Feature Interaction Detection Using Testing and Model-Checking Experience Report

Lydie du Bousquet

Laboratoire LSR-IMAG, BP 72,
38402 Saint Martin d'Hères cedex, France
tel: +33 4 76 82 72 32, fax: +33 4 76 82 72 87
ldubousq@imag.fr
http://www-imag.lsr.fr/

Abstract. We present an experiment in feature interaction detection. We studied the 12 features defined for the first feature interaction contest held in association with the 5th international Feature Interaction Workshop. We used a synchronous approach for modeling features, and both, a model-checker and a test generator for revealing interactions. The first part of the paper describes the feature modeling. The second part deals with the feature interaction detection carried out with a testing tool, and the last part addresses the use of a model-checker for the detection.

1 Introduction

Telecommunication software is a variety of safety-critical software. Its requirements in terms of dependability are high since a malfunction may result in environment harm. The disastrous financial consequences of failures impose on this kind of software strong correctness and quality of service constraints.

This is why modeling, analysis and risk assessment activities take a large part of its development process. For critical components, the requirements engineering phase usually ends in a formal specification which is provided in some logics; therefore, validation can be performed in a very rigorous and formal way using proof tools and/or specification-based testing techniques. Examples of such critical pieces are protocols and telephone services.

The expansion of new telephone supplementary services (called features) has reinforced the need for formal, mathematically sound and well equiped specification and validation techniques. Indeed, a new supplementary service can change the behavior of pre-existing ones, alter them, or even crash the system. The phenomena are known as the "feature interaction problem" in the telecommunication industry [9].

Besides, one can note that a telecommunication system has most of the characteristics of reactive programs [18]. Such programs continuously react with their environment at their own speed. They must satisfy some strong timing dependencies between external events and their role is to bring about or maintain desired relationships in the environment.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 622–641, 1999.
© Springer-Verlag Berlin Heidelberg 1999

We applied a reactive synchronous approach to the feature validation problem, using Lustre [6]. Lustre is an executable specification language which can be used for programming and formally verifying (using model-checking) [11] as well as for testing [7] synchronous critical software.

In 1998, Nancy Griffeth et al. organized the first feature interaction detection tool contest [10]. This contest was held for the 5th International Workshop on Feature Interactions. We participated in this contest and our tool won the "Best Tool Award". Our approach consisted in two major points: a synchronous technology to express a model of a telecommunication system and using test techniques to detect feature interactions.

We decided to favor a testing approach for a couple of reasons.
(1) Feature interaction detection can be viewed as finding some errors in a program, and testing is the process of executing a program with the intent of finding errors [15]. Moreover, testing allows to establish the fitness or the worth of a software product for its operational mission, and interaction occurrences are strongly connected to the telecom system operations [3].
(2) It is our experience that model-checking often fails for lack of time or memory. Besides, model-checking is not appropriate to evaluate the adequacy of a property. When searching for interaction and no interaction is found, the relevance of the property must be questioned. Deciding on this relevance can be easily performed using a testing tool.

We have applied a model-checker to confirm the results which have been obtained with our testing tool, contrary to most situations where testing and proving (by deduction or model-checking) are jointly used.

In this paper, we first sum up the contest instructions (section 2). Then, we describe the synchronous approach (section 3). In sections 4 and 5, we detail three major points of our model: our definition of interaction, our approach principles and our feature composition operation. Section 6 describes the first part of our validation experiment using a test generator, and section 7 describes the second part of the validation experiment using a model-checker on the same model.

2 Feature Interaction Detection Tool Contest

The goal of this contest was to compare different feature interaction detection tools according to a single benchmark collection of twelve features. The contest had two phases. The first one required the contestants to analyze the ten first features in a five month period. The second phase required the analysis of the two last features, in sixteen days.

The contest instructions were made of the description of a network model, given as a collection of black boxes communicating with each other via defined interfaces, the description of a feature description formalism, the specification of

the basic call service feature, and an informal description and the specification of the twelve features to be examined.

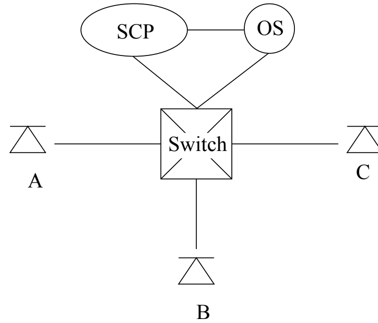


Fig. 1. The network specification

The network consists of 4 types of elements (fig. 1): some end-user equipments (telephones), a Switch, a Service Control Point (SCP) processing IN features, and an Operations System (OS) that does the billing.

A user can perform 4 actions: go off the hook, go on the hook, dial a number, a flash-hook. The telephones are assumed to have a flash button.

A service formal specification is represented as a Chisel sequence diagram [1]. Such a diagram is a directed graph whose vertices are labeled by events or messages exchanged at the various interfaces. A diagram defines the set of event sequences of a single call, one for each path through the graph. Event sequences involving multiple calls can be interleaved to define the global system activity. Figure 2 represents the Plain Old Telephone Service (POTS) diagram which stands for two phones and a single call, originated by party A.

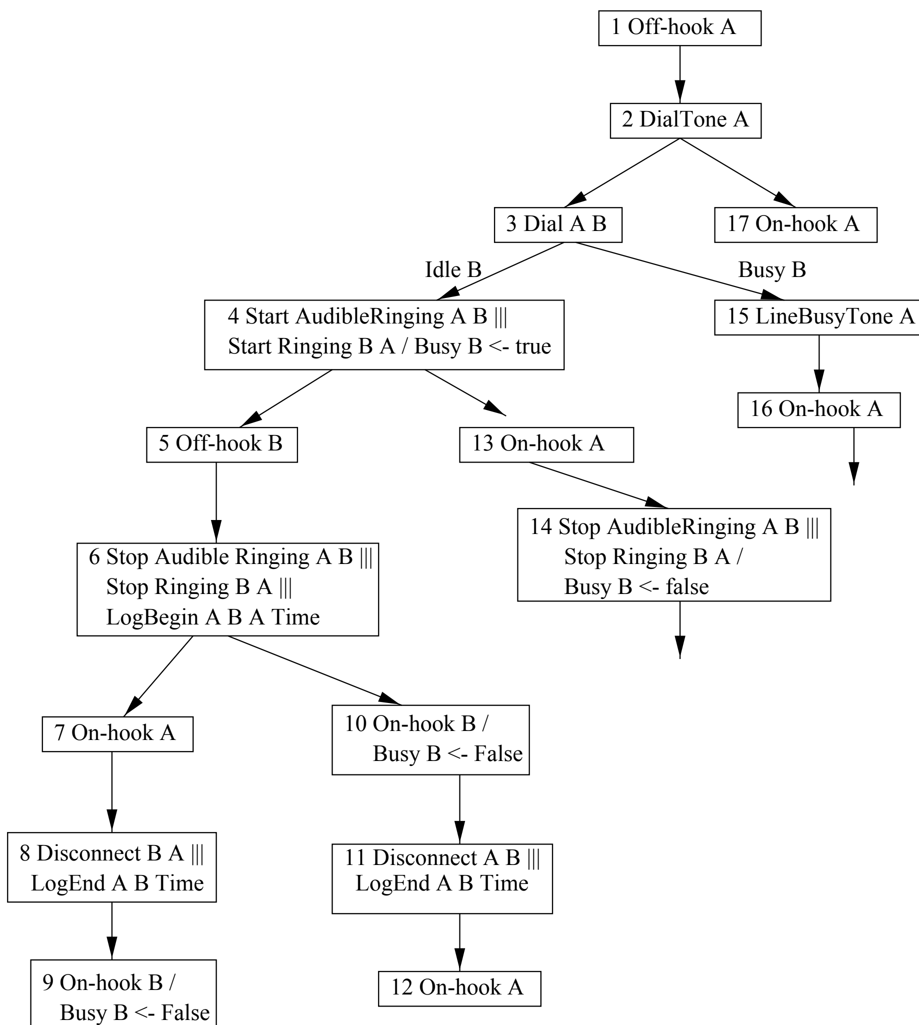
3 Synchronous Approach and Lustre Language

To enter the contest, we have used a synchronous framework. In this section, we describe the synchronous approach.

3.1 The Synchronous Software Technology

Synchronous programs [2] are a sub-class of reactive software programs: every reaction of a program to its inputs is theoretically instantaneous. Synchronous programs have cyclic behaviors: at each tick of a global clock (also called instant of time), all inputs are read and all outputs are emitted.

Synchronous languages rely on the ideal synchrony hypothesis, which says that synchronous machines have zero-time response delay and synchronous systems are systems of dynamical equations. Practically, the synchrony hypothesis



Variables:
Busy A: true between an Off-hook A event and the next On-hook A event; between a Start Ringing A B event and the next Stop Ringing A B event, if no Off-hook A intervenes; or between a Start Ringing A B event and the next On-hook A.
Ringing A B: true between a Start Ringing A B event immediately following a Dial B A event and the next Stop Ringing A B event.
AudibleRinging A B: true between a Start AudibleRinging A B event immediately following a Dial A B event and the next Stop AudibleRinging A B event. All of the POTS event sequences start and end with Busy A = False (Idle A = True).

Fig. 2. POTS formal description (Chisel diagram)

consists in checking that the program always reacts quicker than its environment. From the software developer's point of view, the main characteristics of the synchronous languages are [2]: a precise mathematical semantics, a flexible concept of hierarchy, precise notions of modularity and encapsulation, and an automatic generation of executable code.

Synchronous languages are well-adapted to the specification and programming of reactive software [11]. They allow to avoid the combinatorial explosion problem, which impairs the approaches based on parallel and communicating processes. Indeed, all parallel components of a synchronous system react simultaneously and, thus, their executions are not intertwined. An additional consequence of this characteristic is that all state transitions (which take place at each reaction) become visible.

Lustre

Our work is more precisely based on Lustre, a synchronous declarative data-flow language [4]. Lustre is an executable specification language. It corresponds to a linear past temporal logic [17] which offers usual arithmetic, boolean and conditional operators and two specific operators: **pre**, the "previous" operator, and -> the "followed-by" operator[1]. Lustre allows the specifier to define its own logical and/or temporal operators to express invariants or safety-oriented properties. Those properties are used for the system validation.

3.2 Reactive System Specification

An important feature of a reactive system is that it is developed under assumptions about the possible environment behavior. So a complete specification of a reactive system is a three step process. First, the environment has to be described, in order to be able to specify system reactions only in response to valid environment behaviors. The second step provides a set of properties, which describes the system requirements. Those properties are commonly safety-oriented. The last step objective is to provide a functional specification of the reactive software. If the functional specification is performed in Lustre, it can be compiled into an executable code.

3.3 Reactive System Validation

We focus here on the high level validation phase of reactive systems, which consists in showing that the functional specification, providing the specification of the environment, satisfies the properties.

Thus, the reactive system validation is done with respect to environment constraints. Clearly, when one is not concerned with the system robustness,

[1] Let E and F be two expressions of the same type denoting the sequences $(e_0, e_1, ..., e_n...)$ and $(f_0, f_1, ..., f_n, ...)$; $\mathbf{pre}(E)$ denotes the sequence $(nil, e_0, e_1, ..., e_{n-1}...)$ where nil is an undefined value. $E \rightarrow F$ denotes the sequence $(e_0, f_1, ..., f_n...)$.

it makes no sense to take into account unrealistic environment behaviors. For example, in a telecommunication system, it is physically impossible to go on the hook twice without going off the hook in between. Consider a program which simulates the network reactions when some users go on the hook, go off the hook or dial a number. This program should observe for each user a sequence of actions among which "go off" et "go on" actions alternate.

The validation tool has to produce a verdict which indicates whether the program to be validated satisfies the properties under the environment assumptions (fig. 3).

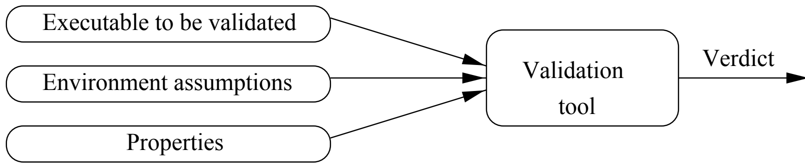


Fig. 3. Reactive system validation method

Lustre has been equipped with various dedicated validation tools: for example, a model-checker (Lesar) [11, 12] and a testing environment (Lutess) [7, 16]. Lutess is presented in section 6, and Lesar is presented in section 7.

4 Feature, Feature Composition, and Feature Interaction

Our definition of a feature relies on the feature interaction detection contest instructions. A single feature is a modification of the Plain Old Telephone Service (POTS). Thus, the POTS is successively redefined by the application of features. A feature (or a service) waits for an input and then reacts to it by producing outputs.

A general framework has been defined in [5], which inspired our approach. In this framework, let $F_1, F_2 ... F_n$ be a set of feature specifications, and N the network specification. Let $\oplus$ denote the composition of the network, and one or possibly several features ($N \oplus F_1 \oplus ... \oplus F_n$). Let $P_1, P_2 ...$ be a set of formulae expressing the respective feature requirements. Let $S \models P$ denote that the specification S satisfies (is a model of) the formula P. By definition, we say that there is an interaction when

$$\begin{cases} N \oplus F_i \models P_i, 1 \leq i \leq n \\ N \oplus F_1 \oplus ... \oplus F_n \not\models P_1 \wedge ... \wedge P_n \end{cases} \qquad \text{(T)}$$

The basic principle of our approach consists of building an executable description of the system to be studied. The network in the sense of (T) is only made of POTS, and the composition operation consists of modifying the POTS.

The POTS and each feature is represented as an automaton. This automaton results from the translation of the corresponding Chisel diagram. The automata are coded in Lustre.

Then, the composition of the POTS and the features is done in a "software unit" (fig. 4b). This unit is called a *logical telephone* (LT). There is one logical telephone for each user.

Logical telephones are gathered into a single program, which is called a *simulation program*. For the contest experiment, all the simulation programs have dealt with 4 users. These programs have been limited to up to two features (each logical telephone contains at most 2 different features). The simulation program is then used for feature interaction detection.

To detect occurrences of feature interactions, we expressed feature requirements as properties (in Lustre) and we defined interactions as the system's inability to satisfy these properties.

5 Modeling Choices

Each simulation program has been built applying the reactive system specification method (3.2). We had to identify the system environment, the functional specification of the system and some properties.

5.1 The Simulation Program Environment

The simulation program deals with the Switch and the Service Control Point. The Operations System is a passive element, which only receives some messages. Thus, it has been considered as part of the simulation environment. The end-user equipments (telephones) compose the other part of the environment (fig. 4a).

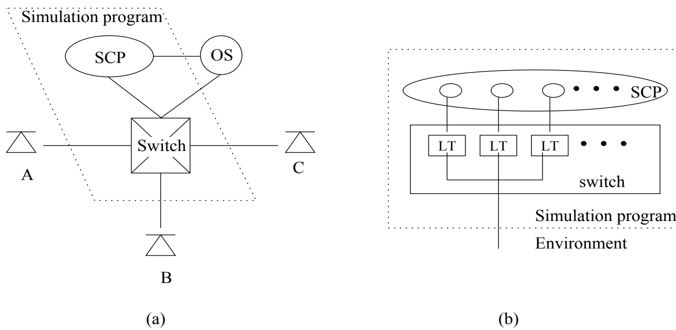


Fig. 4. Network executable specification

5.2 The Network Specification

The network executable specification (in the sense of (T)) relies only on the POTS specification. This specification is the result of a translation of a Chisel diagram into automata. The translation is performed in two steps. First, the different users involved in the feature execution and their respective actions are identified. Each user represents a "role" in the communication. For instance, POTS involves two roles: the caller and the callee.

For each role, the Chisel diagram is duplicated and then simplified to keep only the events with which it is concerned. For the POTS, we thus obtain two automata (fig. 5). Each automaton is then coded in a Lustre node. The node inputs are the user-to-switch messages. The outputs are the switch-to-user messages and the events produced for the billing systems.

Message encoding

There are 4 types of messages (ringing, display, billing messages and user action). All the messages are represented by boolean vectors. Moreover, for each message or event type, a specific message is coded. Indeed, at each cycle, a reactive synchronous program should read all its inputs and produce all its outputs. The absence of message or event is also a piece of information, and therefore, it must be "coded". For this purpose, a specific value (usually named "no_message") exists for each type of message.

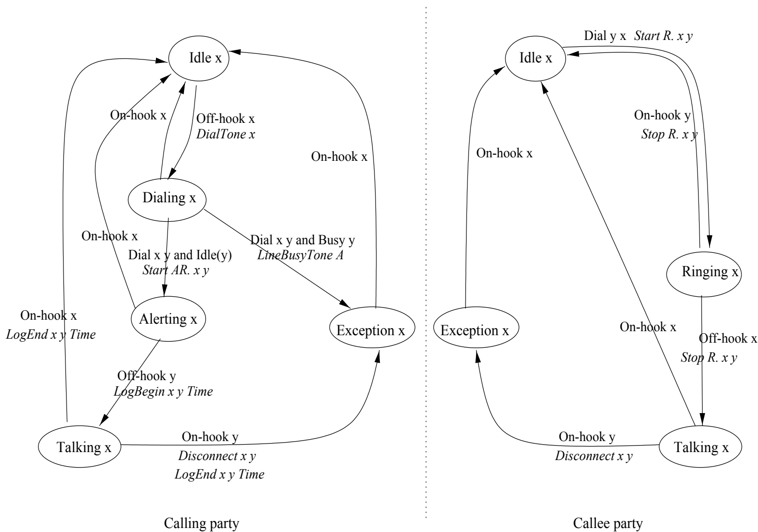


Fig. 5. The two automata for POTS

5.3 Feature Specification F_i

The feature executable specifications F_i were obtained systematically by generalizing the method used for POTS. For instance, Call Forwarding (CF) feature involves three users, the caller, the callee (the user to whom the CF is provided) and the forwarded-to user (the user to whom the call is redirected as a result of forwarding). Three automata were built.

As for POTS, the inputs of a feature executable specification are the user-to-switch messages. The outputs are also the switch-to-user messages and the events produced for the billing system. One supplementary output has been introduced: it is a boolean variable *af* (*active_feature*) which states whether the feature is active.

An inactive feature always emits the message "no_message". An active one can emit a message from the interface defined, or choose to answer "no_message". For instance, in the 4th vertex of POTS Chisel diagram (fig. 2), there are no display message and no billing event. So, a "no_message" can be emitted in two cases, and the boolean variable *af* allows us to make difference between them.

5.4 Feature Composition $\oplus$

The composition of the features is done at the switch level, in the logical telephones. We tried two different feature composition operations $\oplus_1$ and $\oplus_2$. Both rely on a multiplexing operation and on the *af* variables.

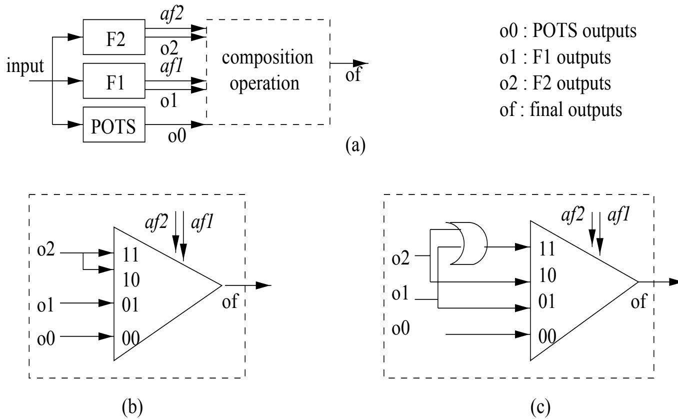


Fig. 6. Composition operations of POTS and two features

Let us consider two features $F1$ and $F2$. When the features are inactive (fig. 6a), the POTS outputs are chosen. When there is only one active feature, its outputs are chosen. When both features are active, we implemented two ways of selecting the outputs.

The first composition operation $\oplus_1$ (fig. 6b) gives priority to one feature over the other. The underlying hypothesis is: if there is an interaction between $F1$ and $F2$, it should be detected by $F1$ property violation.

The second composition operation $\oplus_2$ (fig. 6c) gives no priority to any of the features. When both features are active, the final output messages are built with a simple boolean *or* operator on each type of message. When both features produce no message, the final message is "no_message" (fig. 7a). When only one feature produces a message m, the final message is m (fig. 7b). When both features produce a message, the final message is the "composition" of both feature messages. If the messages are equal, the resulting message is unchanged; and if the messages are different, the resulting message is undefined (fig. 7c). Message codes were defined so that any composition of two different messages produces an undefined message. The second composition operation underlying hypothesis is: if there is an interaction between $F1$ and $F2$, it should be detected by $F1$ or $F2$ property violation or when an undefined message appears.

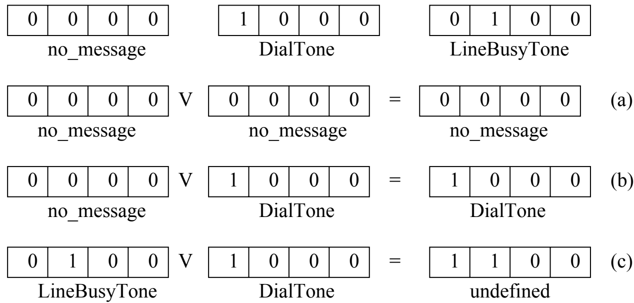


Fig. 7. Message composition for $\oplus_2$

5.5 Feature Properties

Two sets of properties are to be stated. The first one is intended to ensure the consistency of the services (POTS and feature) implementation in Lustre. For instance, the messages between the Switch and the telephones should be well-formed. The second one collects the POTS and the feature expected behavior properties. As the former set is implementation-dependent, we focus only on the latter. Those properties correspond to the feature informal requirements expressed in English.

For example, let us considered the Calling Number Delivery feature. The informal requirements are:

> [CND1] CND feature enables the subscriber's telephone to receive and display the number of the originating party on an incoming call.

[CND2] For the purpose of the contest, we assume the capability of delivering the number of the calling party whenever an idle called party receives the Ringing event.

and the corresponding properties, written in Lustre, are:

$$\begin{cases} \{ \text{ CNDsubs(x)} \Leftrightarrow \text{x is a subscriber of CND feature. } \} \\ \\ (1) \triangleright \text{CNDsub(x) and ConnectRequest(z,x) and pre Idle(x)} \Rightarrow \text{Display(x,z)} \\ (2) \triangleright \text{CNDsub(x)} \Rightarrow \text{Display(x,z)} \Leftrightarrow \text{StartRinging(x,z)} \end{cases}$$

6 Detection of Interactions Using Lutess

6.1 Lutess Testing Tool

Testing reactive systems can not easily be based on manually generated data. The software input data depend on the software outputs produced at the previous step of the software cyclic behavior. Such a process is facilitated by an automatic and dynamic generation of input data sequences.

Lutess [7, 16] is the testing tool we have developped to validate reactive synchronous software. It requires three elements: an environment description written in Lustre (A), a system under test (Π) and an oracle (B) (fig. 8). Lutess builds a random generator from the environment description and constructs automatically a test harness which links the generator, the system under test and the oracle. Lutess coordinates their executions and records the sequences of input-output relations and the associated oracle verdicts (trace collector module). Components are just connected to one another and not linked into a single executable code.

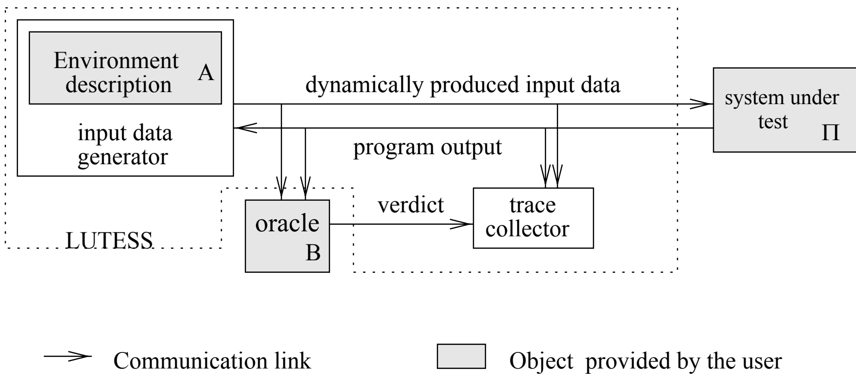


Fig. 8. Lutess

The system under test and the oracle are both synchronous and reactive executable programs, with boolean inputs and outputs. Optionally, they can be supplied as Lustre programs.

The test is operated on a single action-reaction cycle, driven by the generator. The generator simulates the environment behavior. At each cycle, the generator randomly selects an input vector which is valid with respect to the environment description (A). The chosen input vector is then sent to the system under test. This latter then reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is repeated. The oracle observes the program inputs and outputs, and determines whether the software specification is violated. The testing process is stopped when the user-defined length of the test sequence is reached.

Basically, the Lutess generator selection algorithm chooses a valid input vector in an equally probable way. In each environment state, each valid input vector has the same probability to be selected as the others. This selection method is easy to use and requires no supplementary work for the tester. However, this selection method is not always efficient. For instance, in a 4-user simulation program, each user can dial his own number 1 time out of 4.

Lutess allows the tester to guide the generator [7]. Three methods are proposed:

- the user can define some safety properties; the selection algorithm will select inputs which potentially drive the system under test toward those properties violation;
- the user can define some scenarios (behavioral patterns); the selection algorithm will select inputs which follow the scenario;
- the user can also define input statistical (partial) distribution; the selection algorithm will produce the inputs following the given distribution.

6.2 Use of Lutess for Feature Validation

The environment constraints were mainly constraints on sequences of operations that the user can perform (e.g. a user cannot go off the hook twice in a row without going on the hook in between). These constraints may be enriched with probability distributions or specific behavioral patterns corresponding to typical sequences of users' operations.

The detection procedure consists in running Lutess over a user-defined number of exchanges between the feature executable specifications and the environment simulator. During these exchanges, the interactions are detected by the oracle which signals them by setting the verdict to false. The exchange trace and the verdict trace are compiled by Lutess in a readable format which can be subsequently analyzed to find out the reason of the interaction.

6.3 Experiment Description

78 configurations were to be tested, each including one or two features [8]. In each case, 5 to 10 feature properties were available. The test process for each

configuration involved 10 to 20 sequences of 1000 to 10000 steps each. On the whole, each configuration has been evaluated on around 1 million test cases.

Building the generator corresponding to a given environment is the most expensive part of the testing process. In our experimentations, environments included from 32 to 45 constraints, plus up to 20 testing guides.

It was always possible to perform this computation and to run the test on a Sparc Ultra-1 station with 128 MB of memory. Maximum virtual memory required amounts to 100 MB. As the number of constraints describing the environment increases, the environment generation lasts longer. For the less-constrained environments that we produced, 6 seconds on CPU were necessary, while the most-constrained environments required 2000 seconds to be generated. As a comparison, a 1000 test run lasts 120 seconds once the generator has been built[2]. One can notice that here a trade-off has to be found: the more the environment is constrained, the more relevant is the test (since the whole test case is more realistic), but the longer is the generation.

6.4 Experiment Lessons

Test Automation

From the tester's perspective, the tool allows a significant relief by automating the test. Building the oracle appeared to be the most difficult part of the testing process. One has first to master the temporal logic paradigm, then to find out the better terms to express a given property. This requires an adequate training and experience.

From the specifier's point of view, Lutess has shown to be very helpful to debug specifications. First, Lutess has been used to validate the oracles: the oracle specifications are put in place of the unit under test and a human observation is substituted for the Lutess oracle. Second, prior to the search for interactions, the service specifications to be tested have been validated using oracle properties. For instance, in the specifications, some possible transitions were missing in a diagram, or an expected output message was never sent in a given situation. These problems were automatically exhibited as oracle violations.

Composition operation and feature interaction detection

We tried the two composition operations presented in section 5.2 (fig. 6b and 6c). It appears that the second one is more efficient in terms of "quantity of feature interaction detected". Let us consider the Calling Number Delivery (CND) and Call Forward (CF) features. CND properties are given in 5.5.

As it can be noticed in figure 9, CND feature displays the number of the subscriber calling party as soon as his number is dialed. Let A be a user who is both a CND and CF subscriber, let B the forwarded-to number of A. B is also a CND subscriber. Whenever A's number is dialed when B is idle, CF

[2] This second phase of the testing process is proportional to the sequence length.

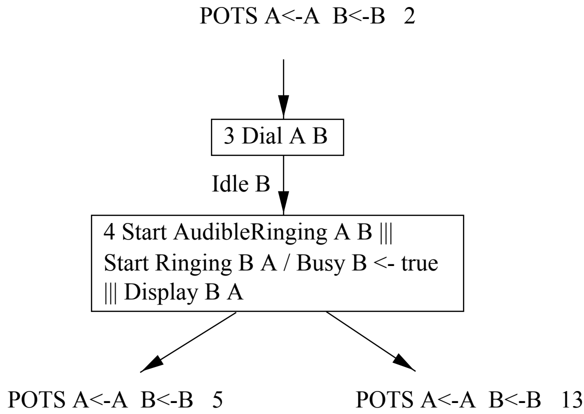


Fig. 9. CND formal description (Chisel diagram)

feature diverts the call. Moreover CND feature displays the caller's number on A's phone, but not on B's phone (because of the dialing action). Thus there are two interactions between CF and CND:

– After a call forwarding due to CF feature, the number of the calling party is not displayed even if the forwarded-to user is a CND subscriber.
– After a call forwarding due to CF feature, the number of the calling party is displayed on the callee's phone even if the forwarded-to user is idle.

When the first composition operation is used and CF feature is given the higest priority, (i.e. $POTS \oplus_1 CND \oplus_1 CF$) the second interaction does not occur. Indeed, when A's Call Forward feature is invoked, CND outputs are hidden by the CF's ones: CF is active and produce a "no_message" for the callee's display.

With the second composition operation, a feature can not completely mask another feature reaction. In this case, the second interaction appears and is detected by Lutess.

Feature subscription list configuration

An interaction may depends on the feature subscription configuration. If the subscription configuration is not adequate, some ineractions could be missed.

To use Lutess, the configurations of the feature subscription list were defined manually. Several possible configurations were chosen for each pair of features in order to decrease the probability to miss an interaction.

Importance of the properties

The Contest Committee defined a set of *valid* interactions. Those are the feature interactions that the Committee believes exist between the features as defined for the contest.

We found 83 interactions using Lutess. 3/4 of them were valid features. We found 14 invalid interactions and we also missed 19 interactions.

The invalid interactions we found result mainly from our interpretation of the informal requirements: we produced properties which were not considered to be "requirements" of a feature for the contest.

For instance, let us consider the Teen Line (TL) informal description: "TL restricts outgoing calls based on the time of the day, (i.e. hours when homework should be the primary activity). The restriction can be overridden on a per-call basis by anyone with the proper identity code." We deduced from this informal specification that *no call should be charged to the TL subscriber during the restriction hours, unless the correct identity code has been dialed.* Let us consider a user who is both a Call Forward and a Teen Line feature subscriber. When a call is forwarded during the restriction hours, the extra charges are "charged" and the TL property is thus violated. This is one of the invalid interaction we found.

Concurrently, the valid interactions we didn't find are also due to our properties, which were inadequate to reveal them. For instance, for most of the features, an on-hook action ends a phone call. This is not the case of Call Waiting (CW) feature. Thus there are some interactions between CW and some other features when a on-hook action is both interpreted as a step of the communication by CF and as the end of communication by the other features. The notion of "communication ending" was never taken into account since it was always implicit in the informal feature description.

When to stop testing ?

One of the major problem we had during this experiment phase was to decide when the test should be stopped. Since testing does not provide a definite verdict on the absence of the interaction, if no problem is found, one may wonder whether the test was significant enough to detect all interactions. Thus, to evaluate how significant were various test sequences, and increase confidence in testing, we have worked along two directions:

- since very often, a feature interaction appears during features invocation, we have produced some specific observers (i.e. properties which are included in the oracle) to measure how often a feature is invoked; thus we have checked whether interesting situations were explored during the test and so, if relevant data were produced;
- we have applied a model-checker, since we expected to evaluate the ability of such methods to detect feature interactions.

7 Detection of Interaction Using a Model-Checker (Lesar)

7.1 Lesar

Lesar [11, 12] has been designed to prove the correctness of a Lustre program with respect to some critical safety properties. A safety property usually states that a given situation will never occur or that a given statement should always hold.

- Safety properties can be checked on program abstractions. Let P a program and S a safety property to be proven. Let P' an abstraction of P. Intuitively, P' has more "behaviors" than P. Therefore, if S holds for P', it holds for P.
- Safety properties can be checked on program states rather than on execution paths.

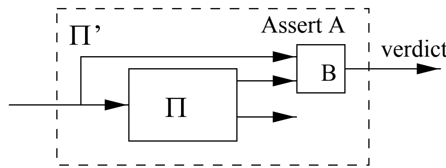


Fig. 10. Verification program structure

Lesar operates on a *verification program.* A *verification program* is a specific Lustre program Π' built out of three elements (fig. 10) [11]:

- a program Π to be verified,
- a property P expressed by a boolean expression B which should be invariably *true*,
- some assumptions on the environment (environment constraints); those assumptions are boolean expressions (A) which can be assumed to be always true.

These three elements are also those required by Lutess (fig. 8)

The verification is performed on a finite boolean state abstraction Π'' of the program Π'. Any numerical abstraction is ignored. Boolean expressions depending on numerical variables (such as comparisons) are considered nondeterministic. The verification principle is the following: proving that Π'' holds is equivalent to enumerating its finite set of states, checking that in each state (belonging to a path starting from initial state and on which the assertions are always true) and for each input vector, Π'' output evaluates to true.

7.2 Experiment Description

Lesar was used on the same computer configuration as Lutess. We had to modify the original simulation programs (Π) to use Lesar, since Lesar cannot prove properties involving numerical variables. The major modification was to translate integer variables into boolean vectors.

We also split up the feature properties (B). Most of our oracle properties were built as a conjunction of sub-properties. To make Lesar proofs shorter, we proved separately the sub-properties. Proving property $p \wedge q$, p and q being both safety-like properties, is equivalent to proving p and proving q.

The environment constraints (A) had not to be changed.

Only twelve pairs of features[3] have been studied with Lesar: CND+(CND, TCS, CFBL, CF, CELL), TCS+(TCS, CFBL, CF, CELL), CELL+CELL, CFBL+CFBL, CF+CF. The pairs were chosen in order to progressively increase the simulation program size. The size of a feature executable specification in Lustre is estimated by the number of Chisel diagram states (and/or transition) of the feature (CND has 2 states and 4 transitions, TCS has 3 states and 6 transitions, CF 16 states and 20 transitions, CELL has 21 states and 27 transitions, CFBL has 14 states and 19 transitions).

Lesar managed to build the boolean abstraction (Π''), for 8 pairs out of 12 and failed because of lack of memory or lack of time for 4 pairs: CF+CF, CF+CND, TCS+CFBL, TCS+CF. No other pair of features were therefore considered.

For the 12 listed pairs, Lutess found 12 interactions. Lesar exibited only 6 of these interactions. The 6 other interactions affect the 4 pairs of services for which Lesar was unable to build the boolean abstraction. Each time Lesar was able to produce a scenario counter-example, it revealed the same interactions as those produced by Lutess.

The simplest pairs of features (CND+CND and CND+TCS) required more than 2000 states and 17000 transitions to be explored. Lesar managed to prove each property without requiring the feature subscription list to be fixed. In this case, Lesar has proved to be more powerful than Lutess, since the subscription list has to be fixed for Lutess, and since the test only gives a partial result.

For the two pairs of features (CFBL+CFBL) and (CF+CND), Lesar found an interaction, but was not able to terminate the model-checking. In each case, several properties had to be considered, and Lutess exhibited a counter-example for one of these properties. Lesar was able to build the boolean abstraction, and exhibited the same counter-example as Lutess; it found the interaction after exploring less than 100 states of the verification program. However, Lesar was not able to explore the whole model (due to a lack of memory), to prove the properties for which Lutess exhibited no counter-example.

[3] Call Number Delivery (CND), Terminating Call Screening (TCS), Call Forwarding Busy Line (CFBL), Call Forwarding (CF), Cellular (CELL).

7.3 Experiment Lessons

Since some simulation programs were too big to be handled by Lesar, we simplified them. The first simplification we carried out was to fix the subscription list. Thus, several subscription situations had to be proved. For example, when proving CELL+TCS, one has to check the following cases:

- a user is both a Cellular and a Terminating Call Screening subscriber,
- a user is only a Cellular or a Terminating Call Screening subscriber,
- a user has no feature subscription...

The second simplification we carried out was to produce simulation programs using only 3 users. This simplification was useless since Lesar still fails to prove (CF+CF, CF+CND, TCS+CFBL, TCS+CF). Some other simplifications were not studied, since we wanted to keep the simulation program unchanged as most as possible.

In section 6.4, we showed that Lutess was helpful and convenient to debug feature properties, feature specifications or the environment specification. Lesar is less convenient: there was no way to check whether the system or the environment behavior was realistic, since feature properties do not describe the complete behavior of the simulation program. For instance, Lesar can not discover a missing transition (in the Lustre automata) if the properties do not explicitly concern these transitions. Also, we managed to prove a property under an over-specified environment. Actually, this property was false with respect to the correct environment description. This means that one can not consider Lesar verdict without simulating the environment description and/or the properties to be proved.

8 Conclusion

We have shown that within a unified validation framework (fig. 3), testing and model-checking are applicable to the feature interaction detection problem.

First, thanks to this framework, both methods can be used with few additional efforts. Testing can be used for environment and simulation program behavior validation. Proofs can be used to obtain a definitive verdict as far as it is possible. And when model-checking fails for lack of time and/or memory, testing can be used to get a partial verdict.

Second, this experiment has confirmed the fact that the feature detection problem is better tackled using testing than model-checking.

It appears that very few attempts to put in practice the synchronous approach to telephony systems have been carried out [14, 13]. The one reported in this paper is more thorough. Applying synchronous modeling with Lustre and testing with Lutess has revealed itself to be the most efficient approach in the first feature interaction detection tools contest.

9 Acknowledgments

I would like to thank Nicolas Zuanon, Remi Cave and Jérome Vassy who worked hard during the first part of the experiment, and Nicolas Halbwachs and Pascal Raymond who welcomed me at Verimag and helped me use Lesar during the second part of the experiment.

References

[1] A. Aho, S. Gallagher, N. Griffeth, C. Schell, and D. Swayne. Scf3^{TM}sculptor with chisel: Requirements engineering for communications services. In *Feature Interactions in Telecommunications Systems V*, pages 45–63. IOS Press, 1998.
[2] A. Benveniste and al. Synchronous technology for real-time systems. In *The 1994 Real-Time Conferences*, pages 104–122, Teknea, 1994.
[3] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.
[4] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL 87), Munich*, pages 178–188. ACM, 1987.
[5] P. Combes and S. Pickin. Formalization of a user view of network and services for feature interaction detection. In *Feature Interactions in Telecommunications Systems*, pages 120–135. IOS Press, 1994.
[6] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation : a synchronous point of view. In *Feature Interactions in Telecommunications Systems V*, pages 262–275. IOS Press, 1998.
[7] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *21st International Conference on Software Engineering*, Los Angeles, USA, May 1999. ACM.
[8] L. du Bousquet and N. Zuanon. Feature interaction detection contest: Lutess testing tool. technical report PFL, IMAG - LSR, Grenoble, France, 1998.
[9] N. D. Griffeth and Y.-J. Lin. Extending telecommunication systems: The feature-interaction problem. In *IEEE Computer*, pages 14–18, August 1993.
[10] N.D. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Otha. Feature interaction detection contest. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications Systems V*, pages 327–359. IOS Press, 1998.
[11] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language LUSTRE. *IEEE Transactions on Software Engineering*, pages 785–793, september 1992.
[12] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A-C. Glory. Specifying, Programming and Verifying Real-Time Systems, using a synchronous declarative language. In *Workshop on automatic verification methods for finite state systems, LNCS 407*, Grenoble, France, 1989. Springer Verlag.
[13] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Safety Property Verification of Esterel Programs and Applications to Telecommunications Software. In *7th Conference on Computer-Aided Verification*, 1995.
[14] G. Murakami and R. Sethi. Terminal call processing in Esterel. In *Proc. IFIP 92 World Computer Congress*, Madrid, Spain, 1992.
[15] G. Myers. *The Art Of Software Testing.* Wiley-Interscience, 1979.

[16] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, Monterey, USA, 1994.

[17] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In *Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Warwick, 1988. Springer Verlag.

[18] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems : a survey of current trends. *Current Trends in Concurrency, LNCS, Springer-Verlag*, 224:510–584, 1986.