

Automotive and Finance Case Study in CSCC

Car Break and Credit Request Scenarios

Luís Caires

João Costa Seco

Hugo Torres Vieira

July 2, 2007

Abstract

We describe the encoding of the Car Break scenario of the SENSORIA Automotive case study and of the Credit Request scenario of the SENSORIA Finance case study using the Conversation Calculus (CSCC). These scenarios consist of an orchestration of services and service clients which are typefully encoded here in a modular way. Namely the latter scenario consists of a workflow involving different actors: a client willing to submit a credit request, a bank employee, and its supervisor. We show how the workflow is well described in the type assigned to the processes implementing it. We first informally describe the CSCC calculus, and then show how the two scenarios can be encoded using the CSCC calculus and the corresponding typing.

1 Introduction

This report documents the developments of the Conversation Calculus (CSCC) [9] when applied to the case studies of the SENSORIA project [1]. We describe the encoding of the Car Break and Credit Request scenarios of the SENSORIA case studies Automotive [4] and Finance [3].

These scenarios consist of an orchestration of services and service clients which are typefully encoded here in a modular way. Namely the latter scenario consists of a workflow involving different actors: a client willing to submit a credit request, a bank employee, and its supervisor. We show how the workflow is well described in the type assigned to the processes implementing it. We first informally present the CSCC calculus, and then show how the two scenarios can be encoded using the CSCC calculus and the corresponding typing.

Section 2 informally presents CSCC, Section 3 presents encodings for the two case studies highlighting different capabilities of the calculus, and the report follows, in Section 4, by discussing some typing issues of CSCC, using the running examples. We end the document with some concluding remarks, in Section 5, about the kinds of systems that can be typefully expressed by the calculus.

2 CSCC

We briefly present the motivation, structure, and informal semantics of the Conversation Calculus. The Conversation Calculus integrates a small set of abstractions for expressing and analyzing service based systems [9]. It focuses on the aspects of distribution, process delegation, communication and context sensitiveness, and loose coupling, believed to be essential to the service oriented computational model. Distinguishing aspects of our model are the adoption of a very simple, context

$$\begin{array}{l}
P, Q \quad ::= \text{stop} \\
\quad | P \mid Q \\
\quad | (\text{new } n)P \\
\quad | \text{out } \textit{direction } m(v_1, \dots, v_n).P \\
\quad | \text{in } \textit{direction } m(x_1, \dots, x_n).P \\
\quad | !P \\
\\
\quad | n \textit{role } [P] \\
\quad | \text{here}(x).P \\
\quad | \text{instance } n \textit{role } s \Leftarrow P \\
\quad | \text{def } s \Rightarrow P \\
\\
\quad | \text{try } P \text{ catch } Q \\
\quad | \text{throw}.P \\
\\
\textit{role} \quad ::= \blacktriangleleft \mid \blacktriangleright \\
\textit{direction} ::= \leftarrow \mid \uparrow \mid \downarrow
\end{array}$$

Figure 1: The Calculus

sensitive, message passing, local communication mechanism, and a novel mechanism for handling exceptional behavior. Technically, we proceed by extending a fragment of the π -calculus; the resulting calculus may nevertheless be seen as a possible evolution of the preliminary SCC presented in [5]. While focusing on the notion of process delegation, first class conversation contexts, and the loosely coupled message passing mechanism, the Conversation Calculus distinguishes itself from other descendants of SCC [12, 6], which explore different interaction mechanisms. We informally describe the several primitives of the Conversation Calculus. The abstract syntax of the calculus is given in Figure 1, where we assume given a set Λ of names (m, n, s) .

Context A context is a delimited container where closely related computation and communication happen. In more general terms, a context is a general abstraction that may be used to model locations (e.g., a unit of distribution), service endpoints (e.g., a delimited scope of communication), contexts of conversation (e.g., a correlated set of interacting partners), and other forms of localized interaction. Contexts encapsulate functionality and appear to the surrounding environment as a plain local process, thus allowing system descriptions to abstract away from particular implementation details. Any context participates in one of two possible dual roles (initiator \blacktriangleleft and responder \blacktriangleright).

$$\begin{array}{l}
\textit{role} ::= \blacktriangleleft \mid \blacktriangleright \\
\textit{ContextName } \textit{role } [Process]
\end{array}$$

For example, $n \blacktriangleleft [P]$ represents a process P inside the initiator endpoint of the context n .

Service publication and instantiation A distinguishing feature of service oriented computing is the emphasis on the remote delegation of *interactive processes*, rather than on the remote delegation

of individual operations. We consider service instantiation as a higher level mechanism, allowing a service client to delegate to a remote server not just a single operation or task, but the execution of a whole interactive activity (technically, a process). By instantiating a service, a client is able to incorporate a new process in its workflow (a dynamic stateful interactive entity) that, although executing remotely in the server environment, appears to the client as a local subsystem.

Any context may publish one or more service definitions. Service definitions are located in contexts – seen as sites. Published services may be instantiated upon clients request, resulting in the creation of a new context of conversation composed by two endpoints (initiator and responder). We must differentiate service instantiation, that creates new process delegates in the service provider and client sites, from service invocation that is modeled by some communication mechanism, more concretely, message passing. The interaction of these primitives with contexts and communication abstractions is inspired on, but differs from, the basic “session” mechanism of other session based service calculi, and allows contexts to be used with great flexibility (for example, to model multi-party interactions, abstract correlation, and session delegation). Service definition and service instantiation are thus written

```
def ServiceName ⇒ ServiceBody
instance ServiceProvider role ServiceName ⇐ ClientProtocol
```

Context awareness A process executing in a given context should be able to dynamically access the identity of such context, in order to correlate its behavior with other partners, and act accordingly. We thus introduce the primitive

```
here(x).Process
```

Communication Interaction between subsystems (e.g., contexts, endpoints, sites) is realized by message passing. However communication is only allowed between “adjacent” contexts. The possible communication paths are: between two dual endpoints (*other*, written \leftarrow), between a context and its enclosing context (*up*, written \uparrow), and internally to a context (*here*, written \downarrow).

```
direction ::=  $\leftarrow$  |  $\uparrow$  |  $\downarrow$ 
in direction Message(x, ..., x).Process
out direction Message(v, ..., v).Process
```

Exception handling Exceptional behavior, in particular fault signaling, fault detection, and resource disposal, are aspects orthogonal to the existing communication mechanisms, for which specific abstractions are provided.

```
try GuardedProcess catch HandlerProcess
throw SignalingProcess
```

3 Examples

We here illustrate CSCC by means of two examples inspired by the SENSORIA Automotive and Finance case studies [4, 3].

We first present the Car Break scenario of the SENSORIA Automotive case study, where a set of distributed services collaborate to assist a driver of a malfunctioning car. It involves a signal sent

```

ServicePlanner ▶ [
  def CarBreak ⇒
    in ← LocalDiagnosis(data).in ← GPSLoc(myLoc).
    (instance RepairShop ▶ BookRepair ⇐
      out ← BookRepairOperation().
      in ← BookingAccepted(shopLoc, bookingRef).
      (out ↑ RepairShopBookingOK(shopLoc, bookingRef)
        |
        out ← LocalDiagnosis(data))
    |
    in ↓ RepairShopBookingOK(shopLoc, bookingRef).
    (instance CarRental ▶ Booking ⇐
      out ← RequestCar(shopLoc).
      in ← CarAvailable(carId).
      out ↑ CarRentalOK(carId)
    |
    instance TowTruck ▶ Order ⇐
      out ← CallTruck(myLoc, shopLoc).
      in ← TruckAvailable().
      out ↑ TowTruckOK()
    |
    in ↓ TowTruckOk().in ↓ CarRentalOK(carId).out ← ServicePlanCompleted())
]

```

Figure 2: Car Break Scenario (Implementation I)

by the car’s onboard computer asking for assistance, the booking of a tow-truck, the booking of a repair shop, and the booking of a car rental.

We also illustrate in this section the encoding of the general workflow application of the Credit Request scenario of the SENSORIA Finance case study. This example involves the orchestrated interaction of three participants. A client of a bank, an employee that reviews the credit request application and his supervisor that must agree with his judgment before the credit is given. The challenge of encoding this scenario in the Finance case study is to represent the workflow as a centralized and well known (and typefull) entity while maintaining the loosely-coupled structure of the whole system.

3.1 Automotive Case Study

Briefly, the Car Break scenario of the SENSORIA Automotive case study describes the role of a car’s service planner on an onboard computer that, activated by a car failure, initiates a process to discover and automatically book a repair shop, call a tow truck to take the car from its current location to the assigned repair shop, and rent a car to be delivered to the driver either at the shop.

In order to informally introduce the calculus we first present a possible specification of the Car Break scenario in Figure 2 that assumes that all goes well and no compensations are needed. We

define a context *ServicePlanner* defining service *CarBreak*.

$$\textit{ServicePlanner} \blacktriangleright [\text{def } \textit{CarBreak} \Rightarrow \dots]$$

This service is instantiated by the running diagnosis process in the car. Two dual occurrences of a new context are then created to represent the two endpoints of the service instance. The service starts running by receiving from the initiator end-point, held by the running diagnosis process, some sensor data and the location of the car.

$$\dots \text{in} \leftarrow \textit{LocalDiagnosis}(\textit{data}). \text{in} \leftarrow \textit{GPSLoc}(\textit{myLoc}). \dots$$

It then instantiates a booking service in the repair shop context and interacts by sending message *BookRepairOperation*

$$\text{instance } \textit{RepairShop} \blacktriangleright \textit{BookRepair} \Leftarrow \text{out} \leftarrow \textit{BookRepairOperation}(). \dots$$

Upon receiving the response (message *BookingAccepted*) follows by posting the message *RepairShopBookingOk* in the enclosing context. Notice that the instantiation of the repair shop service creates an endpoint (a context) within the server endpoint of service *CarBreak*.

$$\begin{array}{l} \text{instance } \textit{RepairShop} \blacktriangleright \textit{BookRepair} \Leftarrow \dots \\ \quad \text{out} \uparrow \textit{RepairShopBookingOK}(\textit{shopLoc}, \textit{bookingRef}). \dots \\ | \\ \text{in} \downarrow \textit{RepairShopBookingOK}(\textit{shopLoc}, \textit{bookingRef}). \dots \end{array}$$

The *CarBreak* service then proceeds to concurrently instantiate both the booking service of the rental car and the call service of the tow truck. The location of the repair shop is passed both to the tow truck service and to the car rental service. The service ends when both confirmations are received.

We present a different specification of the same scenario in Figure 3, with the aim of illustrating a more loosely coupled design, and the important role of the local message passing interaction mechanism in achieving such a design. A service based computation usually consists in (1) a collection of remote partner service instances, to which functionality is to be delegated, (2) some locally implemented processes, and (3) one or more control (or orchestration) processes. The flexibility and openness of a service based design, or at least an aimed feature, results from the loose coupling between these various ingredients. For instance, an orchestration describing a “business process”, should be specified in a quite independent way of the particular subsidiary service instances used, paving the way for dynamic binding and dynamic discovery of partner service providers. In the orchestration language *WSBPEL* [10], loose coupling to external services is enforced, to some extent, by the separate declaration of “partner links” and “partner roles” in processes. In the modeling language *SRML* [11], the binding between service providers and clients is mediated by “wires”, which describe plugging constraints between otherwise hard to match interfaces. The CSCC idiom we next describe represents such specification styles in an abstract way.

To implement this loosely-coupled design variant, we separate the remote interfaces from the orchestration code. Notice the separate introduction of three service instances (*BookRepair*, *Booking*, and *Order*), while the control flow is assured by the orchestration code. All interactions between the orchestration and the local endpoints of the service instances are loosely coupled, and realized

```

ServicePlanner ▶ [
  def CarBreak ⇒
    instance RepairShop ▶ BookRepair ⇐
      in ↑ BookRepairShop(data).
      out ← BookRepairOperation().
      in ← BookingAccepted(shopLoc, bookingRef).
      (out ↑ RepairShopBookingOK(shopLoc, bookingRef)
      |
      out ← LocalDiagnosis(data))
    |
    instance CarRental ▶ Booking ⇐
      in ↑ BookCarRental(loc).
      out ← RequestCar(loc).
      in ← CarAvailable(carId).
      out ↑ CarRentalOK(carId)
    |
    instance TowTruck ▶ Order ⇐
      in ↑ CallTowTruck(depLoc, destLoc).
      out ← CallTruck(depLoc, destLoc).
      in ← TruckAvailable().
      out ↑ TowTruckOK()
    |
    in ← LocalDiagnosis(data).in ← GPSLoc(myLoc).
    (out ↓ BookRepairShop(data)
    |
    in ↓ RepairShopBookingOK(shopLoc, bookingRef).
    (out ↓ BookCarRental(shopLoc)
    |
    out ↓ GetTowTruck(myLoc, shopLoc)
    |
    in ↓ TowTruckOk().in ↓ CarRentalOK(carId).out ← ServicePlanCompleted())
  ]

```

Figure 3: Car Break Scenario (Implementation II – Loosely Coupled Design)

through messages exchanged in the context of each particular *CarBreak* service instance. The body of this service definition follows the general pattern

```
instance Partner1 ▶ Service1 ⇐ Wire1 |  
...  
instance Partnern ▶ Servicen ⇐ Wiren |  
OrchestrationProcess
```

where *OrchestrationProcess* is a process communicating with the several instances via messages, and the *Wire*_{*i*} descriptions adapt the remote endpoint functionalities (or protocols) to the particular roles performed by the instances in this local process.

3.2 Finance Case Study

The Credit Request of the SENSORIA Finance case study is a simple case of a service based system with a well defined workflow involving three client participants. This differs from the previous example in the sense that, beyond an orchestration of services, there is also the need to orchestrate clients invoking services. The "state" of the workflow must be maintained by the system to allow clients to interact in a correct way.

The evolution of a credit request application is described in [2] as follows:

Step 1: Customer starts credit request application in the credit portal and uploads his data.

The customer invokes the credit portal of his bank in the browser. After the login process (...). Additionally he must insert further data like security values. At finalization (...) these information will be checked regarding to consistency and to validation. For validating (...) a web service will be invoked. If the verification was positive the data will be uploaded to the bank, in negative case the customer has to update his information.

Step 2: Bank employee reviews credit request application of the customer

The bank employee invokes the credit portal in the browser. After the login process he navigates to the task list which also includes such credit requests. (...) he proceeds to evaluate the securities. (...) The rating is computed in the same way. Now the bank employee can provide an offer to the customer or he can reject the credit request application.

Step 3a: Bank credit supervisor reviews the offer

(...) The bank employee supervisor invokes the credit portal in the browser. (...) After the selection of the specific confirmation request he is able to accept or reject that offer. If the supervisor rejects it, the customer will get a message and the process will be finished here. If he accepts it, an offer will be sent out to the customer. (...)

Step 3b: Customer may change his data after the rejection of the credit request

If the employee has decided against an offer to the customer. Now the customer can log into the credit portal and is able to update his request information. (...)

To avoid cluttering this report with many accessory code, we start by considering a simplified setting of the case study, where we only admit one credit request at a time and thus, one client, one bank employee, and one supervisor. We then illustrate a case where many requests may be processed concurrently by using the correlation mechanism present in CSCC (local communication within a shared conversation context). For the sake of simplicity and without loss of generality we do not

```

Bank ► [def DataValidation ⇒ ... | def RateCalculator ⇒ ... | def RiskAssessment ⇒ ...]
FinanceCreditPortal ► [
  def CreditRequest ⇒ in ← Login(userId).in ← Request(data).
    out ↑ ValidateUserData(userId,data).
      in ↑ DataIsInvalid().out ← InvalidDataNotification()
      ⊕
      in ↑ DataIsValid().out ↑ RequestForEvaluation(userId,data).
        in ↑ RequestApproved(userId,data,rate).out ← RequestApproved()
        ⊕
        in ↑ RequestDenied(userId).out ← RequestRejected()

  def ReviewApplication ⇒ in ← Login(clerkId).in ↑ RequestForEvaluation(userId,data).
    out ↑ CalculateRate(data)
    |
    out ↑ AssessRisk(userId,data)
    |
    in ↑ Risk().out ↑ RequestDenied(userId)
    ⊕
    in ↑ NoRisk().in ↑ Rate(rate).out ← ShowRequest(userId,data,rate).
      in ← Pass().out ↑ RequestForApproval(userId,data,rate)
      ⊕
      in ← Deny().out ↑ RequestDenied(userId)

  def AuthorizeCredit ⇒ in ← Login(managerId).in ↑ RequestForApproval(userId,data,rate).
    out ← ShowRequestForApproval(userId,data,rate).
      in ← Accept().out ↑ RequestApproved(userId,data,rate)
      ⊕
      in ← Reject().out ↑ RequestDenied(userId)

  instance Bank ► RiskAssessment ⇐ in ↑ AssessRisk(userId,data).out ← Request(userId,data).
    in ← Safe().out ↑ NoRisk()
    ⊕
    in ← UnSafe().out ↑ Risk()

  instance Bank ► RateCalculator ⇐ in ↑ CalculateRate(data).out ← CalculateRate(data).
    in ← Rate(rate).out ↑ Rate(rate)

  instance Bank ► DataValidation ⇐ in ↑ ValidateUserData(userId,data).
    out ← DataValidationRequest(userId,data).
      in ← DataIsValid().out ↑ DataIsValid()
      ⊕
      in ← DataIsInvalid().out ↑ DataIsInvalid()
]

```

Figure 4: The Credit Request Scenario (Implementation I)

treat the updating of data by the client after a rejection and we assume that the user’s session is active during the whole workflow.

The general architecture of the example depicted in Figure 4 comprises web-services in two different portals represented here by contexts *Bank* and *FinanceCreditPortal*. Context *Bank* contains auxiliary services, external to the credit request scenario, used to perform tasks such as validation of user data, computation of interest rates or assessment of risk regarding a credit request. Context *FinanceCreditPortal* contains services and processes that implement the credit request workflow. We follow the pattern described in the previous section to achieve a loosely-coupled structure.

```

FinanceCreditPortal ▶ [
    def CreditRequest ⇒ ...
    def ReviewApplication ⇒ ...
    def AuthorizeCredit ⇒ ...
    instance Bank ▶ RiskAssessment ⇐ ...
    instance Bank ▶ RateCalculator ⇐ ...
    instance Bank ▶ DataValidation ⇐ ...
]

```

Besides containing three service instances bound to the services in the *Bank* portal with the corresponding wiring processes, context *FinanceCreditPortal* also defines the following services: *CreditRequest* which is invoked by a client when applying for a credit; *ReviewApplication* which is invoked by an employee of the bank in-charge of reviewing credit applications; and *AuthorizeCredit* which is invoked by an employee with higher responsibility in the bank to authorize (or reject) the proposal made in response to the already reviewed applications, thus ending the workflow for a credit request.

The first step of the credit request workflow starts with the client accessing the web-banking portal and instantiating service *CreditRequest*. Once instantiated, the protocol is triggered by a message from the client’s endpoint (*Login*) carrying the user identification and proceeds by receiving a message (*Request*) with the request data. The protocol described in section 3.2 continues by using service *DataValidation*, defined in context *Bank*. This step is triggered by a message exchange between the *CreditRequest* instance and the endpoint of service *DataValidation* in context *FinanceCreditPortal*. The exchange uses local communication in the enclosing context *FinanceCreditPortal*.

```

def CreditRequest ⇒ ... out ↑ ValidateUserData(userId,data)....
|
instance Bank ▶ DataValidation ⇐ in ↑ ValidateUserData(userId,data)....

```

The instance of service *DataValidation* then exchanges information with the bank’s endpoint (using the ← direction) and replies either with *DataIsValid* or *DataIsInvalid*, again using local communication in context *FinanceCreditPortal*. Depending on the input, the service instance *CreditRequest* proceeds by notifying the user of a preliminary rejection due to invalid data (message *InvalidDataNotification*), or registers the request for evaluation by a bank employee by means of a message *RequestForEvaluation* posted in the *FinanceCreditPortal* context. The second step of the workflow is triggered by the bank employee when starting a session in the *FinanceCreditPortal* portal and instantiating service *ReviewApplication*. The message regarding the pending request is received and the role of the bank

employee proceeds.

```

def CreditRequest ⇒ ... out ↑ RequestForEvaluation(userId, data). ...
|
def ReviewApplication ⇒ in ← Login(clerkId).
                        in ↑ RequestForEvaluation(userId, data). ...

```

The next step in this role is to concurrently invoke the remote computations for risk assessment and rate calculation (services *RiskAssessment* and *RateCalculator* in *Bank* portal already bound with the *FinanceCreditPortal* portal). If no risk is detected, (message *NoRisk*), the request data and rate are shown to the employee (in the browser, on the other endpoint of service instance *ReviewApplication*). Upon reply from the employee, messages *Pass* or *Deny*, the request evolves to the next step for the final approval or is rejected. This is once more achieved by posting a message in context *FinanceCreditPortal* (message *RequestForApproval* or *RequestDenied*).

The third step of the protocol follows when the credit supervisor logs into the system and the request waiting for approval is shown to him in the browser. Upon a reply sent from his endpoint, the credit is either approved (message *RequestApproved*) or denied (message *RequestDenied*). The workflow is then completed by the *CreditRequest* instance which is waiting for one of these messages to continue. In closing, the outcome of the process is sent to the client's endpoint and thus shown in the browser where the user is logged in.

This example illustrates well the expressiveness of the calculus regarding orchestration of processes and web-services. In this particular case, all components are orchestrated by a service instance, notice that the workflow associated with a credit request is entirely controlled by an instance of service *CreditRequest* by means of messages exchanged locally to context *FinanceCreditPortal*. In section 4 we show that the typing of such a process is revealing of the expected workflow. This example also shows how remote service instances act as local processes in a loosely coupled way. If, for instance, we replace the remote computation performed by service *RiskAssessment* in *Bank* portal by a local process inside *FinanceCreditPortal* portal, the remaining implementation of the credit request workflow would remain untouched. The orchestrating process shown in Figure 4 is intentionally decoupled from the two-step authorization procedure consisting of the evaluation and approval states, the *CreditRequest* protocol is independent of that particular intermediate step.

Using this programming idiom we cannot distinguish concurrent credit requests. We next implement a “conversation“ based service using contexts as correlation mechanism. This avoids mixing messages in context *FinanceCreditPortal* which could be related to different requests.

3.3 Correlation Based Communication

We rewrite the credit request example in Figure 5 taking advantage of a correlation mechanism established using the context awareness primitive.

In service *CreditRequest*, the context denoted by variable *thisConversation* is the endpoint created when the service is instantiated. That name is passed in the request for validation of user data. This means that the answer to the validation request, sent by the instance bound to service *DataValidation* in context *Bank*, can be posted directly into *thisConversation* context. Notice that the direction of the input of messages *DataIsValid* and *DataIsInvalid* in *CreditRequest* implemen-

```

FinanceCreditPortal ▶ [
  def CreditRequest ⇒ in ← Login(userId).in ← Request(data).
    here(thisConversation).
    out ↑ ValidateUserData(thisConversation,userId,data).
      in ↓ DataIsValid().out ← InvalidDataNotification()
      ⊕
      in ↓ DataIsValid().out ↑ RequestForEvaluation(thisConversation,userId,data).
        in ↓ RequestApproved(userId,data,rate).out ← RequestApproved()
        ⊕
        in ↓ RequestDenied(userId).out ← RequestRejected()

  def ReviewApplication ⇒ in ← Login(clerkId).
    in ↑ RequestForEvaluation(clientConversation,userId,data).
    here(thisConversation).
    out ↑ CalculateRate(thisConversation,data)
    |
    out ↑ AssessRisk(thisConversation,userId,data)
    |
    in ↓ Risk().clientConversation ▶ [out ↓ RequestDenied(userId)]
    ⊕
    in ↓ NoRisk().in ↓ Rate(rate).out ← ShowRequest(userId,data,rate).
      in ← Pass().out ↑ RequestForApproval(clientConversation,userId,data,rate)
      ⊕
      in ← Deny().clientConversation ▶ [out ↓ RequestDenied(userId)]

  def AuthorizeCredit ⇒ in ← Login(managerId).
    in ↑ RequestForApproval(clientConversation,userId,data,rate).
    out ← ShowRequestForApproval(userId,data,rate).
      in ← Accept().clientConversation ▶ [out ↓ RequestApproved(userId,data,rate)]
      ⊕
      in ← Reject().clientConversation ▶ [out ↓ RequestDenied(userId)]

  instance Bank ▶ RiskAssessment ⇐ in ↑ AssessRisk(conversation,userId,data)...

  instance Bank ▶ RateCalculator ⇐ in ↑ CalculateRate(conversation,data)...

  instance Bank ▶ DataValidation ⇐ in ↑ ValidateUserData(conversation,userId,data).
    out ← DataValidationRequest(userId,data).
      in ← DataIsValid().conversation ▶ [out ↓ DataIsValid()]
      ⊕
      in ← DataIsInvalid().conversation ▶ [out ↓ DataIsInvalid()]
]

```

Figure 5: The Credit Request Scenario (Correlation Based Implementation)

tation changed from \uparrow in the previous example to \downarrow in this one.

```

def CreditRequest  $\Rightarrow$  ...
  here(thisConversation).
  out  $\uparrow$  ValidateUserData(thisConversation, userId, data).
    in  $\downarrow$  DataIsInvalid()...
     $\oplus$ 
    in  $\downarrow$  DataIsValid()...
|
instance Bank  $\blacktriangleright$  DataValidation  $\Leftarrow$ 
  in  $\uparrow$  ValidateUserData(conversation, userId, data).
  ...
  ... .conversation  $\blacktriangleright$  [out  $\downarrow$  DataIsValid()]
   $\oplus$ 
  ... .conversation  $\blacktriangleright$  [out  $\downarrow$  DataIsInvalid()]

```

The same happens with message *RequestForEvaluation* which also carries the conversation handler, allowing the response to be posted back in the context of service instance of *CreditRequest* (using local communication) by both *ReviewApplication* and *AuthorizeCredit* service instances. Both syntactic contexts (in the *CreditRequest* and *ReviewApplication* for instance) are semantically correlated by the context name and thus may exchange locally posted messages.

4 Typing Services and Processes

Conversation contexts are natural candidates to be subject to interesting typing disciplines, both in terms of the message interchange patterns that may happen at its border (along the lines of behavioral or session types), and in terms of the resources they use or expose to external clients. We would thus expect types (or logical formulas) specifying various properties of interfaces, of service contracts, of endpoint session protocols, of security policies, of resource usage, and of service level agreements, to be in general assigned to context boundaries. Enforcing boundaries between subsystems is also instrumental to achieve loose coupling of systems. Along these lines, we are developing a compositional type system for CSCC, based on (spatial-) behavioral types, along the lines of [8]. An interesting challenge addressed by our type system is the verification of the delegation of conversation references according to quite strict usage disciplines, involving the context-awareness primitive. This feature allows the static verification of systems where several (more than two) partners join and leave dynamically a conversation (represented by a context) in a coordinated way. For a simple example, consider the code fragment from the Car Repair scenario of the Automotive case study (Figure 3)

```

instance RepairShop  $\blacktriangleright$  BookRepair  $\Leftarrow$ 
  in  $\uparrow$  BookRepairShop(data).
  out  $\Leftarrow$  BookRepairOperation() .
  in  $\Leftarrow$  BookingAccepted(shopLoc, bookingRef).
    (out  $\uparrow$  RepairShopBookingOK(shopLoc, bookingRef)
    |
    out  $\Leftarrow$  LocalDiagnosis(data))

```

this code fragment is assigned type

$$! \downarrow \text{BookRepairShop}(\text{DataType}); ? \downarrow \text{RepairShopBookingOk}(\text{LocType}, \text{RefType})$$

in a type environment of the form

$$\text{RepairShop} : \blacktriangleright [\begin{array}{l} \text{BookRepair}(? \leftarrow \text{BookRepairOperation}()); \\ ! \leftarrow \text{BookingAccepted}(\text{LocType}, \text{RefType}); \\ ? \leftarrow \text{LocalDiagnosis}(\text{DataType}) \end{array}]$$

Notice that the type assigned to the service instance is a behavioral type, describing the sequence of messages exchanged at the context border, while the type assigned to service *BookRepair* describes the protocol of the conversations with its clients. The type assigned to context *RepairShop* specifies the interactions that take place in the context and its published services.

Consider the code fragment from the Credit Request scenario of the Finance case study (Figure 4),

```
def CreditRequest ⇒ in ← Login(userId).in ← Request(data).
  out ↑ ValidateUserData(userId, data).
    in ↑ DataIsInvalid().out ← InvalidDataNotification()
  ⊕
  in ↑ DataIsValid().out ↑ RequestForEvaluation(userId, data).
    in ↑ RequestApproved(userId, data, rate).out ← RequestApproved()
  ⊕
  in ↑ RequestDenied(userId).out ← RequestRejected()
```

that implements the *CreditRequest* service which is assigned type

$$\text{CreditRequest}(! \leftarrow \text{Login}(\text{IdType}); ! \leftarrow \text{Request}(\text{DataType}); \\ (? \leftarrow \text{InvalidDataNotification}()) \text{ and } ? \leftarrow \text{RequestApproved}() \text{ and } ? \leftarrow \text{RequestRejected}()))$$

describing the behavior of its clients. In this case the user endpoint is expected to emit a message *Login* and then a message *Request* and then wait for the emission of either a message *InvalidDataNotification*, *RequestApproved*, or *RequestRejected*.

The instance created inside context *FinanceCreditPortal* expects the enclosing context to have the following behavior

$$\begin{array}{l} ? \downarrow \text{ValidateUserData}(\text{IdType}, \text{DataType}); \\ (! \downarrow \text{DataIsInvalid}()) \\ \text{or} \\ ! \downarrow \text{DataIsValid}(); ? \text{RequestForEvaluation}(\text{IdType}, \text{DataType}); \\ (! \text{RequestApproved}(\text{IdType}) \text{ or } ! \text{RequestRejected}(\text{IdType})) \end{array}$$

We can see here that the control over the credit request workflow is clearly specified by the type of an instance of service *CreditRequest*. The *CreditRequest* server endpoint is a process that expects the enclosing context to accept data to be validated (by another local process), and that the request is accepted for evaluation, and that, in sequence, the context is capable of giving an answer (either *RequestApproved* or *RequestRejected*).

We can also see the loose-coupling between this instance and the remaining components. The concrete implementation of *CreditRequest* service is independent on how the remaining computations are achieved. The whole evaluation and approval process can be made either by a single process or by a set of processes and external services as in this case.

5 Concluding remarks

We have briefly presented the Conversation Calculus, a model for service oriented computation. We have illustrated the calculus by using it to encode two examples taken from the SENSORIA case studies using the Conversation Calculus. The model underlying the calculus is focused on the essential aspects of distribution, process delegation, communication, context sensitiveness, and loose coupling.

The examples we have shown here illustrate well the usage of the novel primitives in the orchestration of several processes in a loosely-coupled style where service instances act as local processes both in the client and the server side. The novel communication primitives are of key importance to achieve such designs. We used, in the case of the Finance case study, a single process as the control point of a workflow. We have also shown the usefulness of the context awareness operator in isolating and correlating the communication among processes involved in a single workflow.

We have presented a glimpse of the typing of processes and contexts which describe the behavior of the processes in terms of message interchange patterns that may happen at their context borders.

References

- [1] Sensoria project. <http://www.sensoria-ist.eu/>.
- [2] Sensoria S&N Finance Case Study Definition and Requirements, Sensoria Report. <http://www.pst.ifi.lmu.de:8080/FinanceCaseStudy>, 2006.
- [3] Michel Alessandrini. Finance Case Study - Scenario descriptions. Technical Report 2007-SandN-1, S&N AG, April 2007.
- [4] M. Banci, A. Fantechi, S. Giannini, and F. Santanni. Automotive Case Study: a UML Description Scenario. Technical report, Sensoria, 2006.
- [5] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In *Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods*, Lecture Notes in Computer Science. Springer-Verlag, 2006.
- [6] Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. A service oriented process calculus with sessioning and pipelining. Technical report, 2007. Draft.
- [7] M. J. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *25 Years Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2004.

- [8] L. Caires. Spatial-Behavioral Types for Distributed Services and Resources. In U. Montanari and D. Sanella, editors, *Proceedings of the Second International Symposium on Trustworthy Global Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2006.
- [9] L. Caires, H. T. Vieira, and J. C. Seco. A Model of Service Oriented Computation. TR-DI/FCT/UNL 6/07, Universidade Nova de Lisboa, 2007.
- [10] A. Alves et al. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2006.
- [11] J. L. Fiadeiro, A. L., and L. Bocchi. A formal approach to service component architecture. In M. Bravetti, M. N., and G. Zavattaro, editors, *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer-Verlag, 2006.
- [12] Ivan Lanese, Vasco T. Vasconcelos, Francisco Martins, and Antonio Ravara. Disciplining Orchestration and Conversation in Service-Oriented Computing. In *5th IEEE International Conference on Software Engineering and Formal Methods*, 2007.
- [13] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 2006.