Detecting and Resolving Email Feature Interactions Through Constraints

Rui Gustavo Crespo
INESC-ID / Technical University of Lisbon, Lisboa, Portugal
E-mail: rgc@comp.ist.utl.pt

Abstract

The introduction and modification of features in Internet applications may result in undesired behaviors, and this effect is known as feature interaction-FI. We advocate that CLP-Constraint Logic Programming is suitable enough to detect and resolve FIs, within a non-monotonic system modeled in layers, with interfaces defined by predicate negation.

We illustrate the specification of Email basic services and the ten most widely-used Email features. CLP provides mechanisms to detect FIs through model-checking. FI resolution is implemented above following priority and tail elimination strategies.

1 Introduction

Internet applications have been enhanced with many features. A feature is defined as an atomic 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 raises inconsistencies.

Example 1 Suppose that Bob subscribes to the AutoResponder feature, which informs the message initiator that he will read the message later. Suppose also that Alice instructs the Email server to execute the ForwardMessage feature.

Consider now that Alice sends a message to Bob, to inform that all messages directed to her will be sent forward to him. Because Bob subscribed
to the AutoResponder, a notification message is sent to Alice. However, because Alice subscribed to the ForwardMessage, the notification message is bounced back to Bob. Her Email server discards the message, due to looping, and the notification message is lost.

Three basic FI problems have been studied [3]:

- **Avoidance** intervenes at the protocol or design stages to prevent FIs, before features are executed. In Internet, avoidance is unachievable due to the distributive characteristic.
- **Detection** aims at the identification of FIs, with suitable methods.
- **Resolution** exercises runtime actions over triggered features, which averts FIs.

The rest of the paper starts with a brief introduction to feature interaction problem, in section 1.1, and to constraint logic programming, in section 1.3. Section 2 depicts the feature layered structure, analyzes how interfaces are defined in constraint logic programming, and depicts how basic Email and features are specified. The method to identify FIs is briefly explained in section 3. Section 4 describes FI resolution. Finally, section 5 provides a performance analysis of our approach: we describe how to calculate the number of different messages are sent among \( p \) subscribers in \( nR \) time slots and display field values for the CPU time spent as function of the number of parties and rounds.

## 1.1 FI issues

The FI problem, first identified in circuit-switched networks, also occurs in many Internet applications, e.g. Email [4], VoIP [5], WWW [6] and networked home appliances [7]. Nowadays, with the evolution of distributed information systems, such as Cloud computing, FI becomes a more critical issue.

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, ten of which have been held from 1992 to 2009.

The choices for representing features are vast. We divide representation schemes into informal, semi-formal and formal. The informal schemes have been based on scenario based presentation schemes, such as UCM [8]. The semi-formal schemes may be divided between (i) state machine, such
STR [9], and (ii) diagram languages, see [10]. Several formal schemes have been explored, such as (ii) concurrent schemes, such as Petri-nets [11], process algebra [12] and event-based [13], and (iii) temporal logics [14].

Resolution may only be tackled after detection. FI detection and resolution may be exercised in static and runtime contexts.

FIs can be detected with standard tools of simulation [15], theorem proving [16], prediction [17] and model checking [18]. Three approaches have been explored for FI runtime resolution: one phase (based on tables [19] and on state trees [20]), two phase [21] and negotiation (direct [22], indirect [23] and arbitrated [24]).

Velthuijsen was the first to observe that features are augmentations to a system [3]. We adopt the [18] approach of using constraints to model non-monotonic systems. Accorsi et al. work focus on the FI detection of telecommunication features, which only analyzes call control between parties. Email features raises extra challenges against communication features, because they involve extra manipulation over messages, such as ciphering, and parties may perform specialized tasks, such as serving when a client-server paradigm is adopted.

We direct our attention to the integration of detection and resolution of Email features. Our approach is static, i.e., detection are resolution are performed prior to the feature installation.

### 1.2 Logic Programming

Logic programming is a declarative style of programming that represents and manipulates knowledge. Prolog [25] is one of the best-known language of logic programming.

The language is composed of a number of elements:

- Constant symbols, represented by sequence of characters initiated by a lowercase letter and numbers. Examples of constants are *alice*-one party and *server*-an application node.

- Variables, which bind to constant symbols, are represented by sequence of characters initiated by one uppercase letter. For example, *I* binds to parties and application nodes and *N* binds to rounds.

- Function symbols, which map symbols to symbols, are term constructors and do not have computational meaning.

- Logical operators of negation and disjunction.
• Predicates, representing events and states.

A term is a constant, variable or function applied to terms. A ground term is a term containing no variables. An atom, or literal, is a constant or a predicate.

Knowledge is represented by set of formulas, or rules, expressed in a particular form, the Horn clauses - a disjunction of one positive literal and zero, one, or more negative literals. The Horn clause $A \lor \neg B_1 \lor \neg B_2 \lor \ldots \lor \neg B_n$, represented in (1) stands that, if $B_1$ and $B_2$ and ... and $B_n$ are satisfied (that is, hold the truth value), then $A$ is also satisfied.

$$A : -B_1, B_2, \ldots, B_n.$$  \hspace{1cm} (1)

A program model is a subset of atoms such that, when receiving the value true, all rules are satisfied. Moreover, any proper subset of the members of the model fails to satisfy all rules.

The aim of Prolog programs is to evaluate a single query, i.e., to infer that a formula is inferred from the set of rules. Prolog adopts SLD-resolution as the inference system.

### 1.3 Constraint Logic Programming

Constraint Logic Programming extends (classical) logic programming with concepts from constraints satisfaction [26]. The major extensions on constraint logic language in the rule body are

• variable constraints may be included, e.g. $N < nR$,

• no functions may be used, and

• atoms in the body may be negated with not operator, e.g.

$\text{snd}(I, P, T, L, D) : -\text{out}(I, P, T, L, D),$
$\text{not err}(I, P, T, L, D), \text{not sndDenial}(I, P, T, L, D).$

For constraint logic programming, we follow the stable semantics [27]. The formula represented in (2) stands that, if $B_1$ and $B_2$ and ... and $B_n$ are members of the model, $C_1$ and $C_2$ and ... and $C_m$ are not members of the model, then $A$ is a member of the model.

$$A : -B_1, B_2, \ldots, B_n, \text{not } C_1, \text{not } C_2, \ldots, \text{not } C_m.$$  \hspace{1cm} (2)

The stable model semantics for a constraint logic program identifies all possible models of the program.
To process constraint logic programs, in our work we adopt \texttt{lparse} and \texttt{smodels} tools \cite{28}, available at http://www.tcs.hut.fi/Software/smodels. \texttt{lparse} computes the grounded version of range-restricted constraint logic programs and \texttt{smodels} implements the stable model semantics. Integer functions, such as \texttt{5 mod 3}, are internally evaluated by \texttt{lparse}. Ground functions, such as \texttt{foo(a)} and \texttt{foo(b)}, are treated by \texttt{smodels} as constants.

\texttt{smodels}  can be adapted to execute model checking with the statement \texttt{compute number\{B_1,B_2,...,not C_1,not C_2,...\}}.

The statement indicates a selection of a number of models that include grounded atoms \texttt{B_1,...} and do not include grounded atoms \texttt{C_1,...}. All models are listed when \texttt{number} equals to \texttt{all} or \texttt{0}. If extra models satisfy (no extra models satisfy) the compute statement \texttt{smodels} adds the \texttt{False} (\texttt{True}) statement.

\textbf{Example 2} Consider a simple computing system with two alternative traces with a common element. Figure 1 depicts, on the left, the automaton representation of the computing system. On the right, we depict the equivalent constraint logic program.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{automaton.png}
\caption{Automaton and constraint logic program}
\end{figure}

\begin{verbatim}
sta
st0.
ev1 :- not ev2.
ev2 :- not ev1.
stA :- st0,ev1.
stB :- st0,ev2.
false :- ev1,ev2,st0.
compute 0 \{not false\}.
\end{verbatim}

In the program, states and events are represented by logic constants. Time sequence, which is not represented in the program, is described in section 2.1.1.

The starting state is represented by \texttt{st0}, which is a fact. \texttt{sta} and \texttt{stB} represent the two states reached from \texttt{st0} through the occurrence of, respectively, \texttt{ev1} and \texttt{ev2} events.

Rule \texttt{false} : \texttt{ev1,ev2,st0}. forbids the two events to occur simultaneously.
The program output reports, after the models, the amount CPU time consumed in the model search.

The model check is performed upon a statement made of ground messages that represent one particular FI manifestation. For example, ForwardMessage feature interacts with itself. The undesirable behavior of message looping is revealed by a stable model containing one message from one party to another, which is followed by another message from the second to the first party.

All Email services, basic and features, are coded after the identification of the automaton representation. Moreover, Email services involve a number of parties.

All programs in this paper were executed on a Intel Core 2 machine with 2GB RAM, 3 GHz clock and Linux Kernel 2.6.31 operating system. The specification files are public available at [29].

2 Layered Architecture in CLP

Layering is a widely-used structuring technique, such as Internet and Operating Systems, which separates different units of functionality. Communication between layers happens through predefined, fixed interfaces. Interfaces define the services offered and layer information is hidden to the other layers.

Usually, once the layer interfaces have been defined, the layered system is implemented by programming languages that provide separate compilation, such as C [30] and Java [31]. However, separate compilation makes more difficult to reason about the system. To infer properties, programming languages such as Prolog require the knowledge of all “world” under discussion—the domain or universe of discourse.

We advocate that the layered approach may be applied to non-monotonic systems [32], maintaining the advantages of separation of concerns and the ability to reason about properties. The existence of an upper layer may
change the number of stable models generated by the lower layers, hence the system becomes non-monotonic.

The interface between a layer and above is defined, in constraint logic programming, by a predicate. In the lower layer, the predicate is negated at the body. This approach allows the determination of properties of one layer without the definition of the upper layer: if the predicate is absent from the head of rules, it value is considered to be false and does not interfere on the identification of stable models. The verification of properties about the whole system requires all interface properties to be negated.

\[
\text{event} : \ldots \quad \text{predicateAbove} : \ldots \\
\quad \text{not predicateAbove}, \quad \ldots \\
\quad \ldots \\
\text{Layer N} \quad \text{Layer N+1}
\]

Figure 2: Layer interface in Constraint Logic Programming

We center features around messages, processed by 3-layers. From the lowest layer, we have (1) Basic Email Services defining message read and write. (2) Features, subscribed by users, which extend basic email services. This layer detects FIs through model-checking. (3) FI Resolution that avoids FI detected at the second layer.

There are two interface predicates between Basic Email Services and Feature layers: \text{rdDenial/2} that forbids party to read an incoming message, and \text{sndDenial/5} that forbids party to send a message. The interface predicate between Features and FI Resolution layers is \text{resolvDenial/4}.

### 2.1 Specification of Basic Email Services

For the specification of basic Email services, we first define rounds and message legs. Thereafter, we identify the transition system that describes the party behavior.

Table 1 lists the syntax and meaning of events and states used for describing Email basic services. \(P\) stands for party processing the message, \(I\) stands for party initiator, \(O\) stands for party originator and \(T\) stands for intended party destination. \(L\) stands for message leg (see section 2.1.1). \(D\) stands for processing code (see sections 2.2.2 and 2.2.3).
Table 1: Predicates describing events and states

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>req(I, P, T, L, D)</td>
<td>Party P requests message to be sent.</td>
</tr>
<tr>
<td>snd(I, P, T, L, D)</td>
<td>Message is sent.</td>
</tr>
<tr>
<td>err(I, P, T, L, D)</td>
<td>Message is not sent, due to error.</td>
</tr>
<tr>
<td>States</td>
<td></td>
</tr>
<tr>
<td>idl(P, N)</td>
<td>Party P is idle at round N.</td>
</tr>
<tr>
<td>in(I, O, P, L, D)</td>
<td>Party P receives a message.</td>
</tr>
<tr>
<td>out(I, P, T, L, D)</td>
<td>Party P send a message.</td>
</tr>
<tr>
<td>fl(P, N)</td>
<td>Party P rejects reading the message.</td>
</tr>
<tr>
<td>rd(P, N)</td>
<td>Party P reads the message.</td>
</tr>
</tbody>
</table>

2.1.1 Rounds and Message Legs

We define round as the sequence of events that occur and states that are transversed during the service or features execution, from the idl state back to idl.

An Email message is sent from one party at a specific round to a, possibly different, party and arriving at a later round. We designate the pair of departure and arrival rounds as the message leg.

The search space grows very fast with the number of constants that predicate parameters may bind. Constraint logic programs must then eliminate, to the maximal extension, the number of invalid legs. If we indicate in the snd predicate as parameters both the departure and the arrival rounds, the number of possible combinations is \( nR^2 \). We may reduce the search space, by noting that messages always arrive later than the departure.

We code one leg \((\text{depart}, \text{arrival})\) as \(\text{depart} \ast nR + (\text{arrival} - \text{depart})\).

The number of legs is reduced to

\[
\sum_{n=1}^{nR-1} n = \frac{nR \ast (nR - 1)}{2}.
\]

Example 3 If \( nR = 4 \), the valid legs \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\} are coded as \{5, 6, 7, 9, 10, 13\}.

As an example for a round, party P requests a message to be sent at period 1 to T and arriving at period 3. The party state changes to out. From this state, event snd indicates that the message is sent. Finally, the round concludes with party state changing to idl at period 2.

\[
\text{idl}(P, 1) \xrightarrow{\text{req}(P, P, T, 6, D)} \text{out}(P, P, T, 6, D) \xrightarrow{\text{snd}(P, P, T, 6, D)} \text{idl}(P, 2)
\]
2.1.2 Basic Transition System for a Party

Figure 3 represents a round of the party behavior. Write service is depicted in the upper part of the figure and Read service is depicted in the lower part. Starting at the idle state, \textit{idl}, three alternative events may occur: \textit{req}, which indicates that the party requests the Email server to write a message, \textit{snd}, which indicates a message arrival, and \textit{nil} that indicates nothing is to be done in round.

![Transition system of Basic Services for one party](image)

2.1.3 Encoding Basic Services

The encoding of Email basic services is divided in two parts, domain definitions and service specifications.

Domain definitions define the number of rounds \( nR \), valid legs \( leg \), and the subscribers \( subs \).

The encoding of a transition system in constraint logic program is implemented, for all states, by three sets of rules.

- Rules specifying the transitions from the current state. Each rule identifies the trigger event and the precondition that must be satisfied.
- Rules specifying the following state.
- Constraints that stable models must satisfy.

\textbf{Example 4} In the basic Write service, the \textit{req} parameters of message initiator and originator are always equal to the current party.
%%% Transitions from idl state %%%

\[ \text{req}(P,P,T,L,1) :\text{- idl}(P,1), \]
\[ \text{departs}(L,1), \text{not nil}(P,1). \]
\[ \text{req}(P,P,T,L,1) :\text{- idl}(P,N), N>1, \]
\[ \text{departs}(L,N), \text{arrives}(M,N), \text{not nil}(P,N), \text{not in}(I,O,P,M,E). \]

%%% Next state from idl state %%%

\[ \text{out}(I,P,T,L,D) :\text{- idl}(P,N), \]
\[ \text{departs}(L,N), \text{req}(I,P,T,L,D). \]
\[ \text{in}(J,O,P,M,D) :\text{- idl}(P,N), \]
\[ \text{arrives}(M,N), \text{snd}(J,O,P,M,D). \]
\[ \text{idl}(P,\text{plus}(N,1)) :\text{- idl}(P,N), N<nR, \text{nil}(P,N). \]

In every round the parties are constrained in a way that may only react to one single message, unless a feature (which triggers a reply) is executed.

%%% Local constraints at idl state %%%

\[ \text{false} :\text{- idl}(P,N), \text{req}(I,P,T,L,D), \]
\[ \text{departs}(L,N), \text{snd}(J,O,P,M,E), \text{arrives}(M,N), \]
\[ \text{not equalTime}(J,O,P,M,E). \]

For the transition triggered by \text{snd}(I,P,T,L,D), the constraints are distributed over the messages processed by the originator \( P \) and messages processed by the intended terminator \( T \).

- The originator \( P \) must satisfy two constraints: no other message arrives \( P \) at the depart round (except for features that triggers current message, as explained in section 2.2.1),
\[ \text{false} :\text{- out}(I,P,T,M,E), \text{snd}(I,P,T,M,E), \]
\[ \text{departs}(M,N), \text{snd}(J,O,P,L,D), \text{arrives}(L,N), \]
\[ \text{not equalTime}(J,O,P,L,D). \]

and no other message leaves \( P \) at the depart round.
\[ \text{false} :\text{- out}(I,P,T,M,E), \text{snd}(I,P,T,M,E), \]
\[ \text{departs}(M,N), \text{snd}(J,P,O,L,D), \text{departs}(L,N), \]
\[ I!=J \mid T!=O \mid M!=L \mid D!=E. \]

- The intended terminator \( T \) must satisfy the constraint that no other message arrives at the arrival round.
\[ \text{false} :\text{- out}(I,P,T,M,E), \text{snd}(I,P,T,M,E), \]
\[ \text{arrives}(M,U), \text{snd}(J,O,T,L,D), \text{arrives}(L,U), \]
\[ I!=J \mid P!=O \mid M!=L \mid D!=E. \]

The interface predicate \text{sndDenial}, defined in \text{Features} layer, is part of the rule body that specifies the transition from \text{out} to \text{idl} - see section 1.3. □
Consider the case the system is composed only by basic services. In this case, \textit{sndDenial} predicate is evaluated to \textit{false}. If formulas of the rule body, which specify the transition from \textit{out} to \textit{idl}, are satisfied, the message the rule header is also satisfied and the message is sent.

If we now add \textit{Features} layer to the system, and if this layer evaluates \textit{sndDenial} predicate to \textit{true}, the body evaluation becomes \textit{false} and the message is not sent. In this case, adding a layer results in message removal and, therefore, reveals a non-monotonic characteristic.

2.2 Feature Specification

We specified ten Email features, defined by [4], which are divided into three classes: Simple, which do not manipulate code parameters in the message, Code based, which manipulate code parameter in the message, and Server based, which requires extra parties acting as servers. All simple class features were implemented with support of James, an open-source Email server [24].

The \textit{subscribe/3} predicate indicates \textit{party} subscribes to feature \textit{featureName} and \textit{partyOther} represents an extra parameter (e.g., to whom the incoming messages are sent by the \textit{ForwardMessage} feature).

The specification of a feature is divided in four parts:

- Identification of new states in the automaton representation of the feature behavior, as well the message that triggers the feature, the starting state and the conditions to be satisfied in order the feature to be executed.
- Affected actions.
- Generated messages.
- Constraints to be satisfied.

2.2.1 Simple Features

Based on the automaton representation of the feature behavior, the creation of new states and the generation of messages are implemented in the same way as depicted in section 2.1.3. The affected actions are encoded by \textit{False} rules. In this way, features AddressBook, AutoResponder, FilterMessage, ForwardMessage and MailHost were specified in constraint logic programming.

The basic Write service imposes the constraint that no other message arrives at the depart round, see section 2.1.3. This constraint forces the
message sent forward to depart after the incoming message arrival. As a result of this constraint, the number of rounds to be covered increases. To avoid the expansion of search space, we allow the arrival of the incoming message to share the depart of the message sent forward, by the satisfaction of equalTime/5 predicate.

**Example 5** Figure 4 depicts, in dashed arrows and circles, the automaton that describes the behavior of ForwardMessage feature and is inserted over the basic services automaton (see figure 3).

![Figure 4: Transition system of ForwardMessage](image)

The incoming message snd(J, O, P, M, E) triggers the ForwardMessage feature to change the automaton current state to fwST/4. Feature activation is encoded by the following rule:

\[
\text{fwST}(J, P, T, M) \leftarrow \text{idl}(P, N), \text{snd}(J, O, P, M, E), \\
\text{arrives}(M, N), \text{subscribe(forwardMsg, P, T)}.
\]

For the affected events, the incoming message cannot be read by the party that subscribes to the ForwardMessage feature. In figure 4 message rd is cross out.

\[
\text{rdDenial}(P, N) \leftarrow \neg \text{resolvDenial(forwardMsg, P, T, M)}, \\
\text{fwST}(J, P, T, M), \text{arrives}(M, N).
\]

Concerning the generated messages, ForwardMessage requests the basic Email server to send forward the incoming message. The incoming message departs again at round equal to the arrival.

\[
\text{req}(J, P, T, L, E) \leftarrow \text{fwST}(J, P, T, M), \\
\neg \text{resolvDenial(forwardMsg, P, T, M)}, \\
\text{arrives}(M, N), \text{departs}(L, N).
\]

\[
\text{equalTime}(J, O, P, M, E) \leftarrow \text{fwST}(J, P, T, M), \\
\neg \text{resolvDenial(forwardMsg, P, T, M)}.
\]
2.2.2 Code Based Features

For code based features, we consider that parties hold capabilities of asymmetric cryptography. Each party holds a pair of unique keys, one public and the other private.

When messages are ciphered, the code parameter contains a prime factor denoting the party key used in ciphering. To decrease search space, we make cipher codes available only to Alice and Bob. Carl may only subscribe to the VerifySignature feature and to the EncryptMessage with Alice and Bob public keys.

Example 6 Concerning EncryptMessage feature, two alternative states are triggered from the request event, cipherSTA/5 and cipherSTB/5. The predicate satisfaction depend on the requested message already being signed, or not, with the subscriber secret key.

\[
\text{cipherSTA}(I, P, T, M, \text{times}(D, E)) : - \\
\text{req}(I, P, T, M, D), \text{idl}(P, N), \text{departs}(M, N), \\
pK(O, E), (D \text{ mod } E) \neq 0, sK(O, G), (D \text{ mod } G) \neq 0, \\
\text{not ciphered}(I, P, T, M, \text{b}), \\
\text{subscribe}(\text{encryptMsg}, P, O).
\]

\[
\text{cipherSTB}(I, P, T, M, \text{div}(D, G)) : - \\
\text{req}(I, P, T, M, D), \text{idl}(P, N), \text{departs}(M, N), \\
sK(O, G), (D \text{ mod } G) = 0, \\
\text{not ciphered}(I, P, T, M, \text{a}), \\
\text{subscribe}(\text{encryptMsg}, P, O).
\]

The predicate ciphered/5 indicates the ciphering case and its purpose is to avoid the message result triggers again the EncryptMessage feature.

\[
\text{ciphered}(I, P, T, M, \text{a}) : - \text{cipherSTA}(I, P, T, M, D). \\
\text{ciphered}(I, P, T, M, \text{b}) : - \text{cipherSTB}(I, P, T, M, D).
\]

Sending the requested message is affected by the EncryptMessage feature. For cipherSTA state the rule is (rule for cipherSTB is similar, with times replaced by div): 

\[
\text{sndDenial}(I, P, T, M, D) : - \\
\text{not resolvDenial}(\text{encryptMsg}, P, T, M), \\
\text{cipherSTA}(I, P, T, M, \text{times}(D, E)), \\
\text{subscribe}(\text{encryptMsg}, P, O), pK(O, E).
\]

2.2.3 Server Based Features

Server based features requires the definition of special parties. A server is modeled as a special party, named server.
The RemailMessage feature is implemented by a remailer server that replaces user addresses with pseudonyms. Pseudonym nodes, designated by anon1 and anon2, implement the reverse operation of replacing their name with the real party.

Example 7 Figure 5 depicts the remailing steps of Alice sending a message to Carl, through messages (1) and (2), and reply through messages (3) and (4). Dashed arrows represent messages with pseudonym address and Alice is mapped to Anon1.

![Figure 5: RemailMessage parties and messages](image)

To assure anonymity, the mapping between the two real parties and the two pseudonyms is set in the stable model. The mapping is specified by the following set of rules:

\[
\text{map}(X, \text{anon1}) : \neg \text{map}(X, \text{anon2}).
\]

\[
\text{map}(X, \text{anon2}) : \neg \text{map}(X, \text{anon1}).
\]

\[
\text{false} : X \neq Y, \text{map}(X, \text{anon1}), \text{map}(Y, \text{anon1}).
\]

\[
\text{false} : X \neq Y, \text{map}(X, \text{anon2}), \text{map}(Y, \text{anon2}).
\]

The specification of RemailMessage feature is divided in three parts:

- re-routing to the server of the outgoing messages for the parties that subscribe to RemailMessage,

- server behavior, and

- re-routing to the real address the messages sent to pseudonyms.

The re-routing to the server is specified by two rules, one to forbid the outgoing of the initial message and the other to require a message to the server. Because server is the terminator of the re-routed message, the intended terminator of the initial message, \( T \), is communicated to the server in place of first parameter.
When receiving a message from the party subscribing RemailMessage, the Server requests a new message initiated by himself, originated by the pseudonym and directed to the party indicated at the first parameter of the incoming message.

3 FI Detection

In our approach, the identification of FIs is static, and exercised over basic service and feature layers. If we do not include the Resolver layer, lparse considers not resolvDenial to be satisfied always.

We follow [18] approach, by setting set parties to subscribe the involved features and select one stable model, which includes messages that reveal unacceptable behaviors.

Although we focus only on property analysis, not on implementation, concurrency is an issue when a message (incoming or outgoing) triggers two different features subscribed by same party.

Example 8 It is known that ForwardMessage interacts with many features, such as AutoResponder, MailHost, DecryptMessage, SignMessage and with itself [4]. To detect FI occurrence between DecryptMessage and ForwardMessage, we set party a to subscribe to EncryptMessage with b public key and party b to subscribe to DecryptMessage and ForwardMessage. The facts are subscribe(encryptMsg,a,b).
subscribe(forwardMsg,b,c). subscribe(decryptMsg,b,b).

the computation directive for smodels is
compute 1 { not false,pMSG(a,a,1,b,2,3),pMSG(a,b,2,c,3,1) }.

The number of rounds is 3, code values are \{1,2,3,5,7,6,35,14,15\} and
domain initialization is pK(a,2),pK(b,3),sK(a,5),sK(b,7).

The smodels output confirms a stable model that depicts Alice sending a cipher message to Bob, which sends in clear to Carl.

smodels version 2.33. Reading...done
Answer: 1
Stable Model: pMSG(a,b,2,c,3,1) pMSG(a,a,1,b,2,3)
rMSG(a,b,2,c,3)

True
Duration: 3.525
4 FI Resolution

The elimination of the undesirable behaviors is dealt on a layer above the feature specification. We adopt the [24] approach of feature interdiction, expressed by the resolvDenied/4 predicate: \( \text{resolvDenied}(F, P, T, M) \) stands that party \( P \), when subscribing to feature \( F \) with second parameter \( T \), is not allowed to execute the feature at leg \( M \).

There are two main differences between [24] and this paper approach on FI resolution. Here, resolution requires the knowledge of all system and is implemented off the line.

Experience on FI detection led the identification of two classes of resolution mechanisms, according to the number of parties subscribing to the features: one and more. When all features are subscribed by the same party, we follow a resolution strategy by priority. When features are subscribed by different parties, we follow a resolution by tail elimination.

4.1 Resolution by Priority

In this resolution strategy, users attach a priority value to every feature. When a FI occurs between two features that the same party subscribes to, the feature holding a lower priority is interdicted.

The rule that implements resolution by priority is listed below. \( \text{samePartieFI}/2 \) predicate defines the relative priority of two features that, when subscribed by same party, interact. First parameter indicates the feature holding higher priority.

\[
\text{resolvDenial}(F, P, T, M) :\text{ samePartie}(H, F),
\text{ subscribe}(F, P, T), \text{ subscribe}(H, P, 0).
\]

Example 9 By adding \( \text{samePartieFI}(\text{decryptMsg}, \text{forwardMsg}) \) fact we eliminate DM/FM interaction, depicted in example 8.

smodels version 2.33. Reading...done
False
Duration: 3.689

The priority assignment is a creative process, guided by experience. For example, DecryptMessage feature holds higher priority than ForwardMessage and AutoResponder. Moreover, VerifySignature holds higher priority than RemailMessage.
4.2 Resolution by Tail Elimination

In this resolution strategy, to minimize the number of interdicted features, whenever possible we interdict the last feature in the sequence.

The rule that implements resolution by tail elimination when the second feature is ForwardMessage, is listed below:

\[
\text{resolvDenial(forwardMsg,P,T,M)} :- \\
\text{subscribe(H,O,J), sequenceFILoop(H),} \\
\text{snd(I,O,P,M,D), subscribe(forwardMsg,P,O).}
\]

AutoResponder, ForwardMessage and MailHost are features that interact when ForwardMessage is executed secondly.

Until now, all FIs involve two features. There are, however, some FIs involving three features. Consider that Alice subscribes to EncryptMessage with Bob public key and Bob subscribes to SignMessage and to ForwardMessage. In this case, the message is sent forward in clear. This FIs occur only when EncryptMessage parameter is equal to the party subscribing to SignMessage and to ForwardMessage. The rule that implements resolution by tail elimination is listed below:

\[
\text{resolvDenial(F,P,T,M)} :- \\
\text{sequence3FI(H,R,F), snd(I,O,P,M,D),} \\
\text{subscribe(H,O,P), subscribe(R,P,P), subscribe(F,P,J).}
\]

The features EncryptMessage, SignMessage and AutoResponder constitute an example of a 3-way FI.

5 Performance Analysis

To calculate the number of stable models for \( p \) parties and \( nR \) rounds, we note that the maximal number of messages, \( Max \), is given by equation (3) when all messages hold length equal to 1.

\[
Max = \left\lfloor \frac{p \times nR}{2} \right\rfloor (3)
\]

Moreover, the maximal length of any message, \( L \), is

\[
L = (nR - 1) (4)
\]

First, identify the number of messages in each partial stable model, where a partial stable model contains a number \( k \in 1, \ldots, Max \) of messages holding lengths between 1 and \( L \).
For each $k$, sum the number of models for all combinations with repetition of messages holding lengths between 1 and $L$. The number of combinations with repetition is calculated as

$$\binom{k + L - 1}{k}$$

(5)

For example, for the partial model defined by $L = 2$ and $k = 3$, there are 4 combinations with repetition of messages holding lengths 1 and 2 that are: zero messages of length 1 and three messages of length 2, one message of length 1 and two messages of length 2, two messages of length 1 and one message of length 2, and three messages of length 1 and zero messages of length 2.

The number of stable models is the summation of the number of all partial stable models.

The number of stable models, as function of $nR$ and $p$ is depicted in figure 6. The theoretical values match the tool results.

![Figure 6: Number of stable models as function of $p$ and $nR$](image)

To compare the effect of the number of rounds on the increment of CPU time consumed by smodels, we detected FI between FilterMessage and MailHost [4]. With three rounds and one code, smodels consumes 0.044 seconds.

Increasing the number of codes and the number of parties requires more time to detect FI. For example, DecryptMessage and ForwardMessage interaction requires 9 code values with 3 rounds and 3 parties, takes 1.97 seconds. RemailMessage and VerifySignature interaction requires 9 code values with 3 rounds and 6 parties, consumes 60.72 seconds.
6 Conclusions and Future Work

In this work we have achieved the integration of FI detection and resolution phases within one single framework, the constraint logic programming. Moreover, our proposal is based on a layered architecture, which makes possible the separation of concerns and deduction of properties on each layer.

To make the feature specifications as simple as possible, we restricted FIs to message presence and absence, which does not cover FIs generated from message ciphering errors [4].

Our work suffers the same limitation of model checking tools, the state-explosion problem. The majority of Email features only involve two, or three parties. Server features require a larger number of parties, and this is the only known case that dimension is an issue. Approaches may be explored to alleviate the state-explosion problem, such as composition verification [33].

In section 5 of this paper, we depicted the algorithm for the mathematical calculation of the number of stable models generated by Email Basic Services. To identify the computational cost of FI detection and resolution, this work should be extended to the mathematical calculation of the number of stable models involving features.

Acknowledgments

We gratefully acknowledge the members of the Communications and Services research group, at the University of Stirling/Scotland, for the insights that have made this paper possible. In particular, I want to thank Prof. Kenneth Turner for all support.

This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds, and project PTDC/EEA-ELC/66472/2006.

References


