ELSEVIER

Science of Computer Programming 29 (1997) 259–278

Science of Computer Programming

Detecting feature interactions with CÆSAR/ALDÉBARAN[1]

Henri Korver *

CWI, P.O. Box 94079, 1090 GB Amsterdam, Netherlands

Abstract

Bouma and Zuidweg (Dutch PTT) formalised a simple example of feature interaction between two telephone services in LOTOS. The interaction takes place between the Abbreviated Dialling and Originating Call Screening service in the IN CS-1 Global Functional Plane. This paper reports on the results that were gained by analysing the example in CÆSAR/ALDÉBARAN, which is an advanced LOTOS verification toolbox. The results show that even for very small examples, verification goes beyond simulation and testing. © 1997 Elsevier Science B.V.

Keywords: ALDÉBARAN; CAESAR; Feature interaction; Intelligent networks (IN), LOTOS

1. Introduction

Over the last ten years, telecommunication industry has been engaged in increasing the number of services that are supplied by the telephone networks. For instance, in many countries, new services like *Call Forwarding* and *Call Waiting* are being added to the conventional telephone service, and in fact, a large and rapid development of such and more advanced services has been started. However, service engineers stress that unwanted interactions cause difficulties in controlling the proper functioning of services. This problem, where unwanted interactions interfere with the desired behaviour of services, is called *feature interaction.*

In this paper, it is demonstrated by a small example how formal methods and verification tools can be used for detecting feature interactions. In particular, an example of feature interaction by Bouma and Zuidweg [1] is verified in CÆSAR/ALDÉBARAN [2]. This work extends the results of [1] where the example is only tested.

The example centres around a LOTOS specification of two telephone services, Abbreviated Dialling (ABD) and Originating Call Screening (OCS). ABD allows a user to use abbreviated numbers, which will be expanded by the ABD service into

* E-mail: henri@cwi.nl.

[1] This work was supported by the European Communities under RACE project No. 2076, Broadband Object Oriented Service Technology (BOOST).

PII S0167-6423(96)00037-8

network addresses. OCS offers the possibility to forbid call set-up to numbers which are included in a screening list, e.g. your mother-in-law. In principle, these two services can exhibit unwanted interaction: if a dialled number is expanded too late, it might not be recognised as belonging to the list of numbers to be screened.

In [1], a desired property (feature) of a service is represented by a formula of a modal/temporal logic. In this approach, feature interactions can be detected by checking whether the conjunction of individual service features still holds. For example, suppose that the services S_1 and S_2 satisfy the properties ϕ_1 and ϕ_2, respectively. When both services run in parallel, the property $\phi_1 \wedge \phi_2$ (the conjunction of ϕ_1 and ϕ_2) should hold, else there has been some (unwanted) interaction between the two services.

In the example considered in this paper, modal/temporal formulas are only used for recording feature interactions in a formal way. Unfortunately, the formulas used in the example contain datatype definitions, which currently cannot be checked automatically. (Although there is sufficient technology, such tools have not yet been implemented.)

To cope with this complication, in [1] an alternative route was stipulated by using *testers*. A tester is a simple LOTOS specification, which encodes a property to be checked, and runs in parallel with the original specification. As soon as the property is violated, the tester generates a special error transition. Bouma and Zuidweg used this technique in LITE[2] for checking negative properties about services in their example. However, they claimed that LITE was not powerful enough for proving positive properties (correctness). This was mainly due to the fact that the verification tools in LITE cannot yet handle full LOTOS.

In this paper, a simple extension of the testing method is presented which also allows for proving positive properties in CÆSAR/ALDÉBARAN. It works as follows. Hide all the gates except the error gate in the parallel composition of the tester and the original specification. If the generated graph (obtained by using CÆSAR) of the resulting process contains an error transition, then the property is violated; otherwise the property is satisfied. (ALDÉBARAN was merely used for reducing the size of generated graphs with respect to Milner's observation equivalence.)

By using CÆSAR/ALDÉBARAN, I was able to verify all the service features (and interactions between them) that are stated in [1]. Moreover, during the checking a bug was found in the GPF model of [1]. This was due to a subtlety in one of the initial values of the main LOTOS specification. To repair the error, the implementation of the ABD service had to be changed. This is a typical illustration that even in this very simple example, one can benefit from formal methods as set-up in [1]; in particular, when automatic verification tools are used as is shown here.

The paper is organised as follows. In the next section, the IN CS-1 GFP model as given in [1] is quickly reviewed. In Section 3, the example of feature interaction between the ABD and OCS service is presented. Then the example is analysed with the CÆSAR/ALDÉBARAN verification toolbox in Section 4. Conclusions drawn from the analysis are discussed in Section 5.

[2] LITE has been developed within the ESPRIT project 3204 (LOTOSPHERE).

2. The GFP model in LOTOS

In [1], a LOTOS specification of the IN CS-1 Global Functional Plane (GFP), following the CCITT recommendations as close as possible, was given. The interested reader can find the LOTOS code in the Appendix.

One of the objectives was that formal specifications allow for computer-assisted analysis of feature interactions. In this section, the example will be explained informally. For a more thorough treatment, one is referred to [1].

2.1. Datatype definitions

The GFP model has abstract datatypes definitions for the following data:

Network addresses abstractly identify points in the network. In our example, we actually have three addresses: `a1`, `a2` and `null`. In fact, `null` is a special case: it is the address which is not associated with any point in the network.

Dialled numbers represent the numbers that can be dialled on a terminal. In the example, there are four numbers that can be dialled: `d1`, `d2`, `wrong_number` and `abd2` (the abbreviation of `d2`). Furthermore, there is a function `translate` which expands abbreviated numbers. The following definition is specific for the example:

```
translate(abd2)= d2 ;
not(dn eq abd2) => translate(dn) = dn;
```

where `dn` is a variable ranging over dialled numbers. There is also a function `get_address` which computes the destination address of a dialled number. In our particular example, we have

```
get_address(d1) = a1;
get_address(d2) = a2;
get_address(wrong_number) = null;
get_address(abd2) = null;
```

Lastly, there is a function `screen` which is used by the OCS service for screening telephone numbers. The following definition is specific for the example:

```
screen(d1) = no_match;
screen(d2) = match;
screen(abd2) = no_match;
screen(wrong_number) = error;
```

Call reference provides a unique identifier for each basic call process. Because we shall only consider one incoming telephone call, the call reference (which is represented by a natural number) will always be zero.

Call instance data is the record carrying the information associated with a Basic Call Process. It contains a call reference (which is here always set to zero), a calling line identity (which is not used here), a dialled number and a destination number.

SIB end is the type covering all possible termination values for SIBs.

Detection points are used for modelling the (dis)arming of triggers which invoke the telephone services.

2.2. Processes

The LOTOS model is built around two gates: `poi` (point of invocation) and `por` (point of return). Values of the `detection_point` type are used to identify particular points in the Basic Call Process where telephone services are invoked. All interactions in the LOTOS model are of the form:

```
poi <detection point> <call instance date>
por <detection point> <call instance date>
```

For example,

```
poi! address_collected !cid(call_ref, cli,dialled_nr, dest_nr)
por! continue_as_is ?new_cid: call_instance_data
```

The following processes are distinguished in the LOTOS model:

SIB processes: Each Service Independent Building Block (SIB) is represented by a LOTOS process that performs a particular function, such as `Screen` and `Translate`. These functions are used for building services. A service can be composed by the usual LOTOS operators like parallel composition, enabling, disabling and choice.

Basic call process (BCP): This process describes the interactions (`poi` and `por`) in a telephone network. An example of this is given in Fig. 1.

Trigger detection: This process determines whether a trigger is armed and calls the 'Invoke service' process if appropriate.

Invoke service: This process determines which service script to call if a particular trigger point is detected.

Service logic processes: A service is modelled by a LOTOS process that calls one or more SIBs.

2.3. Reformulating the specification

For being able to analyse the LOTOS specification in CÆSAR several parts had to be reformulated. For the interested reader some modifications are mentioned here:

- Some datatypes, e.g. Dialled numbers, had to be polished, as they were not accepted by the CAESAR compiler. In polishing the datatypes, the function `mk_dialled_number` was removed and the constants `d1`, `d2` were added. Moreover, I changed the name of the constant `ab` (the abbreviated number of `d2`) in `abd2` which is in my opinion a more appealing name. At last, for coherent notation, I redefined the equality function for dialled numbers via the equality between natural numbers, as was already done for the other types.

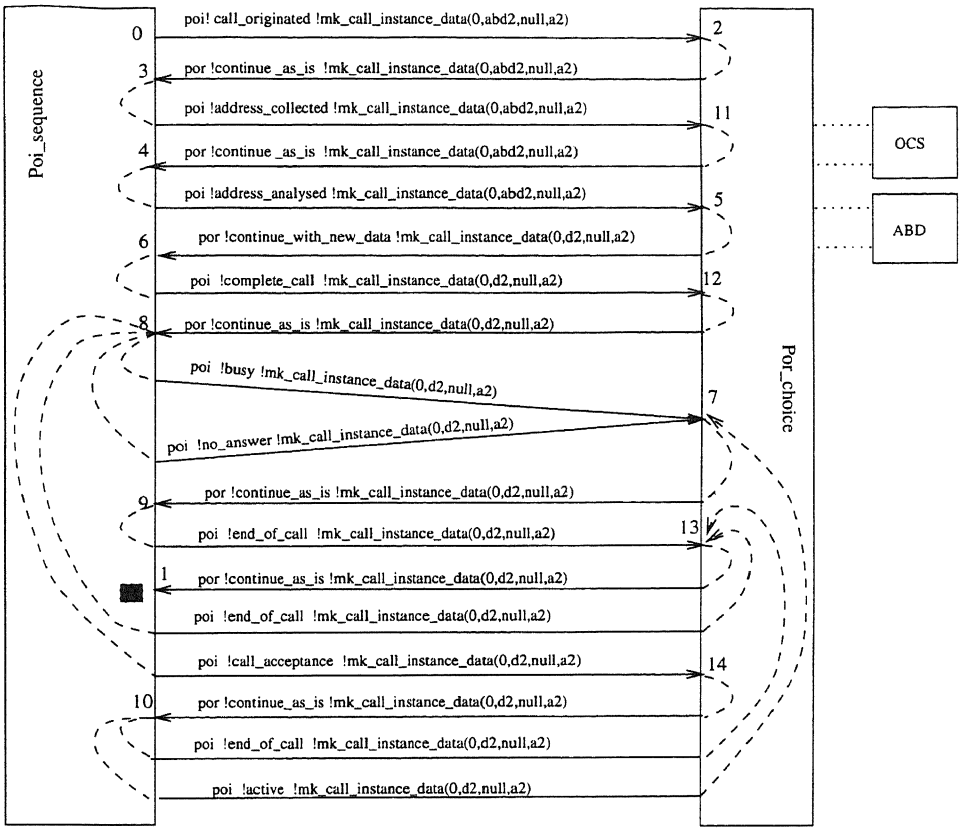

Fig. 1. IN CS-1 global functional plane.

- The function `update_destination_number` was added for revising the ABD service.
- We changed the BCP process because CÆSAR does not allow recursive process instantiation on the left (and also the right) side of a parallel operator. In the original specification of the BCP process such infinitely growing recursion is used for modelling arbitrary many incoming phone calls from the external world. We remedied this by changing the specification in such way that only one particular phone call can be considered at the time.

The code that was actually analysed in CÆSAR/ALDÉBARAN can be found in the appendix.

3. The example: ABD and OCS

The example of Bouma and Zuidweg consists of two services: Abbreviated Dialling (ABD) and Originating Call Screening (OCS). ABD allows a user to use abbreviated numbers which then will be expanded by the ABD service into network addresses.

OCS gives the possibility to forbid call set-up to numbers included in a screening list. In principle, these two services can exhibit unwanted interaction: if a dialled number is expanded too late, it might not be screened.

A desired property of the ABD service could be that the dialled number must have been translated before the call is completed, i.e. the connection is established. This feature is formalised by the following ACTL formula:

ϕ_1 : **AF**{poi!complete_call!cid}
(get_destination_number(cid) eq
get_address(translate(get_dialled_number(cid))))

This is a reformulation of a formula given in [1]. Note that this formula has not been checked directly. As far as I know currently no tools exist for checking formulas that are parametrised by data. However, it can be checked by encoding the formula into a tester as is described in the next section. Here logic formulas are only used for recording service features (and their interactions) in an elegant way.

For OCS a similar formula can be written:

ϕ_2 : **AF**{poi!complete_call!cid}
(screen(get_dialled_number(cid)) eq no_match)

The specification of these services is straightforward. ABD is realised by definition of a LOTOS process ABD that invokes a SIB called `Translate`. This SIB in its turn consults a function `translate:dialled_number->dialled_number`. ABD is instantiated through update of the function `trigger_ABD:trigger_points,call_instance_data->Bool`.

The OCS service is defined in a similar manner: define a process OCS invoking an SIB taking care of the actual screening. The screening is realised by a function `screen:dialled_number->SIB_end`, which has the output values `match` and `no_match`.

The full LOTOS specification of the IN CS-1 GFP, the ABD and OCS service and the relevant SIBs, can be found in the appendix.

The next section reports on how I checked that GFP + ABD $\vdash \phi_1$ and GFP + OCS $\vdash \phi_2$. Moreover, to discover interaction, I checked the property $\phi_1 \wedge \phi_2$ relative to GFP + ABD + OCS. It is proved that GFP + ABD + OCS $\not\vdash \phi_1 \wedge \phi_2$. This confirms that indeed dialled numbers are expanded too late such that they could not be screened. It also has been verified that if the order of invocation of the ABD and OCS service is reversed, no (unwanted) interaction occurs.

All these results confirm the statements made in [1]. However, it turned out that still something was not in order. Namely, after switching the ABD service off, property ϕ_1 was still satisfied (GFP $\vdash \phi_1$) which certainly is undesirable. This was due to a subtility in the initialisation of the main process in the LOTOS specification. In the next section, one can read how the bug is repaired.

4. Analysis in CÆSAR/ALDÉBARAN

CÆSAR/ALDÉBARAN is an advanced verification toolbox for LOTOS programs, and it basically consists of two tools. CÆSAR is a tool that allows for generating the transition graph of a LOTOS specification. To our knowledge, CÆSAR is at the moment the only tool which can handle 'full' LOTOS up to some reasonable restrictions. The graphs that are generated by CÆSAR can be used by several other tools like ALDÉBARAN, AUTO, MEC and XESAR. One of these tools called ALDÉBARAN has also been integrated in CAESAR. This tool is used for reducing and comparing transitions graphs with respect to several behavioural equivalences, e.g. Milner's *observation equivalence*. In the analysis of the example, I used both tools.

4.1. Generating graphs

As a first experiment, I generated with CÆSAR the graph of the main specification (the GPF including the ABD and OCS service) which is denoted by the following LOTOS process header:

```
IN_Global_Functional_Plane [ poi, por ]
    (mk_call_instance_data(0, abd2, null, a2))
```

Here the initial values `mk_call_instance(0,abd2,0,a2)` are taken from [1]. For this situation, a graph containing 23 states and 26 edges was generated by using CÆSAR. By ALDÉBARAN, the graph was reduced to 15 states and 18 edges with respect to Milner's observation equivalence. The minimised graph is given below:

```
des (0, 18, 15)
(0,"POI !CALL_ORIGINATED !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, A2)",2)
(2,"POR !CONTINUE_AS_IS !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, A2)",3)
(3,"POI !ADDRESS_COLLECTED !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, A2)",11)
(4,"POI !ADDRESS_ANALYSED !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, A2)",5)
(5,"POR !CONTINUE_WITH_NEW_DATA !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",6)
(6,"POI !COMPLETE_CALL !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",12)
(7,"POR !CONTINUE_AS_IS !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",9)
(8,"POI !BUSY !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",7)
(8,"POI !NO_ANSWER !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",7)
(8,"POI !CALL_ACCEPTANCE !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",14)
(8,"POI !END_OF_CALL !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",13)
(9,"POI !END_OF_CALL !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",13)
(10,"POI !END_OF_CALL !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",13)
(10,"POI !ACTIVE !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",7)
(11,"POR !CONTINUE_AS_IS !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, A2)",4)
```

```
(12,"POR !CONTINUE_AS_IS !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",8)
(13,"POR !CONTINUE_AS_IS !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",1)
(14,"POR !CONTINUE_AS_IS !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",10)
```

In Fig. 1, this graph is represented in the style adopted from [3] which is (hopefully) more readable. This picture can be interpreted as the transition graph given above in the following sense. The points where the arrows bounce against the boxes in the picture correspond to the states in the transition graph. These points have been labelled with the original state names. Furthermore, to guide the intuition, I have also visualised the places where the ABD and OCS services are invoked.

Due to the small size of the graph one can easily check that property ϕ_1 is satisfied: every time an attempt is made to establish the connection (`COMPLETE_CALL`) the destination address corresponds with the expansion of the abbreviated number that was dialled.

On the other hand, property ϕ_2 does not hold because one can see in the same graph that the call has not been rejected (by returning a `CLEAR_CALL`). This is caused by the fact that the number `d2` could not be screened while it was abbreviated (as `abd2`) when the OCS service was active. This is a typical example of feature interaction because property ϕ_2 holds when the OCS service operates in isolation (which I also checked), but does not hold when the ABD service is involved.

To this point the computer analysis confirms the statements of Bouma and Zuidweg. However, we are not done yet. Remarkably, I found out that property ϕ_1 was still satisfied when the ABD service was switched off, which means that ϕ_1 is always true. Clearly, this does not meet with our expectations, because when the ABD service is switched off one would like to have that abbreviated numbers cannot be used any more. This inconsistency is due to the strange initialisation of the GFP process, where the *call instance data* is initialised by

```
mk_call_instance(0,abd2,null,a2).
```

However, the telephone network may not know in advance that `a2` is the destination address of the (abbreviated) dialled number `abd2`. We corrected this by changing the last initialisation parameter as follows:

```
mk_call_instance(0,abd2,null,get_address(abd2))
```

saying that initially the system tries to find the destination address of the dialled number itself. In this example (see the Appendix) this means that in the beginning the destination address is undefined as the dialled number is an abbreviation. Recall that in Section 2 we defined that `get_address(abd2)=null`. But then, it appeared that property ϕ_1 was not satisfied any more when turning the ABD service on again. This was due to the fact that in the original specification the ABD service only updates dialled numbers (if abbreviated), but it should also update the corresponding destination address, as this is not done by the telephone network in the example. After fixing this, the service behaved properly.

As a final example of our verification, the following graph shows there is no feature interaction when the ABD and OCS service are invoked in reverse order. (The graph is generated by CÆSAR and minimised with respect to observation equivalence with ALDÉBARAN.)

```
des (0, 6, 7)
(0,"POI !CALL_ORIGINATED !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, NULL)",2)
(2,"POR !CONTINUE_AS_IS !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, NULL)",3)
(3,"POI !ADDRESS_COLLECTED !MK_CALL_INSTANCE_DATA (0, ABD2, NULL, NULL)",4)
(4,"POR !CONTINUE_WITH_NEW_DATA !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",5)
(5,"POI !ADDRESS_ANALYSED !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",6)
(6,"POR !CLEAR_CALL !MK_CALL_INSTANCE_DATA (0, D2, NULL, A2)",1)
```

In this graph, one can see that the abbreviated number is expanded before the screening took place. And, as we wished, the call is rejected (`CLEAR_CALL`). Note that in our specific example, number `d2` has been inserted in the screening list; `screen (d2)=match` as is defined in Section 2.

4.2. *Checking features with testers*

In the previous section, we just looked into the generated graph for checking whether certain properties (features) were satisfied. Of course, this is not the way we want to do it in general, for the graphs are mostly much larger and the properties to be verified more intrinsic. It would be better to check the properties directly with an automatic model checker. Unfortunately, we are not aware of a tool supporting a modal/temporal logic which incorporates data. (This is despite the fact that we have the technology for implementing such tools.)

For this reason, we follow here an alternative route which is based on *testers* as described in [1]. A tester is a simple LOTOS specification which encodes a property to be checked and runs in parallel with the original specification. As soon as the property to be tested is violated, the tester fires a special error transition. In [1], testers are only used for discovering errors, but not for proving positive properties.

However, here we describe a simple trick which also allows us drawing positive conclusions by using testers. It works as follows: hide all gates except the error gate, in the parallel composition of the tester and the process to be verified. Then generate with CÆSAR the graph of the resulting process. If the graph contains an error transition than we know the property is violated (false), and otherwise the property is true.

Next, it is shown how property ϕ_1 is actually checked with this method in the revised example (see the previous section).

> We placed a tester (`ABD_tester`) encoding the property ϕ_1 of the ABD service in parallel with the main specification (`IN_Global_Functional_Plane`) and hid all gates except the error gate. The resulting process is denoted by `Test`.

In LOTOS:

```
specification Test [ error ]    : exit
   hide poi, por  in
      (
         IN_Global_Functional_Plane  [ poi, por ]
               (mk_call_instance_data(0, abd2, null,
                 get_address(abd2)))
            |[poi]|
         ABD_tester [poi, error]
      )
endspec (* Test *)
```

CÆSAR/ALDÉBARAN generated the following minimised graph for process `Test`:

```
des (0, 0, 1)
```

This represents a graph containing just one state. It is trivial to see that this trivial graph does not contain an error transition. Thus, it can be concluded that the ABD service *does* satisfy property ϕ_1.

Note that when graphs are larger, say 1000.000 states, one can search for an error transition by the various pattern matching algorithms that are available on the UNIX operating system, e.g. the `grep` command. Such commands can be used without any risk as CÆSAR always generates connected graphs (unless CÆSAR is wrong). So, it can not be the case that the error transition was found by a pattern matching command (`grep`) although the error occurred harmless in a disconnected part of the graph.

In an analogous way, we should be able to verify the other properties ϕ_2 and $\phi_1 \wedge \phi_2$. However, we did not verify these properties by testers as Bouma and Zuidweg did not specify the tester encoding ϕ_2. Note that we did verify these properties by just observing the generated graphs as we did in the previous section.

5. Discussion

I consider the experiment as successful: the results of Bouma and Zuidweg are strengthened by the application of verification tools. In particular, a bug was found in the example during verification in CÆSAR which was not detected while testing in LITE.

It turned out that the graphs generated by CÆSAR were extremely small: not more than 23 states and 26 edges. Maybe a lot of possible branches had to be explored internally but due to the presence of data most branches could be cut off. Nevertheless, it indicates that far more complicated examples can be handled than the one that is analysed in this paper.

There are also several points for improvement. For instance, I am looking forward using modal/temporal property checkers that are parametrised with data. In my opinion, it really would be a step forward if we reach the point where one can check the logic properties given in this paper directly, without first having to encode them into testers. Since one can already introduce errors while translating properties into testers.

Another improvement would be to generalise the example such that it can handle more than one incoming phone call in parallel. Then, it can be investigated whether the properties checked in this paper still hold in such more realistic setting. Moreover, it would be interesting to see how fast the state space of this new example grows in the number of incoming phone calls. Maybe the CÆSAR/ALDÉBARAN toolbox will already be pushed to its limit for just a small number of incoming calls.

Acknowledgements

In the first place, I would like to thank Wiet Bouma and Han Zuidweg for their correspondence and making their LOTOS code available to me. Furthermore, I am indebted to Hubert Garavel for answering all my questions about the CÆSAR/ALDÉBARAN tools. Lastly, I would like to thank Frits Vaandrager for spotting a bug in, and for his helpful feedback on, an earlier version of this paper.

Appendix. The main specification

In the definition of the abstract datatypes one can find annotations of the form (*!...*). These are used by a preprocessor of CÆSAR called `caesar.adt` for compiling the datatypes into efficient C code.

```
specification Example [ poi, por ] : exit

library Boolean endlib
library NaturalNumber endlib

behaviour

  IN_Global_Functional_Plane  [ poi, por ]
    (mk_call_instance_data(0, abd2, null, get_address(abd2)))

    (* In the original specification the instance is
       'mk_call_instance_data(0, abd2, null, a2)' *)

where

type Address is Boolean, NaturalNumber
  sorts address (*! implementedby ADT_ADDRESS comparedby ADT_CMP_ADDRESS
                  enumeratedby ADT_ENUM_ADDRESS printedby ADT_PRINT_ADDRESS *)
  opns null (*! implementedby ADT_NULL constructor *),
```

```
      a1   (*! implementedby ADT_A1 constructor *),
      a2   (*! implementedby ADT_A2 constructor *) : -> address
      - eq -   (*! implementedby ADT_EQ_ADDRESS *) : address, address -> Bool
      ord        (*! implementedby ADT_ORD_ADDRESS *) : address -> Nat
  eqns forall ad1, ad2 : address
      ofsort Nat
      ord(null) = 0;
      ord(a1) = Succ(ord(null));
      ord(a2) = Succ(ord(a1));
      ofsort Bool
      ad1 eq ad2 = ord(ad1) eq ord(ad2);
endtype

type Dialled_Number is
  Boolean, Address, SIB_End, NaturalNumber
  sorts dialled_number
    (*! implementedby ADT_DIALLED_NUMBER comparedby ADT_CMP_DIALLED_NUMBER
        enumeratedby ADT_ENUM_DIALLED_NUMBER printedby ADT_PRINT_DIALLED_NUMBER *)
  opns wrong_number (*! implementedby ADT_WRONG_NUMBER constructor *),
      abd2 (*! implementedby ADT_ABD2 constructor *),
      d1 (*! implementedby ADT_D1 constructor *) : -> dialled_number
      d2 (*! implementedby ADT_D2 constructor *) : -> dialled_number
      get_address (*! implementedby ADT_GET_ADDRESS *) : dialled_number -> address
      - eq - (*! implementedby ADT_EQ_DIALLED_NUMBER *) : dialled_number,
        dialled_number -> Bool
      screen  (*! implementedby ADT_SCREEN *) : dialled_number -> SIB_end
      translate  (*! implementedby ADT_TRANSLATE *) : dialled_number -> dialled_number
      abbreviated  (*! implementedby ADT_ABBREVIATED *),
      ok  (*! implementedby ADT_OK *) : dialled_number -> Bool
      ord  (*! implementedby ADT_ORD_AD *) : dialled_number -> Nat
  eqns forall ad, ad1, ad2 : address, dn, dn1, dn2 : dialled_number
      ofsort address
        (* get_address(mk_dialled_number(ad)) = ad; *)
        get_address(d1) = a1;
        get_address(d2) = a2;
        get_address(wrong_number) = null;
        get_address(abd2) = null;
      ofsort Nat
        ord(wrong_number) = 0;
        ord(abd2) = Succ(ord(wrong_number));
        ord(d2) = Succ(ord(abd2));
      ofsort Bool
        ok(dn) = not(dn eq wrong_number);
```

```
        dn1 eq dn2 = ord(dn1) eq ord(dn2);
        abbreviated(abd2) = true;
        not(dn eq abd2) => abbreviated(dn) = false;
      ofsort dialled_number
        (* The following equations are specific for the example *)
        translate(abd2) = d2;
        not(dn eq abd2) => translate(dn) = dn;
      ofsort SIB_end
        screen(d1) = no_match;
        screen(d2) = match;
        screen(abd2) = no_match;
        screen(wrong_number) = error;
endtype

type SIB_End is
  Boolean, NaturalNumber
  sorts SIB_end (*! implementedby ADT_SIB_END comparedby ADT_CMP_SIB_END
                 enumeratedby ADT_ENUM_SIB_END printedby ADT_PRINT_SIB_END *)
  opns match (*! implementedby ADT_MATCH constructor *),
       no_match (*! implementedby ADT_NO_MATCH constructor *),
       success (*! implementedby ADT_SUCCESS constructor *),
       error (*! implementedby ADT_ERROR constructor *) : -> SIB_end
       ord (*! implementedby ADT_ORD_SIB_END *) : SIB_end -> Nat
       - eq - : (*! implementedby ADT_EQ_SIB_END *) SIB_end, SIB_end -> Bool
  eqns forall x, y : SIB_end
       ofsort Bool
         x eq y = ord(x) eq ord(y);
       ofsort Nat
         ord(error) = 0;
         ord(success) = Succ(ord(error));
         ord(no_match) = Succ(ord(success));
         ord(match) = Succ(ord(no_match));
endtype (* SIB_End *)

type Call_Instance_Data is
  Address, Dialled_Number, NaturalNumber
  sorts call_instance_data (*! implementedby ADT_CALL_INSTANCE_DATA
                              comparedby ADT_CMP_CALL_INSTANCE_DATA
                              enumeratedby ADT_ENUM_CALL_INSTANCE_DATA
                              printedby ADT_PRINT_CALL_INSTANCE_DATA *)
  opns mk_call_instance_data (*! implementedby ADT_MK_CALL_INSTANCE_DATA constructor *)
                  :Nat, dialled_number, address, address -> call_instance_data
       get_call_reference (*! implementedby ADT_GET_CALL_REFERENCE *)
```

```
        :call_instance_data -> Nat
    get_calling_line_identity (*! implementedby ADT_GET_CALLING_LINE_IDENTITY *)
        :call_instance_data -> address
    get_dialled_number (*! implementedby ADT_GET_DIALLED_NUMBER *)
        :call_instance_data -> dialled_number
    get_destination_number (*! implementedby ADT_GET_DESTINATION_NUMBER *)
        :call_instance_data -> address
    update_dialled_number (*! implementedby ADT_UPDATE_DIALLED_NUMBER *)
        :dialled_number, call_instance_data -> call_instance_data
    update_destination_number (*! implementedby ADT_UPDATE_DESTINATION_NUMBER *)
        :address, call_instance_data -> call_instance_data
  eqns forall cr : Nat, dn, dn1 : dialled_number,
           cli, dst, ad1 : address
    ofsort Nat
      get_call_reference(mk_call_instance_data(cr,dn,cli,dst)) = cr;
    ofsort dialled_number
      get_dialled_number(mk_call_instance_data(cr,dn,cli,dst)) = dn;
    ofsort address
      get_calling_line_identity(mk_call_instance_data(cr,dn,cli,dst)) = cli;
      get_destination_number(mk_call_instance_data(cr,dn,cli,dst)) = dst;
    ofsort call_instance_data
      update_dialled_number(dn1, mk_call_instance_data(cr, dn, cli, dst)) =
          mk_call_instance_data(cr, dn1, cli, dst);
        update_destination_number(ad1, mk_call_instance_data(cr, dn, cli, dst)) =
          mk_call_instance_data(cr, dn, cli, ad1)
endtype (* Call_Instance_Data *)

type Trigger_Points is
  NaturalNumber, Boolean, Call_Instance_Data
  sorts trigger_points
    (*! implementedby ADT_TRIGGER_POINTS comparedby ADT_CMP_TRIGGER_POINTS
        enumeratedby ADT_ENUM_TRIGGER_POINTS printedby ADT_PRINT_TRIGGER_POINTS *)
  opns call_originated (*! implementedby ADT_CALL_ORIGINATED constructor *)
       address_collected (*! implementedby ADT_ADDRESS_COLLECTED constructor *),
       address_analysed (*! implementedby ADT_ADDRESS_ANALYSED constructor *),
       complete_call (*! implementedby ADT_COMPLETE_CALL constructor *),
       busy (*! implementedby ADT_BUSY constructor *),
       no_answer (*! implementedby ADT_ANSWER constructor *),
       call_acceptance (*! implementedby ADT_CALL_ACCEPTANCE constructor *),
       active (*! implementedby ADT_ACTIVE constructor *),
       end_of_call (*! implementedby ADT_END_OF_CALL constructor *),
       continue_as_is (*! implementedby ADT_CONTINUE_AS constructor *),
       continue_with_new_data (*! implementedby ADT_CONTINUE_WITH_NEW_DATA constructor *),
```

```
        handle_as_transit (*! implementedby ADT_HANDLE_AS_TRANSIT constructor *),
        initiate_call (*! implementedby ADT_INITIATE_CALL constructor *),
        party_handling (*! implementedby ADT_PARTY_HANDLING constructor *),
        clear_call (*! implementedby ADT_CLEAR_CALL constructor *) : -> trigger_points
        ord (*! implementedby ADT_ORD_TRIGGER_POINTS *) : trigger_points -> NAT
        - eq - (*! implementedby ADT_EQ_TRIGGER_POINTS *) :
          trigger_points, trigger_points -> Bool
        is_armed (*! implementedby ADT_IS_ARMED *) :
          trigger_points, call_instance_data -> Bool
        trigger_ABD (*! implementedby ADT_TRIGGER_ABD *) :
          trigger_points, call_instance_data -> Bool
        trigger_OCS (*! implementedby ADT_TRIGGER_OCS *) :
          trigger_points, call_instance_data -> Bool
         eqns
        forall t, t1, t2 : trigger_points, cid : call_instance_data,
               cr : Nat, dn : dialled_number, cli, dst : address
        ofsort Nat
          ord(call_originated) = 0;
          ord(address_collected) = Succ(ord(call_originated));
          ord(address_analysed) = Succ(ord(address_collected));
          ord(complete_call) = Succ(ord(address_analysed));
          ord(busy) = Succ(ord(complete_call));
          ord(no_answer) = Succ(ord(busy));
          ord(call_acceptance) = Succ(ord(no_answer));
          ord(active) = Succ(ord(call_acceptance));
          ord(end_of_call) = Succ(ord(active));
          ord(continue_as_is) = Succ(ord(end_of_call));
          ord(continue_with_new_data) = Succ(ord(continue_as_is));
          ord(handle_as_transit) = Succ(ord(continue_with_new_data));
          ord(initiate_call) = Succ(ord(handle_as_transit));
          ord(party_handling) = Succ(ord(initiate_call));
          ord(clear_call) = Succ(ord(party_handling));
        ofsort Bool
          t1 eq t2 = ord(t1) eq ord(t2);
          trigger_ABD(t1, cid) = t1 eq address_analysed;
          trigger_OCS(t1, cid) = t1 eq address_collected;
          is_armed(t1, cid) = trigger_ABD (t1, cid) or
          trigger_OCS(t1, cid);
        endtype (* Trigger_Points *)

process Poi_sequence [poi,call_terminate] : exit
:=
  poi! call_originated ?cid : call_instance_data;
```

```
  poi! address_collected ?cid : call_instance_data;
  poi! address_analysed ?cid : call_instance_data;
  poi! complete_call ?cid : call_instance_data;
(
  (
    poi! busy ?cid : call_instance_data;
    exit
   []
    poi! no_answer ?cid : call_instance_data;
    exit
   []
    poi! call_acceptance ?cid : call_instance_data;
    poi! active ?cid : call_instance_data;
    exit
  )
  [>
  (
    poi! end_of_call ?cid : call_instance_data;
    exit
   []
    call_terminate; exit
  )
)
endproc (* Poi_sequence *)
Process Por_choice [poi, por, call_setup call_terminate]
                   (cid: call_instance_data) : exit
:=
  (
    poi?dp : trigger_points !cid;
    (
      por! continue_as_is ?new_cid: call_instance_data;
      Por_choice[Poi, Por, call_setup, call_terminate](cid)
    []
      por! continue_with_new_data ?new_cid: call_instance_data;
      Por_choice[Poi, Por, call_setup, call_terminate](new_cid)
(* []
      por! initiate_call ?new_cid: call_instance_data;
      call_setup! new_cid;
      Por_choice[poi, por, call_setup, call_terminate](cid)
    []
      por! handle_as_transit ?new_cid: call_instance_data;
      call_setup! new_cid;
      call_terminate; exit  *)
```

```
    []
      por! clear_call ?new_cid: call_instance_data;
      call_terminate; exit
    )
   )
endproc (* Por_choice *)

Process Basic_call [poi, por] (cid: call_instance_data): exit :=
hide call_setup in
 (
  (
  hide call_terminate in
    (
      Poi_sequence[poi, call_terminate]
     |[poi, call_terminate]|
      Por_choice [poi, por, call_setup, call_terminate](cid)
    )
  )
   []
   exit
 )
endproc (* Basic_call *)

Process Trigger_Detection [Poi, por] : exit
:=
Poi ?detection_point: trigger_points ?cid: call_instance_data;
(
(  [is_armed(detection_point, cid)] ->
       (
        Invoke_Service(detection_point, cid) >>
        accept return_point: trigger_points,
            new_cid: call_instance_data in
             (
              por !return_point !new_cid;
              Trigger_Detection [Poi, Por]
             )
       )
)
  []
    [not(is_armed(detection_point, cid)) ] ->
      (
        Por !continue_as_is  !cid;
        Trigger_Detection [Poi, Por]
      )
```

```
  [] exit
)
endproc (* Trigger_Detection *)

Process Invoke_Service (dp: trigger_points, cid: call_instance_data)
: exit(trigger_points, call_instance_data)
:=
   [trigger_ABD(dp, cid)] -> ABD(cid)
  []
   [trigger_OCS(dp, cid)] -> OCS(cid)

(* If we want to switch off a service 'ABD(cid)' or 'OCS(cid)'
   must be changed in 'exit(dp, cid)' *)

endproc (* Invoke_Service *)

Process Screen (d: dialled_number): exit(SIB_end)
:= exit(screen(d))
endproc (* Screen *)

Process Translate (d: dialled_number): exit (SIB_end, dialled_number)
:=
  [ok(d) ] -> exit(success, translate(d))
  []
  [not (ok(d))] -> exit(error, d)
endproc (* Translate *)

Process ABD (cid: call_instance_data)
: exit(trigger_points, call_instance_data)
:=
  Translate(get_dialled_number(cid)) >>
  accept termination: SIB_end, new_number: dialled_number in
  (
    [termination eq success] ->
        exit(continue_with_new_data,
             update_dialled_number(new_number,
             update_destination_number(get_address(new_number),
             cid)))

        (*  Instead of the original:
            'exit(continue_with_new_data,
            update_dialled_number(new_number, cid))'  *)
    []
     [not(termination eq success)] ->
      exit(clear_call, cid)
 )
```

```
endproc(* ABD *)

Process OCS (cid: call_instance_data)
: exit(trigger_points, call_instance_data)
:=
  Screen(get_dialled_number(cid)) >>
  accept termination: SIB_end in
  (
    [termination eq no_match] ->
     exit(continue_as_is, cid)
    []
     [not(termination eq no_match)] ->
      exit(clear_call, cid)
  )
endproc(* OCS *)

Process ABD_tester [Poi, error] : exit
:=
  Poi? dp : trigger_points ?cid :call_instance_data;
  (
    [abbreviated(get_dialled_number(cid))
     and (dp eq address_collected)] ->
    Check[Poi, error] (get_dialled_number(cid))
   []
    [not(aabreviated(get_dialled_number(cid)))
     and (dp eq address_collected))] -> ABD_tester[Poi, error]
  )
endproc (* ABD_tester *)

Process Check [Poi, error](d: dialled_number): exit
:=
  Poi? dp : trigger_points ?cid :call_instance_data;
  (
    [(dp eq complete_call) and
     not(get_destination_number(cid)
     eq get_address(translate(d)))] ->
     error; stop
  []
    [not ((dp eq complete_call) and
     not(get_destination_number(cid)
     eq get_address(translate(d))))] ->
     (Check [poi, error](d))
  []
    exit
```

```
  )
endproc (* Check *)

process IN_Global_Functional_Plane [poi, por] (cid :call_instnace_data) : exit
:=
  (Basic_call [poi, por] (cid)
   |[poi, por]|
    Trigger_Detection [poi, por]
  )
endproc (* IN_Global_Functional_Plane *)

endspec
```

References

[1] W. Bouma and H. Zuidweg, Formal analysis of feature interactions by model checking, *Proc. 2nd Workshop on Protocol Verification*, Eindhoven, Netherlands, 1993.

[2] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez and J. Sifakis, A toolbox for the verification of LOTOS programs, *Proc. 14th Internat. Conf. on Software Engineering ICSE'14*, Melbourne, Australia, 1992.

[3] A.S. Klusener, S.F.M. van Vlijmen and A. van Waveren, Service independent building blocks – I; concepts, examples and formal specifications, Report P9310, Programming Research Group, University of Amsterdam, 1993. A shorter version of this paper appeared in: *Proc. RACE IS & N Conf. (Internat. Conf. on Intelligence in Broadband Services and Networks)*, Paris, France, 23–25 November, 1993.