The Automotive Case Study in the Sensoria Core Calculi*

Luís Cruz-Filipe[1,2] lcfilipe@gmail.com
Francisco Martins[1] fmartins@di.fc.ul.pt
Vasco Vasconcelos[1] vv@di.fc.ul.pt

[1]Dept Informatics, Fac. Sciences, University of Lisbon
[2]SQIG–IT and IST, Tech. Univ. Lisbon

June 29, 2007

1 Introduction

In this note we show how two case studies from the automotive scenario can be represented in SSCC [4] in a satisfactory way. The first case study (a sight service that dinamically shows sights according to the driver's preferences) is straightforward to model. The second case study (a dinner service that allows the driver to make a reservation at a restaurant with some interaction) raises some problems with communication and typing of the resulting processes, which we then show can be solved without compromising the motivation behind SSCC.

It is also shown how the same two scenarios can be implemented in two other variants of the Sensoria Core Calculus, namely pSCC [2] and CSCC [3]. The resulting processes are then briefly compared and the differences and similarities between them discussed.

2 Sight Service (Case study 4.1.1)

Here is the scenario as described in [1].

> The driver has subscribed to the dynamic sight service offered by the car company. The vehicle's GPS coordinates are automatically sent to the dynamic

*Research supported by project FET-GC II IST-2005-16004 Sensoria.

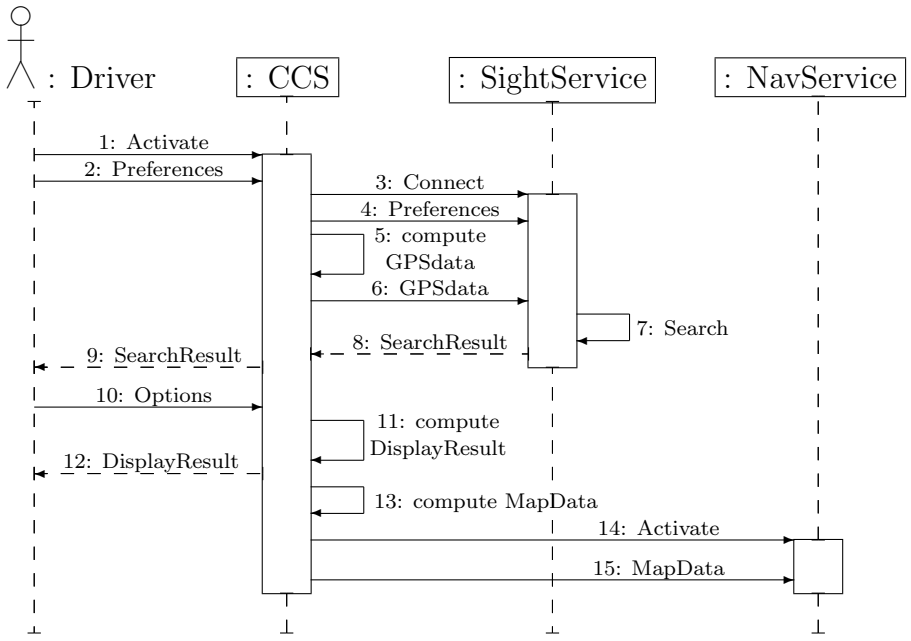

Figure 1: Sequence diagram for sight service scenario.

sight server at regular intervals, so the vehicle's location is known within a specified radius. Based on the driver's preferences that were given at the beginning of the trip, the dynamic sight server searches a sightseeing database for appropriate sights and displays them on the in-car map of the vehicle's navigation system. The driver clicks on sights he would like to visit, which results in the display of more detailed information about this specific sight (e.g. opening times, guidance to parking etc.).

As suggested in the paper cited above, there are four actors in this scenario. The driver enables the sight service (via the vehicle's communication system) and sets his preferences. The Car Communication System (CCS) manages the communication to and from the sight service. The sight service has access to a sightseeing database where it gathers information from. Finally, the vehicle's navigation system displays the results to the driver in a graphical way.

The dialogue between the actors is represented by the sequence diagram in Figure 1, which is essentially the same as in [1].

As a service, the CCS can be implemented as the following process in SSCC. The notation $\langle$action$\rangle$ stands for an internal action of the system; the numbers on the right correspond to the numbers of the actions in the sequence diagram.

```
CCS ⇒ (Preferences).                          // 1:, 2:
      (SightService ⇐ Preferences.            // 3:, 4:
                      ⟨compute GPSdata⟩.      // 5:
                      GPSdata.                // 6:
```

	(SearchResult).	// 8:
	feed SearchResult)	
$>^1$ SearchResult >		
SearchResult.		// 9:
(Options).		// 10:
⟨compute DisplayResult⟩.		// 11:
DisplayResult.		// 12:
⟨compute MapData⟩.		// 13:
(NavSystem ⇐ MapData)		// 14:, 15:

This implementation follows the UML diagram above closely. It is easy to prove that

$$\Gamma \vdash \text{CCS} : \mathsf{?Preferences.!SearchResult.?Options.!DisplayResult.end}$$

whenever Γ is such that

$$\begin{aligned} \Gamma &\vdash \text{SightService} : \mathsf{?Preferences.?GPSdata.!SearchResult.end} \\ \Gamma &\vdash \text{NavSystem} : \mathsf{?MapData.end} \end{aligned}$$

The (anonymous) stream has type $\langle \mathsf{SearchResult} \rangle$.

Observe that the types of all these services (CCS, SightService and NavSystem) correspond precisely to their part of the conversation in the sequence diagram presented above as seen by the party who invokes them. For example, the type of SightService composes the arrows labelled 4:, 6: and 8:, which is the trace of its conversation with CCS.

Also the communication along the stream does not correspond to any action in the sequence diagram. As we will discuss in more detail in the next section, it corresponds to communication between two sessions within the CCS (the session CCS–SightService and the session CCS–Driver), and not to communication between different parties.

3 Dinner Service (Case study 4.1.7)

Once again we present the scenario as given in [1].

> Paul is very hungry since he is driving without any food for 5 hours, so he activates the dinner service and enters a pizzeria as desired restaurant type and a price range between five and ten euros per meal. The navigation system displays a collection of nearby restaurants that match the preferred settings. Paul chooses the option to check for available seats in the participating local restaurants, and as a result the map displays only restaurants with available tables. Paul chooses "Tony's Pizza" and gets his reservation acknowledged. The way to the restaurant's parking lot is now displayed on the navigation system map.

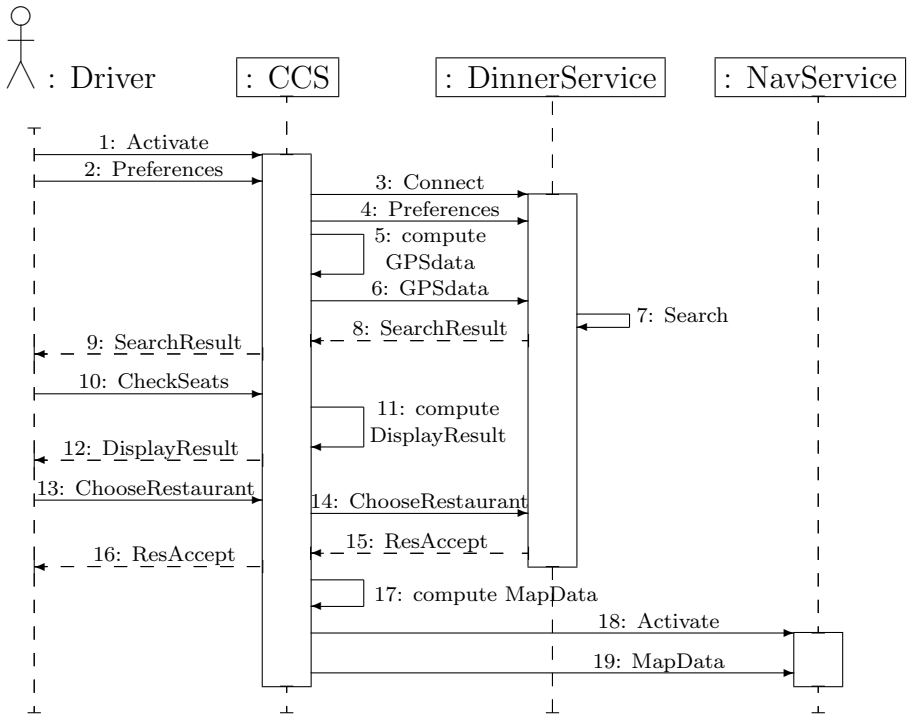

Figure 2: Sequence diagram for dinner service scenario.

Again following the suggestion in the reference paper, we identify four actors in this scenario. The driver enables the dinner service (via the vehicle's communication system) and sets his preferences. The Car Communication System (CCS) manages the communication to and from the dinner service. The dinner service has access to a database of restaurants that it can contact in order to acknowledge the reservation. Finally, the vehicle's navigation system displays the results to the driver in a graphical way.

The dialogue between the actors is represented by the sequence diagram in Figure 2, which is again essentially the same as in [1].

However, it is not so straightforward as before to implement the CCS in SSCC. The problem arises from the need to interact with the driver *after* receiving information from the dinner service, and then give feedback to the latter. We present some alternatives and discuss the drawbacks of each of them.

3.1 Implementation creating a continuation

In this approach, the dinner service, when invoked for the first time, creates a (customized) new service whose unique ID is sent back to the CCS. Afterwards, the CCS invokes *that* new service, which contains some persistent information. This solution deviates slightly from the sequence diagram above, since the session between the CCS and the dinner service is actually split into two sessions, one between the CCS and the dinner service (actions 2: to 8:), another between the CCS and the new service (actions 14: and 15:); see Figure 3.

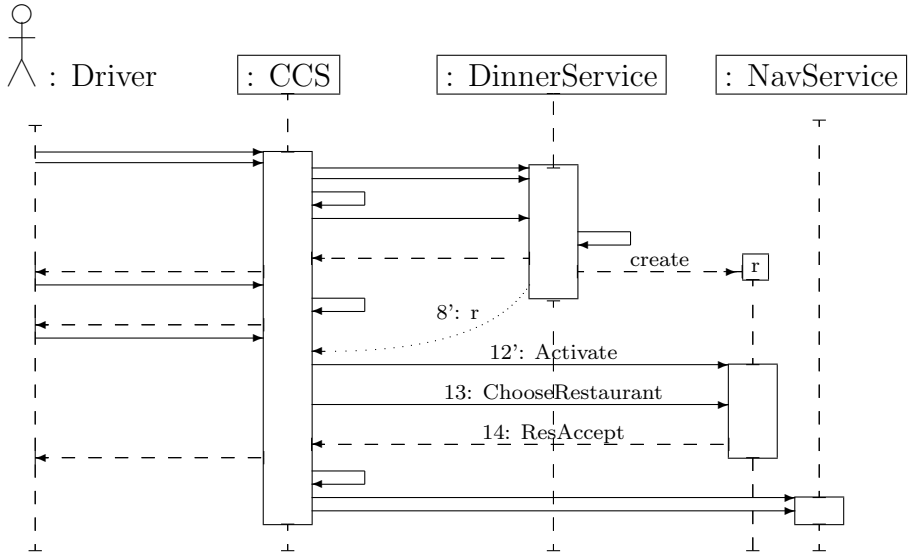

Figure 3: Sequence diagram for dinner service scenario with creation of a continuation. Only the different part is detailed.

```
CCS ⇒ (Preferences).                                    // 1:, 2:
      stream (DinnerService ⇐ Preferences.              // 3:, 4:
                              ⟨compute GPSdata⟩.        // 5:
                              GPSdata.                  // 6:
                              (SearchResult).           // 8:
                              feed SearchResult.
                              (NewService).             // 8':
                              feed NewService)
      as f in (f(SearchResult).
               SearchResult.                            // 9:
               (CheckSeats).                            // 10:
               ⟨compute DisplayResult⟩.                 // 11:
               DisplayResult.                           // 12:
               (ChooseRestaurant).                      // 13:
               f(NewService).
               (NewService ⇐ ChooseRestaurant.          // 13':, 14:
                             (ResAccept).               // 15:
                             feed ResAccept)
               >[1] ResAccept >
               ResAccept.                               // 16:
               ⟨compute MapData⟩.                       // 17:
               (NavSystem ⇐ MapData)                    // 18:,19:
               )
```

This approach is interesting because it shows how continuations can be easily passed as (specialized) services, yielding some form of persistency. However, the CCS as given will not be typable because stream f cannot be typed consistently (since it is used twice to pass two bits of information of different types).

3.2 Implementation using an auxiliary service

An alternative approach is to feed new data from the communication system into the dinner service. However, this is not immediately possible using the syntax of SSCC. In order to achieve this "backwards" communication, a new (linear) service b is created (whose name is private to the communication system). Whenever the communication system needs to send more data to the dinner service, it does so via b. Notice that one service is created for each message that needs to be sent back.

This solution follows the original sequence diagram (Figure 12) faithfully.

```
CCS ⇒ (Preferences).                                          // 1:, 2:
      (νb)(stream (DinnerService ⇐ Preferences.               // 3:, 4:
                                   ⟨compute GPSdata⟩.         // 5:
                                   GPSdata.                   // 6:
                                   (SearchResult).            // 8:
                                   feed SearchResult.
                                   b ⇓ (ChooseRestaurant).
                                   ChooseRestaurant.          // 14:
                                   (ResAccept).               // 15:
                                   feed ResAccept)
      as f in  (f(SearchResult).
                SearchResult.                                 // 9:
                (CheckSeats).                                 // 10:
                ⟨compute DisplayResult⟩.                      // 11:
                DisplayResult.                                // 12:
                (ChooseRestaurant).                           // 13:
                b ⇑ ChooseRestaurant.
                f(ResAccept).
                ResAccept.                                    // 16:
                ⟨compute MapData⟩.                            // 17:
                (NavSystem ⇐ MapData)                         // 18:, 19:
      ))
```

where:

- b ⇑ v.P stands for (b ⇐ v)>^{0}>P,
  which in turn unfolds to **stream** (b ⇐ v) **as** f **in** P.

- b ⇓ (x).P stands for (b ⇒ (z)**feed** z)>^{1}x>P,
  which in turn unfolds to **stream** (b ⇒ (z)**feed** z) **as** f **in** (f(x)P).

Observe that both b ⇑ v.P and b ⇓ (x).P always have the same type as P (but in the second case P might not be typable without the extra information of the type of x); furthermore, v:T ⊢ b:?T, hence the use of these abbreviations always fits well with the type system. Indeed, the only typing problem is the one above – namely stream f can not be adequately given a type.

3.3 Implemenation with communication via services

The constructions b ⇑ v.P and b ⇓ (x).P may also be used to communicate in the same direction as the stream, hereby avoiding the typing problems both previous solutions suffered from. Thus, we arrive at a third proposal for modelling the dinner service scenario within SSCC, which again follows the proposed sequence diagram faithfully.

```
CCS ⇒ (Preferences).                                        // 1:, 2:
      (νa1,a2,b)(
            (DinnerService ⇐ Preferences.                   // 3:, 4:
                             ⟨compute GPSdata⟩.             // 5:
                             GPSdata.                       // 6:
                             (SearchResult).                // 8:
                             a1 ⇑ SearchResult.
                             b ⇓ (ChooseRestaurant).
                             ChooseRestaurant.              // 14:
                             (ResAccept).                   // 15:
                             a2 ⇑ ResAccept)
            |
            (a1 ⇓ (SearchResult).
             SearchResult.                                  // 9:
             (CheckSeats).                                  // 10:
             ⟨compute DisplayResult⟩.                       // 11:
             DisplayResult.                                 // 12:
             (ChooseRestaurant).                            // 13:
             b ⇑ ChooseRestaurant.
             a2 ⇓ (ResAccept).
             ResAccept.                                     // 16:
             ⟨compute MapData⟩.                             // 17:
             (NavSystem ⇐ MapData)                          // 18:, 19:
            )
      )
```

With this solution, it can be easily verified that

$$\Gamma \vdash \text{CCS} : \mathsf{?Preferences.!SearchResult.?CheckSeats.!DisplayResult.?ChooseRestaurant.!ResAccept.end}$$

in any context Γ such that:

$$\Gamma \vdash \text{DinnerService} : \mathsf{?Preferences.?GPSdata.!SearchResult.?ChooseRestaurant.!ResAccept.end}$$
$$\Gamma \vdash \text{NavSystem} : \mathsf{?MapData.end.}$$

The types of services a_1, a_2 and b are exactly as in the previous example. Again, the types of the services reflect their communication in the sequence diagram with the party who invokes them: the type of CCS is the concatenation of actions 2:, 9:, 10:, 12:, 13: and 16:; the type of DinnerService is the concatenation of actions 4:, 6: and 8:; and the type of NavSystem is simply action 19:.

Observe that all communication via a_1, a_2 and b is hidden in the type of the services; thus, the type also hides all internal communication, which is not represented in the sequence diagram.

4 Other variants of SCC

In this section we discuss implementations of the same scenarios within other variants of SCC.

4.1 pSCC

These two scenarios can also be modelled within pSCC [2]. Since pSCC uses $\langle v \rangle$ for output of a value v, we use instead the notation [A] to refer to an internal action A of a process.

The sight service scenario is a straightforward adaptation of the implementation in SSCC, the major differences being notational.

```
CCS. (?Preferences).                          // 1:, 2:
     (SightService. ⟨Preferences⟩.            // 3:, 4:
                    [compute GPSdata].        // 5:
                    ⟨GPSdata⟩.                // 6:
                    (?SearchResult).          // 8:
                    ⟨SearchResult⟩↑)
     >
     (?SearchResult).
     ⟨SearchResult⟩.                          // 9:
     (?Options).                              // 10:
     [compute DisplayResult].                 // 11:
```

```
⟨DisplayResult⟩.                    // 12:
[compute MapData].                  // 13:
NavSystem.⟨MapData⟩                 // 14:, 15:
```

As for the dinner service scenario, the implementation is again very similar to that within SSCC. Besides the differences in notation, the auxiliary services a ⇑ v and a ⇓ (x) are also defined differently:

- a ⇑ v.P stands for $\bar{a}.\langle v\rangle \mid P$ (asynchronous output);
- a ⇓ (x).P stands for $a.(?x)\langle x\rangle^{\uparrow} > (?x).P$ (input).

```
CCS. (?Preferences).                                    // 1:, 2:
     (νa1,a2,b)(
          (DinnerService. ⟨Preferences⟩.                // 3:, 4:
                          [compute GPSdata].            // 5:
                          ⟨GPSdata⟩.                    // 6:
                          (?SearchResult).              // 8:
                          a1 ⇑ SearchResult.
                          b ⇓ (ChooseRestaurant).
                          ⟨ChooseRestaurant⟩.           // 14:
                          (?ResAccept).                 // 15:
                          a2 ⇑ ResAccept)
          |
          (a1 ⇓ (SearchResult).
           ⟨SearchResult⟩.                              // 9:
           (?CheckSeats).                               // 10:
           [compute DisplayResult].                     // 11:
           ⟨DisplayResult⟩.                             // 12:
           (?ChooseRestaurant).                         // 13:
           b ⇑ ChooseRestaurant.
           a2 ⇓ (ResAccept).
           ⟨ResAccept⟩.                                 // 16:
           [compute MapData].                           // 17:
           NavSystem.⟨MapData⟩                          // 18:, 19:
          )
     )
```

4.2 CSCC

Another calculus within which these two scenarios can also be modelled is CSCC [3]. Again this happens in a very similar way to the approach of SSCC. However, the dinner service case study is somewhat simpler due to the more powerful message-sending mechanism.

The sight service becomes the following process.

```
def CCS ⇒ (in ← Preferences(p).                          // 1:, 2:
           instance Server ▷SightService ⇐               // 3:
                        out ← Preferences(p) .           // 4:
                        ⟨compute g⟩.                     // 5:
                        out ← GPSdata(g) .               // 6:
                        in ← SearchResult(s).            // 8:
                        out ↑ SearchResult(s)
           )
           |
           (in ↓ SearchResult(s).
            out ← SearchResult(s) .                      // 9:
            in ← Options(o).                             // 10:
            ⟨compute d⟩.                                 // 11:
            out ← DisplayResult(d) .                     // 12:
            ⟨compute m⟩.                                 // 13:
            instance Server ▷NavSystem ⇐                 // 14:
                        out ← MapData(m)                 // 15:
)
```

The dinner service thus becomes the following process.

```
def CCS ⇒ (in ← Preferences(p).                          // 1:, 2:
           instance Server ▷DinnerService ⇐              // 3:
                        out ← Preferences(p) .           // 4:
                        ⟨compute g⟩.                     // 5:
                        out ← GPSdata(g) .               // 6:
                        in ← SearchResult(s).            // 8:
                        out ↑ SearchResult(s) .
                        in ↑ ChooseRestaurant(c).
                        out ← ChooseRestaurant(c) .      // 14:
                        in ← ResAccept(r).               // 15:
                        out ↑ ResAccept(r)
           )
           |
           (in ↓ SearchResult(s).
            out ← SearchResult(s) .                      // 9:
            in ← CheckSeats(c).                          // 10:
            ⟨compute d⟩.                                 // 11:
            out ← DisplayResult(d) .                     // 12:
```

```
    in ← ChooseRestaurant(c).                    // 13:
    out ↓ ChooseRestaurant(c) .
    in ↓ ResAccept(r).
    out ← ResAccept(r) .                         // 16:
    ⟨compute m⟩.                                 // 17:
    instance Server ▷NavSystem ⇐                 // 18:
                    out ← MapData(m)             // 19:
)
```

5 Conclusions

The second scenario exposes the issue with streams: they render communication assymetric, since a running instance of a service is able to feed information into the process that invoked it, but the latter process has no way to interact back (directly) with that instance of the service[1].

However, in this scenario, the communication system has to syncronize two sessions running concurrently (that with the dinner service and another one with the driver) and information has to run back and forth between them – which is *a priori* not possible due to who invoked whom.

A simple way to work around this problem is using continuations, as done in the approach in Subsection 3.1. Whenever a session needs extra information from the context to proceed, it saves its state in a new service, feeds the name of that service into the context and dies. Later on, the context can invoke the continuation of that session with any extra information that became available in the meantime. Another possibility, explored in Subsection 3.2, is to use ephemerous services to communicate in the "wrong" direction.

Figure 4 depicts the sequence of messages exchanged between the intervenients (forgetting activation commands). Observe that CCS is managing two sessions; the curved arrows denote information that must be transmitted from one into the other (which CCS *should* be able to do). The arrows going left can be implemented by **feed**ing into the appropriate stream; the arrow going right must be dealt with using one of the two mechanisms detailed earlier.

This diagram also explains why typing fails for both alternative solutions. Except for their direction, there is no essential distinction the arrows connecting both sessions CCS is managing. However, streams are specially tailored to capture the right-to-left arrows, while the left-to-right have to be implemented by service invocation. But the purpose of stream communication is quite different: allowing a process invoking several concurrent services to receive answers from them all without regard to order.

Thus the search for a symmetric manner of modelling all the curled arrows in Figure 4 is motivated not only because of technical problems, but also by desire for coherence. The third solution proposed to the dinner problem satisfies both requirements: CCS manages

[1]This is actually the intended motivation behind streams, since when a service is invoked it should go on running on its own.

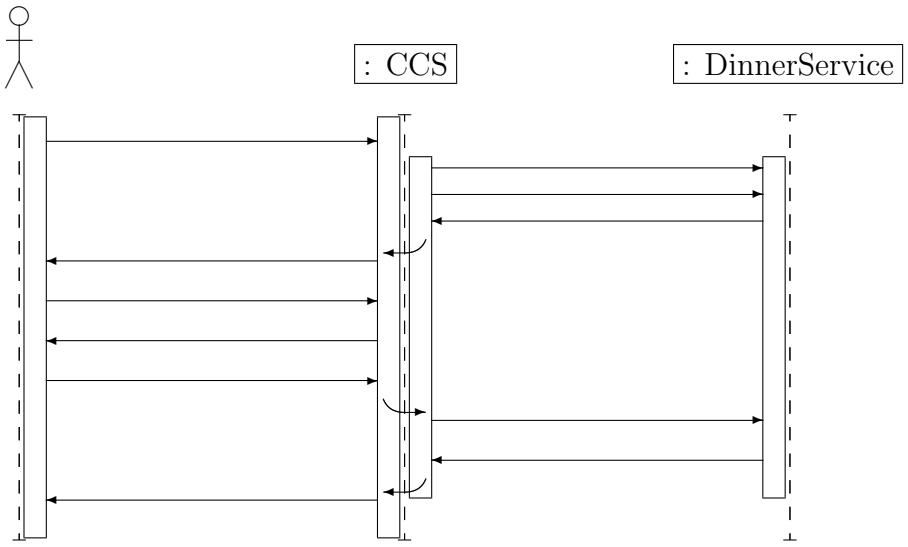

Figure 4: Two services running in parallel. The curved arrows indicate where information needs to be transmitted between sessions.

communication between the sessions it is involved in in a uniform way, and the resulting process is typable.

It is interesting to notice that modelling this scenario in pSCC [2] poses exactly the same questions. The final solution, presented for both calculi, has a slightly dissatisfactory taste in that the characteristic construction of either calculus (streams, for SSCC, or pipelines, for pSCC) is not used at the top level, since the service invocation may run in parallel with the top level process (even must, in the case of pSCC) from the moment when communication is completely managed by auxiliary services. Of course, these will themselves use either streams or pipelining to communicate with their parent, but this is not directly visible.

The same scenarios are easier to implement in CSCC, since the message-passing mechanisms of that calculus is more directly suitable for the kind of communication needed.

References

[1] M. Banci, A. Fantechi, S. Giannini, and F. Santanni. Automotive case study: a UML description of scenarios. Technical report, Sensoria, December 2006.

[2] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. pSCC: Revising SCC.

[3] L. Caires and H.T. Vieira. Note on a model of service oriented computation.

[4] I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. To appear in Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods.