]\!] \;\rightarrow \\
 & \mathbb{C}[\![\, Q[\tilde{v}/\tilde{x}] | (\mathbb{P}[\![\, \mathbb{S}_{r^p}[\![\, P \,]\!] \,]\!] > (?\tilde{x})Q) \,]\!] \\
\text{Struct} & P \equiv P' \;\wedge\; P' \rightarrow Q' \;\wedge\; Q' \equiv Q \;\Rightarrow\; P \rightarrow Q
\end{array}
$$

Fig. 3. Reduction Semantics of CaSPiS

the variables $\tilde{x}$ affected by the pipeline input in the right branch of the pipeline, as in $P >_l (?\tilde{x}^{l_1})Q$. Note that these annotations do not affect the semantics.

Furthermore, to simplify the definition of our Control Flow Analysis in Section 3, we discipline the α-renaming of *bound* values and variables. To do it in a simple and "implicit" way, we partition all the names used by a process into finitely many equivalence classes and we use the names of the equivalence classes instead of the actual names. This partition works in such a way that names from the same equivalence class are assigned a common *canonical name* and consequently there are only finitely many canonical names in any execution of a given process. This is enforced by assigning the same canonical name to every name generated by the same restriction. The canonical name $\lfloor n \rfloor$ is for a name n; similarly $\lfloor x \rfloor$ is for a variable x. In this way, we statically maintain the identity of values and variables that might be lost by freely applying α-conversions. Hereafter, when unambiguous, we shall simply write n (resp. x) for $\lfloor n \rfloor$ (resp. $\lfloor x \rfloor$).

Examples. Consider the following simplified specification of a bank credit request service, where *req* is the service definition of the bank B.

$$
\begin{aligned}
B &\stackrel{def}{=} req_{@1}.(?y_{ba})\overline{val_{@3}}.\langle y_{ba}\rangle(?w_{ans})\langle w_{ans}\rangle^\uparrow \\
V &\stackrel{def}{=} val_{@4}.(?z_{ba})\langle Ans\rangle
\end{aligned}
$$

After the client invocation $\overline{req_{@1}}$, the bank waits for the client balance asset Ba (as input to $?y_{ba}$) that should be passed to the Validation service V, through the service invocation $\overline{val_{@3}}$. The validation answer Ans is sent to B (as input to $?w_{ans}$) and forwarded to the client. Thus a typical client C can be defined as

$$
C \stackrel{def}{=} \overline{req_{@2}}.\langle Ba\rangle(?x_{ans})\langle x_{ans}\rangle^\uparrow
$$

After the service invocation $\overline{req}$, the overall system $S \stackrel{def}{=} C|B|V$ becomes S',

$$
\begin{aligned}
S \rightarrow S' \stackrel{def}{=} (\nu r_{req@1:2})(\; & r^-_{req@1:2} \triangleright \langle Ba\rangle(?x_{ans})\langle x_{ans}\rangle^\uparrow \;| \\
& r^+_{req@1:2} \triangleright (?y_{ba})\overline{val_{@3}}.\langle y_{ba}\rangle(?w_{ans})\langle w_{ans}\rangle^\uparrow) \;|\; V
\end{aligned}
$$

where $r_{req@1:2}$ is the freshly generated session and where the client protocol is running on the left (the session side with negative polarity r^-) and the service protocol on the right (the session side with positive polarity r^+). The two protocols running on opposite sides of the same session can now exchange data, leading to S'':

$$\begin{aligned} S' \to S'' \stackrel{def}{=} (\nu r_{req@1:2})(\ & r^-_{req@1:2} \rhd (?x_{ans})\langle x_{ans}\rangle^\uparrow \mid \\ & r^+_{req@1:2} \rhd \overline{val_{@3}}.\langle Ba\rangle(?w_{ans})\langle w_{ans}\rangle^\uparrow) \mid V \end{aligned}$$

The computation continues as follows:

$$\begin{aligned} S'' \to\ & (\nu r_{req@1:2})(r^-_{req@1:2} \rhd (?x_{ans})\langle x_{ans}\rangle^\uparrow \mid \\ & (\nu r_{val@3:4})(r^+_{req@1:2} \rhd r^-_{val@3:4} \rhd \langle Ba\rangle(?w_{ans})\langle w_{ans}\rangle^\uparrow) \mid r^+_{val@3:4} \rhd (?z_{ba})\langle Ans\rangle)) \\ \to\ & (\nu r_{req@1:2})(r^-_{req@1:2} \rhd (?x_{ans})\langle x_{ans}\rangle^\uparrow \mid \\ & (\nu r_{val@3:4})(r^+_{req@1:2} \rhd r^-_{val@3:4} \rhd (?w_{ans})\langle w_{ans}\rangle^\uparrow) \mid r^+_{val@3:4} \rhd \langle Ans\rangle)) \\ \to\ & (\nu r_{req@1:2})(r^-_{req@1:2} \rhd (?x_{ans})\langle x_{ans}\rangle^\uparrow \mid \\ & (\nu r_{val@3:4})(r^+_{req@1:2} \rhd r^-_{val@3:4} \rhd \langle Ans\rangle^\uparrow) \mid r^+_{val@3:4} \rhd \mathbf{0})) \\ \to\ & (\nu r_{req@1:2})(r^-_{req@1:2} \rhd \langle Ans\rangle^\uparrow \mid (\nu r_{val@3:4})(r^+_{req@1:2} \rhd r^-_{val@3:4} \rhd \mathbf{0} \mid r^+_{val@3:4} \rhd \mathbf{0})) \end{aligned}$$

Since the bank and validation service must typically handle more requests, one can use their replicated versions $!B$ and $!V$.

Now suppose several bank services $B_1, ..., B_n$ are available (together with suitable verification services $V_1, ..., V_m$, possibly shared by different banks), and that a client wants to contact them all and be notified by email about their answers Ans_j (with $j \in [1, m]$) exploiting a suitable service *email*. Then the client could be written as

$$(C_1 \mid \cdots \mid C_n) >_l (?x^{l_1}_{any-ans})\overline{email_{@0}}.\langle x_{any-ans}\rangle$$

where each C_i is the request to a specific bank service, i.e. it has the form $\overline{req_i}.\langle Ba\rangle(?x_i)\langle x_i\rangle^\uparrow$.

3 The Control Flow Analysis

We develop a Control Flow Analysis for the `close`-free fragment of CaSPiS, borrowing some ideas from [20, 7]. Session primitives indeed introduce a nesting hierarchy that resembles the ones used in Ambients-like calculi. Our analysis uses the notion of enclosing scope, recording the current scope due to services, sessions or pipelines. We say that P is in the scope of $s_{@k}$ if $s_{@k}$ is the immediate enclosing service definition. Similarly for $\overline{s}_{@k}$, $r^p_{s@m:k}$, and for l_0 or l_1. The aim of the analysis is over-approximating all the possible behaviour of a CaSPiS process. In particular, our analysis keeps track of the possible contents of scopes, in terms of communication and service synchronizations. The result of analysing a process P is a pair $(\mathcal{I}, \mathcal{R})$, called *estimate* for P, that satisfies the judgements defined by the axioms and rules in the upper (lower, resp.) part of Table 1. The analysis is defined in the flavour of Flow Logic [21]. The first component $\mathcal{I}$ gives information on the contents of a scope. The second component $\mathcal{R}$ gives information about the set of values to which names can be bound. Moreover, let σ, σ' be scope identifiers, ranged over by $s_{@k}, \overline{s}_{@k}, r^p_{s@m:k}, l_0, l_1$.

To *validate* the correctness of a proposed estimate $(\mathcal{I}, \mathcal{R})$ we state a set of clauses operating upon judgements for analysing processes $\mathcal{I}, \mathcal{R} \models^{\sigma} P$. The judgement expresses that when P is enclosed within the scope identified by σ, then $(\mathcal{I}, \mathcal{R})$ correctly captures the behaviour of P, i.e. the estimate is valid also for all the states P', passed through a computation of P. More precisely:

- $\mathcal{I} : (\lfloor \mathcal{N}_{srv} \rfloor \cup \lfloor \mathcal{N}_{sess} \rfloor) \cup \mathcal{L} \to \wp(\lfloor \mathcal{N}_{srv} \rfloor \cup \lfloor \mathcal{N}_{sess} \rfloor \cup \mathcal{L}) \cup \lfloor \mathcal{P}ref \rfloor$, where $\lfloor S \rfloor$ is the set of canonical names in S, $\wp(S)$ stands for the power-set of the set S and $\lfloor \mathcal{P}ref \rfloor$ is the set of action and service prefixes, defined on canonical names. Here, $\sigma \in \mathcal{I}(\sigma')$ means that the scope identified by σ' may contain the one identified by σ; $\pi \in \mathcal{I}(\sigma')$ means that the action π may occur in the scope identified by σ'.
- $\mathcal{R} : \lfloor \mathcal{N} \rfloor \to \wp(\lfloor \mathcal{N} \rfloor)$ is the *abstract environment* that maps a variable to the set of names it can be bound to, i.e. if $v \in \mathcal{R}(n)$ then n may take the value v. We assume that for each free name n, we have that $n \in \mathcal{R}(n)$. Moreover, we write $\mathcal{R}(x_1, ..., x_n)$ as a shorthand for $\mathcal{R}(x_1), ..., \mathcal{R}(x_n)$. Without loss of generality, we suppose to have all the variables distinct.

Validation. Following [20], the analysis is specified in two phases. First, we check that $(\mathcal{I}, \mathcal{R})$ describes the initial process. This is done in the upper part of Table 1, where the clauses amount to a structural traversal of process syntax.

The clause for service definition checks that whenever a service $s_{@k}$ is defined in $s_{@k}.P$, then the relative hierarchy position w.r.t. the enclosing scope must be reflected in $\mathcal{I}$, i.e. $s_{@k} \in \mathcal{I}(\sigma)$. Furthermore, when inspecting the content P, the fact that the new enclosing scope is $s_{@k}$ is recorded, as reflected by the judgement $\mathcal{I}, \mathcal{R} \models^{s_{@k}} P$. Similarly for service invocation $\overline{x}_{@k}$: the only difference is that when x is a variable, the analysis checks for every actual value s that can be bound to x that $\overline{s}_{@k} \in \mathcal{I}(\sigma)$ and $\mathcal{I}, \mathcal{R} \models^{\overline{s}_{@k}} P$. The clauses for input, output and return check that the corresponding prefixes are included in $\mathcal{I}(\sigma)$ and that the analysis of the continuation processes hold as well. There is a special rule for pipeline input prefix, that allows us to distinguish it from the standard input one. Note that the current scope has the same identifier carried by the variables. Similarly, there is a rule for output prefixes occurring inside the scope of a left branch of a pipeline. The corresponding possible outputs are annotated with the label l_0. The rule for session, modelled as the ones for service, just checks that the the relative hierarchy position of the session identifier $r^p_{s@m:k}$ w.r.t. the enclosing scope must be reported in $\mathcal{I}$, i.e. $r^p_{s@m:k} \in \mathcal{I}(\sigma)$. It is used in analysing the possible continuations of the initial process.

All the clauses dealing with a compound process check that the analysis also holds for its immediate sub-processes. In particular, the analysis of $!P$ and that of $(\nu n)P$ are equal to the one of P. This is an obvious source of imprecision (in the sense of over-approximation). The clause for pipeline deserves a specific comment. It checks that whenever a pipeline $>_l$ is met, then the analysis of the left and the right branches is kept distinct by the introduction of two sub-indexes l_0 for the left one and l_1 for the right one. This allows us to predict possible communication over the two sides of the same pipeline. Furthermore, the analysis contents of the two scopes must be included in the enclosing scope identified by σ. This allows us to predict also the communications due to I/O synchronisations, involving prefixes occurring inside the scope of a pipeline.

In the second phase, we check that $(\mathcal{I}, \mathcal{R})$ also takes into account the dynamics of the process under analysis, i.e. the synchronizations due to communications, services and pipelines. This is expressed by the closure conditions in the lower part of Table 1 that mimic the semantics, by modelling, without exceeding the precision boundaries of the analysis, the semantic preconditions and the consequences of the possible actions. More precisely, preconditions check, in terms of $\mathcal{I}$, for the possible presence of the redexes necessary for actions to be performed. The conclusion imposes the additional requirements on $\mathcal{I}$ and $\mathcal{R}$, necessary to give a valid prediction of the analysed action. In the clause for *Service Synch*, we have to make sure that the precondition requirements hold, i.e.:

- there exists an occurrence of service definition: $s_{@k} \in \mathcal{I}(\sigma)$;
- there exists an occurrence of the corresponding invocation $\overline{s}_{@m} \in \mathcal{I}(\sigma')$;

If the precondition requirements are satisfied, then the conclusions of the clause express the consequences of performing the service synchronisation. In this case, we have that $\mathcal{I}$ must reflect that there may exist a session identified by $r^{+}_{s@m:k}$ inside σ and by $r^{-}_{s@m:k}$ inside σ' , such that the contents (scopes, prefixes) of $s_{@m:k}$ and of $\overline{s}_{@m}$ may also be inside $\mathcal{I}(r^{+}_{s@m:k})$ and $\mathcal{I}(r^{-}_{s@m:k})$, resp..

Similarly, in the clause for *I/O Synch*, if the following preconditions hold:

- there exists an occurrence of output in $\mathcal{I}(r^{p}_{s@m:k})$;
- there exists an occurrence of the corresponding input in the sibling session $\mathcal{I}(r^{\overline{p}}_{s@m:k})$.

then the values sent can be bound to the corresponding input variables: i.e., a possible communication is predicted here. Note that the rule correctly does not consider outputs in the form $\langle\tilde{v}\rangle^{l_0}$, because they possibly occur inside a left branch of a pipeline and therefore they are not available for I/O synchronisations.

The clause for *Ret Synch* is similar. The return prefix must be included in $r^{p}_{s@m:k}$, in turn included in $\mathcal{I}(r^{p'}_{s'@n:q})$, while the corresponding input must be included in $r^{\overline{p'}}_{s'@n:q}$, i.e. in the sibling session scope of the enclosing session.

In the clause for *Pipe I/O Synch*, the communication can be predicted, whenever the output and the pipeline input prefixes occur in the scope of the same session identifier (same side, too), and furthermore the output occurs in the left branch of a pipeline, while the pipeline input occurs in the right part of the same pipeline. Note that only pipeline input prefixes are considered here.

Similarly, in the clause for *Pipe Ret Synch*. The only difference is that the return prefix must occur in the session identified by $r^{p}_{s@m:k}$, included in the same scope that includes the corresponding pipeline input.

$\mathcal{I},\mathcal{R} \models^{\sigma} s_{@k}.P$	iff $s_{@k} \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{s_{@k}} P$
$\mathcal{I},\mathcal{R} \models^{\sigma} \overline{x}_{@k}.P$	iff $\forall s_{@m} \in \mathcal{R}(x):\; \overline{s}_{@k} \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{\overline{s}_{@k}} P$
$\mathcal{I},\mathcal{R} \models^{\sigma} (?\tilde{x}).P$	iff $(?\tilde{x}) \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{\sigma} P$
$\mathcal{I},\mathcal{R} \models^{l_1} (?\tilde{x}^{l_1}).P$	iff $(?\tilde{x}^{l}) \in \mathcal{I}(l_1) \;\wedge\; \mathcal{I},\mathcal{R} \models^{l_1} P$
$\mathcal{I},\mathcal{R} \models^{\sigma} \langle\tilde{x}\rangle.P$	iff $\forall \tilde{v} \in \mathcal{R}(\tilde{x})\; \langle\tilde{v}\rangle \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{\sigma} P$
$\mathcal{I},\mathcal{R} \models^{l_0} \langle\tilde{x}\rangle.P$	iff $\forall \tilde{v} \in \mathcal{R}(\tilde{x})\; \langle\tilde{v}\rangle^{l_0} \in \mathcal{I}(l_0) \;\wedge\; \mathcal{I},\mathcal{R} \models^{l_0} P$
$\mathcal{I},\mathcal{R} \models^{\sigma} \langle\tilde{x}\rangle^{\uparrow}.P$	iff $\forall \tilde{v} \in \mathcal{R}(x)\; \langle\tilde{v}\rangle^{\uparrow} \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{\sigma} P$
$\mathcal{I},\mathcal{R} \models^{\sigma} \Sigma_{i\in I}\pi_i P_i$	iff $\forall i \in I : \mathcal{I},\mathcal{R} \models^{\sigma} \pi_i P_i$
$\mathcal{I},\mathcal{R} \models^{\sigma} P\|Q$	iff $\mathcal{I},\mathcal{R} \models^{\sigma} P \;\wedge\; \mathcal{I},\mathcal{R} \models^{\sigma} Q$
$\mathcal{I},\mathcal{R} \models^{\sigma} !P$	iff $\mathcal{I},\mathcal{R} \models^{\sigma} P$
$\mathcal{I},\mathcal{R} \models^{\sigma} (\nu n)P$	iff $\mathcal{I},\mathcal{R} \models^{\sigma} P$
$\mathcal{I},\mathcal{R} \models^{\sigma} P >_{l} (?\tilde{x}^{l_1})Q$	iff $l_0, l_1 \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{l_0} P \;\wedge\; \mathcal{I},\mathcal{R} \models^{l_1} (?\tilde{x}^{l})Q \;\wedge$ $\mathcal{I}(l_0), \mathcal{I}(l_1) \subseteq \mathcal{I}(\sigma)$
$\mathcal{I},\mathcal{R} \models^{\sigma} r^{p}_{s@m:k} \triangleright P$	iff $r^{p}_{s@m:k} \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{r^{p}_{s@m:k}} P$

(*Service Synch*)	$s_{@m} \in \mathcal{I}(\sigma) \wedge \overline{s}_{@k} \in \mathcal{I}(\sigma')$	
	$\Rightarrow r^{+}_{s@m:k} \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I}(s_{@m}) \subseteq \mathcal{I}(r^{+}_{s@m:k}) \;\wedge$	
	$r^{-}_{s@m:k} \in \mathcal{I}(\sigma') \;\wedge\; \mathcal{I}(\overline{s}_{@k}) \subseteq \mathcal{I}(r^{-}_{s@m:k})$	
(*I/O Synch*)	$\langle\tilde{v}\rangle \in \mathcal{I}(r^{p}_{s@m:k}) \wedge (?\tilde{x}) \in \mathcal{I}(r^{\overline{p}}_{s@m:k})$	
	$\Rightarrow \tilde{v} \in \mathcal{R}(\tilde{x})$	
(*Ret Synch*)	$\langle\tilde{v}\rangle^{\uparrow} \in \mathcal{I}(r^{p}_{s@m:k}) \;\wedge\; r^{p}_{@m:k} \in \mathcal{I}(r^{p'}_{s'@n:q}) \wedge (?\tilde{x}) \in \mathcal{I}(r^{\overline{p'}}_{s'@n:q})$	
	$\Rightarrow \tilde{v} \in \mathcal{R}(\tilde{x})$	
(*Pipe I/O Synch*)	$\langle\tilde{v}\rangle^{l_0} \in \mathcal{I}(l_0) \;\wedge\; (?\tilde{x}^{l_1}) \in \mathcal{I}(l_1)$	
	$\Rightarrow \tilde{v} \in \mathcal{R}(\tilde{x})$	
(*Pipe Ret Synch*)	$\langle\tilde{v}\rangle^{\uparrow} \in \mathcal{I}(r^{p}_{s@m:k}) \;\wedge\; r^{p}_{s@m:k} \in \mathcal{I}(l_0)$	
	$\wedge\; (?\tilde{x}^{l_1}) \in \mathcal{I}(l_1)$	
	$\Rightarrow \tilde{v} \in \mathcal{R}(\tilde{x})$	

Table 1. Analysis for CaSPiS Processes

$$
\begin{array}{l}
\mathcal{I}(*) \ni req_{@1}, \overline{req_{@2}}, val_{@4} \\
\mathcal{I}(*) \ni r^{+}_{req@1:2}, r^{-}_{req@1:2}, r_{val@3:4} \\
\mathcal{I}(\overline{req_{@2}}), \mathcal{I}(r^{-}_{req@1:2}) \ni \langle Ba \rangle, (?x_{ans}), \langle x_{ans} \rangle^{\uparrow} \\
\mathcal{I}(req_{@1}), \mathcal{I}(r^{+}_{req@1:2}) \ni \overline{val_{@3}}, (?y_{ba}) \\
\mathcal{I}(\overline{val_{@3}}), \mathcal{I}(r^{-}_{val@3:4}) \ni \langle y_{ba} \rangle, (?w_{ba}), \langle w_{ba} \rangle^{\uparrow} \\
\mathcal{I}(val_{@4}), \mathcal{I}(r^{+}_{va@3:4}) \ni (?z_{ba}), \langle Ans \rangle \\
\mathcal{R}(y_{ba}) \ni Ba,\ \mathcal{R}(z_{ba}) \ni Ba \\
\mathcal{R}(w_{ans}) \ni Ans,\ \mathcal{R}(x_{ans}) \ni Ans
\end{array}
$$

Fig. 4. Some Entries of the Example Analysis

Example 1. Now, we can show how the analysis works on our running example:

$$
\begin{array}{l}
S \stackrel{def}{=} B|V|C \\
B \stackrel{def}{=} req_{@1}.(?y_{ba})\overline{val_{@3}}.\langle y_{ba} \rangle (?w_{ans}) \langle w_{ans} \rangle^{\uparrow} \\
V \stackrel{def}{=} val_{@4}.(?z_{ba}) \langle Ans \rangle \\
C \stackrel{def}{=} \overline{req_{@2}}.\langle Ba \rangle (?x_{ans}) \langle x_{ans} \rangle^{\uparrow}
\end{array}
$$

The main entries of the analysis are reported in Fig. 4, where $*$ identifies the ideal outermost scope in which the system top-level service scopes are. It is easy to check that $(\mathcal{I}, \mathcal{R})$ is a valid estimate, i.e., that $\mathcal{I}, \mathcal{R} \models^* S$, by following the two stages explained above: we just illustrate some steps. We have indeed that $\mathcal{I}, \mathcal{R} \models^* S$ holds if $\mathcal{I}, \mathcal{R} \models^* B$, $\mathcal{I}, \mathcal{R} \models^* V$ and $\mathcal{I}, \mathcal{R} \models^* C$. In particular, $\mathcal{I}, \mathcal{R} \models^* A$ holds because $req_{@1} \in \mathcal{I}(*)$ and $\mathcal{I}, \mathcal{R} \models^* C$ because $\overline{req_{@2}} \in \mathcal{I}(*)$. Now, by *(Service Synch)*, we have that $req_{@1}, \overline{req_{@2}} \in \mathcal{I}(*)$ implies that

$$
\begin{array}{l}
r^{+}_{req@1:2} \in \mathcal{I}(*) \wedge \mathcal{I}(req_{@1}) \subseteq \mathcal{I}(r^{+}_{req@1:2}) \wedge \\
r^{-}_{req@1:2} \in \mathcal{I}(*) \wedge \mathcal{I}(\overline{req_{@2}}) \subseteq \mathcal{I}(r^{-}_{req@1:2})
\end{array}
$$

Furthermore, by *(I/O Synch)*, we have that $Ba \in \mathcal{R}(y_{ba})$, because

$$
\langle Ba \rangle \in \mathcal{I}(r^{-}_{req@1:2}) \quad (?y_{ba}) \in \mathcal{I}(r^{+}_{req@1:2})
$$

Similarly, we can obtain that $\mathcal{R}(x_{any-ans})$ includes Ans_j, for each $j \in [1, m]$.

Semantic Correctness. Our analysis is correct w.r.t. the given semantics, i.e. a valid estimate enjoys the following subject reduction property.

Theorem 1. ***(Subject Reduction)***
If $P \to Q$ and $\mathcal{I}, \mathcal{R} \models^{\sigma} P$ then also $\mathcal{I}, \mathcal{R} \models^{\sigma} Q$.

This result depends on the fact that the analysis is invariant under the structural congruence, as stated below.

Lemma 1. ***(Invariance of Structural Congruence)*** *If $P \equiv Q$ and $\mathcal{I}, \mathcal{R} \models^{\sigma} P$ then also $\mathcal{I}, \mathcal{R} \models^{\sigma} Q$.*

The above results are handled and proved in the extended version of the calculus [9]. Currently, our analysis has not been implemented yet, but it could, e.g., along the lines of the one in [20].

Modelling the Malicious Customer. We need to model a new kind of attacker, different from the classical Dolev-Yao one [17], on which rely many systems for the verification of security protocols. A Dolev-Yao attacker acts on an open network, where principals exchange their messages. He/she can intercept and generate messages, provided that the necessary information is in his/her knowledge, which increases while interacting with the network. Our setting calls for a different model, because our attacker is an *accredited customer of a service* that has no control of the communication channels, apart from the ones established by the sessions in which he/she is involved. Nevertheless, our attacker, that we call *malicious customer* or **M**, does not necessarily follow the intended rules of the application protocol and can try to use the functions of the service in an unintended way, e.g., by sending messages in the right format, but with contents different from the expected ones. Under this regard, our attacker is less powerful of the Dolev-Yao one: **M** just tries to do everything that the application does not prevent him/her to do. More precisely, **M** has a knowledge made of all the public information and increased by the messages received from the service: the attacker can use this knowledge to produce messages to be sent to the server. We assume **M** is smart enough to send only messages in the expected format in each information exchange. We extend our framework in order to implicitly consider the possible behaviour of such an attacker or *malicious customer*. We statically approximate the malicious customer knowledge, by representing it by a new analysis component $\mathcal{K}$. Intuitively, the clauses acting on $\mathcal{K}$ implicitly take the attacker possible actions into account. The component $\mathcal{K}$ contains all the free names, all the messages that the customer can receive, and all the messages that can be computed from them, e.g. if v and v' belong to $\mathcal{K}$, then also the tuple (v, v') belongs to $\mathcal{K}$ and, vice versa, if (v, v') belongs to $\mathcal{K}$, then also v and v' belong to $\mathcal{K}$. Furthermore, all the messages in $\mathcal{K}$ can be sent by the customer. To distinguish the customer actions, we annotate the corresponding prefixes with M, as in π^M, and we use a and b both for M or the empty annotation ϵ. As a consequence, we need to slightly change the rules in Table 1, as shown in Table 2, where only the more significant rules are reported. Consider the rule (*I/O Synch*). Whenever the analysis predicts that a customer may send a certain message $\tilde{v}$, the analysis predicts that the same, possibly malicious, customer can also send every other message $\widetilde{v'}$, obtained by synthetising the information in $\mathcal{K}$, provided that it is in the same format of $\tilde{v}$. Whenever a customer may receive an input, then the analysis predicts that the same, possibly malicious, customer can also acquire the received message in $\mathcal{K}$.

Analysis at Work In the following, we refer to the version of CaSPiS that includes pattern matching into the input construct, as defined by the following syntax (for a similar treatment see also [8]).

$\pi, \pi' ::=$	*Prefixes*	$\mathbf{E} ::=$	*Terms*	$\mathbf{D} ::=$	*Definition Terms*
$(D_1, ..., D_k)$	input	n	names	E	terms
$\langle E_1, ..., E_k \rangle$	output	x	variables	$?x$	definition variables
$\langle E_1, ..., E_k \rangle^{\uparrow}$	return				

$\mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma (?\tilde{x})^M.P$	iff $(?\tilde{x})^M \in \mathcal{I}(\sigma) \ \wedge\ \mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma P$
$\mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma (?\tilde{x}^l)^M.P$	iff $(?\tilde{x}^l)^M \in \mathcal{I}(\sigma) \ \wedge\ \mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma P$
$\mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma \langle\tilde{x}\rangle^M.P$	iff $\forall \tilde{v} \in \mathcal{R}(\tilde{x})\ \langle\tilde{v}\rangle^M \in \mathcal{I}(\sigma) \ \wedge\ \mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma P$
$\mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma \langle\tilde{x}\rangle^{\uparrow M}.P$	iff $\forall \tilde{v} \in \mathcal{R}(\tilde{x})\ \langle\tilde{v}\rangle^{\uparrow M} \in \mathcal{I}(\sigma) \ \wedge\ \mathcal{I}, \mathcal{R}, \mathcal{K} \models^\sigma P$
$(I/O\ Synch)$	$\langle\tilde{v}\rangle^a \in \mathcal{I}(r^p_{s@m:k}) \wedge (?\tilde{x})^b \in \mathcal{I}(r^{\overline{p}}_{s@m:k})$ $\Rightarrow \tilde{v} \in \mathcal{R}(\tilde{x})$ $a = M \Rightarrow \forall \widetilde{v'} \in \mathcal{K} : \widetilde{v'} \in \mathcal{R}(\tilde{x})$ $b = M \Rightarrow \tilde{v} \in \mathcal{K}$
$(Pipe\ I/O\ Synch)$	$\langle\tilde{v}\rangle^{\uparrow a} \in \mathcal{I}(l_0) \ \wedge\ (?\tilde{x}^{l_1 b}) \in \mathcal{I}(l_1)$ $\Rightarrow \tilde{v} \in \mathcal{R}(\tilde{x})$ $a = M \Rightarrow \forall \widetilde{v'} \in \mathcal{K} : \widetilde{v'} \in \mathcal{R}(\tilde{x}^{l_1})$ $b = M \Rightarrow \tilde{v} \in \mathcal{K}$
$(Knowledge\ Rule\ 1)$	$v \in \mathcal{N}_{\mathsf{fn}} \Rightarrow v \in \mathcal{K}$
$(Knowledge\ Rule\ 2)$	$v_1, ..., v_k \in \mathcal{K} \Rightarrow \langle v_1, ..., v_k\rangle \in \mathcal{K}$
$(Knowledge\ Rule\ 3)$	$\langle v_1, ..., v_k\rangle \in \mathcal{K} \Rightarrow v_1, ..., v_k \in \mathcal{K}$

Table 2. Analysis for CaSPiS Processes in the Presence of a Malicious Customer

Our patterns are tuples of definition terms $(D_1, \cdots, D_k)$ that have to be matched against tuples of terms $(E_1, \cdots, E_k)$, upon input. Note that, at run-time, each E_i is a closed term. Intuitively, the matching succeeds when the closed terms, say D_i, elementwise match to the corresponding terms E_i, and its effect is to bind the remaining terms E_j to the remaining variables. Actually, definition terms in $(D_1, \ldots, D_k)$ can be partitioned into closed terms to be matched and definition variables to be bound. We make the partition above explicit, by using the auxiliary functions $\mathbf{Term}(D_1, \ldots, D_k)$ and $\mathbf{Var}(D_1, \ldots, D_k)$. They work on the position of definition terms within the tuples in such a way that if $i \in \mathbf{Term}(D_1, \ldots, D_k)$, then D_i is a closed term, while if $i \in \mathbf{Var}(D_1, \ldots, D_k)$, then D_i is a definition variable. The new semantic rules for message exchange take pattern matching into account, e.g., the *(S Sync)* rule becomes:

$$\begin{array}{l} \mathbb{C}_r[\![\, \langle E_1, ..., E_k\rangle P + \sum_i \pi_i P_i, (D_1, ..., D_k)Q + \sum_j \pi_j Q_j \,]\!] \ \rightarrow\ \mathbb{C}_r[\![\, P, Q[\tilde{E}/\tilde{D}] \,]\!] \\ \text{if} \quad \wedge_{i \in \mathbf{Term}(D_1, \ldots, D_k)}\ E_i = D_i \end{array}$$

Suppose to have the following process

$$(\nu r)(r^- \rhd \langle A, M_A\rangle.P) | (r^+ \rhd (A, ?y_B).Q)$$

In the input tuple (A, y_B), we have that $\mathbf{Term}(A, ?y_B) = 1$ and $\mathbf{Var}(A, ?y_B) = 2$. Here the synchronisation succeeds, because the matching on the first term A succeeds. As a consequence, the second term y_B is bound to M_A in the continuation process $(\nu r)(r^- \rhd P) | (r^+ \rhd Q[M_A/y_B])$.

Extending the presented analysis in order to capture this kind of input construct is quite standard (see e.g. [10, 8]). We recall the new CFA rules in Table 3, where we use $\mathcal{R}(D)$ as a shorthand for $\mathcal{R}(x)$ when $D = ?x$. For the sake of space, we only report some of them.

$$\mathcal{I},\mathcal{R} \models^{\sigma} (D_1,...,D_k).P \text{ iff } (D_1,...,D_k) \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{\sigma} P$$
$$\mathcal{I},\mathcal{R} \models^{\sigma} \langle E_1,...,E_k\rangle.P \text{ iff } \forall v_1,..v_k : v_i \in R(E_i)\; \langle v_1,..v_k\rangle \in \mathcal{I}(\sigma) \;\wedge\; \mathcal{I},\mathcal{R} \models^{\sigma} P$$

$$\begin{array}{ll}(I/O\ Synch) & \langle v_1,..v_k\rangle \in \mathcal{I}(r^{p}_{s@m:k}) \wedge (D_1,...,D_k) \in \mathcal{I}(r^{\overline{p}}_{s@m:k}) \\ & \wedge \bigwedge_{i\in \mathbf{Term}(D_1,...,D_k)}\; v_i = D_i \\ & \Rightarrow \wedge_{j\in \mathbf{Var}(D_1,...,D_k)}\; v_j \in \mathcal{R}(D_j)\end{array}$$

Table 3. Analysis for CaSPiS Processes

We assume that, in each information exchange, a malicious customer sends messages in the expected format and successful w.r.t. the required pattern matching, as can stated by the rule $(I/O\ Synch)$ in the presence of the Malicious Customer, modified to take the pattern matching into account.

$$\begin{array}{ll}(I/O\ Synch) & \langle v_1,..v_k\rangle^a \in \mathcal{I}(r^{p}_{s@m:k}) \wedge (D_1,...,D_k)^b \in \mathcal{I}(r^{\overline{p}}_{s@m:k}) \\ & \wedge \bigwedge_{i\in \mathbf{Term}(D_1,...,D_k)}\; v_i = D_i \\ & \Rightarrow \wedge_{j\in \mathbf{Var}(D_1,...,D_k)}\; v_j \in \mathcal{R}(D_j) \\ & ... \\ & a = M \Rightarrow \forall \langle v'_1,..v'_k\rangle^a \in \mathcal{K} : \\ & \wedge \bigwedge_{i\in \mathbf{Term}(D_1,...,D_k)}\; v'_i = D_i \\ & \Rightarrow \wedge_{j\in \mathbf{Var}(D_1,...,D_k)}\; v'_j \in \mathcal{R}(D_j)\end{array}$$

Price Modification Example. We are now ready to model and analyse the example informally introduced in Section 1. The global system is composed by two processes put in parallel, the e-shop service S and the customer C. Also a data base DBI storing item prices is modeled:

$$(S \mid DBI) \mid C$$

Essentially, the e-shop S allows costumers to choose among several items $item_i$; when the costumer returns its selection, S asks the DBI service for the price of the selected item. In the first specification, shown below, the service S does not check if the form sent by the costumer contains the right price, i.e. the one computed by the DBI service, because the DBI sends the price directly to the client, without sending it also to the bank. For the sake of readability, we distinguish the part of the output tuples relative to the order form (payment form, resp.) by writing: $order\_form(...)$ ($payment\_form(...)$, resp.).

$$
\begin{aligned}
S \;\; &= !selling.\ \textstyle\sum_i((item_i)(\nu\ code)\\
&\quad (\overline{price\_db}\ \langle item_i\rangle\ (?x_{price_i})\langle order\_form(code, item_i, x_{price_i})\rangle^{\uparrow}\\
&\quad |\\
&\quad\ (ok, payment\_form(code, item_i, ?y_{price_i}, ?y_{name}, ?y_{cc})).PAY +\\
&\quad\ (no\_payment)))\\
DBI &= !price\_db\ \textstyle\sum_i((item_i)\langle price_i\rangle)\\
C \;\; &= \overline{selling}.\ \langle item_i\rangle^M (order\_form(?z\_code, item_i, ?z_{price_i}))^M\\
&\qquad\qquad \langle ok, payment\_form(z\_code, item_i, z_{price_i}, name, cc)\rangle^M +\\
&\qquad\qquad \langle no\_payment\rangle^M
\end{aligned}
$$

In the second formulation, the service S' matches the price sent by the customer against the one provided by the DBI. The other processes are unmodified.

$$
\begin{aligned}
S' = !selling.\ \textstyle\sum_i((item_i)(\nu\ code)\overline{price\_db}.\ &\langle item_i\rangle\ (?x_{price_i})\langle x_{price_i}\rangle^{\uparrow}\ >_l\\
&(?y^{l_1}_{price_i})\ \langle form(code, item_i, y_{price_i})\rangle\\
&(ok, payment\_form(code, item_i, y_{price_i}, ?y_{name}, ?y_{cc}))) +\\
&(no\_payment)
\end{aligned}
$$

The main entries of the analysis of the first formulation are reported in Fig 5. The variable $?y_{price}$, used by S, may be bound to any value the costumer sends, in particular to any possible faked price value. This depends on the fact that there is no pattern matching on the values received; more generally, no control on this part of input is made. Furthermore, note that since $\langle ok, payment\_form(code, item_i, price_i, name, cc)\rangle^M$ belongs to $\mathcal{I}(\overline{selling})$ and $faked\_price \in \mathcal{K}$, then $\langle ok, payment\_form(code, item_i, faked\_price, name, cc)\rangle$ can synchronise with the input $(ok, payment\_form(code, item_i, price_i, name, cc))$.

In the second formulation, the problem does not arise, because: (i) the DBI sends the price to the bank that, in turn, forwards it to the client; (ii) the shop service checks if the price returned by the costumer matches against the one returned by the DBI component, thus avoiding the attack. The first fix has to do with the overall design of the service. The second has to do with input validation and, technically, is obtained by using pattern matching on the third value received in the payment form, that should match with the price $y^{l_1}_{price}$, as correctly predicted by the analysis of the second formulation, shown in Fig 6.

4 Conclusion

Often, component-based applications, i.e. those that are mostly implemented by connected existing applications, as web services, only limit the analysis of their component applications to the functionalities they actually use. A good practice could be to consider all the functionalities an application offers, before including it in the composition, in order to check all the possible inputs the overall web

$$
\begin{array}{ll}
\mathcal{I}(\underline{selling}), \mathcal{I}(r^+_{sell}) & \ni item_i, \overline{price\_db} \\
\mathcal{I}(\overline{price\_db}), I(r^-_{price}) & \ni \langle item_i \rangle, (?x_{price_i}), \langle order\_form(code, item_i, price_i)\rangle^{\uparrow}, \\
& \quad (ok, payment\_form(code, item_i, price_i, name, cc)), \\
& \quad (ok, payment\_form(code, item_i, faked\_price, name, cc)), \\
& \quad (no\_payment) \\
\mathcal{I}(\overline{selling}), \mathcal{I}(r^-_{sell}) & \ni \langle item \rangle^M, (order\_form(code, item_i, price_i))^M, \\
& \quad \langle ok, payment\_form(code, item_i, price_i, name, cc)\rangle^M \\
& \quad \langle no\_payment \rangle^M \\
\mathcal{I}(price\_db), I(r^+_{price}) & \ni (item_i), \langle price_i \rangle \\
\mathcal{R}(x_{price_i}), \mathcal{R}(z_{price_i}) & \ni price_i \\
\mathcal{R}(y_{price_i}) & \ni price_i, faked\_price \\
\mathcal{K} & \ni price_i, faked\_price, payment\_form(code, item_i, price_i, name, cc), \\
& \quad payment\_form(code, item_i, faked\_price, name, cc)
\end{array}
$$

Fig. 5. Some Analysis Entries of $(S \mid DBI) \mid C$

$$
\begin{array}{ll}
\mathcal{I}(selling), \mathcal{I}(r^+_{sell}) & \ni (item_i), \underline{l_0} \\
\mathcal{I}(\underline{l_0}) & \ni \langle item_i \rangle, \overline{price\_db} \\
\mathcal{I}(\overline{price\_db}), I(r^-_{price}) & \ni \langle item_i \rangle (?x_{price}) \langle price \rangle^{\uparrow}, \\
\mathcal{I}(l_1) & \ni (?y^{l_1}_{price}), \langle form(code, item, price_i)\rangle, \\
& \quad (ok, payment\_form(code, item_i, price_i, ?y_{name}, ?y_{cc})), \\
& \quad (no\_payment) \\
\mathcal{R}(x_{price_i}) & \ni price_i \\
\mathcal{R}(y^{l_1}_{price_i}) & \ni price_i
\end{array}
$$

Fig. 6. Some Analysis Entries of $(S' \mid DBI) \mid C$

service can eventually provide and to keep trace where data-validation is applied or is missing.

We applied a Control Flow Analysis to an example inspired by a known logic-flawed application for e-commerce, specified in service oriented calculus as CaSPiS. Ours is a proof-of-concept work: we chose to detect the application misuse, by analysing the static approximation of the behaviour of the system. Our analysis can be easily specialised to capture specific properties of interest, e.g., data integrity in this example. We do not describe the specialisation process here, but only remark that the core of the analysis here introduced is preserved (for a similar process, see, e.g., the analyses for the LySa calculus in [10, 8]). Our proposal shows that the chosen level of abstraction of services and of their features is suitable to investigate logic flaws. In fact, a calculus like CaSPiS abstractly expresses the key aspects of service oriented computing as primitives, without requiring any further codification, thus allowing us to focus on the security flaws that arise at the level of services, and to ignore those arising at the level of the underlying protocols. Indeed different forms of basic interactions are distinguished and regulated on their own (services are globally available, while ordinary I/O communications are context sensitive); and sessions are an implicit mechanism for enclosing the communications between a caller and its callee, avoiding external interferences.

References

1. L Acciai, M. Boreale. Type Abstractions of Name-Passing Processes. In *Proc. of International Symposium on Fundamentals of Software Engineering (FSEN'07)* , LNCS, vol 4767, pp. 302–317. Springer, 2007.
2. L Acciai, M. Boreale. A Type System for Client Progress in a Service-Oriented Calculus. In *Concurrency, Graphs and Models* , LNCS, vol 5065, pp. 642–658. Springer, 2008.
3. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.C. Hem, O. Kouchnarenko, J. Mantovani, S. Mdersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proc. of CAV'05*, LNCS, vol 3576, pp. 281–285. Springer, 2005.
4. M. Bartoletti, P. Degano, G.L. Ferrari, R. Zunino. Semantics-Based Design for Secure Web Services. *Software Engineering, IEEE Transactions*, Vol 34(1): 33–49, 2008.
5. K. Bhargavan, C. Fournet, A.D. Gordon Verified Reference Implementations of WS-Security Protocols In *Proc. of Web Services and Formal Methods (WS-FM 06)*, LNCS, vol 4184 pp. 88–10 Springer, 2006.
6. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. *Computer Security Foundations Workshop (CSFW)*, 2001.
7. C. Bodei, A. Bracciali and D. Chiarugi. Control Flow Analysis for Brane Calculi. ENTCS, 227:59–75. Elsevier, 2009.
8. C. Bodei, L. Brodo, P. Degano, H. Gao. Detecting and Preventing Type Flaws at Static Time. To appear in *Journal of Computer Security*, 2009.
9. C. Bodei, L. Brodo, R. Bruni. Static Detection of Logic Flaws in Service Applications. Technical Report, Dipartimento di Informatica, Università di Pisa, 2009.
10. C. Bodei, M. Buchholtz, P. Degano, F. Nielson and H.R. Nielson. Static Validation of Security Protocols. *Journal of Computer Security*, 13(3):347–390, 2005.
11. M. Bond, J Clulow. Extending Security Protocol Analysis: New Challenges. ENTCS, 125(1):13–24. Elsevier, 2005.
12. E. Bonelli, A. Compagnoni, E. Gunter. Typechecking Safe Process Synchronization. In *Proc. Foundations of Global Ubiquitous Computing* ENTCS, 138(1):3–22. Elsevier, 2005.
13. M. Boreale, R. Bruni, R. De Nicola, M. Loreti. Sessions and Pipelines for Structured Service Programming. In *Proc. of Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*, LNCS, vol 5051, pp. 19–38, Springer, 2008.
14. R. Bruni. Calculi for service-oriented computing. In *Proc. of 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM'09)*, LNCS, vol 5569, pp. 1–41, Springer, 2009.
15. R. Bruni, L.G. Mezzina. Types and Deadlock Freedom in a Calculus of Services, Sessions and Pipelines. In *Proc. of Algebraic Methodology and Software Technology (AMAST'08)*, LNCS, vol 5140, pp. 100–115, Springer, 2008.
16. D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *Proc. of CONCUR'06*, LNCS, vol 4137, pp. 477–491. Springer, 2006.
17. D. Dolev and A.C. Yao. On the Security of Public Key Protocols. IEEE TIT, IT-29(12):198–208, 1983.
18. M. Kolundzija. Security Types for Sessions and Pipelines. In *Proc. of the 5th International Workshop on Web Services and Formal Methods (WS-FM'08)*, LNCS, vol 5387, pp. 175–189, Springer, 2009.
19. F. Nabi. Secure business application logic for e-commerce systems. Computers & Security 24(3):208–217, Springer, 2005.
20. F. Nielson, H. Riis Nielson, C. Priami, and D. Schuch da Rosa. Control Flow Analysis for BioAmbients. ENTCS, 180(3):65–79. Elsevier, 2007.
21. H. Riis Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. The Essence of Computation: Complexity, Analysis, Transformation, LNCS, vol 2566, pp. 223–244, Springer, 2002.
22. OASIS Technical Commitee. Web Services Security (WS-Security), 2006.
23. Neohapsis Archives. *Price modification possible in CyberOffice Shopping Cart* **http://archives.neohapsis.com/archives/bugtraq/2000-10/0011.html**
24. M. Backes, S. Mödersheim, B. Pfitzmann, L. Viganò. Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario. In *Proc. of Foundations of Software Science and Computation Structures (FOSSACS 2006)* LNCS, vol 3921, pp. 428–445, Springer.
25. T.Y.C. Woo, S.S. Lam. A semantic model for authentication protocols. In *Proc. of IEEE Symposium on Security and Privacy*, 1993.