Checking choreography conformance using SLMC

Luís Caires, David Tavares Sousa, Hugo Torres Vieira

October 16, 2007

Abstract

We illustrate with a simple example how the Spatial Logic Model Checker can be used to check choreography conformance properties.

1 Introduction

Spatial Logics [4, 3, 5, 2] have proven their adequacy to express properties of distributed systems. Distributed processes run, communicate, access resources, and so on, in a distributed structured space, which makes them natural candidates for the specification of properties using spatial logics that are specially equipped with structural observations.

One example of a distributed model is service oriented computing, a paradigm of gaining importance given its widespread application. The purpose of a service relationship is to allow the incorporation of certain activities in a given system, without having to engage *local* resources and capabilities to support or implement such activities. By delegating activities to an external service provider, that will perform them using its own *remote* resources and capabilities, a computing system may concentrate on those tasks for which it may autonomously provide convenient solutions. Thus, the notion of service makes particular sense when the service provider and the service client are separate entities, with access to separate sets of resources and capabilities. This understanding of the service relationship between provider and client assumes an underlying distributed computational model, where client and server are located at least in distinct (operating system) processes, more frequently in distinct sites of a network.

As services collaborate to reach a determined goal, an important aspect subject to verification is that the protocols of interaction are designed correctly, *i.e.*, in accordance to an interaction specification. The idea is that one can statically set the rules of interaction in a choreography plan and check that interacting partners conform to such a specification. Furthermore it might be interesting to verify that such interactions do not cause systems to get stuck, and other liveness and also safety properties.

We illustrate here with a familiar example a natural specification of a choreography conformance property using spatial logics, and we demonstrate the verification of such property using the SLMC tool [9,

1], that uses π-calculus [8] to specify systems and the spatial logics of [2] to describe the properties.

2 Buyer Seller Shipper revisited

We consider the Buyer, Seller, Shipper pattern. Our system will be built out of three distinct parts, each one representing a separate entity: either the Buyer, or the Seller or finally the Shipper. For the sake of simplicity each one of these participants will be identified by a name. The top level system description will then look like:

```
defproc System = Buyer(bu) | Seller(se) | Shipper(sh);
```

The names given as arguments will function as the identifiers. The general idea is that the Buyer is going to try to buy something off the Seller, deciding upon the price requested, being that in a successful purchase the Seller interacts with the Shipper to obtain the delivery details, which are then sent back to the Buyer. We turn to the definition of the Buyer:

```
defproc Buyer(buyer) =
    new session in (
        quoteCh!(session,buyer).
        session?(quote).
        select {
            session!(accept).
                session?(deliverydetails).
                Buyer(buyer);
            session!(reject).
                Buyer(buyer)
        });
```

The behaviour of the Buyer can be described as: first tries to communicate with the Seller using channel `quoteCh`, and establishes a session by emitting the private name `session`. The Buyer then expects a message indicating the price on `session`, depending on which the decision to purchase is made. If the Buyer decides to purchase then it will inform the Seller, sending `accept` on `session`, and expects the Seller to inform on the delivery details. At the end of each run, both in a successful purchase or not, the Buyer returns to the initial state.

We now turn to the description of the Seller:

```
defproc Seller(seller) =
    quoteCh?(session,bu).
    session!(price).
    session?(choice).
    select {
        [choice=accept].
            new t in (
                deliveryCh!(t,seller).
```

```
                t?(deliverydetails,sh).
                session!(deliverydetails).
                Seller(seller));
        [choice=reject].
            Seller(seller)
    };
```

The Seller will start by following the interaction described for the Buyer: first expects a request for a price and replies informing on the price on a received name `session`, afterwards expecting a decision on the purchase. In the case of acceptance by the Buyer, signalled by the matching of the received name and `accept`, the Seller will place a request to the Shipper to obtain the delivery details. After receiving the delivery details from the Shipper, the Seller forwards them to the Buyer and returns to the initial state. In the case of rejection of the purchase the Seller directly returns to the initial state.

Finally we have the Shipper description, that expects requests for delivery details, post replies to these requests and returns to the initial state:

```
defproc Shipper(shipper) =
    deliveryCh?(t,se).
    t!(deliverydetails,shipper).
    Shipper(shipper);
```

3 Choreography

We start with the specification of a message link or connection, *i.e.*, two parts of a system are linked or connected if they hold dual capabilities of a channel. We also specify and identify the source or sender and the destination or receiver:

```
defprop sArrow(message,src,dst) =
    inside((1 and @src and <message!>true)
           | (1 and @dst and <message?>true ) | true);
```

The property `sArrow` here specified states that opening up all restricted names, that can bind parts of the system, the system can be broken down into three parts `|`, two that are indivisible `1`, and another that can be anything `true`. The two indivisible parts hold occurrences of the names `src` and `dst` that identify them as sender and receiver, respectively, being that the sender must exhibit an output `!` on `message` while the receiver must exhibit the input `?`, being the continuations specified as any possible state.

Using this predicate `sArrow` we can then be more specific of the links we intend to describe:

```
defprop sBuyer2Seller(message) = sArrow(message,bu,se);
defprop sSeller2Buyer(message) = sArrow(message,se,bu);
defprop sShipper2Seller(message) = sArrow(message,sh,se);
defprop sSeller2Shipper(message) = sArrow(message,se,sh);
```

We can express that the Buyer is able to send a message to the Seller that is in turn able to receive that message `sBuyer2Seller`. We can express the same property reverting the communication roles `sSeller2Buyer`. We can express that the Shipper and Seller can be connected both ways: either `sShipper2Seller` or `sSeller2Shipper`.

Let us now describe the initial interaction between Buyer and Seller:

```
defprop buyerSellerInteraction(session, A) =
    sBuyer2Seller(quoteCh) and
    [] ( sSeller2Buyer(session) and
         [](  sBuyer2Seller(session) and
             [][]( A )));
```

We have that initially Buyer and Seller can communicate on message `quoteCh`, corresponding to the price request, and also any possible evolution of the system `[ ]` will lead to a state where Seller is connected to Buyer on the parametered name `session`, corresponding to the reply to the price request. After any possible evolution Buyer is connected to Seller again on the channel given by `session`, to inform on the decision taken. After two steps, involving the decision procedure, the system is in a state specified by the parameter `A`.

We turn to the description of the interaction between Seller and Shipper:

```
defprop sellerShipperInteraction(A) =
    sSeller2Shipper(deliveryCh) and
    [] hidden t.
        ( sShipper2Seller(t) and
          [] ( A ));
```

Property `sellerShipperInteraction` specifies that at starting point, Seller is connected to Shipper in channel `deliveryCh`, corresponding to the request for delivery details. Also, after one step, there is a restricted name `t` under which Shipper is connected to Seller, where the delivery details will be passed along. Afterwards Seller is connected to buyer, corresponding to the forwarding of the delivery details. Finally, after one step, the system is in a state specified by the parameter `A`.

We are at this point able to devise a specification that expresses the overall behaviour of a run of this protocol between Buyer, Seller and Shipper:

```
defprop sGlobalDescription =
    maxfix X.(
        hidden session.
        buyerSellerInteraction(session,
            X
            or
            sellerShipperInteraction(
                sSeller2Buyer(session) and []X)));
```

So, recalling that the interaction properties require an argument that specifies the final state of the interaction we have that: We start

by revealing the restricted name session and then we proceed to specify that the interaction between Buyer and Seller takes place using that name. Afterwards the system either returns to the initial state or the interaction between Seller and Shipper takes place. If the interaction between Seller and Shipper occurs, at the end of it the Seller is once again connected to the Buyer, in channel `session`, after which any possible evolution leads back to the initial state.

3.1 Checking choreography conformance

After loading the example on the SLMC tool one can write:

```
check System |= sGlobalDescription;
```

to which the tool will reply with:

```
* Process System satisfies the formula sGlobalDescription *
```

thus giving our intended assertion.

3.2 Checking other properties

Other interesting properties include liveness, which can be writen as:

```
defprop aLive = always (<tau>true);
```

Property `aLive` specifies that a system in every possible state can always perform an internal action `tau`, hence the system never gets stuck.

Also we may consider of interest to check safety properties like, for instance, the existence of races. We start by specifying that a writer is an indivisible process that can perform an output on a given channel.

```
defprop write(x) = (1 and < x! >true);
```

We characterize a reader similarly, being the channel capability the input:

```
defprop read(x) = (1 and < x? >true);
```

We can now express that a configuration that holds a race is one where there exists a channel where two distinct components are trying to write to and there is one other trying to read from:

```
defprop hasRace =
    inside (exists x.( write(x) | write(x) | read(x) | true));
```

Finally we can state that a race free system is a system that in every possible configuration holds no races.

```
defprop raceFree = always (not hasRace);
```

Once again we load the example in the SLMC tool:

```
check System |= raceFree;
```

to which the tool will reply with:

```
* Process System satisfies the formula raceFree *
```

thus giving our intended assertion.

4 Concluding remarks

We have shown by means of a simple example how one can use the SLMC tool to check for choreography conformance. We presented a simple specification of the familiar buyer shipper seller pattern in π-calculus, and showed a natural description of the choreography of interaction using spatial logics.

An interesting follow up to this illuminating exercise would be to use an analogous approach considering service-world languages, such as `WS-BPEL` [6] to provide system specification and `WS-CDL` [7] to give the choreography plan.

References

[1] Spatial logic model checker. http://www-ctp.di.fct.unl.pt/SLMC/.

[2] L. Caires. Behavioral and Spatial Properties in a Logic for the Pi-Calculus. In Igor Walukiwicz, editor, *Proc. of Foundations of Software Science and Computation Structures'2004*, number 2987 in Lecture Notes in Computer Science. Springer Verlag, 2004.

[3] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). In *CONCUR 2002 (13th International Conference)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[4] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.

[5] L. Cardelli and A. D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In *27th ACM Symp. on Principles of Programming Languages*, pages 365–377. ACM, 2000.

[6] A. Alves et al. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2006.

[7] N. Kavantzas et al. Web Services Choreography Description Language Version 1.0. Technical report, W3C Working Draft, 2005.

[8] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[9] H. Vieira and L. Caires. Spatial logic model checker user's guide. Technical Report TR-DI/FCT/UNL-03/2004, DI/FCT Universidade Nova de Lisboa, 2004.