Static Analysis Techniques for Session-Oriented Calculi*

Lucia Acciai[1], Chiara Bodei[2], Michele Boreale[1], Roberto Bruni[2], and Hugo T. Vieira[3]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze, Italy
{lacciai,boreale}@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Pisa, Italy
{chiara,bruni}@di.unipi.it
[3] CITI/Departamento de Informática, FCT Universidade Nova de Lisboa, Portugal
htv@fct.unl.pt

Abstract. In the SENSORIA project, core calculi have been adopted as a linguistic means to model and analyze service-oriented applications. The present chapter reports about the static analysis techniques developed for the SENSORIA session-oriented core calculi CaSPiS and CC. In particular, it presents a type system for client progress and control flow analysis in CaSPiS and type systems for conversation fidelity and progress in CC. The chapter gives an overview of the these techniques, summarizes the main results and presents the analysis of a common example taken from the SENSORIA financial case-study: the credit request scenario.

1 Introduction

In Chapter 2-1 the core calculi for service specification and analysis developed within SENSORIA have been introduced. These calculi are classified according to the approach adopted to maintain the link between the caller and the callee. We focus here on *session-oriented calculi*, where a private channel is implicitly instantiated upon service invocation between the caller and the callee. Specifically, we report on the static analysis techniques developed within the project for CaSPiS and CC. Recall that in CaSPiS sessions are binary, whereas in CC sessions, also called *conversations*, may dynamically involve multiple parties.

As far as CaSPiS is concerned, we provide contributions towards developing techniques for safe client-service interaction and preventing misuses at the so-called application logic level.

We first introduce a type system providing guarantees of *client progress* [1]. This system ensures that, in a well-typed CaSPiS process and in absence of divergence, any client invoking a service is guaranteed not to deadlock in a conversation with a service. The type system builds upon behavioral types techniques [11], as the behavior of a CaSPiS process is abstracted by means of a simpler ccs-like term. A key point, though, is that types account only for flows of I/O value-types, ignoring the rest. In particular, it is not necessary to equip the language of types with constructs to describe sessions. Indeed, considering the tree describing the nesting of sessions, a service invocation can

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

produce effects only at the parent level. It is then sufficient to associate to each process a two-level type taking into account just the current-level interactions and the upper-level effects. In order to guarantee progress of the invoker, the system relies on a notion of *compliance* between the client and the service protocols, which is essential to avoid deadlocks.

The second contribution is a Control Flow Analysis for CaSPiS [3] that can detect and prevent certain misuses at the application logic level. The presence of bugs at this level, called *application* or *business logic flaws*, may often lead to undesired behavior or to security attacks. This Control Flow Analysis system statically approximates the behavior of CaSPiS processes, in terms of the possible service and communication synchronizations. More precisely, what the analysis predicts includes 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, the analysis can focus on each client-server conversation separately and second, it can focus on the application logic.

We also propose analysis techniques for systems specified in CC, addressing *conversation fidelity* and *progress* [8]. Conversation fidelity captures the fact that all participants in a multiparty conversation follow the protocols of interaction, while progress – differently from client progress discussed above – guarantees absence of deadlocks in the whole system. We introduce two separate but complementary techniques: to discipline multiparty conversations we introduce conversation types, a novel and flexible type structure, able to uniformly describe both the internal and the interface behavior of systems, referred respectively as choreographies and contracts in web-services terminology. To guarantee deadlock freedom we introduce a progress proof system that relies on a notion of ordering of events and, crucially, propagation of orderings in communications.

Structure of the chapter. The chapter is organized in three main sections. Section 2 introduces a type system guaranteeing client progress in CaSPiS; it discusses the main results and proves that client progress is guaranteed in the considered scenario. A simple variation of the scenario is also considered in order to show how the system rules out processes not guaranteeing the client progress property. Section 3 introduces a Control Flow Analysis preventing business logic flaws in CaSPiS, proves that the proposed analysis technique enjoys the subject reduction property and, in order to show how logic flaws are detected, applies this technique to (a variation of) the running example. Section 4 introduces the type system for conversation fidelity and the progress proof system in CC, together with their main properties. Both proposals are then applied to the running example in order to prove that it enjoys conversation fidelity and progress. Finally, Section 5 concludes the chapter.

2 A Type System for Client Progress in CaSPiS

In this section we introduce a type system providing guarantees of client progress. There are three key aspects involved in its design. A first aspect concerns abstraction: types

focus on flows of I/O value-types and ignore the rest (actual values, service calls, ...). Specifically, akin to [11], types take the form of ccs-like terms describing I/O flows of processes. In fact, a tiny fragment of ccs, with no synchronization and restriction, is employed, where the role of atomic actions is played by basic types. A second aspect concerns compliance of client protocols with service protocols, which is essential to avoid deadlocks. In the type system, the operational abstractions provided by types are employed to effectively check client-service compliance. To this purpose, types are required to account for process I/O behavior quite precisely. Indeed, approximation might easily result into ignoring potential client-service deadlocks. A final aspect concerns the nesting of sessions. A session at a lower level can exercise effects on the upper level, say the level of any enclosing session. To describe this phenomenon, the system keeps track of the behavior both at the current level and at the level of a (fictitious) enclosing session, along the lines of those in [7,12,13]. This results in type judgments of the form $P : [\mathsf{S}]\mathsf{T}$, where S is the current-level type and T is the upper-level effect of P. Note that the distinction between types and effects we make here is somehow reminiscent of the type-and-effects systems of [15], with the difference that our effects are very simple (sequences of outputs) and are exercised on an upper level of activity rather than on a shared memory.

2.1 Language Fragment

We consider here a sub-calculus of the close-free fragment of CaSPiS [6], that we call CaSPiS$^-$, where return prefixes have always an empty continuation and service protocols do not return any value. From the technical point of view, both limitations are necessary in order to guarantee a two-way operational correspondence between processes and the corresponding types (see [1] for the details). From the practical point of view, the latter limitation means that, once a session is started, for the service there will be no "feedback" of sort as to what is going on inside the session. This is somehow consistent with the idea that services should be stateless entities. Hence, terms of the form $r \triangleright Q$, where Q is a service protocol, cannot produce any visible effect and they might be executed anywhere in the system, not necessarily on the service side. Indeed, under these restrictions, it turns out to be technically convenient to slightly modify the operational semantics so that Q, the service protocol, and P, the client protocol, are both executed at the side of the invoking client. The resulting session will be denoted by $[P \| Q]$. Note that session names, which in the full language are used to locate the two session endpoints, become redundant, as the endpoints now share the same location. Hence session names are discarded right away. To sum up, the set $\mathcal{P}$ of CaSPiS$^-$ processes is generated by the following grammar (where F and V are respectively the patterns and values defined in [6] and u can be either a name or a variable)

$$\pi ::= (F) \mid \langle V \rangle \qquad P ::= \textstyle\sum_{i \in I} \pi_i.P_i \mid \langle V \rangle^{\uparrow} \mid s.P \mid \overline{u}.P \mid [P \| Q] \mid P > Q \mid P | Q \mid (\nu s)P \mid !P \ .$$

Both the structural congruence and the labeled transition relation defined in [6] can be modified as expected to accommodate these changes. In particular, some operational rules must be replaced by the homonymous rules shown in Fig. 1. Notice that upon a synchronization of a service call with the corresponding service definition, the service

$$\text{(Def)}\ \frac{}{s.P \xrightarrow{s\langle P\rangle} \mathbf{0}} \qquad \text{(Call)}\ \frac{}{\overline{s}.P \xrightarrow{\overline{s}(Q)} [P \| Q]}$$

$$\text{(S-Ret)}\ \frac{P \xrightarrow{(\nu\hat{v})\langle v\rangle^{\uparrow}} P'}{[P\|Q] \xrightarrow{(\nu\hat{v})\langle v\rangle} [P'\|Q]} \qquad \text{(S-Sync)}\ \frac{P \xrightarrow{(\nu\tilde{n})s\langle R\rangle} P' \quad Q \xrightarrow{\overline{s}(R)} Q'}{[P\|Q] \xrightarrow{\tau} (\nu\tilde{n})[P'\|Q']}$$

Fig. 1. Labeled semantics

protocol R is sent to the invoker and executed on its side, (S-Sync). Returns are enabled only on the client side of sessions, (S-Ret).

2.2 Proving the Client Progress Property

The client progress property will be defined in terms of an error predicate. Informally, an error occurs when the client protocol of an active session tries to send a value to (or receive from) the service side, but the session as a whole is blocked. This is formalized by the predicate $\rightarrow_{\text{ERR}}$ defined below. In the definition, we rely on the standard notion of *contexts*, $C[\cdot], C'[\cdot], \ldots$. We say a context is *static* if its hole is not under the scope of a dynamic operator (input and output prefixes, replication, service definitions and invocations and the right-hand side of a pipeline). In essence, active subterms in a process P are those surrounded by a static context.

Definition 1 (error). *$P \rightarrow_{\text{ERR}}$ if and only if whenever $P \equiv C[[Q\|R]]$, with $C[\cdot]$ static, and $Q \xrightarrow{\eta}$, with $\eta ::= (v) \mid (\nu\hat{v})\langle v\rangle$, then $[Q\|R] \not\xrightarrow{\eta'}$, with $\eta' ::= \tau \mid \overline{s}(P')$.*

Note that "pending" returns are not taken into account in this definition. Indeed, a return is seen as an output at the upper level, (S-Ret), and the error, if any, is detected in the parent session, if it exists.

A process guarantees client progress if it is error-free at run-time.

Definition 2 (client progress). *Let $P \in \mathcal{P}$. We say P* guarantees client progress *if and only if whenever $P \rightarrow^* P'$ then $P' \not\rightarrow_{\text{ERR}}$.*

The above definition of error may seem too liberal, as absence of error does not actually guarantee progress of the session if $[Q\|R] \xrightarrow{\overline{s}(P')}$ and service s is not available. In fact, we are interested in processes where such situations do not arise: we call these processes *available*.

Definition 3 (available process). *We let* available *be the largest predicate on processes satisfying the following conditions. If P is* available *then* (i) *whenever $P \equiv (\nu\hat{s})C[\overline{s}.P']$, for some static $C[\cdot]$, and $C[\mathbf{0}] \rightarrow^* Q$ then $Q \rightarrow^* \xrightarrow{(\nu\tilde{n})s\langle R\rangle}$ for some $\tilde{n}$ and R; and* (ii) *whenever $P \rightarrow Q'$ then Q' is available.*

Here, clause (i) guarantees that the system (without interacting with service invocation $\overline{s}.Q$) can always reduce into a state where service s is ready to be invoked and clause (ii) guarantees that availability is preserved by reductions.

$$\text{(T-Out)}\ \frac{\Gamma \vdash P : [\mathsf{S}]\mathsf{T} \quad \Gamma \vdash u : \mathsf{b}}{\Gamma \vdash \langle u \rangle.P : [!\mathsf{b}.\mathsf{S}]\mathsf{T}} \qquad \text{(T-Call)}\ \frac{\Gamma \vdash u : \mathsf{V} \quad \Gamma \vdash P : [\mathsf{S}]\mathsf{T} \quad \mathsf{S} \propto \mathsf{V}}{\Gamma \vdash \overline{u}.P : [\mathsf{T}]0}$$

$$\text{(T-Ret)}\ \frac{\Gamma \vdash u : \mathsf{b}}{\Gamma \vdash \langle u \rangle^{\uparrow} : [0]!\mathsf{b}} \qquad \text{(T-Pipe)}\ \frac{\Gamma \vdash P : [\mathsf{S}]\mathsf{T} \quad \Gamma \vdash Q : [\sum_{i\in I} ?\mathsf{b}_i.\mathsf{U}_i]\mathsf{V} \quad \text{out}(\mathsf{S}) \subseteq \bigcup_{i\in I}\{\mathsf{b}_i\} \quad \text{NoSum}(\mathsf{S})}{\Gamma \vdash P > Q : [\mathsf{S} \bowtie \sum_{i\in I} ?\mathsf{b}_i.\mathsf{U}_i](\mathsf{T}|\mathsf{S} \mathbin{@} \mathsf{V})}$$

Fig. 2. Rules of the type system

Types. Types are essentially a fragment of ccs corresponding to BPP processes [9]. We presuppose a set $\mathcal{B}t$ of *base types*, $\mathsf{b}, \mathsf{b}', \ldots$ which include name *sorts* $\mathcal{S}, \mathcal{S}', \ldots$. Moreover, we presuppose a generic base-typing relation, mapping base values and service names to base types, written $v : \mathsf{b}$, with the obvious proviso that service names are mapped to sorts and base values are mapped to the remaining base types. The set $\mathcal{T}$ of types is defined by the grammar below.

$$\alpha ::= !\mathsf{b} \mid ?\mathsf{b} \mid \tau \qquad \mathsf{T}, \mathsf{S}, \mathsf{U}, \mathsf{V} ::= \sum_{i\in I} \alpha_i.\mathsf{T}_i \mid \mathsf{T} \mid \mathsf{T} \mid !\mathsf{T}$$

Notice that, like in [7,12], we need not nested session types in our system, because in order to check session safety it is sufficient to check local, in-session communications. In what follows we abbreviate with 0 the empty summation type.

The operational semantics of types can be found in [1]. It is worth to recall that input and output prefixes, $?\mathsf{b}$ and $!\mathsf{b}$, cannot synchronize with each other – we only have interleaving in this fragment of ccs.

The basic requirement for ensuring client progress is *type compliance* between client and service protocols involved in sessions. In the following, we indicate with $\overline{\lambda}$ the *coaction* of λ: $\overline{?\mathsf{b}} = !\mathsf{b}$ and $\overline{!\mathsf{b}} = ?\mathsf{b}$. This notation is extended to sets of actions as expected. Moreover, we indicate with $I(\mathsf{S})$ the set of initial actions S can perform: $I(\mathsf{S}) = \{\lambda \neq \tau \mid \exists \mathsf{S}' : \mathsf{S} \xrightarrow{\lambda} \mathsf{S}'\}$. Type compliance is defined co-inductively and guarantees that, given two compliant types S (the client's protocol) and T (the service's protocol), at any stage of a computation either S is stuck or there is at least one (weak) action from S matched by a (weak) coaction from T.

Definition 4 (type compliance). *Let be $\mathsf{S}, \mathsf{T} \in \mathcal{T}$.* Type compliance *is the largest relation on types such that whenever S is compliant with T, written $\mathsf{S} \propto \mathsf{T}$, it holds that*

- *either $I(\mathsf{S}) = \emptyset$ and $\mathsf{S} \not\xrightarrow{\tau}$*
- *or (a) either $\mathsf{S} \xrightarrow{\tau}$, or $\mathsf{T} \xrightarrow{\tau}$, or $I(\mathsf{S}) \cap \overline{I(\mathsf{T})} \neq \emptyset$; and (b) the following holds true:*
  1. *for each S' such that $\mathsf{S} \xrightarrow{\tau} \mathsf{S}'$ it holds that $\mathsf{S}' \propto \mathsf{T}$;*
  2. *for each T' such that $\mathsf{T} \xrightarrow{\tau} \mathsf{T}'$ it holds that $\mathsf{S} \propto \mathsf{T}'$;*
  3. *for each S' and T' such that $\mathsf{S} \xrightarrow{\lambda} \mathsf{S}'$ and $\mathsf{T} \xrightarrow{\overline{\lambda}} \mathsf{T}'$ it holds that $\mathsf{S}' \propto \mathsf{T}'$.*

Type system and results. The type system is along the lines of those in [7,12]; the most interesting rules are reported in Fig. 2 (the missing ones can be found in [1]). We presuppose a mapping ob from sorts $\{\mathcal{S}, \mathcal{S}', \ldots\}$ to types $\mathcal{T}$, with the intended meaning that if $ob(\mathcal{S}) = \mathsf{T}$ then names of sort $\mathcal{S}$ represent services whose abstract protocol is T. We take $s : \mathsf{T}$ as an abbreviation of $s : \mathcal{S}$ and $ob(\mathcal{S}) = \mathsf{T}$ for some $\mathcal{S}$. A *context* Γ is

a finite partial mapping from types to variables. For u a service name, a base value or a variable, we take $\Gamma \vdash u : \mathsf{b}$ (resp. $\Gamma \vdash u : \mathsf{T}$) to mean either that $u = v : \mathsf{b}$ (resp. $u = s : \mathsf{T}$) or $u = x \in \mathrm{dom}(\Gamma)$ and $\Gamma(x) = \mathsf{b}$ (resp. $\Gamma(x) = \mathsf{T}$). In the process syntax, we attach type annotations to input variables as expected. Type judgments are of the form $\Gamma \vdash P : [\mathsf{S}]\mathsf{T}$, where Γ is a context, P is a possibly open process with $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma)$ and S and T are types. Informally, S and T represent respectively the *in-session*, or *internal*, and the *external* types of P. The first one describes the actions P can perform at the current session level (see (T-Out)), while the second one represents the outputs P can perform at the parent level (see (T-Ret)). Notice how, in (T-Call), the premises ensure compliance between client and service internal types. Rule (T-Pipe) deserves some explanations. We impose some limitations on the types of the pipeline operands. First, the right-hand process is a summation of input-prefixed processes. Second, we make sure that the left-hand type does not contain any summation. Third, we make sure, through $\mathrm{out}(\mathsf{S}) \subseteq \bigcup_{i \in I} \mathsf{b}_i$, that each (type of) output on left-hand side of a pipeline can be matched by (the type of) an input on the other one. Formally, $\mathrm{out}(\mathsf{S})$ corresponds to the set of all bs that occur in output prefixes ($!\mathsf{b}$) in S. The auxiliary functions $\bowtie$ and $@$ are used to build respectively the internal and the external type of $P > Q$ starting from the types of P and Q. In essence, both $\mathsf{S} \bowtie \mathsf{U}$ and $\mathsf{S} @ \mathsf{V}$ spawn a new copy of type U and V, respectively, in correspondence of each output prefix in S. The main difference is that in $@$ inputs and silent prefixes in S are discarded, while in $\bowtie$ they are preserved. Both $\bowtie$ and $@$ are defined by induction on the structure of types, the case of output prefixes is described below, definitions for the omitted cases can be found in [1].

$$!\mathsf{b}.\mathsf{S} \bowtie \textstyle\sum_{i \in I} \alpha_i.\mathsf{U}_i = \sum_{i \in I} \tau.(\mathsf{U}_i | \mathsf{S} \bowtie \sum_{i \in I} \alpha_i.\mathsf{U}_i) \qquad !\mathsf{b}.\mathsf{S} @ \mathsf{U} = \mathsf{U} | (\mathsf{S} @ \mathsf{U})$$

The first step towards proving that well-typed processes guarantee client progress is establishing the usual subject reduction property (Proposition 1). Then, we prove a type safety result (Theorem 1), stating that a well typed process cannot immediately generate an error. These are sufficient to conclude that in well-typed and available processes sessions never stuck, unless the client has terminated its protocol, Corollary 1. The proofs follow the lines of those in [1] and are omitted.

Proposition 1 (subject reduction). *Suppose $\emptyset \vdash P : [S]T$. Then whenever $P \xrightarrow{\tau} P'$ then either $\emptyset \vdash P' : [S]T$ or $S \xrightarrow{\tau} S'$ and $\emptyset \vdash P' : [S']T$.*

Theorem 1 (type safety). *Suppose P is well typed. Then $P \not\rightarrow_{\mathrm{ERR}}$.*

Corollary 1 (client progress). *Suppose P is well typed. Then P guarantees client progress.*

2.3 Client Progress in the Credit Request Scenario

We reconsider the CaSPiS specification of the Credit Request Scenario (see Chapter 0-3) introduced in Chapter 2-1. Here we add typing annotations for input variables and ignore termination handling, which is not dealt with by the type system. So, termination handlers are discarded and close actions are replaced by the empty process. E.g., *CreditPortal* becomes

$$\mathit{CreditPortal} \triangleq !\mathtt{CrReq}.(?\mathit{id} : \mathsf{id})\mathbf{select}\ (?\mathit{logged} : \mathsf{bool})\ \mathbf{from}\ \overline{\mathtt{CheckUser}}(\mathit{id})$$
$$\mathbf{inif}\ \mathit{logged}\ \mathbf{then}\ \langle \text{“Valid”}\rangle \mathit{Creation}(\mathit{id})$$
$$\mathbf{else}\ \langle \mathbf{excpt}(\text{“InvalidLogin”})\rangle$$

We assume that values of the form **excpt**(“string”) are of base type exception. Moreover, we assume that each name has associated a homonymous sort, e.g. *creditD* : creditD. Finally, we assume that each call to an auxiliary service in the system is well typed and returns either boolean values or strings as expected. Suppose now $\mathtt{CrReq} : \mathsf{T}_{\mathtt{CrReq}}$, where:

$$\mathsf{T}_{\mathtt{CrReq}} \triangleq ?\mathsf{id}.\big(\ \tau.!\mathsf{exception} + \tau.!\mathsf{string}.?\mathsf{creditD}.(\ \tau.!\mathsf{exception} + \tau.!\mathsf{string}.?\mathsf{bals}.!\mathsf{string}.?\mathsf{secs}.(\tau.!(\mathsf{offer}).(?\mathsf{bool}+?\mathsf{bool}) + \tau.\mathsf{S}_{\mathit{eval}} + \tau.\mathsf{S}_{\mathit{eval}}))\big)$$
$$\mathsf{S}_{\mathit{eval}} \triangleq \tau.!(\mathsf{offer}).(?\mathsf{bool}+?\mathsf{bool})\ +\ \tau.!(\mathsf{decline}).(?\mathsf{bool}+?\mathsf{bool})\,.$$

Then, the whole system *Sys* is well typed, indeed the client protocol has type

$$\mathsf{U}_{\mathit{ClPr}} \triangleq !\mathsf{id}.\big(?\mathsf{exception}+?\mathsf{string}+?\mathsf{string}.!\mathsf{creditD}.(?\mathsf{exception}+?\mathsf{string}+?\mathsf{string}.!\mathsf{bals}.?\mathsf{string}.!\mathsf{secs}.(?(\mathsf{offer}).!(\mathsf{bool})\ +\ ?(\mathsf{decline}).(\tau.!\mathsf{bool}\ +\ \tau.!\mathsf{bool}))\big)$$

and it is easy to check that $\mathsf{U}_{\mathit{ClPr}} \propto \mathsf{T}_{\mathit{CrReq}}$. Therefore, client progress is guaranteed. Notice that service CrReq is persistent therefore, assuming that all auxiliary services are persistent too, *Sys* (see Chapter 2-1) is *available* and this guarantees that *CreditRequester*, the invoker, will never block.

Consider now another client, *CreditRequester′* below, that does not expect an exception after sending its credentials:

$$\mathit{CreditRequester}' \triangleq \overline{\mathtt{CrReq}}\langle \mathit{id}\rangle((\text{“Valid”})\mathit{CR}\ +\ (\text{“Valid”})\mathbf{0})\,.$$

Clearly, if we replace the previous client with this one, the system will not be well typed. Indeed, the client protocol in *CreditRequester′* is well typed under

$$\mathsf{U}_{\mathit{ClPr}'} \triangleq !\mathsf{id}.(?\mathsf{string}+?\mathsf{string}.!\mathsf{creditD}.\ldots)$$

but $\mathsf{U}_{\mathit{ClPr}'} \not\propto \mathsf{T}_{\mathit{CrReq}}$.

3 From Discovering Type Errors to Preventing Business Logic Flaws

The type system we have seen in Section 2 gives important guarantees about the overall compatibility of interaction protocols between callers and callees, but cannot prevent application logic flaws.

We have investigated this issue, in [3], by adapting the techniques used in the field of network security (see e.g., [5,4]) to that of services. In particular, we have provided a

Control Flow Analysis of CaSPiS, that is shown able to detect some possible misuses and to help prevent them, by taking appropriate counteractions. The analysis statically approximates the behavior of CaSPiS processes, in terms of the possible service and communication synchronizations. More precisely, what the analysis predicts includes everything that may happen, while what the analysis *does not* predict corresponds to something that cannot happen. The model of attacker we are interested in is a bit different from the classical Dolev-Yao one: the *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. The analysis implicitly considers the possible behavior of such an attacker.

3.1 Language Fragment

For the sake of brevity, we focus here on a simplified version of the `close`-free fragment of CaSPiS, where the pipeline construct is rendered as $P > (?\tilde{x})Q$: it spawns a fresh instance $Q[\tilde{v}/\tilde{x}]$ of Q on any value $\tilde{v}$ produced by P. Note that the variables $\tilde{x}$ to be bound after pipeline synchronization are included in a special input $(?\tilde{x})$, called *pipeline input* preceding the right branch process. Moreover, we employ sessions polarities to mark a clear distinction between the two sides involved: $r^{-} \triangleright P$ and $r^{+} \triangleright Q$. To distinguish among different occurrences of the same service, we annotate each of them with a different index, as in $s_{@k}$. The synchronization on the service s, on the occurrences $s_{@k}$ and $\overline{s}_{@m}$, results then in a session, identified by $r^{p}_{s@m:k}$, where p is the polarity. Similarly, we distinguish each pipeline operator with a different label l, as in $>_l$ and we identify the left branch with a label l_0 and the right branch with l_1. The variables $\tilde{x}$ affected by the pipeline input in the right branch of the pipeline, are also identified by the label l_1, as in $P >_l (?\tilde{x}^{l_1})Q$. Note that these annotations do not affect the semantics, whatsoever.

3.2 The Control Flow Analysis

The analysis over-approximates all the possible behaviors of a CaSPiS process, in terms of communication and service synchronizations. The analysis uses the notion of enclosing scope σ, recording the current scope due to services, sessions or pipelines. The result of analyzing a process P is a pair $(\mathcal{I}, \mathcal{R})$, called *estimate* for P. 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.

A proposed estimate $(\mathcal{I}, \mathcal{R})$ is correct, if it satisfies the judgements defined by the axioms and rules in the upper (lower, respectively) part of Fig. 3.

First, we check that $(\mathcal{I}, \mathcal{R})$ describes the initial process. This is done in the upper part of Fig. 3, where the clauses amount to a structural traversal of process syntax (we have omitted rules for parallel composition, restriction and replication, on which the analysis is just propagated to the arguments).

The clause for service definition checks that whenever a service $s_{@k}$ is defined in $s_{@k}.P$, then the relative hierarchy position with respect to 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,

$$
\begin{array}{ll}
\mathcal{I},\mathcal{R} \models^{\sigma} s_{@k}.P & \text{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 & \text{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} r^{p}_{s@m:k} \triangleright P & \text{iff } r^{p}_{s@m:k} \in \mathcal{I}(\sigma) \ \wedge \ \mathcal{I},\mathcal{R} \models^{r^{p}_{s@m:k}} P \\
\mathcal{I},\mathcal{R} \models^{\sigma} \Sigma_{i\in I}\pi_i P_i & \text{iff } \forall i \in I : \mathcal{I},\mathcal{R} \models^{\sigma} \pi_i P_i \\
\mathcal{I},\mathcal{R} \models^{\sigma} (?\tilde{x}).P & \text{iff } (?\tilde{x}) \in \mathcal{I}(\sigma) \ \wedge \ \mathcal{I},\mathcal{R} \models^{\sigma} P \\
\mathcal{I},\mathcal{R} \models^{\sigma} \langle \tilde{x} \rangle.P & \text{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^{\sigma} \langle \tilde{x} \rangle^{\uparrow}.P & \text{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} P >_l (?\tilde{x}^{l_1})Q & \text{iff } l_0, l_1, \mathcal{I}(l_0), \mathcal{I}(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 \\
\mathcal{I},\mathcal{R} \models^{l_0} \langle \tilde{x} \rangle.P & \text{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^{l_1} (?\tilde{x}^{l_1}).P & \text{iff } (?\tilde{x}^{l}) \in \mathcal{I}(l_1) \ \wedge \ \mathcal{I},\mathcal{R} \models^{l_1} P
\end{array}
$$

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

Fig. 3. Analysis for CaSPiS processes

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, modeled as the one on service definition and invocation, just checks that the relative hierarchy position of the session identifier $r^{p}_{s@m:k}$ with respect to the enclosing scope must be reflected in $\mathcal{I}$, i.e. $r^{p}_{s@m:k} \in \mathcal{I}(\sigma)$. It is used in analyzing the possible continuations of the initial process.

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 synchronizations, 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 consideration, i.e. the synchronizations τ due to communications, services and pipelines. This is expressed by the closure conditions in the lower part of Fig. 3 that mimic the semantics, by statically modeling the semantic preconditions and

the consequences of the possible actions. More precisely, the precondition checks, in terms of $\mathcal{I}$, for the possible presence of the redexes necessary for an action to be performed. The conclusion imposes the additional requirements on $\mathcal{I}$ and on $\mathcal{R}$, necessary to give a valid prediction of the analyzed action. In the clause for *Service Synch*, we have to make sure that the precondition requirements are satisfied, i.e. that: (i) there exists an occurrence of service definition: $s_{@k} \in \mathcal{I}(\sigma)$; (ii) 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 synchronization. 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 $\mathcal{I}(r^{-}_{s@m:k})$, respectively. Similarly, in the clause for *I/O Synch*, if the following preconditions are satisfied: (i) there exists an occurrence of output in $\mathcal{I}(r^{p}_{s@m:k})$; (ii) 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. In other words 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 synchronizations. The other rules are analogous.

Our analysis is correct with respect to the given semantics, i.e. a valid estimate enjoys the following subject reduction property.

Theorem 2 (Subject Reduction). *If* $P \xrightarrow{\tau} Q$ *and* $\mathcal{I}, \mathcal{R} \models^{\sigma} P$ *then also* $\mathcal{I}, \mathcal{R} \models^{\sigma} Q$.

In the following, we refer to the version of CaSPiS that includes pattern matching into the input construct. Furthermore, we need to consider the possible presence of the *malicious customer*, that is an *accredited customer of a service* that has no control of the communication channels, apart from the one established by the sessions in which he/she is involved. Nevertheless, our attacker 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. More precisely, he/she has a knowledge made of all the public information and increased by the messages received from the service: the attacker can use his/her knowledge to produce messages to be sent to the server. The presented analysis is part of a more complex analysis, that implicitly considers the possible behavior of such an attacker. The complete analysis takes care of the malicious customer presence, by statically approximating its possible knowledge, represented as 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.

3.3 Control Flow Analysis of the Scenario

For brevity, we refer to a quite simplified version of the Credit Request Scenario, leaving aside authentication through the credit portal and approval by supervisor. We also omit

$$
\begin{aligned}
\mathit{Prtl} &\equiv \mathrm{CrReq}.(?x_{usr}, ?x_{cred}, ?x_{bals}, ?x_{secs}) \\
&\qquad (\overline{\mathrm{Upd}}.\langle x_{usr}, x_{bals}, x_{secs}\rangle(?x_{ack})\langle \mathsf{go}\rangle^{\uparrow} >_{l} (\mathit{Dcsn}(x_{usr}, x_{cred}) >_{l'} \mathit{Offer})) \\
\mathit{Dcsn}(u, c) &\equiv (?x_{go})\overline{\mathrm{Rate}}.\langle u, c\rangle((\mathsf{AAA}, ?x_{offer})\langle \mathsf{true}, x_{offer}\rangle^{\uparrow} + (\mathsf{BBB}, ?x_{risk})\mathit{Clerk}(x_{usr}, x_{risk})) \\
\mathit{Clerk}(u, r) &\equiv \overline{\mathrm{ReqCk}}.\langle u, r\rangle(?w_{response}, ?w_{offer})\langle w_{response}, w_{offer}\rangle^{\uparrow} >_{l''} \\
&\qquad (?u_{response}, ?u_{offer})\langle u_{response}, u_{offer}\rangle^{\uparrow} \\
\mathit{Offer} &\equiv (?z_{response}, ?z_{offer})\langle z_{response}, z_{offer}\rangle((\mathsf{true}, ?z_{cred}) + (\mathsf{false}, ?z_{decline})\mathbf{0}) \\
\mathit{CR}(\mathsf{usr}) &\equiv \overline{\mathrm{CrReq}}.(\nu \mathsf{cred}, \mathsf{bals}, \mathsf{secs})\langle \mathsf{usr}, \mathsf{cred}, \mathsf{bals}, \mathsf{secs}\rangle \\
&\qquad (\mathsf{true}, ?y_{offer})\langle \mathbf{eval}(y_{offer}, \mathsf{cred})\rangle\mathbf{0}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{I}(*) \ni \mathrm{CrReq}, \overline{\mathrm{CrReq}}, r^{+}_{CrReq}, r^{-}_{CrReq} \\
&\mathcal{I}(\mathrm{CrReq}) \ni (?x_{usr}, ?x_{cred}, ?x_{bals}, ?x_{secs}), l_0, l_1, \mathcal{I}(l_0), \mathcal{I}(l_1) \\
&\mathcal{I}(r^{+}_{CrReq}) \ni r^{-}_{Upd}, r^{-}_{Rate}, r^{-}_{ReqCk} \\
&\mathcal{R}(x_{usr}) \ni \mathsf{usr}, \mathcal{R}(x_{cred}) \ni \mathsf{cred}, \mathcal{R}(x_{bals}) \ni \mathsf{bals}, \mathcal{R}(x_{secs}) \ni \mathsf{secs} \\
&\mathcal{I}(l_0) \ni \overline{\mathrm{Upd}} \\
&\mathcal{I}(\overline{\mathrm{Upd}}) \ni \langle \mathsf{usr}, \mathsf{bals}, \mathsf{secs}\rangle, (?x_{ack}), \langle \mathsf{go}\rangle^{\uparrow} \\
&\mathcal{I}(l_1) \ni l'_0, l'_1, \mathcal{I}(l'_0), \mathcal{I}(l'_1) \\
&\mathcal{I}(l'_0) \ni (?x_{go}), \overline{\mathrm{Rate}} \\
&\mathcal{I}(\overline{\mathrm{Rate}}) \ni \langle \mathsf{usr}, \mathsf{cred}\rangle((\mathsf{AAA}, ?x_{offer}), \langle \mathsf{true}, \mathsf{offer}\rangle^{\uparrow}, \mathcal{I}(l''_0) \ni \overline{\mathrm{ReqCk}} \\
&\mathcal{I}(l''_1) \ni (?u_{response}, ?u_{offer})\langle \mathsf{response}, \mathsf{offer}\rangle^{\uparrow}(\mathsf{BBB}, ?x_{risk}), \overline{\mathrm{ReqCk}}, l''_0, l''_1, \mathcal{I}(l''_0), \mathcal{I}(l''_1) \\
&\mathcal{I}(\overline{\mathrm{ReqCk}}) \ni \langle \mathsf{usr}, \mathsf{risk}\rangle, (?w_{response}, ?w_{offer}), \langle \mathsf{response}, \mathsf{offer}\rangle^{\uparrow} \\
&\mathcal{I}(l''_1) \ni (?z_{response}, ?z_{offer}), \langle \mathsf{response}, \mathsf{offer}\rangle, (\mathsf{true}, ?z_{cred}), (\mathsf{false}, ?z_{decline}) \\
&\mathcal{I}(\overline{\mathrm{CrReq}}) \ni \langle \mathsf{usr}, \mathsf{cred}, \mathsf{bals}, \mathsf{secs}\rangle, (\mathsf{true}, ?y_{offer}), \langle \mathsf{offer}, \mathsf{cred}\rangle, \langle \mathsf{offer}, \underline{\mathsf{cred}'}\rangle, \langle \mathsf{false}, \mathsf{decline}\rangle
\end{aligned}
$$

Fig. 4. Specification of our scenario and some entries of its analysis

some service counterparts, whose specification is trivial. In this particular toy example, the application logic flaw has been introduced on purpose, in order to illustrate our methodology. Still, we think it is representative of analogous flaws, reported in the literature, as the price modification one [14], handled in [3].

The introduced application logic flaw is related to the handling of requested amount of credit, here not considered for the outcome of the clerk approval request. Note that, as in the original specification, the evaluation of the clerk is not dependent on the requested amount of credit.

The scenario specification and some of 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. We assume **eval**(o, c) either evaluates to $(\mathsf{false}, \mathsf{decline})$ (the conditions offered by the bank are not convenient) or to (true, c) where c is the amount for which the credit is requested. The analysis directly considers the resulting values for **eval**.

Note that the variable $?z_{credit}$ in *Offer* may be bound to any value the costumer sends, in particular to any credit value *cred′*, possibly higher than *cred*. This is reflected by the analysis, because if $\mathit{cred}' \in \mathcal{K}$, we also have that $\langle \mathsf{offer}, \mathsf{cred}'\rangle$ belongs to $\mathcal{I}(\overline{\mathrm{CrReq}})$. This application flaw 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. To avoid this problem, we can modify the specification of *Offer*, by assuming it expects to receive a specific value c on the critical input, in our example the value *cred* initially bound to x_{cred}:

$$
\mathit{Offer}'(c) \equiv (?z_{response}, ?z_{offer})\langle z_{response}, z_{offer}\rangle((\mathsf{true}, \mathsf{c}) + (\mathsf{false}, \mathsf{decline})\mathbf{0})\,.
$$

4 Conversation Types

In the previous sections we have focused our analysis on the interaction between two parties, typically a client and a server. However, service-oriented applications often rely on collaborations between several partners, usually established dynamically and without centralized control. A central concern in the development of service-oriented systems is thus the design of interaction protocols that allow for the decentralized and dynamic collaboration between several parties in a reliable way. Within the SENSORIA project we have developed a novel type-based approach that copes with such challenging scenarios involving dynamically established multiparty collaborations [8].

In the remainder of this section we discuss the analysis techniques introduced in [8] that support the verification of key properties, namely conversation fidelity and progress, addressing scenarios where multiple parties interact in a conversation, even when some of them are dynamically called in to participate, and where parties interleave their participation in several of such collaborations, even in the dynamically established ones. Such challenging scenarios are of interest as they can be found in realistic service-oriented applications, and fall out of scope of previous approaches—namely the works on multiparty session types [10,2]. Our techniques, although independent, may be viewed as complementary. In fact, it is their combined use that allows us to prove the progress property.

4.1 Analyzing Multiparty Protocols with Conversation Types

In [8] we introduced a type theory for analyzing concurrent multiparty interactions as found in service-oriented computing based on the notion of conversation (see Chapter 2-1). A conversation is a structured, not centrally coordinated, possibly concurrent, set of interactions between several participants. The notion of conversation builds on the fundamental concept of session, but generalizes it along directions. In particular, conversation types discipline interactions in conversations while accounting for dynamical join and leave of an unanticipated number of participants.

Our type system combines techniques from linear, behavioral, session and spatial types (see [8] for references): the type structure features prefix $M.B$, parallel composition $B_1 \mid B_2$ (to represent concurrent behavior), and also choice and branch types that capture alternative behavior: the former characterizes processes that can perform one of the $M_i.B_i$ choices, and the latter characterizes processes that can perform either one of the $M_i.B_i$ branches. Messages M describe external (receive ? / send !) exchanges in two views (d): with the *enclosing* ($\uparrow$) and *current* conversations ($\downarrow$). They also describe internal message exchanges (τ). The type language is shown in Fig. 5. Notice that conversation types mix, at the same level in the type language, internal/global specifications (τ message exchanges) with interface/local specifications (output ! and input ? types).

Key technical ingredients in our approach to conversation types are the amalgamation of global types and of local types (in the general sense of [10]) in the same type

$$B ::= B_1 \mid B_2 \mid \mathbf{0} \mid \mathtt{rec}\mathcal{X}.B \mid \mathcal{X} \mid \oplus_{i\in I}\{M_i.B_i\} \mid \&_{i\in I}\{M_i.B_i\} \quad \text{(Behavioral)}$$

$$M ::= pl^d([B]) \quad \text{(Message)} \qquad p ::= \mathtt{!} \mid \mathtt{?} \mid \tau \quad \text{(Polarity)}$$

$$L ::= n : [B] \mid L_1 \mid L_2 \mid \mathbf{0} \quad \text{(Located)} \qquad T ::= L \mid B \quad \text{(Process)}$$

Fig. 5. Conversation types syntax.

$$\frac{\forall_{i\in I}(B_i = B_i^- \bowtie B_i^+)}{\oplus_{i\in I}\{\tau\, l_i^{\downarrow}(C).B_i\} = \oplus_{i\in I}\{!\, l_i^{\downarrow}(C_i).B_i^+\} \bowtie \&_{i\in I}\{?\, l_i^{\downarrow}(C_i).B_i^-\}}(\mathit{Plain})$$

$$\frac{M_1 \# M_2.B_2 \qquad B_1' \mid B_2' = B_1 \bowtie M_2.B_2}{M_1.B_1' \mid B_2' = M_1.B_1 \bowtie M_2.B_2}(\mathit{Shuffle}) \qquad \frac{B_1 \# B_2}{B_1 \mid B_2 = B_1 \bowtie B_2}(\mathit{Apart})$$

Fig. 6. Behavioral type merge relation selected rules.

language, and the definition of a merge relation ensuring, by construction, that participants typed by the projected views of a type will behave well under composition. Merge subsumes duality, in the sense that for each τ-free B there are types $\overline{B}, B'$ such that $B \bowtie \overline{B} = \tau(B')$, so dyadic sessions are special cases of conversations. But merge of types allows for extra flexibility on the manipulation of projections of conversation types, in an open-ended way. In particular, our approach allows fragments of a conversation type (e.g., a choreography) to be dynamically distributed among participants, while statically ensuring that interactions follow the prescribed discipline.

Building on the capability to mix local and global specifications, we are able to describe, via the merge relation, arbitrary decompositions of the protocol in the roles of one or more parties. This allows, in particular, for a single participant to be initially typed with a fragment of the protocol that will be dynamically delegated away, which is crucial to support conversation join. We write $B = B_1 \bowtie B_2$ to say that B is a particular (in general not unique) behavioral combination of the types B_1 and B_2. The merge of two independent types (measured up to *apartness* # which determines if the types have distinct message alphabets) yields the independent composition of the two types. However, when the types specify behaviors that may synchronize, then the merge relation introduces an internal message exchange τ in the type to represent such synchronization potential. Thus, the merge of two behaviors is defined not only in terms of spatial separation, but also, and crucially, in terms of merging behavioral "traces". Fig. 6 shows a selection of the merge relation rules.

We may then characterize the behavior of CC systems by means of conversation types. Our type system singles out CC processes that enjoy some safety properties, namely that are free from a certain kind of runtime errors, and also that their processes, at runtime, follow the protocols prescribed by the types. The behavioral types capture the protocols of interaction in a single conversation. Since processes, in general, may interact in several conversations, characterizing a CC system involves describing the several protocols the process has in each conversation. Then, the typing judgment: $P :: B \mid n : [B_n] \mid m : [B_m] \mid o : [B_o] \mid \ldots$ specifies that the behavior of process P in conversations $n, m, \ldots$ is captured by behavioral types $B_n, B_m, \ldots$, respectively. Notice that, since CC processes may interact in the current and enclosing conversations, the

$$\frac{P :: T_1 \quad Q :: T_2}{P \mid Q :: T_1 \bowtie T_2}(Par) \qquad \frac{P :: L \mid B}{l^d!\langle n\rangle.P :: (L \bowtie n : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\}}(Output)$$

$$\frac{P :: T \mid a : [B] \quad (closed(B))}{(\nu a)P :: T}(Res) \qquad \frac{P :: L \mid B}{n \blacktriangleleft [P] :: (L \bowtie n : [\downarrow B]) \mid loc(\uparrow B)}(Piece)$$

Fig. 7. Selected typing rules.

typing judgment considers an unlocated behavioral type B. Such a typing judgement $P :: T$ intuitively says that if process P is placed in a context where a process that behaves like T is expected then we obtain a safe system.

We show a selection of the rules of our type system in Fig. 7. Rule (*Par*) says that the composition is well typed under the merge of the types of the branches. Recall the merge explains the composition of two processes by synchronizing behavioral traces. Rule (*Res*) types the name restriction by checking if the behavioral type of the restricted conversation is closed, in such case eliding it in the conclusion of the rule. Closed behavioral types characterize processes that have matching receives for all sends—roughly, a closed type is defined exclusively on messages of polarity τ. The type associated to a process only describes the behaviors in the visible conversations, and the closed condition avoids hiding a conversation where there are unmatched communications.

The premise of rule (*Output*) specifies that the continuation process P defines some located behavior L and some unlocated behavior B. Then, the output prefix is typed by the merge of the delegated conversation fragment $n : C$ with the located behavior L, along with a choice type that includes the output action specified in the prefix with respective continuation. Notice that the conversation fragment piece that is delegated away is actually a separate $\bowtie$ view of conversation n, which means that the type being sent may actually be some separate part of the type of the conversation. This mechanism is crucial to allow external partners to join in on ongoing conversations in a disciplined way. The behavioral interface of the output prefixed process is a choice type, as the process can choose the specified action from any set of choices that contains it.

We may now state our type safety results. To describe how typings are preserved under process reduction we introduce a notion of type reduction which is a reflexive relation obtained by the closure under static contexts of rule $\tau\, l^d(C).B \to B$: hence, synchronizations in the process are explained by τ message types. We may then precisely characterize the preservation of typing by means of type reduction.

Theorem 3 (Subject Reduction). *Let process P be such that $P :: T$. If $P \to Q$ then there is $T \to T'$ such that $Q :: T'$.*

Subject reduction thus guarantees that well-typedness is invariant under process reduction. Moreover, each reduction in a process is explained by a reduction in the type. Reflexivity in the type reduction relation is required since it allows us to abstract away from message synchronizations that occur at the level of restricted conversations.

Our type safety result asserts that certain error processes are unreachable from well-typed processes. Error processes are configurations where there is an active race on a (linear) message, which means two processes are willing to send or waiting to receive the same message. Thus, a process is not an error only if for each possible immediate

interaction in a message there is at most a single sender and a single receiver. We prove well-typed systems are error free, and thus are error free throughout their evolution.

Proposition 2. *If P is a well-typed process then P is not an error process.*

Corollary 2 (Type Safety). *Let P be a well-typed process. If there is Q such that $P \xrightarrow{*} Q$, then Q is not an error process.*

Our type safety result ensures that, in any reduction sequence arising from a well-typed process, for each message ready to communicate there is always at most a unique input/output outstanding synchronization. Subject reduction also entails that any message exchange in the process must be explained by a τM prefix in the related conversation type, implying conversation fidelity: all conversations follow the prescribed protocols.

4.2 Proving Progress of Conversations

In this section we present the progress proof system, introduced in [8], which allows us to verify that systems enjoy a progress property. While the conversation type system allows us to guarantee that conversations follow the prescribed protocols, it is not enough to guarantee that the systems do not get stuck, due to, e.g., communication dependencies between distinct conversations. As most traditional deadlock detection methods (see [8] for references), we build on the construction of a well-founded ordering on events. Roughly, we must check that the events specified in the continuation of a prefix are of greater rank with respect to the event relative to the prefix itself.

The challenge is how to statically account for the orderings of events on conversations which will only be dynamically instantiated. To solve this problem we attach to our events a notion of prescribed ordering: the ordering that captures the event ordering expected by the receiving process, that the emitted name will have to comply to. In such way, we are able to statically determine the orderings followed by the processes at "runtime", through propagation of orderings in the analysis of message exchanges that carry conversation identifiers. Technically, we proceed by developing a notion of event and of event ordering that allow us to verify that CC processes can be ordered in a well-founded way, including when conversation references are passed around.

Definition 5 (Event orderings and Events). *We say relation Γ between events is an event ordering if it is a well-founded partial order of events. We denote by $(x)\Gamma$ an event ordering parameterized by x. Events, noted e, are defined as $e, e_1, \ldots ::= n.l.(x)\Gamma$.*

Event orderings capture the overall ordering of events. Parameterized event orderings are used to capture the prescribed ordering of conversation fragments that are passed in messages. Events describe a message exchange by identifying the name of the conversation, the label of the message, and the parameterized event ordering.

We may then characterize the event ordering in CC systems by means of a proof system that associates CC systems to event orderings. The proof system is presented by means of judgments of the form $\Gamma \vdash_\ell P$, which state that the communications of process P follow a well determined order, given by event ordering Γ, where ℓ keeps track of the identities of the current and enclosing conversations of P.

We show the rule that orders the output prefix process:

$$\frac{(\ell(d).l.(x)\Gamma' \perp \Gamma) \vdash_\ell P \quad \Gamma'\{x{\leftarrow}n\} \subseteq (\ell(d).l.(x)\Gamma' \perp \Gamma)}{\Gamma \vdash_\ell l^d!(n).P}(\mathit{Output})$$

In rule (*Output*), we take the event associated to the prefix $(\ell(d).l.(x)\Gamma'$ where $\ell(d)$ identifies the conversation, l is the message label and $(x)\Gamma'$ specifies the prescribed ordering for the conversation reference passed in the message) and verify that the continuation is ordered by events greater than $\ell(d).l.(x)\Gamma'$. Also, and crucially, we check that the name being passed in the output complies with the ordering prescribed in the event $(x)\Gamma'$ by verifying that such prescribed ordering, where the variable x is replaced by the name to be sent n, is contained in the ordering of events greater than $\ell(d).l.(x)\Gamma'$. In such way, we ensure the overall ordering is still respected after the name passing.

We may now present our progress results. First, we prove orderings are preserved under process reduction. Then, we present our main progress result, where we exclude processes that are stuck. We distinguish stuck processes from finished processes, which, intuitively, can be viewed as collections of service definitions, where there are no pending service calls, neither linear protocols to be fulfilled. We then consider finished processes to be in a stable state, despite the fact they have no reductions.

Theorem 4 (Preservation of Event Ordering). *Let process P be such that $\Gamma \vdash_\ell P$. If $P \rightarrow Q$ then $\Gamma \vdash_\ell Q$.*

Theorem 5 (Progress). *Let P be a process such that $P :: T$, where closed(T), and $\Gamma \vdash_\ell P$. If $P \stackrel{*}{\rightarrow} Q$ then either Q is a finished process or there is $Q \rightarrow Q'$.*

Theorem 5 thus ensures that in systems that get past our rules, services are always available upon request and protocols involving interleaving conversations never get stuck.

4.3 Typing the Credit Request Scenario

In this section we show the typing for the credit request scenario CC implementation given in Chapter 2-1. The CC code for the credit request, the *CreditRequestSystem*, presents a challenging scenario for analysis techniques as it involves collaborations between multiple parties established in a dynamic way and where parties interleave their participation in several conversations, including ones to which they have dynamically gained access to. The typing judgment for the entire system is shown in Fig. 8, where we focus on the typing of the Finance Portal conversation, along with the definitions of the abbreviations introduced for the service types (e.g., acsT for **A**uth**C**redit**S**ervice**T**ype).

The typing of the *FinancePortal* conversation captures the interactions in services `CreditRequest`, `ReviewApp` and `AuthCredit`, along with the messages `requestApp` and `requestEval` which are exchanged between the service instances, each identifying in the argument type the conversation fragment delegated in the communications. For instance, the (first) argument type of messages `requestEval` and `requestApp` is $\oplus\{!\,\mathtt{approved}^{\downarrow}(); !\,\mathtt{denied}^{\downarrow}()\}$, which captures the client conversation fragment delegated from the `CreditRequest` instance to the `ReviewApp` service instance, and from the `ReviewApp` service instance to the `AuthCredit` service instance, respectively, allowing for the reply to the client to originate in the other service instances.

$$
\begin{array}{l}
\mathit{CreditRequestSystem} :: \\
\quad \mathit{FinancePortal} : [\\
\qquad \tau\,\mathtt{CreditRequest}([\mathsf{crsT}]) \mid \tau\,\mathtt{ReviewApp}([\mathsf{rasT}]) \mid \tau\,\mathtt{AuthCredit}([\mathsf{acsT}]) \\
\qquad \mid \tau\,\mathtt{requestApp}([\oplus\{!\,\mathtt{approved}^{\downarrow}(); !\,\mathtt{denied}^{\downarrow}()\}], \mathit{idT}, \mathit{dataT}) \\
\qquad \mid \tau\,\mathtt{requestEval}([\oplus\{!\,\mathtt{approved}^{\downarrow}(); !\,\mathtt{denied}^{\downarrow}()\}], \mathit{idT}, \mathit{dataT})\ ] \\
\quad \mid \mathit{Bank} : [\ \tau\,\mathtt{RateCalc}([\mathsf{rcsT}]) \mid \ldots\ ] \\
\quad \mid \mathit{Client} : [\ \ldots\ ] \mid \mathit{Clerk} : [\ \ldots\ ] \mid \mathit{Manager} : [\ \ldots\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{crsT} \triangleq ?\,\mathtt{login}^{\downarrow}(\mathit{idT}).?\,\mathtt{request}^{\downarrow}(\mathit{dataT}).\tau\,\mathtt{userData}^{\downarrow}(\mathit{dataT}). \\
\qquad\qquad \tau\,\mathtt{rateValue}^{\downarrow}(\mathit{rateT}).\oplus\{!\,\mathtt{approved}^{\downarrow}()\ ;\ !\,\mathtt{denied}^{\downarrow}()\} \\
\mathsf{rasT} \triangleq ?\,\mathtt{login}^{\downarrow}(\mathit{idT}).!\,\mathtt{show}^{\downarrow}(\mathit{idT},\mathit{dataT}).\&\{?\,\mathtt{pass}^{\downarrow}()\ ;?\,\mathtt{deny}^{\downarrow}()\} \\
\mathsf{acsT} \triangleq ?\,\mathtt{login}^{\downarrow}(\mathit{idT}).!\,\mathtt{show}^{\downarrow}(\mathit{idT},\mathit{dataT}).\&\{?\,\mathtt{accept}^{\downarrow}()\ ;?\,\mathtt{reject}^{\downarrow}()\} \\
\mathsf{rcsT} \triangleq ?\,\mathtt{userData}^{\downarrow}(\mathit{dataT}).!\,\mathtt{rateValue}^{\downarrow}(\mathit{rateT})
\end{array}
$$

Fig. 8. Typing the credit request scenario CC Implementation.

The conversation types for each service interaction are obtained by merging service provider and service user behaviors. For instance, the interaction between the client and the finance portal is captured by the merge of the `CreditRequest` service type (crsT) with the client role in the service conversation:

$$
\begin{array}{l}
\tau\,\mathtt{login}^{\downarrow}(\mathit{idT}).\tau\,\mathtt{request}^{\downarrow}(\mathit{dataT}).\tau\,\mathtt{userData}^{\downarrow}(\mathit{dataT}). \\
\qquad \tau\,\mathtt{rateValue}^{\downarrow}(\mathit{rateT}).\oplus\{\tau\,\mathtt{approved}^{\downarrow}()\ ;\ \tau\,\mathtt{denied}^{\downarrow}()\} \\
= \\
?\,\mathtt{login}^{\downarrow}(\mathit{idT}).?\,\mathtt{request}^{\downarrow}(\mathit{dataT}).\tau\,\mathtt{userData}^{\downarrow}(\mathit{dataT}). \\
\qquad \tau\,\mathtt{rateValue}^{\downarrow}(\mathit{rateT}).\oplus\{!\,\mathtt{approved}^{\downarrow}()\ ;\ !\,\mathtt{denied}^{\downarrow}()\} \\
\bowtie \\
!\,\mathtt{login}^{\downarrow}(\mathit{idT}).!\,\mathtt{request}^{\downarrow}(\mathit{dataT}).\&\{?\,\mathtt{approved}^{\downarrow}()\ ;?\,\mathtt{denied}^{\downarrow}()\}
\end{array}
$$

Notice that the merge yields a closed type: hence, all communications are matched. Notice also that the `CreditRequest` service type (crsT) still refers some internal communications (τ message types in messages `userData` and `rateValue`). This is crucial to support the dynamic join of the `RateCalc` service to the `CreditRequest` service conversation, as it allows for the *CreditRequest* code to further along delegate a conversation fragment to `RateCalc`, while ensuring that the overall protocol is followed.

We may also show that the events in the *CreditRequestSystem* are well ordered, and thus ensure that the *CreditRequestSystem* enjoys some fundamental properties.

Corollary 3. *The CreditRequestSystem enjoys conversation fidelity and progress.*

5 Conclusion

We have reported on the static analysis techniques developed for CaSPiS and CC, two session oriented calculi developed within the Sensoria project [1,3,8]. Each technique aims at guaranteeing a specific property one would expect from service-oriented applications. Our models and techniques may be complementary used and combined in

order to provide provide as many guarantees as possible on the correctness of services' behavior.

The most relevant related works have been discussed throughout the chapter. More extensive discussions on related techniques and possible extensions can be found in the original papers introducing the approaches reported here [1,3,8].

References

1. L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 642–658. Springer-Verlag, 2008.
2. L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In F. van Breugel and M. Chechik, editors, *CONCUR 2008*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, 2008.
3. C. Bodei, L. Brodo, and R. Bruni. Static detection of logic flaws in service-oriented applications. In P. Degano and L. Viganò, editors, *ARSPA-WITS*, volume 5511 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 2009.
4. C. Bodei, L. Brodo, P. Degano, and H. Gao. Detecting and preventing type flaws at static time. *Journal of Computer Security*, 18(2):229–264, 2010.
5. 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.
6. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F. S. de Boer, editors, *FMOODS*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer-Verlag, 2008.
7. R. Bruni and L. G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In J. Meseguer and G. Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2008.
8. L. Caires and H. Vieira. Conversation Types. In G. Castagna, editor, *ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 285–300. Springer-Verlag, 2009.
9. S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation equivalence is decidable for basic parallel processes. In E. Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, 1993.
10. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In G. Necula and P. Wadler, editors, *POPL 2008*, pages 273–284. ACM Press, 2008.
11. A. Igarashi and N. Kobayashi. A Generic Type System for the π-Calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.
12. I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM*, pages 305–314. IEEE Computer Society, 2007.
13. L. Mezzina. *Typing Services*. Phd thesis in computer science, IMT Institute for Advanced Studies, Lucca, 2009.
14. Neohapsis Archives. *Price modification possible in CyberOffice Shopping Cart*. **http://archives.neohapsis.com/archives/bugtraq/2000-10/0011.html**.
15. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.