# Service Specification and Validation for the Intelligent Network[1]

Pierre-Alain Etique
etique@vptt.ch
Swiss Telecom PTT
Research & Development
FE324
CH-3000 Bern 29

Jean-Pierre Hubaux
hubaux@tcom.epfl.ch
Swiss Federal Institute of Technology
TCOM Laboratory
Telecommunication Services Group
DE-TCOM
CH-1015 Lausanne

Xavier Logean
logean@tcom.epfl.ch
Swiss Federal Institute of Technology
TCOM Laboratory
Telecommunication Services Group
DE-TCOM
CH-1015 Lausanne

***Abstract***

We propose an object-oriented specification language, FUS++, for expressing the functional behaviors and the desired properties of each telecommunications service at the analysis phase. Our approach is based on Fusion, an object-oriented method that consists of analysis, design, and implementation phases for software development. Accompanying FUS++ is a tool we developed for translating FUS++ specifications to Promela statements; validation of service specifications and detection of feature interactions are thus possible by applying Spin tools to these Promela statements. To ensure a correct implementation of services with respect to their specifications, we exercise a novel concept of adding a service modeler and observer (SMO) to the target system where the execution of these services takes place. Combined with a test scenario generator, SMO is quite effective in identifying implementation errors on the fly. This paper presents the FUS++ language, the specification of telecommunications services based on FUS++, the concept of SMO, and the realization of SMO in a CS-1 (Capability Set No 1) intelligent network.

***Index terms*** : telecommunication software engineering, validation, software lifecycle, property verification, intelligent network.

---

[1] This work has been partially funded by Swiss Telecom PTT

# 1. Introduction

New architectures for telecommunications services, e.g. the Intelligent Network (IN) proposal [1], provide an easier way to introduce new services into a network. These architectures give network operators and independent service providers the opportunity to react rapidly to customer needs. With the deregulation of the telecommunications market in Europe, such opportunities will become increasingly important, and one can see considerable research and standardization efforts in the domain of telecommunications services architectures and service life-cycle (from specification to maintenance): the IN recommendations [1], the TINA initiative [2], and several RACE projects like SCORE [3], ROSA [4], PRISM [5], CASSIOPEIA [6].

However, these new architectures and the opportunity they provide to rapidly introduce new services into complex networks make the problem of proving that the services conform to their specification more acute: in fact, reacting rapidly to customer or market needs requires introducing new services only a few months after the first specification; such a short interval makes it quite impossible to go through the tedious and long (several months) tests usually performed for new services. This difficulty is getting worse since there are always more services added to the networks, contributing to the overall complexity. Services must all work correctly without hindering the functionality of other services; this last problem is often referred to as the "feature interactions problem" [7].

These obstacles on the road towards rapid service introduction call for new approaches to increasing confidence in the service. In order to address this problem, this paper presents a method which allows the validation of the service specification thus ensuring that the implementation is correct with respect to the specification.

We consider that the service development process up to implementation is split into three different phases. The first one is the *analysis*. Its goals are twofold: first to allow the analyst to understand the problem domain correctly and to model it; secondly to produce a document, called *specification*, describing the analyst's understanding of the problem domain and defining *what* the system must do. The specification is then used in the next phase, called *design*, where one defines *how* the system realizes the specification. The *implementation* phase transforms the result of the design phase in such a way that it can be executed by the target system.

In this paper we propose to use an object-oriented language for the analysis phase. We call this language FUS++; it is an extended and formal version of the language used in the analysis phase of Fusion [8]. FUS++ is designed for the specification of telecommunications services; it allows the specification of an object model defining the problem domain for a given service; then the behavior of the service can be specified using constructs inspired by the Intelligent Network. Last but not least, FUS++ allows the specification of properties in linear time temporal logic (LTL)[18]. A FUS++ specification contains a behavioral part and a property part. It is therefore possible, using appropriate validation tools, to prove that the specified properties are satisfied by the behavior part of the specification.

Being convinced of the specification quality is an important first step towards a good implementation. However, errors can be introduced in the different phases leading to the implemented service. The traditional way of proving that an implementation conforms to its specification is to prove that each step leading to this implementation satisfies the properties that

were valid at the preceding step. Such an approach implies that each step in the development process must conform to some well defined rules which can be complex to use.

In this paper we propose a much more pragmatic solution: the implementation is achieved with traditional methods and tools, with a constraint that a mapping can be realized between the objects that are identified in the specification and that in the implementation. When this mapping is given, we provide tools that automatically transform the correctness requirements (the properties) of the specification in such a way that they can be checked in the real system. One can then run automatic tests (for example with random input) until a problem is found. Such random state exploration corresponds to the strategy advocated by West [9] for the validation of protocols. In fact our contribution here is to propose a way to detect a problem automatically in a domain where it can be very difficult for a machine to find out that the system behavior for a random input is not correct. Concretely we shall show how a special entity can be introduced in an Intelligent Network [1] to check the properties specified during analysis.

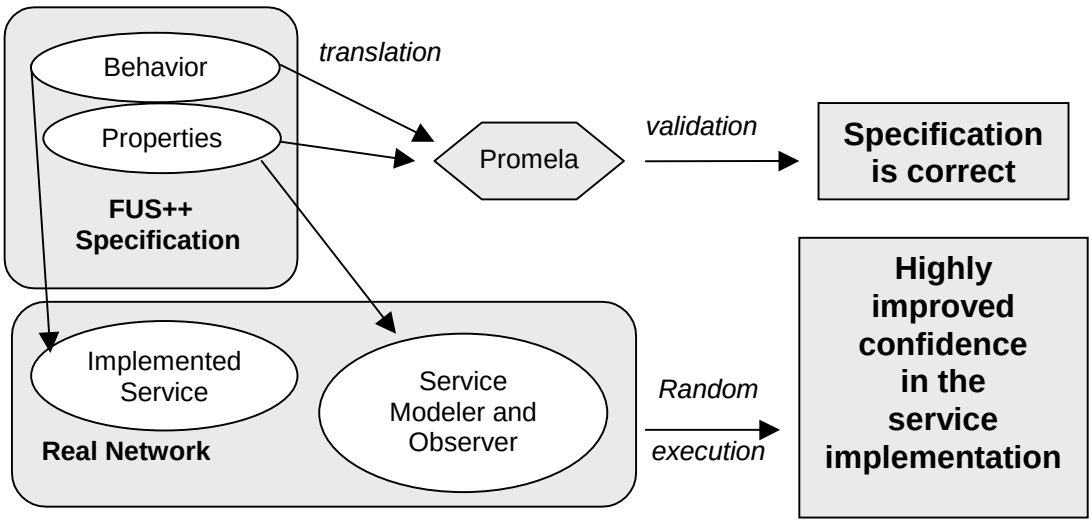


Figure 1: Overview of the approach

Figure 1 presents an overview of the approach described in this paper. Notice that this figure also shows that we are using Promela [10] as an intermediate language for the validation of FUS++ specifications. Promela is a specification language that serves as input to “Spin”, a formal validation system. The two advantages of using Spin in the context of our work are the possibility to validate properties expressed in linear time temporal logic (LTL) and the optimal memory usage during validation.

This paper is composed of two parts: first FUS++ is presented with an emphasis on extensions to the Fusion method which allow several service specifications to be merged. The second part is devoted to the presentation of how our method can be used to increase confidence in the correctness of the service implementation.

# 2. Object-Oriented Service Specification

## *2.1 Introduction*

There exist several service specification methods. The Intelligent Network recommendations [1] propose a method using Service Independent Building Blocks (SIB). Formal description techniques (FDTs), used mainly in the protocol engineering world, can be used to specify services and the network on which they run. Combes and Pickin in [11], and Makarevitch in [27] present an approach using SDL (Specification and Description Language). The use of LOTOS in this context is, for example, extensively studied at the University of Ottawa [12]. The advantage of using such FDTs is obviously the possibility of applying the verification and validation techniques specific to each language.

Another trend in service specification is the use of object-oriented specification techniques [4] [28] [3] [13]. The idea behind object oriented approaches is to promote a structured method in which the concepts of class and encapsulation help the engineers to structure the solution; inheritance allows a progressive refinement of the specification; object-orientation also contributes to specification and code reuse. Our specification method belongs to this trend. Our first goal is to use a methodology well suited to the needs of service specifiers. In a second step, we shall see how such a specification can be validated.

Many object-oriented analysis methods are available [14] [15] [30]. Our approach is based on *Fusion* [8] whose analysis phase borrows several concepts from OMT (Object Modeling Technique). The main reason for this choice is that it allows a clear separation between analysis and design in *Fusion*.

## *2.2 Formalizing the Analysis Phase of Fusion*

### 2.2.1 Introduction

The Fusion methodology considers that the development cycle is divided into three phases: analysis, design and implementation ( Figure 2). The goals of the analysis are twofold: firstly to allow the analyst to understand the problem domain correctly and to model it; secondly to produce a document, called specification, describing the analyst's understanding of the problem domain and defining *what* the system must do. The specification is then used in the next phase, called design, where one defines *how* the system does what was specified. Finally, implementation is the phase where the code is produced.

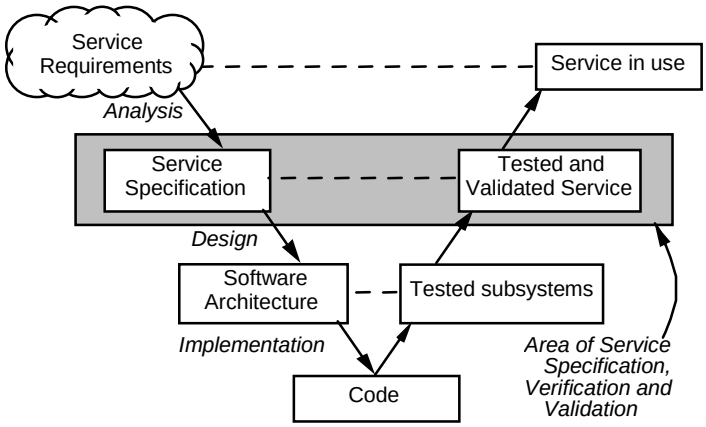


**Figure 2: Service Development phases considered by *Fusion***

In this paper we concentrate on the analysis phase and the document produced during this phase, the specification. We make a distinction between the methodology, which tells the analyst how to structure its ideas, how to model its system and how the different phases are linked together, and the notations used to write down in a document the results of the analysis phase. Fusion provides both a methodology and the corresponding notations. However, these notations are often informal and based on natural language. As we shall see, we need the possibility to have formal notations used for the specification in order to be able to validate it. Therefore we propose a formal notation for a specification resulting from a *Fusion* analysis; we call this formal language FUS++.

### 2.2.2 Analysis with *Fusion*

The analysis in *Fusion* produces two models that capture different aspects of a system:

*Object model***:** This model defines the static structure of the information in the system.

*Interface model***:** This model defines the input and output communication of the system. It uses two models to capture different aspects of behavior. The *operation model* characterizes the effect of each system operation in terms of the state change it causes and the output event it sends. The *life-cycle model* characterizes the allowable sequences of system operations and events.

A data dictionary is produced simultaneously with these models. It is "a central repository of definitions of terms and concepts. Without it, the *Fusion* models have little semantic content" [Cole94].

The *object model* describes graphically the classes and their relationships. A class is represented by a box with the name of the class at the top and the name of the attributes in the lower part of the box (Figure 3 is a typical example of an object model). The type of the attributes is given in the data dictionary.

Relationships are shown as a diamond joined to the participating classes by arcs. The arcs can be annotated to express cardinality constraints on the relation. A number, a range, an asterisk (denoting zero or more) or a plus sign (denoting one or more) are allowed cardinality constraints.

When all classes and relationships are described, a *system object model* is defined as a subset of the object model that relates to the system to be built. The boundary of the *system object model* is represented by a dotted square.

"The *operation model* is expressed as a series of schemata. There must be at least one schema for each system operation" [8]. Operations correspond to input events into the system. The events are produced by external active entities called agents. The *environment* is the set of agents with which a system communicates.

An operation schema is composed of different clauses which are described briefly hereafter. The *Reads* and the *Changes* clauses describe all the values that the operation may access or change. The *Sends* clause gives a list of all the agents and the events that the operation may send them. An operation is described as a state transition between an initial state defined by the *Assumes* clause and a final state described by the *Result* clause. Both are predicates: the former defines the precondition for the operation, and the latter the postcondition.

Finally, the *life-cycle model* defines the allowed order of input/output events combinations. They are described using regular expressions syntax.

### 2.2.3 FUS++

Although some extensions have been proposed [29] in order to even increase the Fusion power to structure the ideas of the analyst and hence the specification, the analysis phase of *Fusion* lacks a well defined formalism allowing the specification to be executed by a computer.

We explained in [16] and [17] that a Fusion specification contains two parts: a constructive part which describes in terms of a state machine what the system does, and a property-oriented part which specifies properties that the constructive part must satisfy. Obviously in a *Fusion* analysis, the constructive part is given by the *operation model*, while the property-oriented part corresponds to cardinality constraints on the relationships, or to explicit properties like invariants. It is shown in both mentioned papers how the informal notations proposed for *Fusion* in [8] can be formalized in order to obtain an executable constructive part, and a formal property-oriented part so that the specification can be validated. We call the language resulting from this formalization process FUS++. Its constructive part has a syntax borrowed from C++; it allows the specifier to define the *object model* in a textual manner and to write the *Assumes* and *Result* clauses of an *operation* in an unambiguous and executable way. In fact, in FUS++ the *Result* clause is no longer a predicate but a function that executes the transition. Thus in the *Result* clause, instead of writing the boolean expression `a == 17`, one writes the assignment statement `a = 17`.

The property-oriented part is used to define invariants or more sophisticated properties in linear time temporal logic (LTL) [18]. For Instance, LTL is used for specifying open and concurrent systems [31] and, in [32], applied for the specification of properties in the framework of POTS.

To validate a FUS++ specification means to prove that the behavior of the constructive part does not violate any properties specified in the property-oriented part. This can be done by translating the FUS++ specification into a language for which validation tools do exist. We have chosen the Promela language because its associated validation tool Spin [10] is capable of validating LTL properties and of handling relatively large state spaces.

In our philosophy an analyst uses the Fusion methodology to structure his ideas and the FUS++ language to write the specification. He can then validate this specification using the appropriate tools.

Notice that the use of *Fusion* and hence FUS++, requires that the designer consider the specified system as one entity reacting to events generated by the agents. These events are treated sequentially; there is therefore no parallelism within the system. Parallelism is introduced for the agents: each agent is considered as an independent process, whereby the system is itself one process.

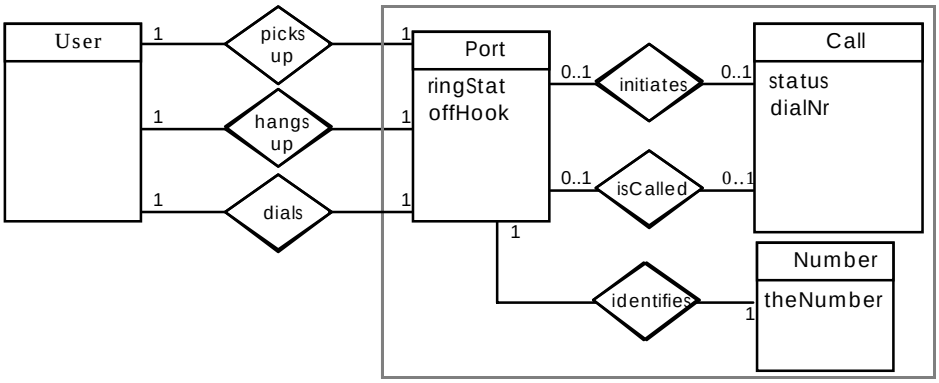


**Figure 3: Part of the Object Model for Basic Call**

Let us consider the system of Figure 3, corresponding to an abstract view of a simple network where users can be attached to ports that are identified by a number, and where calls can be placed between these ports. This object model corresponds to a part of the object model we shall consider for the Basic Call service. The specified system is within the gray/dotted square and represents the network. The agents are outside this square; they are the users. The semantics of the FUS++ specification for this system implies that there is one process for each user, and one single process for the specified system within the dotted square.

Intuitively, the operations defined in the specification correspond to functions that are executed whenever an event generated by a user (`pickup`, `hangup` or `dial`) is received by the system. All the events are queued in one input queue which is emptied by the system process. Therefore, operations are executed sequentially.

FUS++ syntax is provided in the appendix; the detailed of the FUS++ language is described in [22].

## *2.3 Merging Services*

### 2.3.1 Introduction

So far, we have a methodology and the corresponding toolset that allow a specification following the Fusion framework to be written in FUS++ and to be validated. This can already be very useful for software development. Nevertheless, if one wants to use such an environment in the context of telecommunications services and more precisely of Intelligent Network services, a question raises rapidly: how do we merge services?

FUS++ and the associated validation method allow the specification and the validation of a system which, in the context of telecommunications services, can be a network with a single service running on it. However, important problems could appear when several services exist together. Service or feature interactions are recognized to be a serious difficulty on the way towards rapid introduction of new services in telecommunications networks [19].

If one wants to consider several services, using the subpart of FUS++ described in [16], then their specifications must be merged manually to obtain one single system which can then be validated. This approach has several drawbacks:

- Services cannot be specified independently; for each combination of services one may have to modify the specification of individual services differently.
- Merging several service specifications can be a difficult task.

- It does not correspond to the practice of service implementations in real network. At this level, dedicated mechanisms are defined which allow services to operate properly without knowing much of other existing services. This is essential if one accepts that different services can come from different service providers. Notice that what is described here corresponds to an ideal situation. Nowadays, reality is often closer to the manual approach presented above.

All these points call for an automatic merging method of service specifications. We want to be able to obtain a system which is the result of merging n services:

$$system = service_1 \oplus service_2 \oplus \ldots \oplus service_n$$

We assume that services can extend other services, like for example IN services extend Basic Call [1]. Concretely, in FUS++ a system is obtained by including the different files containing the specification of the services to merge.

### 2.3.2 Merging the Data Part

Merging object models can be done with existing constructs, namely inheritance or addition of relationships between classes. See example below, where Basic Call and Call Forwarding are merged (paragraph 2.3.3.3).

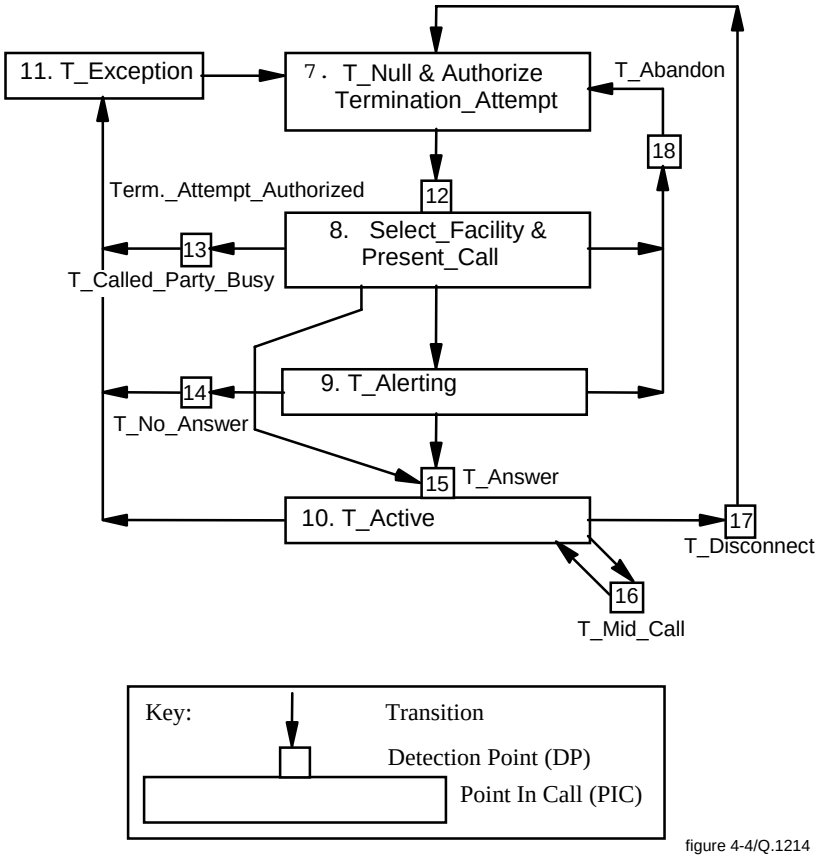


**Figure 4: Terminating Basic Call State Model (BCSM) for Capability Set 1 (CS-1)**

### 2.3.3 Merging Behavior

It is easy in FUS++ to add an *operation* for a new input event. Thus there is no special difficulty if the services to be merged handle different input events. However, this is rarely the case,

especially in a telephony context where the set of possible input events is rather small. Therefore we incorporate the technique proposed in the IN recommendations for the extension of Basic Call. Figure 4 shows the terminating basic call state model (BCSM) for capability set 1 (CS-1) [1]. One can see that call handling is split into different PICs (points in call) where the necessary actions are undertaken. On the transitions between these PICs, there are so called detection points (DP) where an extended service can interrupt call handling and take control. When returning control to basic call handling, the service can either decide to let Basic Call resume from where it was interrupted or to let it jump to another PIC.

#### *2.3.3.1 Detection Points in FUS++*

##### 2.3.3.1.1 Intuitive

In FUS++, we designate places in the service handling where control can be passed to another service requesting it (FUS++ does not enforce that only Basic Call can transfer control; any service could do so). A service must send a *signal* when it is willing to transfer control. Another service can then *respond* to this *signal* in order to take control. A service sending a signal (transferring control) is called an *s-service*, while a service responding to a signal is called an *r-service*.

Remember that FUS++, does not model parallelism within the specified system. Services are therefore similar to functions that are executed either as operations reacting to external events generated by the agents, or as responses to signals generated by other services. Basic Call is the service defining the operations for the POTS events (pickup, hangup and dial). Basic Call sends signals to which other services can respond.

Intuitively, the semantics of sending a signal and responding to it can be understood as follows:

The `response` statement can be considered as declaring a “function” for the given signal. Several distinct responses can be declared for the same signal. Each response corresponds to an r-service.

When the `signal` statement is executed (i.e., the signal is sent) two scenarios can occur:

1. **No explicit response was declared**: in this case, the s-service simply proceeds with the default behavior for the signal (a default behavior has to be defined for each `signal` statement). Notice that this default behavior can be considered as the response to the signal provided by the s-service.
2. **Responses are declared**: A priority number can be associated to each response. The response with the lowest priority number is executed first. If several responses have the same priority, the execution sequence is defined randomly between these responses. The default behavior is considered as the response with the highest priority number, i.e., it is the last one executed. A response without an explicit priority gets the default priority number 1000.

   A response can either return control to the s-service as a traditional function would do or jump to a point in call (`gotoPIC` statement); the latter possibility corresponds to a `goto` statement in a traditional programming language. In this case, control is never returned to the point where `signal` was executed.

If a response returns control normally (i.e., no *gotoPIC*), the next response is executed.

We have seen that when a response returns normally, the next response is executed. There are however cases where it would be wrong to execute a response, namely when the previous response modified the state of the system in such a way that the considered response does no more make sense. This can occur for example if a response is supposed to perform a service when a call is in a given state, but the previous response cleared the call, i.e., put it in an other state. In order to catch such situations, a special variable, called *reference variable*, is considered in the signal statement. As long as this variable is not modified, responses are executed. If it is modified, no more response is considered and execution proceeds with the statement following *signal*.

Formal syntax and semantics of these constructs are given below.

#### 2.3.3.1.2 Formal

We introduce two new statements in FUS++, one for sending the *signal* (*SignalStmnt*) and the other for the response (*SigResp*). The syntax in EBNF of these constructs is given below.

```
SignalStmnt   = "signal" ident [ActualParams]
                "refValue" varRef
                "default" CompoundStmnt.
```

Notice that a *signal* can contain parameters which can be used by the r-services responding to the *signal*. The *CompoundStmnt* placed after the keyword *default* contains the normal behavior of the s-service which must be executed if an r-service responding to the *signal* does not force another behavior (see example in Figure 6 below).

```
SigResp       = "response" ident "to" "signal" ident FormalParams
                ["prio" number] CompoundStmnt.
```

The first *ident* in the *SigResp* construct identifies the r-service responding to a given *signal*. Of course there can be several r-services responding to the same *signal*. In this case they must all have a different name. The number following the keyword *prio* can be used to give a priority to each response. If several r-services respond to the same *signal*, r-services with the smallest priority number will be served first. If several responses have the same priority, they are chosen randomly (see below for more details on the semantics of this construct). The default priority value is 1000. Notice that using FUS++ does not require any particular policy for assigningpriority. The only assumption we make concerning priorities is that there are used to solve interactions problems.Let us now define the semantics associated with these new constructs. Assume that for a given *signal* we have $N$ responses. We call these responses *services*. We identify each of these services by a number ranging from $1$ to $N$. For each service $k$, $1 \leq k \leq N$, we define

*servkResp* as a boolean variable indicating if the service $k$ has got its chance to respond to the considered *signal*.,

$prio_k$ the priority of service $k$.

To define the semantics of the *signal* construct we give an equivalent program in a Promela like syntax. Note that semantically, the *signal* statement contains all the *response* statements for the considered *signal*.

Let

- $SetPrio_i$ be the set of all services of which the priority is $i$, which can be written

$$k \in SetPrio_i \leftrightarrow prio_k == i$$

- *MaxPrio* be the largest priority value of any service responding to the considered *signal*. *MaxPrio* is formally defined as follows:

$$(\exists k,\ 1 \le k \le N : prio_k == MaxPrio) \wedge (\forall k,\ 1 \le k \le N : prio_k \le MaxPrio)$$

We use three temporary variables to define the desired semantics:

- *"prio"* which ranges over the possible priority values
- *"previous"* which stores the original value of the variable identified by the keyword *refValue* in the *signal* statement.
- *"finished"* which indicates if the default behavior has been executed or not.

Thus a *signal* statement in a FUS++ specification can be replaced by the piece of program, expressed in a Promela like syntax, presented in Figure 5:

```
bool finished = false;
prio = 0;
previous = refValue; /* variable reference identified by
                        the keyword "refValue" in the
                        signal statement */
∀k, 1 ≤ k ≤ N: servkResp = false;
do
:: true ->
   if
   :: ⋀ servkResp == true ->
      k, 1 ≤ k ≤ N
      /* default behavior, identified by the keyword "default"
         in the signal statement */
      finished = true;

   :: !servkResp && prio = priok ->
       servkResp = true;
      /* Parameter passing */
      /* Service k's behavior, specified in the "response"
         statement */
      /* ∀k, 1 ≤ k ≤ N <=> this clause appears once for each
         of the N services. */

   :: (prio == i) && ((⋀ servkResp) || (SetPrioi == ∅)) ->
                     k, k ∈ SetPrioi
      prio = prio + 1;
      /* ∀i, 0 ≤ i ≤ MaxPrio <=> this clause appears once for
                                 each value of i between 0 and
                                 MaxPrio */
   fi;

   if
   :: (!VALIDREF(refValue)
      || refValue != previous
      || finished) -> break

   :: (VALIDREF(refValue)
      && refValue == previous
      && !finished) -> skip
   fi
od
```

**Figure 5: Semantics of the *signal* statement**

The function *VALIDREF (varRef)* returns false if the referenced variable belongs to a non-existing object, and true if the object exists.

There are two ways to exit the loop in Figure 5:

- All r-services have responded to the signal and the default behavior of the s-service was executed. In this case the variable *finished* is set to *true*.
- An r-service has modified the variable or attribute identified by *refValue*.

The *refValue* parameter of the *signal* statement is a variable or an object attribute, the value of which is monitored during the execution of the program in Figure 5. An r-service modifying the monitored variable, indicates that no other r-service should be activated on behalf of the considered *signal*, which means that one should exit of the *do* loop of Figure 5. In an Intelligent Network service, the *refValue* would typically be the status of the call. Modifying the call status implies that the situation where the *signal* was sent is no more valid, which means that the call handling is no more at the considered detection point. Thus the other r-services (plus the default behavior) responding to the *signal* should not be activated.

The semantics presented here for the *signal* construct allows the use of FUS++ for the specification of IN services. Yet this semantics is not bounded to a specific capability set. Therefore it does not completely match the detection point handling foreseen in the capability set 1 (CS-1) (see Q.1214 in [1]). If one wants to have this detection point handling exactly, one needs to include a definition of the corresponding behavior in the specification.

#### *2.3.3.2 Points In Call in FUS++*

An IN service taking control at a given detection point in Basic Call can return control to another location in call handling than the one where the service was started. Concretely, the service orders Basic Call to go to a defined PIC. To obtain similar possibilities in FUS++ we introduce the possibility to define a location in the code corresponding to a PIC and to transfer control to such a location. Therefore, two new constructs are defined:

Marking a place in the code is done with the following syntax:

```
PICDecl        = ":" ident FormalParams.
```

This looks very like a function declaration, however it is important to notice that it is only the labeling of a given place in the code. The formal parameters must be all the variables that are in the scope of the operation at the place where the PIC is defined. They must all be present in the formal parameters list. Such a list could theoretically be generated automatically by an appropriate tool. Its role is to allow type checking when a *gotoPic* (see below) statement is encountered.

Ordering control to go to a location marked by a PIC is done with the following statement:

```
GotoStmnt      = "gotoPic" ident ActualParams.
```

The statement *gotoPic* changes the control variable of the process executing the statement to be equal to the value identified by the corresponding PIC definition. The variables identified in the formal parameter list are set to the value given in the actual parameter list. This corresponds to a *by value* parameter passing and is different from the *by reference* mechanism used for procedure calls in FUS++ (see [16]).

### *2.3.3.3 Example*

Let us consider the operation corresponding to the *dial* event for the POTS Basic Call service within the network presented in the object model of Figure 3. Figure 6 presents the interesting part of this operation in this context. Skipped parts are represented by the characters *<…>*.

```
1  Operation:
2     dial
3
4  Description:
5     The user indicates with which partner he/she would like to
6     establish a connection
7
8  Reads:
9     supplied Port thePort;
10    supplied DialNumber dialNr;
11
12 <...>
13 Assumes: {
14    assert (initiates has (Port: thePort));
15    Call call;
16    call = relatant of (Port : thePort) in initiates;
17    assert (thePort.offHook == true && call.status == overlapRec);
18 } // Assumes
19
20 Result: {
21    Call call;
22    call = relatant of (Port : thePort) in initiates;
23    call.dialNr = dialNr;
24    signal Collected_Info(thePort, call)
25    refValue call.status default {
26       :PIC3 (Port thePort, Call call); // Analyse_Info
27       signal Analyzed_Info(thePort, call)
28       refValue call.status default {
29          Number number;
30          select number where number.theNumber == call.dialNr;
31          :PIC4 (Port thePort, Call call, Number number); // Routing & Alerting
32          call.status = callProc;
33          if (number != NIL) {
34             Port destPort;
35             destPort = relatant of (Number : number) in identifies;
36            :PIC7 (Port thePort, Call call, Number number, Port destPort);
37            // T_Null & Authorize Termination_Attempt
38             signal T_Attempt_Auth(destPort, call, thePort)
39             refValue call.status default {
40                :PIC8 (Port thePort, Call call, Port destPort);
41                // Select Facility & Present_Call
42 <...>
43 } // Result
```

**Figure 6: Operation *dial***

In Figure 6 the different clauses necessary to define an operation in FUS*++ are presented: Description, Reads, Assumes* and *Result*. Let us look at the *Result* clause which contains the statements executed when the *dial* event arrives in the system. Notice the progression through the different PICs corresponding to call setup as defined for CS1 [1]: PIC3 (line 26) and PIC4 (line 31) which correspond to the originating BCSM and the PIC7 (line 36) and PIC8 (line 40) corresponding to the terminating BCSM presented in Figure 4. Between PIC 7 and 8, a detection point is foreseen (DP 12 *Term._Attempt_Authorized* see Figure 4); i.e., control can be transmitted to a service wanting to do something when a call arrives at the considered port. In FUS++ this is represented by the *signal* statement on line 38.

Call Forwarding Unconditional (CFU) is a service that will take control of call processing when DP 12 is encountered. Figure 7 presents the additions to the object model for the CFU service. There is a new relationship *forwards* indicating if a *call* was forwarded by a given *port*. A new class *CfuData* is included in *Number* to contain the settings of the service for a given *port* (activated or not and the destination number calls must be forwarded to).

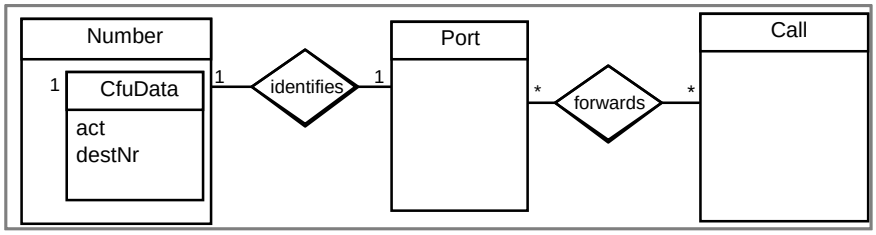


**Figure 7: Object Model for Call Forwarding Unconditional**

Figure 8 shows the FUS++ code for the response corresponding to the CFU service. In the case that no forwarding is necessary, the response returns normally (line 14), but if the call has to be forwarded a gotoPIC statement is used (line 13), which implies that control will never return to the signal statement in Basic Call. The test on line 8avoids forwarding loops.

```
1  response CFU to signal T_Attempt_Auth (Port destPort, Call call, Port thePort)
2  {
3     Number theNumber;
4     theNumber = relatant of (Port : destPort) in identifies;
5     CfuData cfuData;
6     cfuData = relatant of (Number : theNumber) in Number_Incl_CfuData;
7     if (cfuData.act
8         && !forwards has (Port : destPort, Call : call)) {
9        // forward the call
10       forwards with (Port : destPort, Call : call);
11       call.dialNr = cfuData.destNr;
12 Forwarding:
13       gotoPic PIC3 (thePort, call);
14    } // else just continue
15 } // response CFU to signal T_Attempt_Auth
```

**Figure 8: Response for the Call Forwarding Unconditional service**

## *2.4 Summary*

FUS++ gives a powerful framework for the specification and validation of an object-oriented service analysis. With the constructs presented in the previous sections it can be used to specify and validate combinations of telecommunications services and thus address the problem of detecting feature interactions at the specification level. It belongs to the category of detection methods called *satisfaction* approach in [20].

We have developed a compiler, called F2P, which translates a FUS++ specification into a Promela specification according to the FUS++ semantics presented, in [16] and with all the details presented in [22]. The Spin tool can then be used to validate the system[2].

As we have seen, FUS++ supports the modeling of sophisticated services. The described merging technique is especially well suited for Intelligent Network services; nevertheless, the approach is generic enough to take into account a more general family of services, including services on data networks. The F2P compiler provides a straightforward conversion of FUS++ into Promela; this conversion allows the specifier to take advantage of a rich set of existing validation tools.

Of course, the constructs for merging services offered by FUS++ are well adapted to IN service specification. However, even if they are not exploited, the generic nature of the FUS++ tool means that it can be used for any object-oriented specification. Moreover we believe that the

[2] Spin allows the validation of LTL properties by translating them into Büchi automata. We shall see in the next section that we are also using Büchi automata for checking properties in the implemented system. The automata used by Spin can then be reused.

merging techniques presented here are generic enough to model behavior of non-IN telecommunications services.

FUS++ has been used to specify IN services like Call Forwarding and Originating Call Screening at a high abstraction level. In the validation phase we were able to detect errors in a specification: both errors that we introduced willingly in order to test the method and real errors made by the specifier. We also demonstrated that our approach is capable of detecting feature interactions while validating a combination of several services. A very important problem for the success of our approach lies in the choice of the specified properties for a given service. This has already been pointed out by other authors, for example in [20]. The approach proposed by Lin and Lin in [21] which allows a systematic derivation of temporal properties from textual requirements could be integrated into our method in order to help the specifier to find relevant properties.Other teams are working on applying LTL for the detection of feature interactions. For instance, at Uppsala University, they propose a method for the automatic detection of feature interactions in temporal logic[32].

As we will see in the next section, our method also covers the implementation and testing phase on a real network; we will detail the way by which we can check that the run time behavior on the target systems is compliant with the specification.

# 3. Improving Confidence in the Service Implementation

## *3.1 Introduction*

We saw how a service can be specified in FUS++ and how properties can then be validated for this specification. We are now going to show how a FUS++ specification can be used to increase confidence in a service implementation.

The traditional way to increase one's confidence in the correctness of a service implementation is to make long manual functional tests. Some work has been done to help the testing team to define test scenarios in order to increase the coverage of the tests [23]. Another approach proposes to let a machine generate test scenarios randomly; this produces a random state exploration of the system [9]. However, many of these only detect errors when the system under test crashes, though this in general is not the type of errors found in the implementation of telecommunications services. To overcome that difficulty, we propose to add an observer module on the target system executing the service. This software piece will then be able to check on the fly the service properties contained in the FUS++ specification; when a property is violated, an error is discovered. Of course, such a mechanism is not a formal validation, because it can never be guaranteed that all possible states have been visited. However, it can be really helpful discover errors [9].

We shall present in this section how such an observer module can be realized for a service implementation in a CS-1 intelligent network.

Similar approaches, consisting in on-line checking of properties, are undertaken at the University of Rennes. For example in the "Véda" project [34] [25] [26] a method is developed for the dynamic verification of protocols or in [35] for the detection of unstable properties in distributed computations.

The observer concept presented here is similar to what is proposed in [25] or even in [26].

### *3.2 Service Modeler and Observer*

The method proposed here is based on the hypothesis that there is a mapping between each state in the implemented system (implementation state) and a corresponding state in the specification of this system (model state). There is therefore a mapping function, called the μ function, which can at any time be used to know the state in the specification corresponding to the current implementation state.

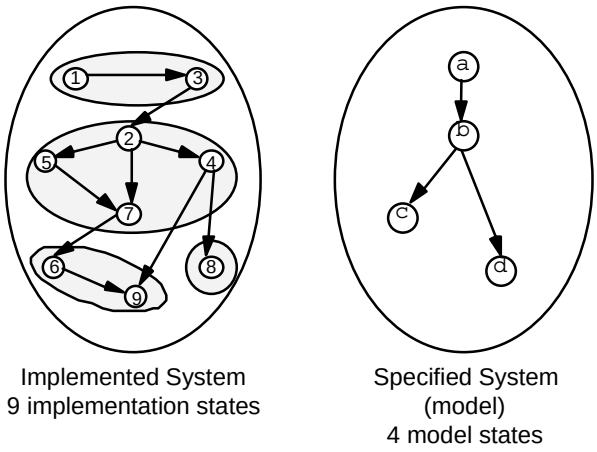


**Figure 9: Example of the different state spaces at specification and implementation levels**

Figure 9 gives a very simple example of the different state spaces resulting from the FUS++ specification of a service (4 states) and its implementation (9 states). Figure 10 shows the corresponding mapping function.

| Implementation States | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Model States | a | b | a | b | b | c | b | d | c |

**Figure 10: Mapping function for the example of Figure 9**

A new entity, called Service Modeler and Observer (SMO), is introduced in the implementation. It is split into two subsystems:

1. The *model manager*, which is responsible for the computation of the mapping function. It must be able to provide the current "equivalent model state" at any moment.
2. The *observer* considers the properties present in the FUS++ specification and checks that they are valid for the model presented by the *model manager*.

Figure 11 shows the context in which we consider the SMO: it is an entity added to the implemented system but which should, as long as possible, act only as an observer for this system, without any influence on the observed system. Notice that in Figure 11, the input to the implemented system is generated by a test generator which can either be a machine running programmed or even random test scenarios, or a person doing manual tests. Random test sequences correspond to a random walk in the system. Colin West showed how such an approach can be efficient [9].

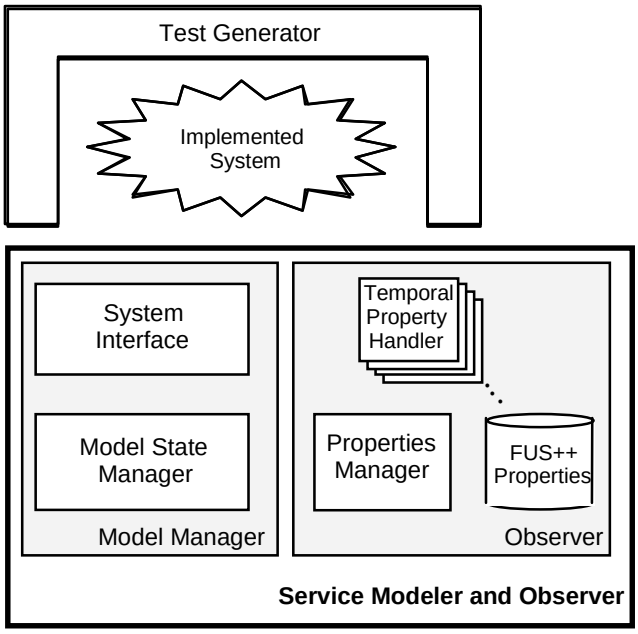


**Figure 11: General structure of the "Service Modeler and Observer" and its logical position in the implementation**

### 3.2.1 The Model Manager

The model manager is responsible for the computation of the mapping function. To achieve this goal, it can use two different strategies:

1. Each time it has to give the value of a data variable it can read the values of the different variables in the implementation necessary to compute the expected result. This approach features several drawbacks:
    a) it implies that most of the variables in the implementation can be read from the SMO, which is not obvious at all;
    b) the number of queries to the variables of the implemented system can be very high, because the SMO cannot know if a variable has changed its value since the last query.
2. The *model manager* stores a data structure containing all the data variables of the specification. It updates the model state each time something has happened in the system, implying a state modification visible at the abstraction level of the specification. Therefore, it must be informed of each transition in the implementation that can have an impact on the model state.

As we shall see, the temporal property handlers in the *observer* need to be informed of each transition in the abstract system managed by the *model manager*. To achieve this, the SMOF must be informed of each transition in the implementation that can have an impact on the model state. This information flow corresponds exactly to what is needed for solution 2; therefore we chose approach 2 and reject the first proposition.

As shown in Figure 11, the *model manager* is itself composed of a *model state manager* and a *system interface.* The former is responsible for the data that represent the system at the abstraction level of the FUS++ specification. It offers a set of operations on the model that can be used to update the model's state. Of course these operations depend on the considered specification and correspond to events that can occur in the implemented system or that can be deduced from what is observed in the implemented system.

The *system interface* is responsible for:

1. the detection of the relevant events in the real system (in fact an event corresponds to a transition in the system);
2. the invocation of the corresponding operations on the *model state manager*.

The SMO must be designed in such a way that the *system interface* is informed of all the relevant transitions in the system. Several solutions can be considered in order to achieve this goal. Let us present three possible design choices:

$\mathcal{X}$ The SMO has access to the messages exchanged between the objects in the system; this information is sufficient to decide which operations must be invoked on the *model state manager*. This solution is probably realistic in a pure object oriented implementation where the SMO can work in collaboration with the run-time system responsible for the transportation of the inter-objects messages.

Advantages: the system is not aware of the SMO. If it is well designed, the SMO could induce very few or even no modification at all on the system behavior.

$\mathcal{Y}$ The SMO can use existing functions to request the system to send notification messages when interesting events occur. This could for example be realized with existing management procedures, if they are complete enough.

Again, with this solution, the system does not need to be aware of the SMO. However, as new messages are sent, the presence of the SMO has an influence on the system, especially on timing aspects.

$\mathcal{Z}$ The system is modified, in order to send notification messages when necessary to the SMO. Disadvantage: the design of the system is influenced by the presence of the SMO.

The appropriate choice between possible solutions $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ or any combination, heavily depends on the considered system. We shall see that for the concrete case of IN CS-1, all three solutions are needed to achieve the best result.

The *model manager*, and thus the way the mapping function is computed, is very dependent on the considered service and on the platform it is implemented on. Notice that although we believe that in pure object-oriented networks as proposed by TINA [2] important parts of the *model manager* could be generated automatically from the specification, we consider in this paper that the *model manager* is designed by hand.

### 3.2.2 The Observer

The *observer* takes as input the properties of the FUS++ service specification and checks their validity on the computations (state sequences) shown by the *model manager*. The properties of the FUS++ specification [16] that are not expressed in linear time temporal logic (LTL) like invariants or absence of non-progress cycles are translated into LTL; thus the *observer* has only to implement a solution for the checking of LTL formulas. Note that assertions can be taken as they are in the *model state manager*.

The checking of temporal properties is based on the same ideas as are used in Spin [10]. An LTL formula can be translated into a Büchi automaton; a Büchi automaton is a finite automaton accepting infinite words [24]. In the context of the verification of temporal properties, the alphabet is the set of states of the considered system, and the words are computations of this

system. To prove that a given program P satisfies a formula *f*, the method used here consists in constructing the Büchi automaton accepting computations satisfying $\neg f$ and prove that there is no computation of P that is accepted by this automaton. In our context such a proof cannot be complete, because we shall never be able to be sure that all possible computations of the implementation have been checked. However, we can run the automaton on the computations generated during our tests, and check that these computations satisfy the desired properties.

Figure 11 shows how an *observer* is composed of several *temporal property handlers* (TPH), one for each temporal property to check. In fact, each TPH implements the Büchi automaton corresponding to the negation of the property it has to check. The TPHs are triggered by the *properties manager* to execute a transition each time the *model state manager* fires a transition. Thus it can be guaranteed that the observed computations satisfy the desired properties.

## *3.3 An SMO for IN Capability Set no 1 (CS-1)*

### 3.3.1 Introduction

A physical Intelligent Network may consist of physical entities coming from different manufacturers. It might be difficult to make modifications in these entities in order to bring some special information to the SMO. Concretely, we should like to be able to realize the SMO using only the solutions $\mathcal{X}$ and $\mathcal{Y}$ presented above. Solution $\mathcal{X}$ means that we want the SMO to be able to intercept all Intelligent Network Application Protocol (INAP) messages [1]. Solution $\mathcal{Y}$ means that the SMO must be able to arm detection points itself, i.e. act as a Service Control Function (SCF). However, this might not be enough. We shall see that to get some information one may need to modify either the SCF or even the Service Switching Function (SSF), i.e. use solution $\mathcal{Z}$ . Such modifications may depend on information considered as "network operator specific" by the recommendations.

We introduce a new functional entity in the Distributed Functional Plane (DFP) which we call the Service Modeler and Observer Function (SMOF). Figure 12 shows how the SMOF is inserted in the DFP.

Notice that the SMOF sees the information flows on the D, E and F relationships (see Q.1211 in [1]). However, this is transparent to the SCF, SRF, SDF and SSF. This corresponds to solution $\mathcal{X}$ .

The SMOF can play the role of an SCF for the SSF. It can do so through the D-relationship. This corresponds to solution $\mathcal{Y}$.

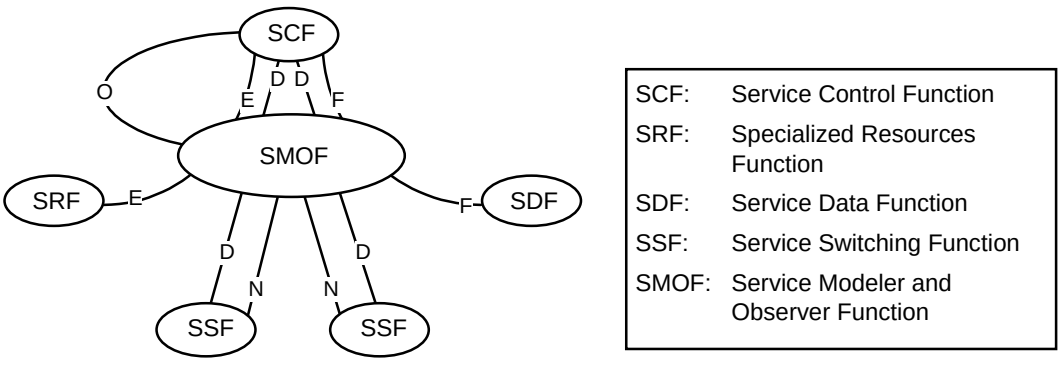


**Figure 12: The SMOF in the CS-1 DFP and the related reference points**

To address the situation where solution ≳ is needed, we introduce two new reference points: the relationship between the SSF and the SMOF, where the SSF could communicate some specific information to the SMOF, coincides with the reference point N; the corresponding relationship between the SCF and the SMOF is the reference point O.

Details on the analysis and design steps of the implementation of an SMOF can be found in [33].

### 3.3.2 The Model State Manager

We have seen that the mapping function must be written by hand; it heavily depends on the model of the service and the network on which this service is implemented. We consider here a simple model for the Basic Call service, where the network is represented by ports between which calls can be established. Each port is identified by a number which can be used as the dial number to reach the port (see Figure 3).

The behavior of our Basic Call service is inspired by the CS-1 call model as presented in recommendation Q.1214 [1] (see also Figure 4). In order to be able to present the abstract behavior of the implemented service, the *model state manager* needs to be informed of all transitions in the basic call state model (BCSM). It must even know when the handling of a given detection point terminated. Therefore we introduce the concept of "proceeded notifications" (PN) corresponding to notifications which are sent to the *model state manager* just after a detection point was completely handled, i.e., just before the next point in call (PIC) is entered (see

Figure **13**).

Figure **13** shows the terminating BCSM as it is seen by the *model state manager*. The latter must be informed of the occurrence of each DP and each PN. It is the responsibility of the *system interface* to find the necessary information in the network where the service is implemented, in order to be able to inform the *model state manager* of all relevant events. This information can be collected by observing the INAP messages exchanged between the SCF and the SSFs. In order to be sure that INAP messages are sent for all the detection points, the *system interface* can play the role of an SCF, arming necessary DPs.

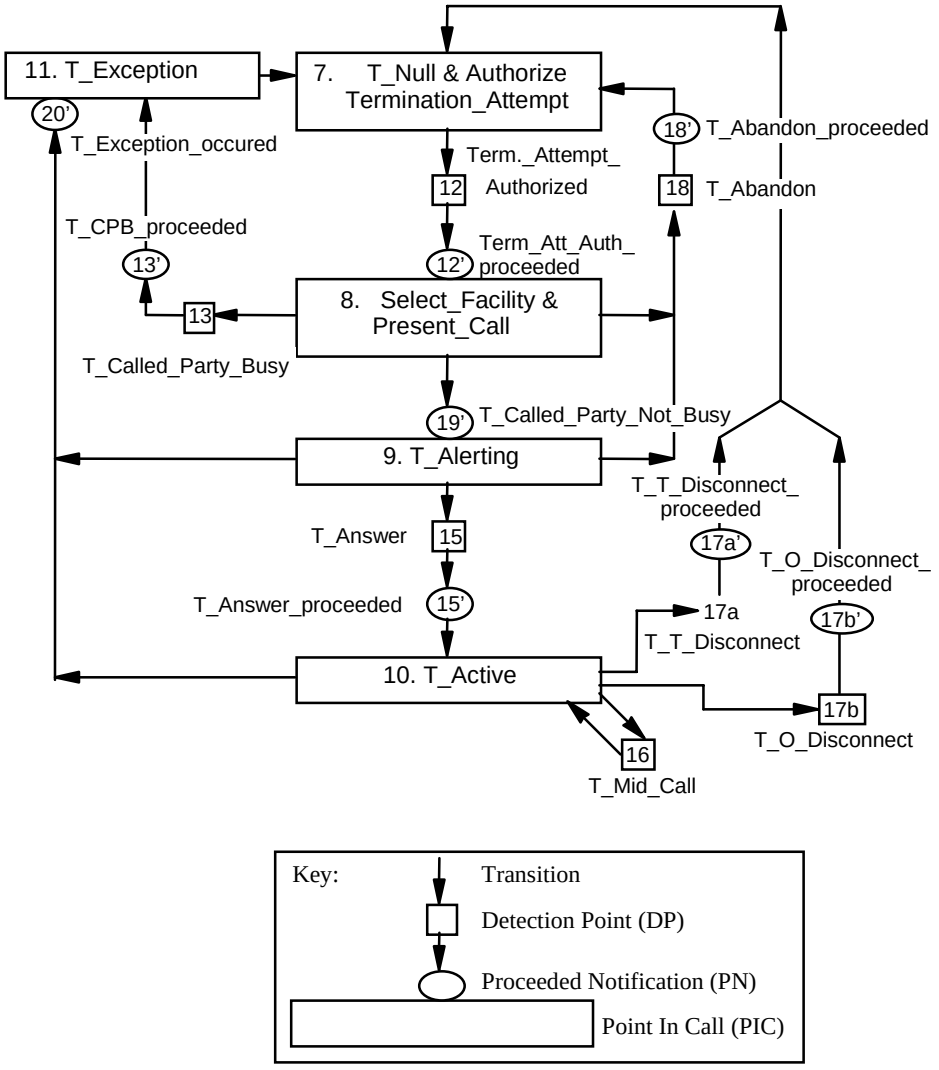


**Figure 13: The terminating BCSM seen by the *model state manager***

Notice in

Figure **13** PN 19' *(T_Called_Party_Not_Busy)* which corresponds to an event which cannot be deduced by observing standardized INAP messages. To be able to inform the *model state manager* of the occurrence of PN 19', the *system interface* must either have access to non standardized information flowing between the functional entities in the IN, or use method ≥ presented above, i.e., modify the SSFs so that they send special notifications between PICs 8 and 9 in the terminating BCSM.

### *3.4 Conclusion*

We have shown how an entity called the *Service Modeler and Observer* (SMO) can be introduced into a real network in order to check the properties of a service specified during the analysis phase. We have shown how an SMO can be introduced into a CS-1 intelligent network. The basic ideas for function μ, which computes for each state of the real system the corresponding state of the specification, were briefly presented. This function has been completely specified in FUS++ which allowed us to validate it. We are currently implementing these concepts on a simulated intelligent network platform which we developed for Swiss Telecom PTT [33].

Combined with a good test scenarios generator, the SMO can be an invaluable aid in increasing the implementation quality of a given set of services.

# 4. Conclusion

The goal of this work has been to address the problem of the validation of IN services at the specification and implementation levels. To support this approach, an object-oriented method coming from the software engineering world has been selected: Fusion. Our claim is that this kind of environment is better suited for our purpose than techniques borrowed from the protocol engineering community such as LOTOS or SDL. In fact, we believe that the most important issue for service specification is to help the specifier to structure his or her ideas; the acceptance of a given specification environment is of crucial importance, and we consider validation as being an interesting additional function, rather than being the primary target.

However, the analysis phase of Fusion, during which the specification is produced, is not formal enough to allow validation. Therefore, it has been necessary to formalize the notations used in the specification in order to obtain a specification language with well defined semantics. We have called this language FUS++. In Addition to the formalization aspect, FUS++ offers two enrichments with respect to Fusion:

- the possibility of specifying several services independently from each other, and of merging them automatically in order to obtain a system containing several services; this possibility is of high interest for the study of feature interactions;
- the possibility of specifying properties that the system must satisfy in linear time temporal logic (LTL).

As yet, no validation tool has been designed for the validation of FUS++ specifications. However, using the semantics of a specification that we have defined, it is possible to translate a FUS++ specification into another language for which validation tools do exist; this is the principle of the F2P compiler, which converts a FUS++ specification into Promela code. This approach makes it possible to use validation tools which are regularly updated and thus to integrate new ideas from the specific domain of validation algorithms.

Through our preliminary experiments, we have been able to draw some conclusions from the usage of FUS++:

- it is a powerful method for the specification of services;
- it facilitates the detection of both internal contradictions within each service and interactions between services;
- the validation phase helps the specifier to get a deeper understanding of the service, which can be very useful in the following phases (design and implementation).

One of the most challenging problems in formal approaches is to maintain the coherence between the specification and the implementation. The method we have proposed here is quite pragmatic: a dedicated entity, called the Service Modeler and Observer, is in charge of computing the mapping function between the events observed in the implemented system and the executable specification. In this way, it is possible to check at run time whether what happens at the implementation level is compliant or not with the properties expressed during the analysis phase.

The work presented in this paper is currently continuing in the framework of the ErnesTINA project. In the ErnesTINA project [36] we propose an integrated approach to facilitate the validation of TINA (Telecommunications Information Networking Architecture) services by verifying at run-time that the service implementation is not violating certain predefined properties [37]. In the ErnesTINA project, there is no mapping to a formal specification of the system (such as the mapping of FUS++ to Promela). Only the properties are specified, based on LTL. Therefore there is no Model Manager and the events retrieved from the implemented system by the observation are directly used to feed the different properties checkers.

## 5. Acknowledgments

The authors would like to thank Yow Jian Lin,Thierry Cattel and Shawn Koppenhoefer for their help and their very useful and constructive comments.

## 6. Bibliography

[1] ITU-T General Recommendations on Telephone Switching and Signalling, “Intelligent Network” – Q-Series Intelligent Network Q.1200 - Q.1290, ITU 1993.

[2] William J. Barr, Trevor Boyd, and Yuji Inoue. “The TINA Initiative”. *IEEE Communications Magazine*, pp. 70 - 76, March 1993.

[3] Service Creation in an Object-oriented Reuse Environment, SCORE, RACE Ref: 2017. “Report on Methods and Tools for Service Creation” (First Version) Part I: Summary. Deliverable D203 - R2017 / SCO / WP2 / DS / P / 027 / b2. SCORE Identifier D2031. January 27th, 1994.

[4] Race Project R1068, RACE Open Service Architecture, ROSA, 2nd Deliverable, “Specifying Services using Objects”, 1989.

[5] Linda Strick and Jens Meinköhn. “Enterprise Modelling for the Design of Telecommunication Management Systems”. *TINA’95*, Conference Proceedings, Vol. 2, pp. 359 - 369,1995.

[6] J. Insulander, P. Schoo, I. Tönnby, S. Trigila. “An Architectural Approach to Integrated Service Engineering for an Open Telecommunication Service Market”. *Proceedings of the International RACE IS&N Conference on Intelligence in Broadband Services and Networks*, November 23-25, Paris, France,1993.

[7] T.F. Bowen, C.H. Chow, F.S. Dworak, G.E. Herman, N. Griffeth and Y. J. Lin. “The Feature Interaction Problem in Telecommunications Systems”. *Proceedings of the Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, pages 59 - 62, Burnemouth, United Kingdom, July 1989.

[8] Derek Coleman et al. *Object-Oriented Development – The Fusion Method*. Prentice Hall International, 1994.

[9] Colin H. West. “Protocol validation – principles and applications”. *Computer Networks and ISDN Systems* , pp. 219 - 242, May 1992.

[10] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[11] Pierre Combes, Simon Pickin. “Formalisation of a User View of Network and Services for Feature Interaction Detection”. *Feature Interactions in Telecommunications Systems*, IOS Press, pp. 120 - 135, 1994.

[12] Mohammed Faci. “Detecting Feature Interactions in Telecommunications Systems Designs”. PhD Thesis, Department of Computer Science, University of Ottawa. 1995.

[13] Subodh Bapat. “Object-Oriented Networks. Models for Architecture”, *Operations and Management.* Prentice Hall, 1994.

[14] P. Coad and E. Yourdon. “Object-Oriented Analysis”. 2nd ed. *Yourdon Press*, Engelwood Cliffs, NJ, 1991.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. “Object-Oriented Modeling and Design”. *Prentice Hall International*, Englewood Cliffs, NJ, 1991.

[16] Pierre-Alain Etique, Tuncay Saydam, Jean-Paul Gaspoz and Jean-Pierre Hubaux. “Validation of an Object-Oriented Service Specification for the Intelligent Network”. *Proceedings of TINA’95*, Melbourne, February 13 - 16, 1995.

[17] Pierre-Alain Etique, Jean-Pierre Hubaux et Tuncay Saydam. “Vérification et validation de services de télécommunications spécifiés par une méthode orientée objets”. *Actes de CFIP’95, Colloque Francophone sur l’Ingénierie des Protocoles, Rennes*, 9 - 12 mai 1995.

[18] Zohar Manna, Amir Pnueli. “The Temporal Logic of Reactive and Concurrent Systems – Specification”. *Springer-Verlag*, 1992.

[19] E. Jane Cameron, Nancy D. Griffeth, Yow-Jian Lin, Margaret E. Nilson, William K. Schnure and Hugo Velthuijsen. “A Feature Interaction Benchmark for IN and Beyond”. *Feature Interactions in Telecommunications Systems*, IOS Press, pp. 1 - 23, 1994.

[20] SCORE-Methods and Tools. “Report on Method and Tools for Service Creation” (Second Version) Volume I: Service Interaction Analysis. Deliverable D204 - R2017/SCO/WP2/DS/P/028/b2. RACE project 2017 (SCORE). December 16th 1994.

[21] F.Joe Lin, Yow-Jian Lin. “A Building Block Approach to Detecting and Resolving Feature Interactions”. *Feature Interactions in Telecommunications Systems*, IOS Press, pp. 86 - 119, 1994.

[22] Pierre-Alain Etique. “Service Specification Verification and Validation for the Intelligent Network”. *PhD Thesis*, Swiss Federal Institute of Technology Lausanne, 1995

[23] Kristofer Kimbler and Claes Wohlin. “A Statistical Approach to Feature Interaction”. *TINA’95*, Conference Proceedings, Vol. 1, pp. 219 - 230, 1995.

[24] Pascal Gribomont and Pierre Wolper. Chapter 4 Temporal logic. “From Modal Logic to Deductive Databases”, A. Thayse Editor, *Wiley*, pp. 165 - 233, 1989.

[25] Roland Groz. “Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulations”. *Ph.D. Thesis from l’Université de Rennes*, November 1988.

[26] Refik Ahmet Molva. “Conception et réalisation d’un observateur d’architectures multicouches dans les réseaux d’ordinateurs”. Thèse de doctorat, Université Paul Sabatier, Toulouse, 1986.

[27] Boris Makarevitch, ”Resolving Service Interactions by Service Components”, Feature Interaction In Telecommunications Systems III, K.E. Cheng and T. Ohta (Eds.), IOS Press, 1995.

[28] C. Abarca, P. Farley, J. Forsloew, T. Hamada, et. al., “Service Architecture”, TINA-C Document, Version 4.0, October 1996.

[29] Ruth Malan, Reed Letsinger and Derek Coleman, “Object-Oriented Development at work, Fusion in the Real World”, Prentice Hall, 1996.

[30] Grady Booch, James Rumbaugh, “Unified Method for Object-Oriented Development”, Rational Software Corporation, Santa Clara, USA, 1995.

[31] L. Lamport,”The Temporal Logic of actions”, ACM Transactions on Programming Languages and Systems, vol 16, no 3, pp 872-923, May 1994.

[32] Johan Blom, Roland Bol and Lars Kempe, ”Automatic Detection of Feature Interactions in Temporal Logic“, Feature Interactions in Telecommunications Systems III, K.E. Cheng and T. Ohta (Eds.), IOS Press, 1995.

[33] Xavier Logean, “Improving Confidence Service Implementation in an Intelligent Network”, EUNICE’96, Swiss Federal Institute of Technology, 1996.

[34] C. Jard, J.F. Monin and R. Groz, “Development of VEDA: a Prototyping Tool for Distributed Algorithms”, IEEE trans. on Software Engineering, Vol. 14 , no 3, March 1988.

[35] E. Fromentin, “Détection de propriétés instables dans les exécutions réparties, application à la mise au point des programmes répartis”, Ph. D: Thesis from l’Université de Rennes, 1996.

[36] X. Logean, F. Dietrich and J.-P. Hubaux, ”TINA service Validation: the ErnesTINA project”, Technical Report No SSC/1997/028, Communication Section Division, Swiss Federal Institute of Technology, Lausanne, 1997.

[37] F. Dietrich, X. Logean, J.-P. Hubaux, “Testing Temporal Logic Properties in Distributed Systems”, Technical Report No SSC/1997/027, Communication Section Division, Swiss Federal Institute of Technology, Lausanne, 1997.

# 7. Appendix: FUS++ Syntax

```
System        = {ObjectModel | InitPart | OperModel | InvDecl | TempProp}.

ObjectModel   = {Class | Relation | TypeDef | ExtensionDef | Agent}.

Typedef       = "typedef" type ident ";".
type          = BasicType | SubrangeType | SetType | enumeration.
BasicType     = ident.
SubrangeType  = "range" "{"number ".." number "}".
SetType       = "set" "of" BasicType.
enumeration   = "enum" EnumContent.
EnumContent   = "{"ident {"," ident} "}".
ExtensionDef  = "typeExtension" [ident] enumeration ident ";".

Agent         = "AGENT" ident ";".

Class         = "CLASS" ident ClassDef.
ClassDef      = "["number"]" [":" ident {"," ident}] [NonDet] [Attributes].
Attributes    = "ATTRIBUTES" AttrDecl {AttrDecl}.
NonDet        = "[~]".

AttrDecl      = BasicType ident [AttrInit] {"," ident [AttrInit]} ";".
AttrInit      = NonDet | "=" expression.

InvDecl       = "SYSTEM" "INVARIANTS" CompoundStmnt.
CompoundStmnt = [ident "::"] "{" {stmnt} "}".
stmnt         = FinalStmnt
                | IfStmnt
                | WhileStmnt
                | ForEachStmnt
                | CompoundStmnt
                | SignalStmnt
                | AttrDecl
                | ident ":" stmnt.
FinalStmnt    = [ Assignement
                | WithStmnt
                | WithoutStmnt
                | AssertStmnt
                | ProcCall
                | SelectStmnt
                | SendStmnt
                | ReturnStmnt
                | DelStmnt
                | RunStmnt
                | UIStmnt
                | PICDecl
                | GotoStmnt] ";".

Assignement   = varRef "=" expression.
WithStmnt     = ident "with" "(" Roles ")".
WithoutStmnt  = ident "without" "(" Roles ")".
AssertStmnt   = "assert" expression.
SelectStmnt   = "select" varRef "where" expression.
ForEachStmnt  = "foreach" varRef ["where" expression] "do" stmnt.
UIStmnt       = "UserInteraction" varRef "->" margs "<-" ident.
PICDecl       = ":" ident FormalParams.
GotoStmnt     = "gotoPic" ident ActualParams.
SignalStmnt   = "signal" margs
                "refValue" varRef
                "default" CompoundStmnt.

ProcCall      = ClassicCall | hasOp | RelatantFnc | NewFnc | ChanCheck
                | AtPred | InvalidPred.
ClassicCall   = ident ActualParams.
ActualParams  = "(" [ExprList] ")".
ExprList      = expression {"," expression}.
hasOp         = ident "has" "(" Roles ")".
Roles         = OneRole {"," OneRole}.
OneRole       = ident ":" expression.
RelatantFnc   = [ident] "relatant" "of" "(" Roles ")"
                "in" ident ["[" expression "]"].
NewFnc        = "new" ident ["(" expression ")"].
ChanCheck     = varRef "?" "[" margs "]".
AtPred        = "at" ident.
InvalidPred   = "invalid" varRef.

SendStmnt     = varRef "!" margs.
margs         = ident [ActualParams].
ReturnStmnt   = "return" expression.
DelStmnt      = "delete" expression.
RunStmnt      = "run" ident "(" expression ")".

IfStmnt       = "if" "(" expression ")" stmnt ["else" stmnt].
```

```
WhileStmnt    = "while" "(" expression ")" stmnt

Relation      = "RELATION" ident RolesDecl.
RolesDecl     = RoleDecl {RoleDecl}.
RoleDecl      = "ROLE" [ident] ident "[" Cardinality "]" ";".
Cardinality   = "*" | "+" | RangeEnum.
RangeEnum     = OneRange {"," OneRange}.
OneRange      = number [".." number].

number        = digit {digit}.
digit         = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
ident         = letter {letter | digit}.
letter        = "a" | .. | "z" | "A" | .. | "Z" | "_".
expression    = "(" expression ")"
                | expression binop expression
                | unop expression
                | expression "cum" CompoundStmnt.
                | terminalExp
                | ProcCall.
binop         = "==" | "!=" | "in" | "<" | "<=" | ">" | ">=" | "<<" | ">>"
                | "+" | "-" | "||" | "|" | "*" | "/" | "%" | "&&" | "&"
                | "U" | "W" | "=>" | "isMemberOf" | "isKindOf".
unop          = "~" | "-" | "!" | "O" | "[]" | "<>".
terminalExp   = ident | number | boolVal | set | varRef | "NIL".
boolVal       = "true" | "false".
set           = [ident] "{" [ExprList] "}".
varRef        = [ident "::"] ident ["." ident].

TempProp      = "TEMPORAL" "PROPERTY" "{"
                {SpecVarDecl} AssertStmnt [";"] "}".
SpecVarDecl   = BasicType ident {"," ident} ";".

InitPart      = "INIT" CompoundStmnt.

OperModel     = {MsgType | IntEvDecl | Function | SigResp | Oper}.
MsgType       = "mtype" "=" MsgEnum [";"].
MsgEnum       = "{"ident [MsgContent] {"," ident [MsgContent]} "}".
MsgContent    = "(" BasicType {"," BasicType} ")".
IntEvDecl     = "InternEvents" "=" MsgEnum [";"].
Function      = "FUNCTION" BasicType ident FormalParams CompoundStmnt.
FormalParams  = "(" [BasicType ident {"," BasicType ident}] ")"
SigResp       = "response" ident "to" "signal" ident FormalParams
                ["prio" number] CompoundStmnt.
Oper          = Operation [Description] [Reads] [Changes] [Sends] Assumes
                Result.
Operation     = "Operation" ":" ident.
Description   = "Description" ":" garbage.
Reads         = "Reads" ":" {"supplied" BasicType ident ";"} garbage.
Changes       = "Changes" ":" garbage.
Sends         = "Sends" ":" garbage.
Assumes       = "Assumes" ":" CompoundStmnt.
Result        = "Result" ":" CompoundStmnt.
```