Feature Execution Trees and Interactions

Rui Gustavo Crespo
Dept Electrical Engineering and Computers
Technical University of Lisbon
Av. Rovisco Pais, 1049-001 Lisboa, Portugal

Luigi Logrippo
School of Information Technology and Engineering
University of Ottawa
Ottawa, ON K1N 6N5, Canada

Tom Gray
Mitel Networks Corporation
350 Legget Drive, Kanata, ON K2K 2W7, Canada

Abstract

Telecommunications networks are becoming more sophisticated, with the development of diversified features working in distributed environments. Increasingly, it is expected that features be created and adapted according to the specific needs of the subscribers. However, the introduction and modification of features may result in undesired behaviour, and this effect is known as feature interaction-FI.

We describe the design language of features, named Feature Execution Tree-FET. We also describe algorithms that identify pairs of features as candidates for possible feature interactions.

Keywords: Communication feature, Condition, Event, Feature interaction, Trees

1 Introduction

Telecommunications networks are continuously evolving, and one of the trends is to provide increasingly sophisticated services to users.

The *Plain Old Telephone Service*-POTS, is made of basic services such as calling another subscriber. Other services, referred as *features*, are constructed using POTS or other features.

Recent trends, such as Intelligent Networks, wireless networks and Internet telephony, emphasize distributed control of services. Each user subscribes to specific sets of features, according to his/her policies.

Internet telephony is expected to considerably expand the role of the user in the selection and customisation of new features and policies.

1.1 Feature characteristics

Nowadays, typical commercial systems have many hundreds of communication features. Most features are very simple, as the vast majority is completely described in a few lines. In a distributed control environment, features are aware of the subscriber status and communicate to other features, of the same or other subscriber, through events. Event-driven systems have been explored in many application areas beyond communication systems, such as active databases [1] and client-server systems [2].

The combination of features may reveal the problem of *feature interaction*-FI, where the combination of features results in undesired behaviours. As an example, suppose that Bob subscribes the feature *Incoming Call Screening*-ICS to forbid calls from Ann. Now, suppose that Carl subscribes the feature *Call Forward Always*-CFA to Bob. If Ann dials Carl, the call is forwarded to Bob and this outcome conflicts with the goal of the ICS feature.

1.2 Architecture

Users must be able to choose the features they wish to subscribe and define policies for feature selection. We propose a three-layer architecture for the construction of communicating systems, adapted to the individual needs. Starting from the lowest layer,

1. Feature layer, with a "stand-alone" definition of features. Examples of the feature

layer are the *Call Direct*-CD basic service, the *Redial on Busy*-RB and the *Call Forward Always*-CFA features.

2. Modality layer, where the features are tailored to the individual needs. For example, a subscriber states that all incoming calls must be forwarded to extension 2398, between 10 and 14 AM: the CFA is adapted with the identification of the time range when it must be executed and the terminal to which calls must be forwarded.

3. Global selection, which describes how to choose the feature to be executed. One example of selection policy is the hierarchical precedence for feature execution, starting at the group policies and ending at the individual policies. The global selection layer also prescribes how FIs are solved, for example through priority lists.

This paper focuses only on the feature layer.

1.3 Representation languages

Many languages have been used to represent features, each with their own range of applicability, advantages and drawbacks. UCM [3] is a high-level notation, directed to general design with no precise semantics. MSC [4] is closer to the implementation level, since it shows message exchanges by which a feature can be implemented. MSCs focus on the system part they represents and, therefore, reasoning about the global behaviour is impractical. *Chisel* [5] is a notation that represents, in a tree format, the end-to-end behavior of entities that are involved in the features.

Specification languages are available, such as LOTOS [6], based on process algebras, or SDL [7], based on extended finite state state machines communicating by message queues.

In Chisel, LOTOS and SDL, developers must present in the specification all features available in the system. Chisel provides "glueing nodes", while in SDL and LOTOS details are added in a top-down development. This approach forces the representation of each feature to include the interface with other features, which makes such representations less readable. In addition, the policies of feature selection must be coded inside the specification, which makes difficult to reason about them.

2 FET language

The *Feature Execution Tree* is a formalization of a graphical notation that has been developed at Mitel Corporation. FET seeks to fulfill the goals of representation based on familiar models and of self-contention the feature descriptions with simple interfaces.

FET represents the control logic of features in terms of subtrees that can either succeed or fail, thus causing other subtrees to perform other actions.

2.1 Abstract Syntax

Figure 1 depicts the abstract syntax of the FET language, with the essential elements of the language and with no concern about the associativity and priority of operators.

$$
\begin{array}{ll}
FET: & \Phi ::= \epsilon T\lambda, X \rightarrow X \rightarrow X \mid \\
 & \quad \epsilon T\lambda, X \rightarrow X \rightarrow X\ [T\lambda] \\
 & \\
Cond: & X ::= X\ \Omega\ X \mid \mathit{not}\ X \mid \mathit{true} \mid \\
 & \quad \Sigma\ T\lambda \mid \gamma : B\lambda \mid \Phi \\
\mathit{Oper}: & \Omega ::= \mathit{seq} \mid \mathit{and} \mid \mathit{or} \\
\mathit{Selector}: & \Sigma ::= \epsilon \mid \epsilon : \Delta \mid \pi \\
\mathit{Direction}: & \Delta ::= \mathit{launched} \mid \mathit{required} \\
 & \\
\mathit{TermList}: & T\lambda ::= \tau, T\lambda \mid \tau \\
\mathit{BranchList}: & B\lambda ::= \alpha - X, B\lambda \mid \alpha - X
\end{array}
$$

Figure 1: Abstract syntax of FET language

The commas, implications, square parenthesis and hyphens are used only to improve the reability of the syntax rules. $\tau, \gamma, \alpha, \epsilon$ and π are the syntactical elements of terminal, group, alternative, event and predicate identifiers.

A feature must have the form shown in (1), with a trigger event (the $\epsilon T\lambda$ of the Φ domain in figure 1) and three conditions X: the first X of the Φ domain is the pre-condition, the last X is the post-condition and the middle X is the execution body.

$$SEQ(event \wedge pre - condition, \\ SEQ(body, post - condition)) \tag{1}$$

The pre-condition states which events are required (launched by other features or by hardware devices) and which predicates must hold, in order to execute the feature.

The body shows which events must have been launched by the subscriber, or by other features (respectively with L and R tag), or which predicates must be satisfied, in order the feature execution be successful.

The post-condition indicates the new status of the subscriber and the tuple space events, after the execution of the body. The post-condition must be expressed as a conjunction of predicates, events and of their negations. All post-condition events are launched by the feature. Unlike the axiomatic semantics of programming languages [8], the post-condition has an operational meaning at the end of the feature execution, with the launching and the clearing of events, and the update of the subscriber predicate values.

2.2 Leaves

Leaves are predicates, events or features. A predicate states a condition about one terminal and its value is made available only to the involved terminal. Events trigger features and are launched by hardware devices (e.g. timers or the telephone hook), or by features. $true$ is a special predicate with no parameters.

Predicates and events may have parameters, that can be:

- subscriber names in lowercase letters $(ann, bob, carl, \ldots)$,
- variables in uppercase letters $(A, B, \ldots)$,
- FET parameters with letter "P" followed by the position parameter $(P1, P2, \ldots)$,
- $self$: in the feature execution, $self$ is replaced by the subscriber's terminal,
- $null$, denoting an empty parameter,
- $tail$, denoting the parameter list starting at $P2$.

Examples of conditions are $idle(self)$, which states that the subscriber terminal is idle, or $talk(self, A)$, which states that feature subscriber terminal is talking to another terminal called A.

The event identifiers are prefixed with $ev\_$. For example, $ev\_ring(A, self, bob)$ states that the subscriber redirects the call, initiated by A, to Bob. Events are placed in one *tuple space* [9] (a globally shared memory, whose tuples are attributes). The attributes are the event name and each of the event parameters. For example, the tuple *(event=ev_offhook, terminal=ann)* indicates that Ann has put her phone offhook. The tuple space is an associatively addressed memory, allowing a fast selection of the tuples that satisfy a template. Operations over the tuple space include reading a tuple, with and without tuple removal, and writing a tuple.

2.3 Connectives

Larger trees are inductively constructed with the binary connectives $SEQ, \wedge, \vee$ and the unary connective $\neg$. SEQ is the only connective whose semantics imposes an evaluation order: the left subtree must be evaluated first and the right subtree is evaluated according to the result of the left subtree evaluation.

The SEQ connective is depicted by an arrow from the left subtree to the right subtree. The $\vee$ is depicted by the $\vee$ symbol connecting the two subtrees and $\neg$ is depicted by the $\bullet$ symbol in the node.

Example 2.1 *The* Call Answer-*CA basic service, graphically depicted in the figure 2, is defined as*

$$SEQ(ev\_ring(A, C, self) \wedge idle(self), \\ SEQ(ev\_offhook(self): L, \neg idle(self) \wedge \\ talk(self, A) \wedge \neg ev\_ring(A, C, self) \wedge \\ ev\_ack(A, self) \wedge \neg ev\_offhook(self)))$$

The event $ev\_ring(A, C, self)$, which triggers the CA service, is launched by the caller and is cleared in the callee post-condition.

The body of the CA is, simply, the requirement that the event $ev\_offhook$ has been launched by the callee.

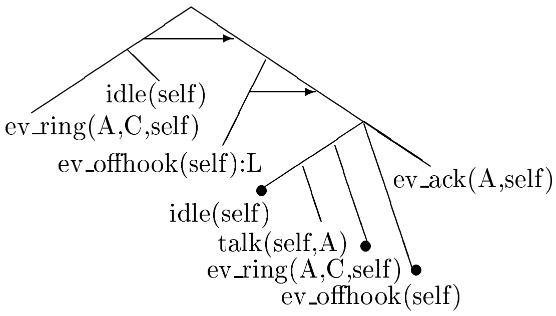

Figure 2: FET of the Call Answer service

In the post-condition, the callee launches the event $ev\_ack(A, self)$, which is necessary for the execution of the caller body. The event ev_ack will later be cleared by the caller. The events ev_ring and $ev\_offhook$ are cleared, the callee is no longer idle and its status shows the callee is talking to A.

2.4 Alternative groups

The FET language enables the merge of several features, that differ in the same nodes. Each group, depicted by one horizontal line, receives an unique name. Each alternative is depicted by a labelled column, drawn from the horizontal line, that represents the group.

Example 2.2 *The* Call Direct-*CD basic service, depicted in figure 3 may have two results, success and failure. There are two causes for failure, the other telephone being busy, and she does not answer.*

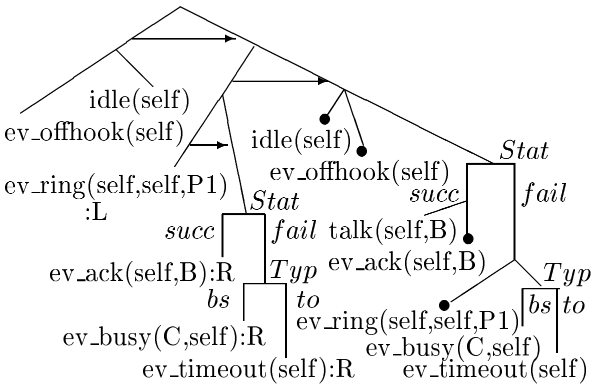

Figure 3: FET of the Call Direct service

The alternative $fail$ of the Stat group contains another group, named Typ. For the failure CD service, by timeout, the body is $SEQ(ev\_ring(self, self, P1) : L, ev\_timeout(self) : R)$ and the post-condition is $\neg idle(self) \wedge \neg ev\_offhook(self) \wedge \neg ev\_ring(self, self, P1) \wedge \neg ev\_timeout(self)$.

Although the call is initially directed to the terminal P1, the connection may be established with another terminal B, for example because terminal P1 forwards the call to B.

2.5 Parameters and replacement

The feature may be parameterised with a terminal list $T\lambda$. The parameterisation allows developers to express features generically. The lenght of the parameter list may not be fixed.

The post-condition leaves may hold a reference to a FET. In this case, the feature execution continues with another FET.

Example 2.3 *The* Redirect on Busy-*RB feature, depicted in figure 4, redirects one call, if the destination is busy.*

The terminals are all declared in the parameter list and the RB starts to try to establish a connection to the first terminal. If the first terminal is busy, then RB tries to establish a connection to the second terminal, and so on. The feature concludes when one of the calls is acknowledged (not necessary by a terminal in the list), or RB tried to establish connections with all the terminals in the list and all failed.

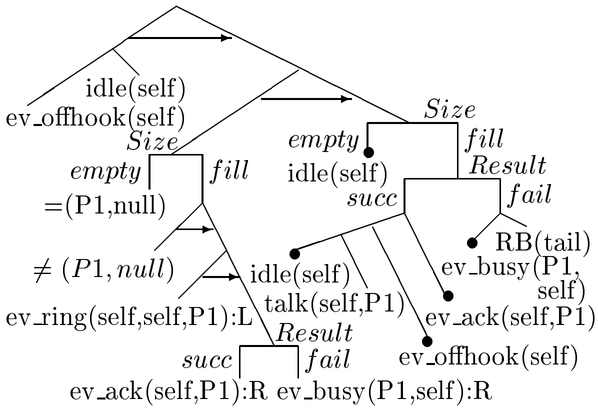

Figure 4: FET of the Redirect on Busy feature

There are two alternative groups, Size and Result, both with dimension equal to two. Size focuses on the dimension of the terminal list, *with empty and fill alternatives. Result deals with the status of the destination phone, with succ and fail alternatives.*

The FET of the RB feature is a merge of three FETs: RB with an empty list of terminals, a RB with a successful redirection and a failure redirection to a specific terminal (the first parameter).

2.6 FET semantics

The formal meaning of MSCs is described in different ways, such as Büchi automata [10] and Petri nets [11]. Chisel semantics has not been described precisely, although a representation on a language based on the process-algebraic language LOTOS has been provided [12]. The formal semantics of *LOTOS* [13] and *SDL* [14] are well-known. The representation languages adopted in the description of features are based on the operational semantics.

The semantics of the FET language follows the denotational approach [15], because it is very well established, having been used for the semantic definition of many programming languages, and tools are available to support it.

3 Feature interaction

The combination of features may reveal the problem of *feature interaction*, where the combination of features results in non-desired behaviours. For some subscribers, a given FI may be seen as conflicting, for others the same FI may be seen as cooperating.

Example 3.1 *The* Call Forward on Busy-*CFB feature is defined in FET as*

$$SEQ(ev\_ring(A,C,self) \wedge \neg idle(self),\ SEQ(true, ev\_ring(A,self,P1) \wedge \neg ev\_ring(A,C,self)))$$

The parameter $P1$ is the terminal to whom the call must be redirected, in case the CFB subscriber is busy.

If two users subscribe to the CFB feature, and are busy, the caller to the first subscriber is redirected to the destination specified by the second subscriber, not to the second subscriber his(her)self. This combination may be seen as conflicting, or as cooperative.

Since features can be arbitrarily complex specifications, the FI detection is an undecidable problem. However, it is possible to try and identify possible feature interactions (FI candidates) by looking for incoherences in the feature definitions.

We propose three algorithms to identify candidates for FI: the direct incoherence, the transitive incoherence and the loop detection. All three algorithms are based on the identification of incoherences in the conjunction of post-conditions of the involved features.

3.1 Direct incoherence

This algorithm detects the case where the same or different users subscribe to two features that are triggered by the same event, if the post-conditions are incoherent. The algorithm is as follows:

1. Select two features with equal trigger events.
2. Make all possible bindings of terminal variables to individuals involved in both features and terminal equalities. Generate all possible values of predicates involved in the feature specifications[1].
3. For each binding, check if both pre-conditions hold.
4. If, for some of the possible bindings, the conjunction of the post-conditions verify, at least, one restriction rule or a logical incoherence, the pair of features is a candidate for a FI.

Table 1 depicts some possible restriction rules. For example, rule $R1$ states that a call cannot be redirected and, simultaneously, denied.

Example 3.2 *Suppose that Bob subscribes the features CFB, described in example 3.1, and* Call Waiting-*CW, defined as*

$$SEQ(ev\_ring(A,C,self) \wedge \neg\ idle(self),\ SEQ(true, hold(A,self) \wedge ev\_ack(A,self) \wedge \neg ev\_ring(A,C,self)))$$

[1] This step is decidable, because the sets of variables and subscribers are finite.

Table 1: Restriction rules

Rule	Formula
1	$ev\_ring(A_1, A_2, A_3) \wedge ev\_denial(A_1, A_3)$
2	$ev\_ack(A_1, A_2) \wedge ev\_denial(A_1, A_2)$
3	$ev\_ring(A_1, A_2, A_3) \wedge ev\_ack(A_1, A_2)$

If the CW subscriber is busy and receives a call, (s)he puts the caller on hold.

The CFB and CW pre-conditions are equal and the variable binding is irrelevant here. The conjunction of the post-conditions reveals a violation of restriction on rule number 3.

$$(ev\_ring(A, bob, P1) \wedge \neg ev\_ring(A, C, bob)) \wedge (hold(A, bob) \wedge ev\_ack(A, bob) \wedge \neg ev\_ring(A, C, bob))$$

3.2 Transitive incoherence

This algorithm detects the case where one feature launches an event that triggers a second feature, and the combination of the second feature's post-condition is incoherent with the post-condition of a third feature. The algorithm of transitive incoherence differs from the direct incoherence in the first step, as follow

1. Select two features, with one feature launching in its post-condition an event that triggers a third feature.

Example 3.3 *Consider the feature* Outgoing Call Screening-*OCS, defined as*

$$\begin{array}{l} SEQ(ev\_offhook(self) \wedge idle(self), \\ \quad SEQ(ev\_dial(self, C) : L \ \wedge \ Stat : \\ \qquad empty- = (P1, null), \\ \qquad equal- = (P1, C), \\ \qquad diff- \neq (P1, B)), \ Stat : \\ \quad empty- ev\_ring(self, self, C) \wedge CD(C), \\ \quad equal- \neg idle(self) \wedge \neg ev\_offhook(self) \wedge \\ \qquad \neg ev\_dial(self, C) \wedge ev\_denial(self, C), \\ \quad diff- OCS(tail))) \end{array}$$

The three alternatives are: the black list is empty, the destination is equal to the head of the list, the destination is not equal to the head of the list. Suppose that Ann subscribes OCS(bob,charles) and calls Carol. Suppose also that Carol subscribes CFB(bob). Finally, consider that the following bindings hold, $\{A \mapsto ann, B \mapsto carol\}$ *for CFB and* $\{A \mapsto bob\}$ *for OCS.*

The pre-conditions are not contradictory. The conjunction of the post-conditions reveal a violation of restriction on rule number 1, for the equal alternative.

$$\begin{array}{l} (ev\_ring(ann, carol, bob) \wedge \\ \quad \neg ev\_ring(ann, carol, carol)) \\ \wedge \\ (\neg idle(ann) \wedge \neg ev\_offhook(ann) \\ \quad \neg ev\_dial(ann, bob) \wedge ev\_denial(ann, bob)) \end{array}$$

3.3 Loop detection

In this algorithm, one features launches an event that triggers a second feature which, in turn, launches an event that triggers again the first feature again.

The algorithm of loop detection differs from the transitive incoherence in the fourth step, as follow

4. If the second feature launches an event, which triggers the first feature, the pair of features is candidate for a FI.

Example 3.4 *Suppose that Ann and Bob subscribe* Call Forward Always-*CFA to each other. CFA is defined as follow*

$$\begin{array}{l} SEQ(ev\_ring(A, C, self), \\ \quad SEQ(true, ev\_ring(A, self, P1) \wedge \\ \qquad \neg ev\_ring(A, C, self))) \end{array}$$

The event $ev\_ring(A, C, bob)$ *triggers features in succession, in a loop.*

- *the event* $ev\_ring(A, C, bob)$ *executes CFA for Bob, which launches the event* $ev\_ring(A, bob, ann)$.
- *the event* $ev\_ring(A, bob, ann)$ *executes CFA for Ann, which launches the event* $ev\_ring(A, ann, bob)$.
- *the event* $ev\_ring(A, ann, bob)$ *executes CFA for Bob, again.*

4 Future Work

The methods of feature interaction definition and detection presented in this paper are, by no means, the only ones in existence. Further studies must be conducted, leading to methods to identify FI between more than two features.

The representation languages for the description of the modality and the global selection layers must be easily integrated with the FET language, namely its semantics.

Acknowledgements

We express our gratitude for the remarks and suggestions of Dr. Daniel Amyot, of Mitel Networks, Jacques Sincennes, of University of Ottawa and Jenny Zhang.

This research has been partially funded by the Fundação Ciência e Tecnologia of Portugal, project SFRH/BASB/201/001, Communications and Information Technology Ontario, Mitel Communications Systems, and the Natural Sciences and Engineering Research Council of Canada.

References

[1] N.H. Gehani and H.V. Jagadish. Ode as an active database: Constraints and Triggers. In *17th Int'l Conference on Very Large Databases*, pages 225–234, 1989.

[2] B. Krishnamurthy and D.S. Rosenblum. Yeast: A Genereal Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[3] R.J.A. Buhr. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.

[4] Ø. Haugen. MSC-2000 interaction diagrams for the new millenium. *Computer Networks*, 35(6):721–732, 2001.

[5] A. Aho, S. Gallagher, N. Griffeth, C. Scheel, and D. Swayne. Sculptor with Chisel: Requirements Engineering for Communications Services. In *5th Int'l Workshop on Feature Interations in Telecommunication and Software Systems*, pages 45–63, September 1998.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[7] ITU. *Recommendation Z.100, Functional Specification and Description Language (SDL)*. International Telecommunications Union, 2000.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, October 1969.

[9] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[10] P.B. Ladkin and S. Leue. What Do Message Sequence Charts Mean? In *6th IFIP Int'l Conference on Formal Description Techniques*, pages 301–316, 1992.

[11] P. Graubman, E. Rudolph, and J. Grabowski. Towards Petri Net based semantics definition for Message Sequence Charts. In *SDL'93-Using Objects*, October 1993.

[12] K.J. Turner. Formalising the Chisel Feature Notation. In *6th Int'l Workshop on Feature Interations in Telecommunication Networks*, pages 241–256, 2000.

[13] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[14] R. Eschbach, U. Glässer, R. Cotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In *Abstract State Machines 2000*, LNCS 1912, pages 242–265, 2000.

[15] R.G. Crespo. The Semantics of Feature Execution Trees. In 3^{rd} *IEEE and IFIP WG10.1 Joint Workshop on Formal Specifications of Computer-based Systems*, pages 39–46, April 2002.