Predicting feature interactions by using inconsistency models

Rui Gustavo Crespo *

INESC-ID/Technical University of Lisbon, DEEC, Av. Rovisco Pais, 1049-001 Lisboa, Portugal

Abstract

Internet applications, such as Email, VoIP and WWW, have been enhanced with many features. However, the introduction and modification of features may result in undesired behaviours, and this effect is known as feature interaction—FI.

In this paper we propose a proactive approach for FI detection. Supported by sets of all possible events, predicates and inconsistent behaviours, we generate hypothetical new features that interact with a given feature. By predicting FIs, the feature subscriber may define, in advance, all mechanisms to resolve the FIs that may occur in the future.

We adopt a semantic model, based on group theory, for the feature axiomatic specification. The algorithms that generate new features do not depend on the particular data structures used in the semantic model of the feature specification.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Internet applications are being enhanced with many features. A feature is defined as a unit of functionality existing in a system and is usually perceived as having a self-contained functional role [1].

The presence of one feature modifies or subverts the operation of another one and this problem is known as feature interaction, or FI for short [2]. More precisely, a FI occurs if the execution of two features precludes the fulfillment of one of the feature’s goals, or satisfies an inconsistency.

The FI problem, first identified in circuit-switched networks, has been studied in many Internet applications, such as Email [3,4], VoIP [5,6], WWW [7] and networked home appliances [8].

Example 1.1. Suppose that Bob instructs the Email server to execute the ForwardMessage feature, forwarding all messages to Carl. Suppose also that Carl subscribes to the AutoResponder feature, by activating the Unix vacation program. A message that Alice sends to Bob is forwarded to Carl. Thereafter, the Email server of Carl accepts Alice’s message and sends a notification message to Bob, not to the message initiator (Alice). This result goes against the initial goal of AutoResponder feature, to notify the initiator that Carl is on vacation. The Email server of Bob, when it receives the notification message, forwards it back to Carl. The Email server of Carl detects a loop, another FI because it satisfies the inconsistency that an unrejected message is never delivered, and discards the notification message.

The increasing number of FIs, and the inconvenience they are causing, has led industry and researchers to meet regularly at the Feature Interactions in Telecommunications and Software Systems conferences, 10 of which have been held from 1992 to 2009.

Three basic problems have been studied [9], avoidance, detection and resolution. Avoidance means to intervene at the protocol or design stages to prevent FIs, before features are executed. Detection aims at the identification of FIs, with suitable methods. In the resolution, actions are exercised runtime over triggered features, which averts a FI.

1.1. Feature interaction types

Several taxonomies have been produced to classify different types of FIs. The most used benchmark of [10] identifies three dimensions, the number of parties involved in

* Tel.: +351 21 8418626; fax: +351 21 8417499.
E-mail address: R.G.Crespo@comp.ist.utl.pt.

1389-1286/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2009.09.010
the interaction (single or multiple), the number of network components involved in the interaction (single or multiple), and the kind of features involved in the interaction (custom or system). We concentrate on custom features because of their greater diversity in production and combination.

1.2. Approaches for FI detection

Research on FI detection has been focused on methods to represent features and check if a given pair of features are likely to interact. This can be done by using one of the many static FI detection algorithms found in the literature. A review of several existing methods is given in [11].

Usually, FI detection focus mainly on message control. Attempts to include conflicts allocation, an essential issue in Multimedia, are addressed in [12]. Such issue is not analyzed in this paper, to avoid the complexity increase in our approach.

FI detection methods adopt property identification, behaviour approaches or both [11]. Properties analyzed in FI detection include inconsistency, releasability and satisfiability. Three approaches for FI detection have been explored so far. They are (i) simulation oriented, where traces of features running together are scanned [13], (ii) model based, where dedicated tools [14,15] unfolds system requirements into a transition system and system properties are checked, and (iii) theorem proving, where it is verified (manually [16] or with tool support [17]) if a conjecture-an unacceptable behaviour-is a logical consequence of the hypothesis-statements hold when features are subscribed.

Security based approach of FI detection, centered on Aspect-Oriented Programming have recently attracted attention [15]. In AOP, an advice declared outside the object, is executed on class/method entries or exit. If a new class or a new method is added, the advice is also executed on its entry or on its exit (if an existing pointcut matches the new class or the new method). However, the security model embedded into the advice code may become outdated with new methods because project teams cannot forecast all possible system updates.

The reactive approach for FI detection listed above reveals the disadvantage that, when a new feature is added or a subscribed feature is updated, a new process of FI detection must be executed. This process may lead to the identification of new interactions, which require the modification of previous resolution practices.

Our proactive approach seeks to derive all hypothetical features that may later be subscribed. Given one feature, one set of possible events that may be generated in the system lifetime and the conditions that reveal a FI, we try to generate new features which would make FIs to occur. The predictive approach anticipates potential FIs, which provides to the feature subscriber the advantage of taking the necessary precautions in advance in the definition of FI resolution methods [18]. The predictive approach to FI is directed to take place between the feature implementation and the definition of FI resolution.

1.3. Article outline

In Section 2 we describe two representation schemes of features, the axiomatic specification and the semantic model.

We propose an innovative approach for the semantic model, called CTR, based on group theory [19]. The semantic model is suitable for FI detection, described in Section 3, and predictive analysis, described in Section 4.

In Section 5 we briefly describe one predictive FI tool and its application to a set of 10 widely-used Email features.

2. Feature representation schemes

We adopt [20] statement of FI problem, where feature representation is divided in two major parts: feature specification and feature constructs. Each part, or both parts, may be absent.

For our goal, feature constructs are irrelevant because we are not concerned in the feature design or programming language used in feature implementation. For the case when feature specifications are present and feature constructs are absent, Bruns [20] identifies FIs by the satisfaction of properties when specifications are conjuncted. Examples of such properties are inconsistency and unattainability.

In our work, features specifications are represented axiomatically with an analytical semantic model.

The axiomatic specification represents features by events, which trigger features and are generated by parties or by features, and the conditions observed. The axiomatic specification is a readable representation, extensively used by industry and researchers. For example, in FI contests features are described by state machines [21].

The analytical semantic model, where implementation details are avoided, is automatically generated from the feature specification.

The examples depicted in this section are based on Email features, but may also be easily applied to other Internet applications, such as WWW and VoIP. This assertion is result of the independence of the axiomatic specification to the implementation and the behaviour similarity of many WWW, VoIP and Email features. For example, WWW Refresh and VoIP CallForwardAlways features hold the same behaviour as ForwardMessage Email feature.

2.1. Feature axiomatic specification

The choices for representing features are vast, many of them adapted to FI detection approaches listed in Section 1.2. Representation schemes may be divided into informal and formal specifications. The former has been based on scenario based presentation schemes, such as UCM [22]. The latter may be further divided between (i) state machine, such as extended FSM [23] and STR [24], (ii) concurrent schemes, such as Petri-nets [25], process algebra [26] and event-based [27], and (iii) temporal logics [28].
We restrict our attention to the problem of identification of FI, not to the feature development. Central to the feature specification are events, or messages, defined as signals generated by a party, or by a feature, that may carry information.

For the feature specification, we extend the axiomatic specification STR language [24], which uses first-order predicate logics [29] to identify properties observed prior and after processing an event. Axiomatic specifications are readable and well known in many fields of computer science.

**Definition 2.1.** A feature specification is axiomatically specified by a non-empty set of quadruples in the form

\[(T, P, G, R)\]  

- \(T \in EE\) is one event that triggers the feature.
- \(P\) is the condition assumed before the feature execution, referred as precondition.
- \(G \in 2^{EE}\) is a set of new events generated in the execution of the feature.
- \(R\) is the condition satisfied after the feature execution, referred as postcondition or assertion.

The third element of the quadruple, \(G\), extends STR definition. Each quadruple specifies a feature behaviour.

Preconditions and postconditions are formulas of predicates, negation and conjunction of formulas.

Features must be deterministic, because users require that feature execution does not vary when triggered by the same event and the same precondition holds. Therefore, preconditions must be mutually exclusive among different feature behaviours, when the feature is triggered by the same event.

Variables denote feature parties and are represented by uppercase letters \(A, B\) or \(C\). The communication community conventioned that the feature subscriber \(\text{self}\) is a person whose name starts with letter \(A\) (e.g., alice), the second party in the connection is a person whose identifier starts with letter \(B\) (e.g., bob) and the third party is a person whose name starts with letter \(C\) (e.g., charles).

Predicates, listed in Table 1, state conditions over parties.

We consider an event as a structured data type. We are not concerned how events may be operated concurrently. Usually, information carried by events are parties involved in the feature execution.

The **retrieve(!)** event allows user \(I\) to collect a message held by the node that manages MailHost feature, for example using the POP3 protocol [30].

**Table 1**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias(A, C)</td>
<td>(C) is an alternative reference for (A)</td>
</tr>
<tr>
<td>held(A, C)</td>
<td>Message from (A) is held for deliver to (C)</td>
</tr>
<tr>
<td>interdict(A, C)</td>
<td>(A) always rejects messages from (C)</td>
</tr>
<tr>
<td>maps(A, C)</td>
<td>Anonymous (A) is mapped to party (C)</td>
</tr>
<tr>
<td>reads(A, C)</td>
<td>Party (A) reads a message written by (C)</td>
</tr>
<tr>
<td>registered(A, C)</td>
<td>Party (C) is known to (A)</td>
</tr>
</tbody>
</table>

In some cases, such as messages involving authentication and confidentiality – see [31], an event extends another base event with extra information. In these cases, the extra information is stored in arguments positioned after those of the base event. In our work we have one base event, \(\text{send}(I, O,T)\), which states that a message, initiated by \(I\) and last processed by the originator \(O\), is sent to the intended terminator \(T\). Extended events are listed in Table 2.

Informally, if the event \(E\) triggers the feature and \(P\) holds, the trigger event is consumed and the feature execution generates a set of new events. At the end of feature execution, \(R\) is satisfied. As an important rule for FI detection, preconditions that remain satisfied after the feature execution must also be stated in the postcondition, even if the feature does not generate new events.

**Definition 2.2.** The direction condition of trigger events and of generated event are, respectively, the precondition, and the postcondition.

In the **Example 2.1** we provide the axiomatic specification for three features described in [3]. Examples 4.1 and 4.2 describe and specify three more features.

**Example 2.1.** Consider the message \(\text{send}(A,B,\text{self})\) initiated by \(A\), delivered to \(\text{self}\), which triggers the **ForwardMessage** feature. The feature generates a new event \(\text{send}(A,\text{self},C)\), which represents a message initiated by \(A\), originated by the **ForwardMessage** subscriber with \(C\) as the intended terminator. The party \(C\) is defined by the feature’s subscriber. **ForwardMessage** feature does not encompass any condition, hence precondition and postcondition are equal to true.

**ForwardMessage** specification is

\[ (\text{send}(A,B,\text{self}), \text{true}, \text{send}(A,\text{self},C), \text{true}) \]

**AutoResponder** axiomatic definition, described in **Example 1.1**, is composed of two quadruples, each quadruple representing a specific behaviour.

\[ (\text{send}(A,B,\text{self}), \text{registered}(\text{self},A), \text{self}, \text{interdict}(\text{self},A), \text{self}) \]
\[ (\text{send}(A,B,\text{self}), \text{reads}(\text{self},A), \text{self}) \]

**FilterMessage** feature screens the message initiator. The message is read only if the initiator does not belong to a “black list”.

\[ (\text{send}(A,B,\text{self}), \text{interdict}(\text{self},A), \text{self}, \text{interdict}(\text{self},A)) \]
\[ (\text{send}(A,B,\text{self}), \text{reads}(\text{self},A), \text{self}, \text{interdict}(\text{self},A)) \]

In first-order predicate logics, variables may only denote elements in the domain of participation, not predicates. To make easier the modification of conditions, which may lead to the satisfaction of properties occurring in a FI, we created an analytical semantic model for the feature specification.

**2.2. Feature analytical scheme**

The semantic model for the feature specification, named CTR-coded transitional representation, is targeted to satisfy the following list of goals:
• Feature specification may be automatically transcribed to CTR, without semantic modifications.
• It is possible to detect candidates for FI with CTR feature representation.
• Preconditions and postconditions of different features, not necessary equal, may coexist in the system while not destroying properties of each feature behaviour.
• CTR must be as simple as possible, with minimal number of operators.
• Although fit to FI detection and generation, CTR must not be tied to implementation details and efficiency concerns.

The universe of discourse, feature parties, is finite. Therefore, quantification over variables may be replaced by a finite set of variable mappings to parties.

To achieve goals listed above, we propose to use a restricted form of Abelian groups [19]. Simpler mathematical structures would not make available essential operators: for example, monoids do not have inverses which are essential to achieve our goals, to be seen in Section 3.3. Other more powerful mathematical structures would not match the requirement of CTR to be as simple as possible: for example, euclidean rings guarantee unique factorization, as stated in Definition 2.4, but require two binary operators.

Definition 2.3. An Abelian group is a tuple $(S, 0, 0^{-1}, \cdot)$, where $S$ is a carrier set (whose elements we designate as codes) and the group axioms (1) to (4) are satisfied

1. $\cdot : S \times S \rightarrow S$ is an associative group operator,
2. $0 \in S$ is the identity element, such as $\forall s \in S : s \cdot 0 = 0 \cdot s = s$,
3. $0^{-1} : S \rightarrow S$ is the inverse operator, such as $\forall s \in S : s \cdot 0^{-1} = s^{-1} \cdot s = 0$,
4. $\cdot$ is commutative.

The semantics of a conditional state containing coded condition and the meaning of an event is a transition between states labeled by the coded event.

The meaning of a predicate is a group element and the meaning of a conjunction is the group operator applied to operand’s meaning.

The algorithm for transcribing a feature behaviour into a CTR representation is depicted in Fig. 1. The group carrier set must have the cardinality at least equal to the number of unary predicates, positive and negative, plus the number of event components.

When transcribing different features, variables must be considered different even if they hold the same identifier.

2.3. Layering groups

In the process of FI detection and FI prediction, states may contain elements beyond those identified in the steps of condition transcription. This combination is implemented by the group operator $\cdot$.

When a FI occurs, the participating features are executed in parallel or in sequence. On these occasions, some predicates are satisfied and some events occur in both features. Therefore, for FI prediction we need to recover predicate and event representations in every representation of feature specification. This goals motivates the definition of C-layered sequence.

Definition 2.4. Given a group $G = (S, 0, 0^{-1}, \cdot)$ and a non-empty set $C \subseteq S \setminus \{0\}$ of fully unclosed elements, designated as base code, the it C-layered sequence of $G, L^1, L^2, \ldots$ is defined as

1. Sequence elements are mutually exclusive, i.e., $i \neq j \Rightarrow L^i \cap L^j = \emptyset$.
2. $L^1 = C$ is fully unclosed to itself, i.e. the group operator applied to every pair of different $C$ elements is never a $L^i$ element.
3. An element $b$ belongs to the subset $L^i$, if the minimal number of different $L^1$ elements $(a_1, a_2, \ldots, a_l)$, such as $b = a_1 \cdot a_2 \cdot \ldots \cdot a_l$, is equal to $i$.

For a base set with $N$ elements, the number of C-layered sequence elements is $\sum_{i=1}^{N-1} \binom{N}{i} = 2^N - 1$.

Example 2.2. The quotient group $\mathbb{Z}/n\mathbb{Z}$, whose carrier set is isomorphic to all integers between 0 and $n - 1$, is an example of Abelian group. The operator is sum modulo $n$, 0 is the identity element and the inverse operator of $a$ is equal to $n - a$ modulo $n$.

Consider group $\mathbb{Z}/16\mathbb{Z}$. The base set $C = \{1, 2, 4, 8\}$ is fully unclosed and the layers are $L^2 = \{3, 5, 6, 9, 10, 12\}, L^3 = \{7, 11, 13, 14\}$ and $L^4 = \{5\}$.

Table 2. Extended events.

<table>
<thead>
<tr>
<th>Event</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendE(I, O, T, PK)</td>
<td>PK is the party whose public key is used to cipher/decrypt data</td>
</tr>
<tr>
<td>sendS(I, O, T, PK)</td>
<td>PK is the party whose secret key is used to cipher/decrypt data</td>
</tr>
<tr>
<td>sendA(I, O, T)</td>
<td>Used in the RemailMessage feature, where I is an anonymous address</td>
</tr>
<tr>
<td>sendB(I, O, T)</td>
<td>Used in the AddressBook feature, where T is an alias</td>
</tr>
<tr>
<td>sendN(I, O, T)</td>
<td>Message unrelated to the triggered message</td>
</tr>
</tbody>
</table>

Please cite this article in press as: R.G. Crespo, Predicting feature interactions by using inconsistency models, Comput. Netw. (2009), doi:10.1016/j.comnet.2009.09.010
Although 11 = 5 + 6, neither 5 nor (6) are base set elements and the element 11 is identified in the C-layered sequence as 8 + 2 + 1.

For the group $(\mathbb{Z}/16\mathbb{Z})$ and base set $C = \{1, 3, 5\}$, the layers are $L^1 = \{4, 6, 8\}$ and $L^2 = \{9\}$.

Table 3 depicts two examples of layering groups, exponential powers under sum and multiplicative group of invertible integers modulus a prime number.

The group of exponential powers is a special case of integers modulus $n$ group, $\mathbb{Z}/n\mathbb{Z}$, with $n$ equals to a power of two. The basic codes used to generate a number are simply the base two representation of the number.

The group of invertible integers modulus prime number is a special case of the multiplicative group, $(\mathbb{Z}/n\mathbb{Z})^*$, with $n$ equals to the smallest prime number equal or larger than the products of the first Max primes.

The number of C-layered elements for $\mathbb{Z}/n\mathbb{Z}$ is $2^n - 1$.

The number of C-layered elements for $(\mathbb{Z}/n\mathbb{Z})^*$ is lower than $2^n - 1$ because there are numbers divided by a power greater than 1 of a prime number. For example, $4 = 2^2$.

Example 2.3. Consider that Alice subscribes ForwardMessage and FilterMessage features, specified in Example 2.1. Consider also that the feature’s environment contains two other parties, named Bob and Charles.

For coded representation of event send, with three arguments, we identify three unary components $send_1, send_2$ and $send_3$. read and interdict hold arity equal to 2, hence both predicates are replaced by two unary predicates.

Table 4 depicts one possible mapping for the events and predicates, using layering group of 2 exponential powers. A minus sign separates positive and negative literals, with positive literal on the left side. Variables used in ForwardMessage and FilterMessage feature specification hold index, respectively, $FW$ and $Fl$. (interdict) predicate has positive and negative connectives. Hence, $interdict_1(alice)$ and $–interdict_1(alice)$ are mapped, respectively, to $2^6$ and $2^{10}$.

Figs. 2 and 3 depict in graphical form, respectively, ForwardMessage and FilterMessage CTR semantics. Events and conditions are depicted by a set of exponential codes. On state, each code represents an unary predicate (true is represented by an empty set). On event, each code represents an event component.

For the ForwardMessage CTR, $s_A$ and $s_B$ represent, respectively, feature starting and ending states. Both states contain value 0, the representation of condition true.

For the FilterMessage CTR, $t_A$ and $t_C$ represent feature starting states for each behaviour. Because $t_C$ condition $\{10, 12\} = 2^{10} + 2^{12}$ is the negation of $t_A$ condition, FilterMessage execution is deterministic.

2.4. Estimating the size of base code sets

Table 3 shows the need to identify the number of basic code elements, Max. The number depends on predicates and events that exist in the feature specification.

We identify the worst-case number of basic codes required to represent a $N$ arity predicate in CTR, when both predicate and its negation are used in the specification. The upper bound of basic codes is then $2 \times N$.

Next, we identify the worst-case number of basic codes required to represent all predicates in CTR, when all pre-condition and postcondition of feature rules are made of different predicates. Considering that all preconditions and postconditions are made of a conjunction of $c$ predicates, the upper bound of different basic codes is $4 \times c \times N + B$ where $B$ stands for the number of specification feature behaviours.

Now, we identify the worst-case number of basic codes required to represent an event in CTR, when all events have a maximal number of arguments $N$. The upper bound of basic codes is $N$.

The number of basic codes necessary for the representation in CTR of all events can be estimated, in the worst case, when all trigger and generated events are different and all features behaviours generate an event. In the worst case, the number is $2 \times N + B$.

Taking the maximum of the size of basic codes necessary for the representation in CTR of all predicates and of all events, we conclude that the number of basic codes nec-
essay for the representation in CTR of one feature is upper bounded by

\[(4 * c + 2) * N * B\]

For the group definition of exponentials, Max is equal to the Eq. (2).

For the group definition of prime numbers, we use function \(\pi(n)\) to identify the number of primes smaller than \(n\). For very large \(n\), Hadamard and Valée-Poussin prime number theorem states that \(\pi(n) = n/(ln(n) - a)\) with \(a \approx 1\) [32]. In our problem, \(\pi(n)\) must be equal to formula (2). The value Max is calculated as the fixpoint of expression \((ln \max - 1) * (4 * c + 2) * N * B\).

It is unusual predicates and events to hold arity greater than 4. Also, conditions usually are constructed with no more than 2 predicates. Therefore, the value of \(B\) gives the greatest contribution to the maximal value contained in a state, or representing an event.

The maximal value contained in a state, or representing an event, is upper bounded with all arguments equal to the variable represented by the largest prime. For groups defined by exponential powers of two, the maximal value is \(2^B\). To identify the asymptotic value of Max value for groups defined by primes, we evaluate how the fixpoint of function \((4 * c + 2) * N * B * ln \max\) evolves. We have \(\frac{\max'}{\max} = \frac{e}{\frac{\ln}{\ln \max}}\) and \(\frac{\ln}{\ln \max}\) is greater than 1, hence Max upper bound is \(\frac{\max'}{\max}\). Therefore, the asymptotic maximal value contained in a state, or representing an event, is \(e^B\).

Although group defined by prime numbers requires less memory, the group operation of sum in groups defined by exponentials is faster.

Example 2.4. Hall [3] describes 10 features with minor restrictions, such as aliases in the AddressBook feature only refer to single member lists, are made of 21 rules. Each condition is constructed with a maximal number of 2 predicates, and predicates and events hold a maximum number of arguments equal to 4.

For a group defined by exponential powers of 2, Max is equal to \((4 * 2 + 2) * 4 * 21 = 840\).

For a group defined by prime numbers, Max is the fixpoint of function \(840 * (ln n - 1) = 6540\). In reality, the 840th prime number is 6473. The maximal value contained in a state, or representing an event, is then \(6473^{2*4} \approx 3 * 10^{30}\). The maximal value is 101 binary digits long, below the 840 bits necessary for the group defined by exponential powers of 2.

Although exponential powers and invertible integers modulus a prime number are groups with same number of C elements, the exponential powers group take more memory because they are allowed to represent conditions with all possible predicates. In reality, the number of predicates in every condition is very small and the register holding the value is mostly filled with zero bits.

3. FI detection

In this section we show that CTR is powerful enough to detect if two subscribed features are candidates for FI.

First, we bind all variables to all possible combinations of feature parties, then we check if a combination of the feature CTR’s is inconsistent.

We depict two algorithms of combining feature CTR’s, derived from [33,17]: parallel walk in Section 3.5 and looping in Section 3.6.

In parallel walk, the system status satisfies two feature preconditions and both features are triggered by the same event. Although only one feature is selected for execution, its postcondition is inconsistent with other feature postcondition.

In looping, the generated event of one feature triggers the second feature and the system enters in a infinite loop of feature execution.

3.1. Inconsistency

In our approach, we consider that one inconsistency rule is satisfied only at the end of the feature execution.

Because every feature is triggered by one single event, each inconsistency rule must contain, at most, one trigger event. To differentiate trigger and generated events, we use \(T\) and \(G\) suffixes in the identifier. Variable \(I\) and \(T\) are bound, respectively, to the message initiator and terminator.

Example 3.1. Table 5 depicts six inconsistency rules, identified with help of experience on FI detection [3]. The table is, by no means, complete and serves to demonstrate the validity of our approach.

The first two rules focus on ciphered messages. Rule 11 states that the same message cannot be sent simultaneously ciphered and in clear. Rule 12 states that the ciphered message cannot simultaneously be read and sent forward in clear. Rules 13 and 14 focus on anonymous messages. Rule 13 states that an incoming signed message cannot be sent anonymously. Rule 14 states that a message cannot be sent signed and anonymously, because the anonymity would be defeated. The last two rules focus on interdict and hold predicates. Rule 15 states a message, initiated from an interdicted user, cannot be read. Rule 16 states that a message cannot be held and sent forward simultaneously.

The inconsistency rule is transcribed to the carrier set of the CTR representation in a way equal to the transcription of a feature specification condition.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>send(G)(I, self, T) \land sendE(G)(I, self, T, PK)</td>
</tr>
<tr>
<td>I2</td>
<td>sendE[T](I, O, self, PK) \land send(G)(I, self, T) \land readIs_self_I</td>
</tr>
<tr>
<td>I3</td>
<td>send(S)[T](I, O, self, PK) \land send(A)(Anon_self, T) \land readIs_self_I</td>
</tr>
<tr>
<td>I4</td>
<td>send(T)(I, I, T) \land send(G)(Anon_anon, T) \land send5(G)(I, I, T, PK) \land mapi_anon_I</td>
</tr>
<tr>
<td>I5</td>
<td>send(T)(I, O, self) \land interdict_self_I \land readIs_self_I</td>
</tr>
<tr>
<td>I6</td>
<td>send(T)(I, O, self) \land send(G)(I, self, T) \land holdI_self</td>
</tr>
</tbody>
</table>
3.2. Constraints over events and predicates

In the axiomatic specification of features, events play a limited role as a trigger for the feature execution and the semantics lie in the feature's conditions. Yet, parameters face constraints on their values across all features. The constraints represent state invariants and depend on the meaning of events and predicates that the axiomatic specifications are made of.

Table 6 depicts, in the upper part, the restrictions that we impose on retrieve and six types of send events, used in [3] features. Predicates also suffer restrictions, which are listed in the lower part of Table 6.

In all events, message originator and terminator must be different because we convey a communication system where information is exchanged between different parties. Also, the only party that may read the ciphered message is the terminator holding the public key.

For some predicates, designated as party invariants, the feature execution cannot modify its value. Therefore, party invariants must be satisfied both in precondition and postcondition. In Table 6, party invariants have a $S$ prefix. The $B$ prefix indicates that, if the predicate is satisfied in the postcondition, then the feature cannot generate events.

Event and predicate restrictions must be checked prior to variable binding to feature parties.

3.3. Unifying inconsistency rule and specification behaviour

For each event component and for each unary predicate, excluding equality predicates, the transcription of a specification behaviour to CTR generates unique group elements. Because we look for events that occur in the CTR of two different features, unary predicate's argument variables of the same condition (precondition or postcondition) and event component's argument variables of the same type (trigger or generated) must be bound to the same party. Moreover, the representation of the unary predicate in the same condition and the event component of the same type should be the same group element.

Definition 3.1. Let $X$ be an inconsistency rule variable and $F$ a feature specification behaviour. $\text{mirror}(X,F)$ is the set of variables and parties in $F$ that are parameters in the same positions of variable $X$ in predicates of same identifiers or events of the same type (trigger or generated) with same identifier prefix.

The algorithm for unifying the variables of an inconsistency rule and a specification behaviour is depicted in Fig. 4.

The algorithm implements the transitive closure of variable bindings. The algorithm starts to identify all variable mirrors, followed by an initial consistency check that includes the bind of specification variables to the rule parties. Then, it follows the main cycle by first checking if there are mirrors with more than one party (in this case the algorithm fails), secondly binding variables whose mirrors contain parties in number of one, and thirdly binding mirrors with no parties.

In the initial consistency check, the same component means the equality of predicate and event identifiers (for events, extension names are acceptable) and feature position equality (for predicates precondition and postcondition, for events only trigger or generated).

Example 3.2. Consider the unification of inconsistency rule 12 and the second behaviour of AutoResponder specification. The mirror set of $I$ over the second behavioural specification of AutoResponder is evaluated as follows.

$I$ appears in I2 as parameter of $\text{send}(G_{1}, \text{sendE}(T_{1})$, and $\text{read}_{1}$ components. The first component in the generated event of the specification is $\text{sendN}_{1}(\text{self})$, hence $\text{self} \in \text{mirror}(1, \text{AR2})$. The first component in the trigger event of the specification is $\text{send}_{1}(A)$ and, because $\text{sendE}$ is an extension of $\text{send}$ event, $A \in \text{mirror}(1, \text{AR2})$. Finally, $\text{read}$ is a predicate in the second behavioural specification of AutoResponder, therefore $A \in \text{mirror}(1, \text{AR2})$. We conclude that $\text{mirror}(1, \text{AR2}) = \text{self}, A$.

Fig. 5 depicts AR2 and I2 events, trigger on the left and generated on the right, and shows how variables $I$ and $A$ are unified.
We have \( \text{mirror}(l, \text{AR2}) = \text{self} \cdot A \), hence we bind \( l \) to \( \text{self} \). Because \( l \) and \( A \) must be bind to the same party, \( A \) is also bind to \( \text{self} \).

Because \( \text{mirror}(T, \text{AR2}) = A \) and \( A \) was bind to \( \text{self} \), \( \text{mirror}(T, \text{AR2}) \) value is updated to \( \text{self} \). We do not continue the propagation of \( \text{self} \) bind from \( T \), because a later MainCycle will perform the \( T \) binding.

As a side note, the bind of \( A \) to \( \text{self} \) will make the variable binding fail, due to the send restriction that the originator and terminator must be different — see Table 6.

Equality and inequality predicates, defined at the direction condition or at the inconsistency rule, restrict variable bindings. As an example, consider a feature with trigger event equals to \( \text{send}(A, \ldots) \). If there are no parties as arguments in the \( \text{send}_l \) component, we may bind variable \( A \) to every party. However, if precondition includes a predicate \( A \neq \text{alice} \), she would be excluded in the \( A \) bindings.

Consider that a condition, which includes the unary predicate \( \phi(Y) \) represented by \( a \), is represented by the group element \( A \). If there is a need to bind \( Y \) to \( X \) and \( \phi(Y) \) is represented by \( b \), then the condition representation changes to \( a \cdot a^{-1} \cdot b \).

### 3.4. Mirrors containing variables only

In some cases, \( \text{mirror}(A, \text{Spec}) \) only contains feature variables. In this case, the actor to whom the variables are bind, must satisfy event and predicates restrictions as well equality conditions.

For all mirrors \( \text{mirror}(A, \text{Spec}) \) that only contain variables, excluding the case where \( A \) is an access code, we look for all combinations of \( A \), person (person is a member of all possible parties) until the event and predicate restrictions are satisfied.

**Example 3.3.** Consider ForwardMessage and FilterMessage features and their CTRs depicted in Example 2.3. The initial set of mirrors has no parties: \( \text{mirror} (A_{FW}, \text{Fl}) = \{A_{1}\}, \text{mirror} (A_{F\text{F}}, \text{FW}) = \{A_{\text{FW}}\}, \text{mirror} (B_{FW}, \text{Fl}) = \{B_{1}\}, \text{mirror} (B_{FW}, \text{FW}) = \{B_{FW}\}, \text{mirror} (C_{FW}, \text{Fl}) = \{\emptyset\} \).

All mirror sets, excluding \( \text{mirror} (C_{FW}, \text{Fl}) \) have the same size, hence we pick up randomly \( \text{mirror} (A_{FW}, \text{Fl}) \) and choose a party. In light of restriction that first and third parties in the \( \text{send} \) event must be different, and third party is \( \text{alice} \), we select \( \text{bob} \). Therefore, we select the same exponential to represent \( A_{FW} \) and \( A_{\text{FF}} \), for example (0).

Similarly, \( B_{FW} \) and \( B_{\text{FF}} \) are bind to the same party. This time there are no restrictions in the party selection and we choose \( \text{alice} \). Therefore, we select the same exponential to represent them, for example 3. \( \text{mirror} (C_{FW}, \text{Fl}) \) is an empty set. Again, there is the restriction that first and third party must be different. Because \( A_{FW} \) in the generated event is bind to alice, we choose \( \text{charles} \) and its representation number remains the same.

With these bindings, all trigger events, from \( s_4 \) to \( s_9 \), from \( t_3 \) to \( t_8 \) and from \( t_2 \) to \( t_9 \), have the same representation \( \{0, 3, 6\} \).

### 3.5. Parallel walk

An inconsistency may be reached if the same sequence of events, which trigger features or are generated by features, walk in parallel along two different features and reaches a state where conditions satisfy an inconsistency rule. The parallel walk along two different features requires that one feature CTR may be integrated into the other feature CTR.

**Definition 3.2.** One feature CTR may be integrated into another feature CTR if all group elements of the starting states of both feature may be joined and represent a consistent state, and the group elements of trigger event component’s representation of the first feature are also group elements of trigger event component’s representation of the second feature.

For example, \( \text{true} \) is a condition, whose representation of identity element is always a group element in any state representation. However, a predicate in one feature precondition and its negation in another feature precondition makes impossible the integration of those features.

The group elements of trigger event \( \text{send}(A, B, C) \) are group elements of trigger event \( \text{send}(l, O, T, PK) \) if and only if \( \text{c} \) \( \text{send}_l(A) = \text{c} \) \( \text{send}_O(C) \) and \( \text{c} \) \( \text{send}_T(C) \).

**Example 3.4.** Consider Example 3.3. \( s_4 \) may be integrated into state \( t_1 \) and \( s_8 \) may be integrated into \( t_9 \), because \( s_4 \) and \( s_8 \) are represented by the identity element. The trigger event has the same representation, therefore it is possible to integrate ForwardMessage CTR into FilterMessage CTR and walk in parallel on both features.

The ending state \( t_9 \) contains the unary predicates \( \text{read}_1(A_{\text{FF}}) \) and \( \text{read}_2(\text{alice}) \) and the ending state \( s_9 \) generates one event with components \( \text{send}_l(A_{FW}) \) and \( \text{send}_2(\text{alice}) \). Example 3.3 binds \( A_{\text{FF}} \) and \( A_{FW} \) to the same party, hence the same message is read and sent forward and this is inconsistent according to rule 12.

### 3.6. Looping

Looping is a FL that occurs when events generated by one feature triggers another feature in a chain, until the initial state is reached. In the looping FL, we look for the variable binding of generated and the trigger events in all possible feature chains.

**Table 7** Mappings for two ForwardMessage CTRs

<table>
<thead>
<tr>
<th>Event</th>
<th>( A_{FW1} )</th>
<th>( B_{FW1} )</th>
<th>( C_{FW1} )</th>
<th>( A_{FW2} )</th>
<th>( B_{FW2} )</th>
<th>( C_{FW2} )</th>
<th>alice</th>
<th>bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>send1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send2</td>
<td>5</td>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>[ \emptyset ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please cite this article in press as: R.G. Crespo, Predicting feature interactions by using inconsistency models, Comput. Netw. (2009), doi:10.1016/j.comnet.2009.09.010
Example 3.5. Consider the ForwardMessage feature, specified in Example 2.1, subscribed by Alice and Bob with messages sent to each other. Table 7 depicts a possible mapping for the send1 component events, coded by the multiplicative group of invertible integers.

The trigger events for Alice and Bob ForwardMessage feature are represented, respectively, by \(2 \times 5 \times 23\) and \(3 \times 7 \times 29\) codes. The generated events for Alice and Bob are represented, respectively, by \(2 \times 11 \times 17\) and \(3 \times 13 \times 19\) codes.

The algorithm of variable bindings unifies the representation of send1 component, for example to 2. This means the message may be initiated by any party.

Alice’s trigger event component \(send2(B_{FW1})\) is unified to the Bob’s generated event component \(send2(bob)\) and, therefore, their representations become the party representation 13. In an similar way, \(send2(B_{FW2})\) representation becomes equal to the \(send2(alice)\) representation, i.e., 11. Also, \(send3(C_{FW1})\) representation becomes equal to the \(send3(bob)\) representation, i.e., 29, and \(send3(C_{FW2})\) representation becomes equal to the \(send3(alice)\) representation, i.e., 23.

All states are represented by the identity element. The representation of the Alice’s trigger event is equal to the Bob’s generated event, \(2 \times 13 \times 29\), and the Bob’s trigger event is equal to the Alice’s generated event, \(2 \times 11 \times 23\).

We then conclude the pair of ForwardMessage features is a candidate of looping FI. Email servers reject looping, by checking if the current node is listed in the processing nodes [34]. However, if a web page depicts a clock, Refresh becomes an acceptable FI.

4. FI prediction

For a given feature (referred as base feature) and a set of possible events that may occur, we look for a hypothetical new feature (referred as generated feature) that could lead to an inconsistency at the ending state. Looping is not considered here, because it is restricted to features with equal trigger and generated events, and precondition equal to postcondition.

We devised two algorithms for the generation of the new feature, simultaneous walk and side by side walk. The two algorithms are based on the model expressed by a set of inconsistency rules and the restrictions that events and predicates must satisfy. Frequently, the two algorithms use events that extend other events.

The two methods differ on the way predicates of the inconsistency rule, in number greater than one, are divided between the postconditions of the feature under analysis and the new feature. For the simultaneous walk, described in Section 4.2, at least one of such predicates must be a sub-formula of the postcondition of the feature under analysis and the predicates of the inconsistency rule, which are not sub-formulas of the feature under analysis, are transferred to the new feature postcondition. Side by side walk, described in Section 4.3, is directed for inconsistency rules with number of generated events greater than the number of generated events of the feature under analysis.

After the generation of new a feature, which interact with a feature under analysis, user must check if the result is a simple basic service. Basic services of read and write are not considered, because they are always surpassed by features.

4.1. Integrating inconsistency rule events and predicates

Inconsistency rules are composed of events and predicates, and we need to define how extended events are merged with base events, and how predicates of the inconsistency rule are split between the two feature postconditions.

All events in the inconsistency rule are used in the feature that is expected to interact with the base feature. The generation of a new feature must satisfy three rules:

1. The events of inconsistency rule and the base feature must be equal or be an extension.
2. The inconsistency rule cannot contain a predicate that is a negation of a predicate in the postcondition of the base feature.
3. If the inconsistency rule contains more than one predicate, at least one must be present in the postcondition of the base feature.

All predicates in the inconsistency rule are split between the postcondition of the base and generated features. Rule 1 is required for a successful variable binding, see algorithm depicted in Fig. 4. Rule 2 is derived from the requirement that illegal states are not allowed. Rule 3 is derived from the cases when FI is caused between predicates. If there are more than one predicate each feature postcondition satisfy a proper subset of the inconsistency rule predicates, and therefore alone the features do not raise inconsistencies.

4.2. FI prediction under simultaneous walk

For FI prediction under simultaneous walk, we inject one inconsistency at the ending state of the base feature. We move then backwards in the analytical representation of the base feature and check if it is possible to combine trigger events, which defines a new feature that interacts. We do not impose that the base and generated features must have equal number of generated events.

The predicates in the inconsistency rule are divided between the two feature postconditions. We adopted a simple strategy for such division. If a predicate that is a sub-formula of the inconsistency rule is not a sub-formula of the feature under analysis postcondition, we insert it in the postcondition of the generated feature. Otherwise, nothing is done.

Fig. 6 depicts the simultaneous walk algorithm, which generates features candidate to interact with the base one.

The number of parties in the simultaneous walk, _self_, alice and bob, is three because this is the maximal number of parties in all known FIs.

Example 4.1. Consider the ForwardMessage feature, with mappings defined in CTR implementation of exponential
In the ending state.

consistent formulas are viable and we may insert predicate hence the generated feature precondition remains equal to modifies its representation to 20 

ForwardMessage precedence than basic services, 

move then backwards in the analytical representation of 

the base feature. The strategy for the division of the inconsistency rule predicates is the same adopted for the simultaneous walk algorithm.

Fig. 7 depicts the side by side walk algorithm, which generates features candidate to interact with a given one. The first part, initialization, is equal to the simultaneous walk algorithm and is not represented in Fig. 7.

Example 4.2. Consider the AddressBook feature, where the target address is replaced if it is an alias for the real address, otherwise the message goes through. The axiomatic specification is

\[(\text{sendB(self, self, B, alias(B,C), (send(self, self, C), alias(B,C))})\]

\[\times (\text{sendB(self, self, B, alias(B,C), (send(self, self, B, alias(B,C))})\]

Consider the alias(B,C) precondition. One possible mapping in CTR implementation of exponential powers is: \(send(I, _{self}) = 0, (send(I, _{self}) = 1, (send(I, _{self}) = 2, (send(I, _{self}) = 3)\) and \(alias(I, _{self}) = 4\) and \(alias(I, _{self}) = 5\).

Now consider inconsistency rule I11, which contains two generated events. \(send(I, _{self}, T)\) is a generated event in the AddressBook feature, with variable bindings of \(I—_{self}\) and \(T—C\) (for example, charles). We only need to inject, into the generated feature, \(sendE(_{self}, _{self}, T, PK)\). The trigger event can be the same of the feature under analysis, or its extension, provided it conforms to the event restrictions depicted in Table 6.

Because I11 does not contain predicates, precondition and postcondition are equal to (true).

The side by side walking algorithm, applied to the first specification behaviour of AddressBook and under the inconsistency rule (I11), generates the EncryptMessage feature. Its axiomatic specification is

\[(send(_{self}, _{self}, B, true, sendE(_{self}, _{self}, B, _{self}, true))\]

4.4. Current limitations

The model of events and predicates that may exist, and the inconsistency rules that may occur, impose a natural limitation on the generation of features. The determination of the inconsistency rules is a creative task and depends on user experience.
Because of these limitations, we cannot guarantee to generate all hypothetical features that interact. A new feature uncovered by the inconsistency rules model, may interact due to an unforeseen inconsistency rule, or may not interact. The former highlights the need to extend our model. As an example of the later, no interactions are known for AddressBook and ForwardMessage features.

Also, our approach does not predicts FIs based on error insertions, such as invalid public keys (see SignMessage and VerifySignature FI in [3]) and based on the message contents (see EncryptMessage and RemailMessage FI in [3]).

5. FI prediction tool

We implemented one tool to check the validity of the ideas expressed in Sections 3.1–3.4 and 4. The tool, programmed in C, implements the layered group of exponential powers and requires Lex and Yacc tool support. The tool, and specifications for 10 Email features defined in [3], is public available at http://comp.ist.utl.pt/rgc/Generator.zip.

Table 8 depicts the results for the Side by Side walk (SS) and Simultaneous Walk (S) methods applied to 10 Email features defined in [3], for the first four inconsistency rules of Table 5 and restrictions depicted in Table 6. Features are indicated by acronyms, where ICS – “Incoming Call Screening” stands for the FilterMessage feature, and the number suffix indicates the feature behaviour.

The generation of a feature that interacts is represented by a pair method:new feature in the column where the inconsistency rule is depicted. For example, the method Side by Side walk identifies that the AddressBook feature interacts with the EncryptMessage feature and the interaction satisfies IR1 inconsistency rule.

We make three observations about the completeness and quality of the generated features.

First, at least one FI is identified for each of 10 features. There are some specification behaviours, such as MailHost, where a fraction of specification behaviours do not predict FIs under the security model expressed in Tables 5 and 6. Not all FIs identified in [3] are identified with our tool, due to the limitations described in Section 4.4.

Second, some of the generated features are useless. For example, the Simultaneous walk algorithm identifies that the ForwardMessage feature interacts with a combination of RemailMessage and VerifySignature features and in the interaction occurs the IR3 inconsistency rule. The combination, given by \(\langle sendS(I, O, self, PK), true, sendA(Anon, _self, T), reads (_self, I)\rangle\), is not useful, but the goal to our work is to identify features that may interact, not to generate features that provide useful work. Our approach requires only the identification of the model of possible events and predicates, with their restrictions, and the set of inconsistency rules. Because our mind is not bound to the problem of feature usefulness, new features are automatically generated by our approach and the extended FIs provide effectiveness of our approach.

Finally, although our approach identifies all candidates for FI, users must decide if they represent undesirable interactions that must be resolved. For example, ForwardMessage forms a loop to itself. In the web, ForwardMessage may be implemented by Refresh feature. However, if the web page depicts a clock, Refresh becomes an acceptable interaction.

6. Conclusions and future work

We have described in this paper a predictive approach for feature interaction detection, based on a model of inconsistency rules that may occur and events and predicates that may exist, together with restrictions in their use. The inconsistency rule is injected at the postcondition of a base feature and propagated backwards to the precondition. We also implemented one tool to demonstrate the validity of our approach.

The limitations, described in Section 4.4, as well the need to avoid conflicts on media allocation, should be addressed in future work. The use Artificial Intelligence techniques in the identification of FIs occurring on error insertions and extending behavioural schemes to enable the visualization of message contents depict promising aspects for such extensions.

Acknowledgements

We gratefully acknowledge the members of the Computer Security Laboratory of UC Davis, for the insights that have made this paper possible. In particular, I want to thank Prof. Matthew Bishop for all support.

Prof. Luigi Logrippo of UQ en Outaouais, Canada has also been a constant source of inspiration. Many thanks also to the anonymous reviewers, who provided useful feedback on earlier drafts of this paper.

References
