An Algebraic Framework for the Feature Interaction Problem[1]

Mohammed Faci and Luigi Logrippo
University of Ottawa,
Telecommunications Software Engineering Research Group,
School of Information Technology and Engineering
E-mail: luigi@site.uottawa.ca

1 Motivation and Background

The problem of augmenting the functionality of a telephone system with new features, without causing unwanted interactions between the features, has received much attention during the last few years [BDCG89]. In industrial practice, detection is done not only by analyzing possible conflicts at the design stage, but also by running extensive libraries of test cases against the new system to see that it still behaves properly. Interestingly, the method we propose in this paper is similar to the one just described. However, being formal, our method allows precise reasoning, leading to precise criteria for choosing the set of test cases and analyzing the test results. Also, by relating the feature interaction problem to the well-known conformance testing problem, our approach makes available in this new area a wealth of well-established results.

Because of space considerations, we consider only features that are defined independently and do not "build" on each other, meaning that these features make no assumptions about the behaviours of other features in the system. Also, we consider only single element features [CGLN94]. In this context, the main idea of our method is that in a system integrating features, the behavior of each feature (which is characterized as the sequences of observable actions generated by the feature) should be the same as its behavior in a system where all features are allowed to execute independently. The fact that this is not always the case is one of the main reasons of interactions in practice. In this framework, feature interactions can be detected at the specification stage by using test cases obtained

1. Appeared in Proc. of the 3rd AMAST Workshop on Real-Time Systems,Salt Lake City, 1996, 280-294.

from a specification which describes the behavior of a 'reference' system where each feature is able to execute independently.

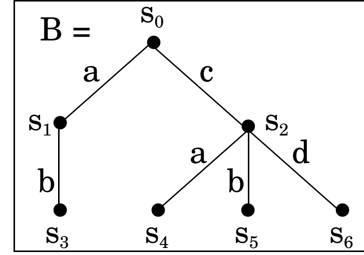
2 Basic Concepts and Notation

We use the algebraic specification language LOTOS [BoBr87][LoFH92] for feature specification [FaLS97]. LOTOS semantics are based on the concept of *labeled transition systems* (LTSs). LTSs are a generalization of finite state machines and provide a convenient way for expressing the step-by-step operational semantics of processes. Processes evolve by executing one action at a time, selected from their alphabet set. Formally,

Definition 1: Labeled Transition System

A *labeled transition system (LTS* is a 4-tuple LTS = $<S, s_0, L, T>$ where

- S is non-empty set of states;
- s_0 in S is the initial state;
- L is a (finite) set of observable actions; and
- T = {—a→ ⊆ S X S | a ∈ L', where L' = L ∪ {i}}, the *transitions*, is a L'-indexed family of binary relations on S. So if s_1 —a→ s_2 such that $s_1, s_2 \in S$ then $<s_1, s_2>$ ∈ —a→.



LTSs are usually represented by *labeled transition trees*, or simply *trees*, for the obvious pictorial advantage. On the right is an example of a behaviour B represented by a tree.

In addition to the basic definition, the following notations and definitions are widely used for interpreting LTSs [BrSS87], [Ledu92].

- $L = \{a, b, c, ...\}$ is the alphabet of observable actions and i is the hidden action;
- B $- a \longrightarrow$ B' means that after executing the observable action a, the behaviour expression B is transformed into another behaviour expression B';
- B $- i^k \longrightarrow$ B' means that after executing a sequence of k hidden actions, the behaviour expression B is transformed into another behaviour expression B';
- B $- ab \longrightarrow$ B' means that ∃ B" such that: B $- a \longrightarrow$ B" **and** B" $- b \longrightarrow$ B';
- B $=a \Longrightarrow$ B' means that B is transformed into another behaviour expression B' by executing zero or more internal actions, followed by the observable action a, then zero or more internal actions. Formally, $\exists\, k_0, k_1 \in N$, such that B $- i^{k0}\ a\ i^{k1} \longrightarrow$ B';
- B $=a \Longrightarrow$ means that B may accept the action a. Formally, ∃ B': B $=a \Longrightarrow$ B';
- B $\neq a \Longrightarrow$ B' means **not**(B $= a \Longrightarrow$ B'), i.e., B **must** refuse the action a;

- B = σ ⟹ B' means that B is transformed into another behaviour expression B' by executing a sequence of observable actions. Formally, if $\sigma = a_1 ... a_n$ then $\exists\ k_0, ..., k_n \in N$: $B - i^{k0}\ a_1\ i^{k1}\ a_2\ \ldots\ a_n\ i^{kn} \longrightarrow B'$;
- B = σ ⟹ means that ∃ B': B = σ ⟹ B';
- B *after* σ = {B' | B = σ ⟹ B'}, i.e., the set of all behaviour expressions reachable from B after executing the sequence σ;
- A *trace* is a sequence of actions; $t \sqsubseteq t'$ expresses the fact that *t* is a, not necessarily contiguous, *subtrace* of *t'*.
- The trace set of B is defined as: *Trace(B)* = {σ | B = σ ⟹}. Note that *Trace(B)* ⊆ L^*;
- *Refuses (B, σ)* is the refusal set of *B* after executing the sequence σ. Formally, *Refuses (B, σ)* = {X | ∃ B' ∈ *B after* σ such that $B' \neq a \Rightarrow$, ∀ *a* ∈ X}. A set X ⊆ L belongs to *Refuses (B, σ)* iff B may engage in the trace σ and, after doing so, refuse every event of the set X.

Although we try to avoid the use of LOTOS notation in this paper, for the sake of clear and concise presentations, we make use of the following constructs:

$|[a_1, ..., a_n]|$	is the parallel composition operator with synchronization on gates $a_1 ... a_n$;
||	is the parallel composition operator with synchronization on all gates;
[]	is the (exclusive) choice operator between two behaviour expressions;
δ	is the successful termination action.

3 A Method for Analyzing and Detecting Feature Interactions

The steps of the method, shown in Fig. 1, are as follows [Faci95]:

❶ *Specify each feature independently, within the context of the existing system, using the notion of constraints[FaLS91][FaLS97][VSVB91].*

❷ *Compose the features into a single specification so that they are able to synchronize on their common interaction points with the system and interleave on the rest of their actions. Consider the results of this composition as a* specification *with respect to the integration obtained in* ❸ *below.*

❸ *Integrate the features into a single specification so that each feature is able to perform its function when other features are disabled. Consider the resulting behaviour as an* implementation *of the features.*

❹ *Derive a set of test cases, using the theory of the derivation of tests for LOTOS processes, from the specification obtained in ❷ above.*

❺ *Simulate the system obtained in step ❸ against the test cases obtained in step ❹, and check for deadlocks.*

❻ *Interpret the results in the following way. A deadlock in ❺ implies that the way the features are integrated in the system does not allow for their simultaneous activation.*

The justification of the method follows.

3.1 Specification of Features in the Context of a System (step ❶)

Figure 2 (a) shows the POTS (Plain Old Telephone Service, denoting a basic featureless telephone system) model defined in [FaLS91] for the specification of basic call processing. The model is based on the concept of constraints which is used to structure the specification [FaLS91][FaLS97]. A specification is expressed as a set of communicating processes representing three types of constraints: *local constraints, end-to-end constraints,* and *global constraints: Local constraints* are used to enforce the appropriate sequences of events at each user's interaction point; they are different according to whether the interaction point connects to a *Caller* telephone or a *Called* telephone. *End-to-End* constraints are related to each connection; they enforce the appropriate sequence of actions between the interaction points for each telephone connection. Finally, *global constraints* involve action sequencing between connections.

This model was generalized to support the specification of features [FaLo94][Faci95][FaLS97] as shown in Figure 2 (b). To do this, we first decide on the *role* of a feature. In general, each feature can be classified as acting on behalf of either the caller process or the called process (or both). Once that decision is made, the integration of the feature's behaviour into the system is accomplished by integrating the feature, using local constraints, into the process on whose behalf the feature acts. This can be done by specifying the feature as a constraint (operator |[A]|, see below), with respect to this process. Of course, a modification of the end-to-end constraints expressed by the controller of POTS is also required. In the above figure, C' is obtained by modifying C in order to support the functionality of the new feature. We refer to the resulting specification of

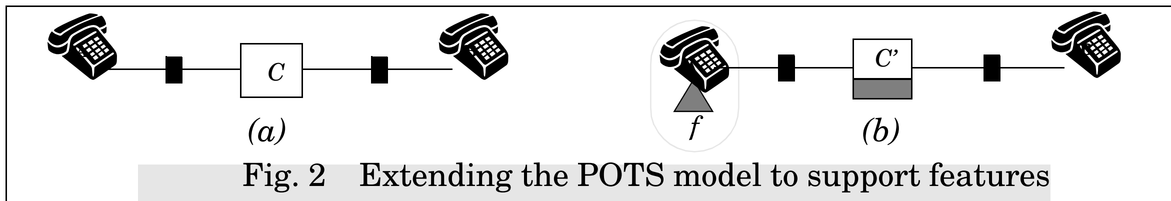


Fig. 2 Extending the POTS model to support features

integrating f_i into POTS as the behaviour of f_i in the *context* of POTS. Formally,

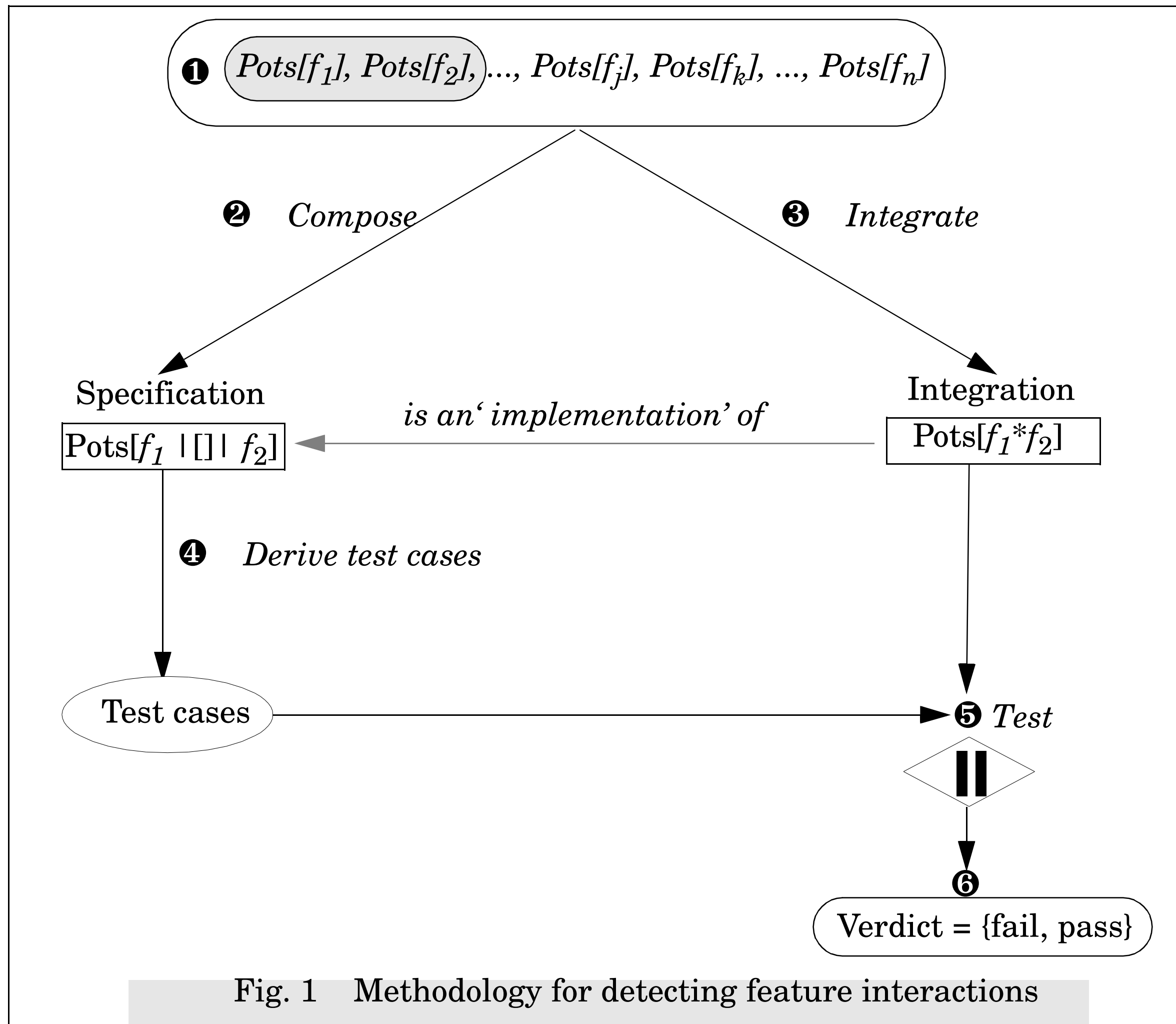


Fig. 1 Methodology for detecting feature interactions

Definition 2: System Context

We say that a feature f_i is specified *in the context of* a system *Pots*, expressed as $Pots[f_i]$, iff the following condition holds: $\forall\ t \in\ Trace(f_i)$, $\exists\ t' \in\ Trace(Pots[f_i])$ such that $t \sqsubseteq t'$.

In other words, a feature is said to be specified in the context of a system if every trace of the feature is a sub-trace of some other trace in the resulting system. For example, suppose that we have specified a POTS system which allows a caller **A** to establish a talking session with a called **B**. Figure 3 (b) shows a portion (obviously very simplified) of the specification. It describes the following sequence, starting from the state where **B** has received a *ring* signal from **A**. User **B** *answers* **A**; the two users engage in a talking session; **A** hangs up; and finally the system exits as shown by δ, the LOTOS successful termination.

Suppose now that we wish to integrate *Call Waiting (Cw)* and *Three Way Calling (Twc)* into the system, independently of each other. In the context of POTS, *Cw*

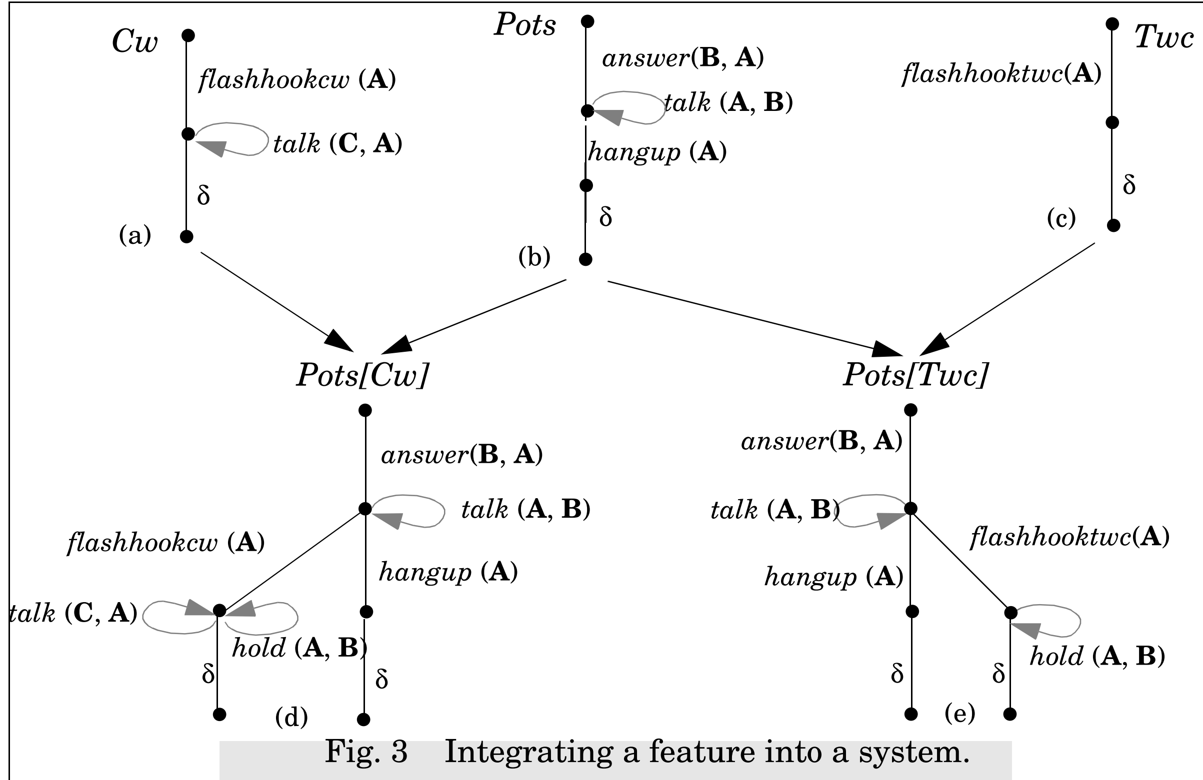


Fig. 3 Integrating a feature into a system.

allows user **A** to respond to another user **C** while still talking to **B**. Starting from a state where **A** is talking to **B** and **C**'s controller has just sent a *Call Waiting Tone* signal to **A**, **A** may send a *flashhookcw* signal to accomplish two things: (1) to put **B** on hold, and (2) to establish a talking session with **C**, as shown in Figure 3 (d).

We can also integrate *Twc* into POTS without taking the behaviour of *Cw* into consideration. *Twc* is a feature which allows **A** to suspend **B** and establish another talking session with another user (**D** for example) by sending a *flashhooktwc* signal to the controller, and after reaching a talking state with **D**, **A** sends another *flashhooktwc* signal to bring **B** back to the connection, thereby establishing a talking session between **A**, **B**, and **D**. Figure 3 (e) shows the integration of *Twc* in the context of POTS, but note that only the first action of *Twc* is shown. The LTSs of Figure 3 can be represented by the following expressions, using a slightly abused LOTOS syntax, where also the *talk* loops have been ignored for simplicity.

(a) *Cw* := *flashhookcw*(**A**); **exit**[1]
(b) *Pots* := *answer* (**A**); *hangup* (**A**); **exit**
(c) *Twc* := *flashhooktwc*(**A**); **exit**
(d) *Pots[Cw] := answer* (**A**); (*flashhookcw*(**A**); **exit** [] *hangup* (**A**); **exit**)

1. In proper LOTOS, we represent this as: *flashhookcw*!**A**; **exit** where ***flashhookcw*** is a gate and **A** is an offer.

(e) *Pots[Twc] := answer* (**A**); (*flashhooktwc*(**A**); **exit** [] *hangup* (**A**); **exit**)

It is easy to verify that both *Cw* and *Twc* are specified according to definition 2 by checking their traces.

- *Trace(Cw)* = {<>, <*flashhookcw*(**A**)>, <*flashhookcw*(**A**); δ>}

  = {t_1, t_2, t_3}

- *Trace(Pots[Cw])* = {<>, <*answer*(**B**, **A**)>, <*answer*(**B**,**A**); *flashhookcw*(**A**)>, <*answer*(**B**,**A**); *hangup* (**A**)>, <*answer*(**B**,**A**); *flashhookcw* (**A**); δ>, <*answer*(**B**,**A**); *hangup* (**A**); δ>}

  = {t_1', t_2', t_3', t_4', t_5', t_6'};

Since $t_1 \sqsubseteq t_1' \in$ *Trace(Pots[Cw])*, and

$t_2 \sqsubseteq t_3' \in$ *Trace(Pots[Cw])*, and

$t_3 \sqsubseteq t_5' \in$ *Trace(Pots[Cw])*

Then it is the case that *Pots[Cw]*, meaning that *Cw* is specified in the context POTS. Using similar deductions we can show that the same holds for *Pots[Twc]*, as shown in Figure 3 (c) and (e).

3.2 Composition Vs. Integration of Features (steps ❷ and ❸)

Our primary objective, as we have already mentioned, is to answer the following question: is there interaction between features *Cw* and *Twc* when they are integrated into an existing system? To answer this question, one must define a reference point against which the answer can be evaluated. Let us first introduce the intuition which motivated the formalism. Our starting point is the notion of *simultaneous* execution. For practical purposes, this notion is interpreted in the context of interleaved semantics. Saying that two features can execute *simultaneously* is equivalent to saying that both features will reach their terminal states and that their actions are allowed to interleave. In many cases, however, when specifiers produce specifications which integrate the functionalities of several features, their primary concern is to include the functionality of each feature, one at time, in the resulting specification. For each feature that is being integrated, the specifier gives no consideration to what effects this will have on other features in the system. The basic idea is explained by way of Figure 4. Parts (a) and (b) express the integration of each of the two features in the context of POTS. Parts (c) and (d) express the composition and the integration of these two specifications, respectively.

Note at this point that, while the approach is applicable for *n* features, for illustration purposes we are using two features only. As mentioned, the composition of features, which reflects the interleaving of the independent actions of *Cw* and *Twc* while synchronising on their common actions with POTS, turns out to be conveniently expressed using the LOTOS composition operator $|[a_{pots}]|$, where a_{pots} are the gates that are common

to *Pots[Cw]* and *Pots[Twc]*. In our example, *answer* is a common gate between *Pots*, *Cw*, and *Twc*. From now on, we will refer to this composition by the following concise notation *(Pots[Cw |[]| Twc])*. Note that this is simply a notation, shorthand for a LOTOS expression, and does not introduce an operator. A question that comes up at this point is: can deadlocks

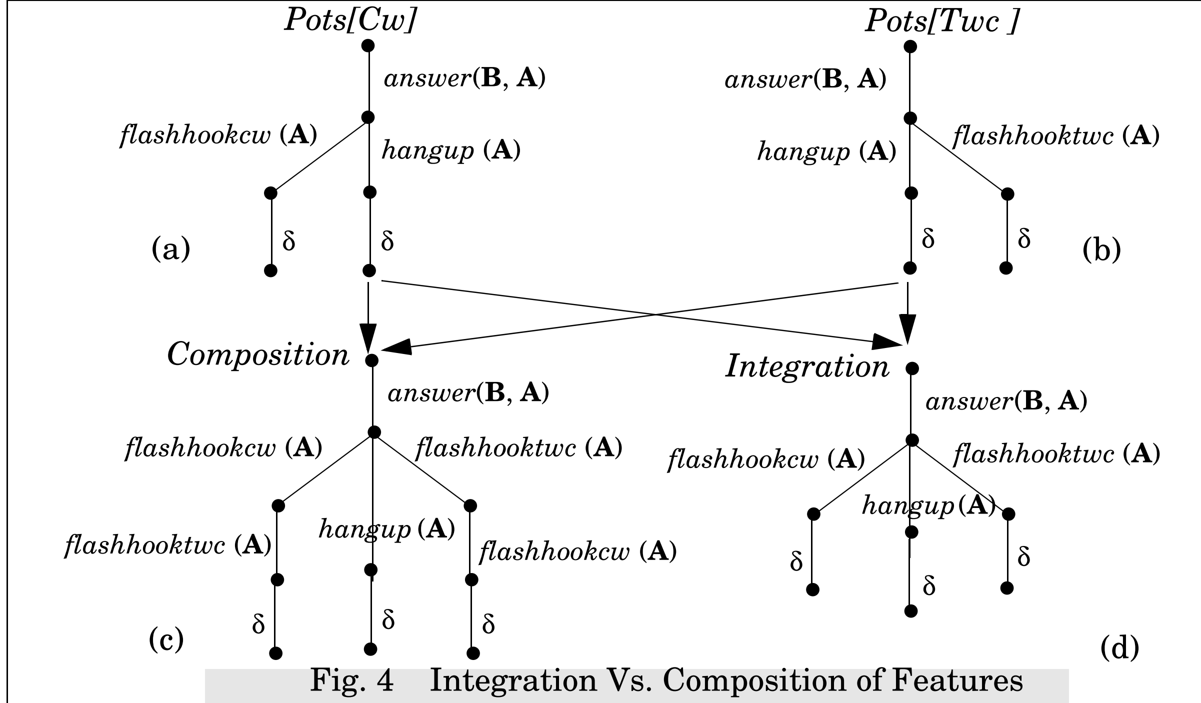


Fig. 4 Integration Vs. Composition of Features

be introduced when doing the composition? This question is worth of further research, and the answer is probably that if such deadlocks come up, this is another symptom of a design problem, which could be a feature interaction. However, we did not encounter such deadlocks in the several features we have specified and composed.

On the other hand, the requirements imposed on the system designer to integrate the two features into the specification, which we will write as *Pots[Cw*Twc]*, are as follows:

- *Trace(Pots[Cw])* $\subseteq$ *Trace(Pots[Cw*Twc])* (r1)
- *Trace(Pots[Twc])* $\subseteq$ *Trace(Pots[Cw*Twc])* (r2)

meaning that the functionalities of both features must be preserved in the final integration of *Cw* and *Twc* in the context of POTS. However, the integration of *Cw* and *Twc* in the context of POTS leaves open the question of whether or not *Cw* can execute all its traces to completion, even when both features are active at the same time. The same holds for *Twc* with respect to *Cw.*

It should be noted that, while composition is formally defined in terms of the LOTOS interleave operator, integration is not. This is because of the fact that integration depends on the way the abstract features are implemented. In other words, composition is a specification-level concept, while integration is an implementation-level concept. The first is formal, the second is not. If the integration is done in such a way that the features cannot exhibit their interleaved behavior, then either the features themselves, or the way they are

integrated, should be modified. For example, we shall see below that both features *Twc* and *Cw* need a user action to become activated. If, in the integration, both these actions are mapped on the same signal *flashhook*, an interaction arises.

It is possible to see if the traces of the composition are also in the integration by applying to the integration testers obtained from the composition. Thus the following section deals with the problem of testing that the interleaved behavior is realized in the integration.

3.3 Derivation of Test Cases to Detect Interactions (step ❹)

Conformance Testing and the Detection of Feature Interactions

In this section we briefly review the results reported in [BrSS87], [Brin88], [BALL90] and show how the notion of *conformance testing* has a direct application in the domain of detecting feature interactions. But first, let us define some concepts which are needed for conformance testing.

Definition 3: Deadlocks in Terms of Refusals

Let L be the set of observable events, L^* be the set of traces. Then, for I a labelled transition system, $A \subseteq L$, $\sigma \in L^*$:

- I ***Refuses*** (σ, A) is defined as: $\exists I' : (I = \sigma \Rightarrow I'$ **and** $\forall a \in A: I' \neq a \Rightarrow)$
- I ***Deadlocks***(σ) is defined as: I ***Refuses***(σ, L)

The first part of the definition says that I may execute the trace σ, and after doing so, refuse every event in the set A. Similarly, the second part of the definition says that I reaches a deadlock if it refuses every observable event, after executing σ.

Conformance allows one to reason about an implementation and a specification using a single formalism. In this context, an implementation is taken to be an abstract representation of a physical realization. *'I is a valid implementation of S'* can be defined as follows [Brin88].

Definition 4: Conformance

Let S and I be processes. We say that :

I **conf** S $\Leftrightarrow$ $\forall \sigma \in$ Trace(S), $\forall A \subseteq L$,
if I Refuses (σ, A) **then** S Refuses (σ, A).

Informally, I conforms to S if, and only if, testing the implementation I against the traces of the specification S does not lead to deadlocks that would not occur while testing S against those same tests. In other words, testing the implementation does not reveal deadlocks that would not be revealed while testing the specification.

In section 3.2 we discussed the relation between *composition* and *integration* of features. Using these two notions and the formal definition of conformance, we can now provide our formal definition of the feature interaction problem.

Definition 5: Formalization of Feature Interactions: Int

Let $f_1, f_2, ..., f_n$ be features,

Let A be the alphabet of POTS and A_i the alphabet of f_i such that:

$$\forall\ i, j: A_i \cap A_j = \emptyset \text{ and } A \cap A_i \neq \emptyset, \text{ for } 1 \leq i \leq n,\ 1 \leq j \leq n.$$

Let S and I be processes, such that:

$$S := Pots[f_1 \,|[]|\, f_2 \,|[]|\, \dots \,|[]|\, f_n] \text{ and } I := Pots[f_1 * f_2 * \dots * f_n],$$

We say that : $\mathbf{int}(f_1, f_2, \dots, f_n) \Leftrightarrow \neg(\text{I } \mathbf{conf} \text{ S})$, meaning that an interaction exists between the n features if, and only if, the *integration* of the features ***does not*** conform to their *composition*.

Note that in the definition we assume that $A_i \cap A_j = \emptyset$. This is consistent with our intuitive view that features are defined independently, in terms of their interactions only with respect to POTS. However, as already mentioned, it is possible that actions of different features are mapped onto the same element in the integration, and this may cause interactions, which can be detected by our method.

Definition 5 is the link that allows us to exploit the *conformance* and *testing theory* of Brinksma et al.

Derivation of Tests

The theory of deriving tests [Brin88] from a LOTOS specification assumes that processes have only successful terminations, meaning that whenever a process reaches a deadlock, it must have done so via the successful termination **exit**. The theory also asserts that there exists a *canonical* tester T_s that can discriminate those implementations which conform to the specification and those which do not. For an implementation I and a canonical tester T_s, this discrimination is accomplished according to the following **passes** relation.

Definition 6: passes relation

Let I and T_s be processes, then

I **passes** T_s iff $\forall\ \sigma \in L^*$: *If* $(I \,||\, T_s)$ after σ **deadlocks**
Then T_s after σ **deadlocks**

For a specification S, a canonical tester T_s is defined as follows:

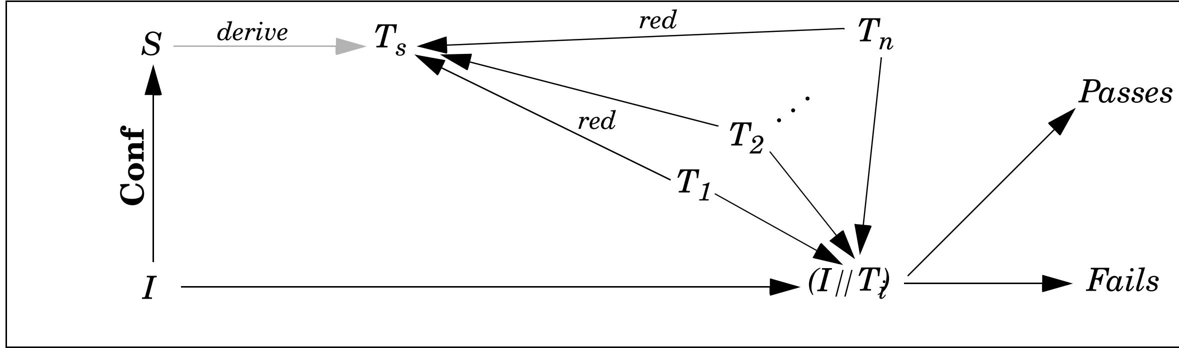
- *Trace* (T_s) = *Trace* (S), and
- $\forall$ I: I **conf** S iff I **passes** T_s

In addition, each T_s can be expressed as a set of testing processes called the *irreducible reductions* (IR_s) of T_s. Formally,

$IR_s =_{def} \{T \mid T \textbf{ red } T_s, \forall T': T' \textbf{ red } T \Rightarrow T' = T\}$, where

B1 **red** B2 iff B1 **conf** B2 **and** *Trace*(B1) ⊆ *Trace*(B2).

The following figure captures the intuition behind these definitions.



Note that, although we use here a nonconstructive definition of T_s, methods for constructing it are known [Weze90].

The integration of two features in a system is driven by the functionalities of both the features and the existing system. Since it is not possible to define the semantics of a general model for the integration of any two given features, we give an example to show that, depending on the results of the integration, an interaction may or may not occur. In either case, our methodology succeeds in reaching the correct verdict. It is important, however, to keep in mind that the ***passes*** verdict simply means that more testing is required, whereas a ***fails*** verdict means that an interaction is detected.

According to our definition **int** of feature interactions, an interaction exists between *Cw* and *Twc* if, and only if, *I* ***does not conform*** to *S*. So, in order to derive the canonical testing process from the composition and execute it against the integration, let us express the trees of Figure 4 (c) and (d) as LOTOS expressions:

I := Pots[*Cw***Twc*] = *answer*(**B**, **A**);
(*flashhookcw*(A);**exit**[]*hangup*(**A**);**exit**[]*flashhooktwc*(A); **exit**)

and

S := Pots[*Cw* |*[]*| *Twc*] = *answer*(**B**, **A**);
(*flashhookcw*(A); *flashhooktwc*(A); **exit**
[] *hangup* (**A**); **exit**
[] *flashhooktwc*(A); *flashhookcw*(A); **exit**
)

Following [Brin88], we now express the canonical tester T_s of *S* in the following way:

T_s = T(*Pots*[*Cw* |*[]*| *Twc*])
= *answer*(**B**, **A**);

(**i**; *flashhookcw*(A); *flashhooktwc*(A); **exit**
[] **i**; *hangup* (**A**); **exit**
[] **i**; *flashhooktwc*(A); *flashhookcw*(A); **exit**
)

The final result is shown in Figure 5 (b).

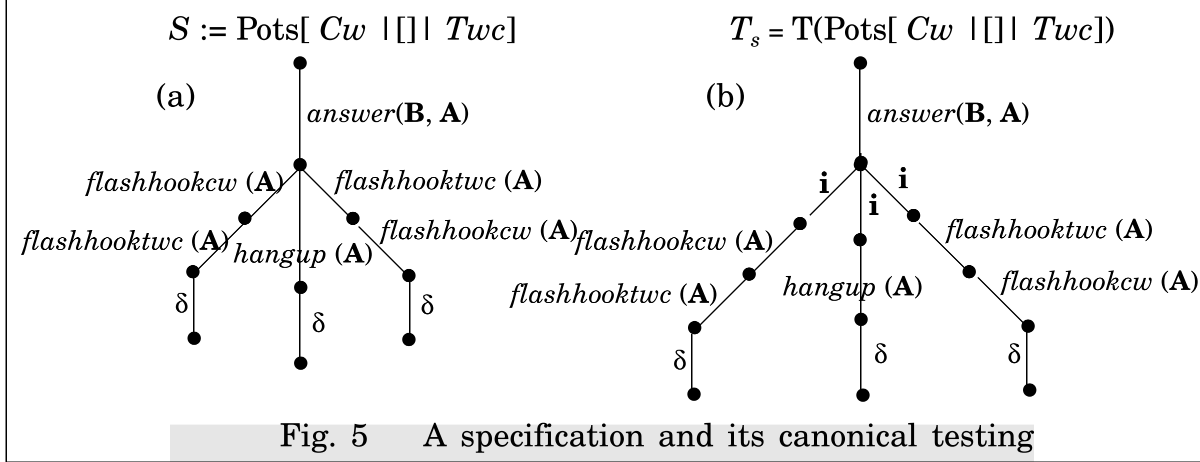


Fig. 5 A specification and its canonical testing

The next step is to express T_s as a set of irreducible test cases, from which a set of useful test cases are selected[BrTV91]. Some such test cases are:

T_1 = *answer*(**B**, **A**); *flashhookcw*(**A**); *flashhooktwc*(**A**); **exit**

T_2 = *answer*(**B**, **A**);*hangup* (**A**); **exit**

T_3 = *answer*(**B**, **A**); *flashhooktwc*(**A**); *flashhookcw*(**A**); **exit**

T_4 = *answer*(**B**, **A**);
(*flashhookcw*(**A**); *flashhooktwc*(A); **exit**
[] *hangup* (**A**); **exit**
[] *flashhooktwc*(A); *flashhookcw*(A); **exit**)

Although in this very simplified example we have assumed finite behavior trees, the presence of loops is not a problem [Jao92][DAV93].

3.4 Executing the System and Analysing the Results (step ❺ and ❻)

The final two steps of the methodology are to execute the specification against a selected subset [BrTV91] of the derived test suite in order to check for deadlocks. For our purposes, the set {T_1, T_2, T_3} is sufficient to test our integration because every trace of T_4 is a member of another test suite in the selected set. An example of testing the integration with T_1 is shown below. Testing with T_2 and T_3 is similar. The verdicts for the three tests are {**fails**, **passes**, **fails**}. Since the integration fails at least one of the tests, we conclude that an interaction exists between *Cw* and *Twc*. The reason is that once the *flashhook* is executed, only the feature which participates in its execution is allowed to continue with

its behaviour, thereby preventing the other one from exhibiting its behaviour in the overall system.

Testing with T_1:

```
Pots[Cw*Twc] || T1 = (  answer(B, A);
                        (      flashhookcw(A); exit
                           []  hangup (A); exit
                           []  flashhooktwc(A); exit
                        )
                     )
                     || answer(B, A); flashhookcw(A); flashhooktwc(A); exit
                   = answer(B, A); flashhookcw(A); stop
```

⇒ test **fails** because it did not reach its **exit**.

4 Conclusions and Research Directions

We have proposed a formal algebraic framework for analyzing and detecting certain types of feature interactions in telephone systems at the design level. We used to advantage the characteristics of process algebras: its formal properties allowed us to establish a theoretical framework for the problem; while its executability allowed us to define a testing framework for actually detecting interactions. In [Faci95], the method is applied on nine examples of feature interactions, mostly very different from the one presented above. Features considered are: Call Waiting, Call Forward on Busy, Call Forward Always, Automatic Recall, Automatic Callback, Originating and Terminating Call Screening, Distinctive Ringing, Calling Number Delivery, Unlisted Numbers.

Of course, many questions remain open. Is the 'conformance' relation the one that best captures the intuition behind detecting feature interaction (it is well-known that this relation has limitations in the area of protocol conformance testing)? Is it possible to better use the concept of 'refusal' for the characterization of features (in our framework, features are characterized by traces and not by refusals)? How can one find appropriate test cases for different types of interactions [BrTV91]? How can the method be extended to cover other cases of feature interaction? Our contribution sets the stage for further research on these and other related problems.

Needless to say, the method could be reformulated in terms of other languages using labelled transition models.

Acknowledgment. Funding sources for our work include the Natural Sciences and Engineering Research Council of Canada, the Telecommunications Research Institute of Ontario, Bellcore, Bell-Northern Research, and the National Institute of Standards and Technology. We like to acknowledge the many fruitful discussions that we have had with members of our LOTOS group.

5 Bibliography

[BALL90] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A Formal Approach to Conformance Testing. Second International Workshop on Protocol Test Systems, eds. J. de Meer, L. Mackert, and W. Effelsberg. North Holland 1990, 349-363.

[BDCG89] T.F. Bowen, F.S. Dworak, C.H. Chow, N. Griffeth, G.E. Herman, and Y-J. Lin. The Feature Interaction Problem in Telecommunications Systems, 7th International Conference on Software Engineering for Telecommunication Switching Systems, July 1989, 59-62.

[BoBr87] T. Bolognesi, and E. Brinksma. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14 (1987) 25-59.

[BrSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specifications, ther Implementations and their Tests. Protocol Specification, Testing, and Verification, VI. Eds. G.V. Bochmann, B. Sarikaya, North Holland 1987, 349-360.

[BrTV91] E. Brinksma, J. Tretmans, L. Verhaard. A Framework for Test Selection, Protocol Specification, Testing, and Verification, XI. Eds. B. Jonsson, J. Parrow, and B. Pehrson, North Holland 1991, 233-248.

[Brin88] E. Brinksma. A Theory for the Derivation of Tests. In: Aggarwal, S., and Sabnani, K., (Eds.) Protocol Specification, Testing, and Verification, VIII, North-Holland, 1988, 63-74.

[CGLN94] E. J. Cameron, N. Griffeth, Y. Lin, M. E. Nilson, W. K. Schnure, H. Velthuijsen. A Feature Interaction Benchmark for IN and Beyond, Second International Workshop on Feature Interactions in Telecommunications Systems, eds. L.G. Bouma and H. Velthuijsen, IOS Press, 1994,1-23. Also in IEEE Communications, vol. 31, No. 3, 64-69, March 1993.

[DAV93] K. Drira, P. Azema, F. Vernadat. Refusal Graphs for Conformance Tester Generation and Simplification: a Computational Framework. In: Protocol Specification, Testing, and Verification, XIII, Eds. A Danthine, G. Leduc, P. Wolper, North-Holland, 1993, 257-272.

[Faci95] M. Faci. Detecting Feature Interactions in Telecommunications Systems Designs, PhD Thesis, University of Ottawa, 1995 (obtainable by ftp on lotos.csi.uottawa.ca).

[FaLS91] M. Faci, L. Logrippo and B. Stepien. Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach, Computer Networks and ISDN Systems, 21, 52- 67, North Holland, 1991.

[FaLS97] M. Faci, L. Logrippo, and B. Stepien. Structural Models for Specifying Telephone Systems. Computer Networks and ISDN Systems 29 (1997) 501-528.

[FaLo94] M. Faci and L. Logrippo. Specifying Features and Analysing Their Interactions in a LOTOS Environment, Second International Workshop on Feature Interactions in Telecommunications Systems, eds. L.G. Bouma and H. Velthuijsen, IOS Press, 1994, 136-151.

[Jao92] Rafik Jaouani. LOTOS Based Conformance Testing. The theory and a tool. Master thesis, University of Ottawa, 1992 (obtainable by ftp on lotos.csi.uottawa.ca).

[Ledu92] G. Leduc. A Framework based on the Implementation relations for Implementing LOTOS Specifications, Computer Networks and ISDN Systems, 25, 23-41, North Holland, 1992.

[LoFH92] L. Logrippo, M. Faci and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples, Computer Networks & ISDN Systems, Vol. 23, No. 5, 1992, 325-342.

[VSVB91] C.A. Vissers, G. Scollo, M. van Sinderen, E. Brinksma. Specification Styles in Distributed Systems Design and Verification, Theoretical Computer Science 89, 1991, 179-206.

[Weze90] C. Wezeman. The CO-OP Method for Compositional Derivation of Conformance Testers. In: Protocol Specification, Testing, and Verification, IX, eds. E. Brinksma, G. Scollo, and C.A. Vissers, North Holland, 1990, 145-158.