An Approach to Feature Interaction via Stable Models

Rafael Accorsi[1], Carlos Areces[2], and Maarten de Rijke[2]*

[1] Albert-Lüdwigs-Universität Freiburg
Freiburg, Germany
accorsi@informatik.uni-freiburg.de

[2] ILLC, University of Amsterdam
Amsterdam, The Netherlands
{carlos|maarten}@wins.uva.nl

Abstract. In this paper we report on ongoing work on using a constraint-based approach towards feature interaction. From a logic specification of the Basic Call Service (BCS), we derive a labelled transition system representing this service. This graphical interpretation is implemented in `smodels` and tested for a variety of properties.
We have devised a stepwise methodology for integrating features. According to out method, features act as constraints on models of the original basic system. On the one hand they forbid some of the original behaviour of the system (thus pruning some models), and on the other they give rise to new models, representing new behaviour. Using this methodology, we have implemented a small number of features on top of the basic call service, and we report on some of the tests that we have performed.

1 Introduction

Software engineering is a one of the most important branches of Computer Science. One of its roles is to develop approaches to design and implement software systems adequately. Modern software engineering has focused on separating the systems in logical pieces. More precisely, it has advocated the distinction of the *basic system* from the set of *features*, i.e., extra functionalities which may be added on top of the basic system. The advantages brought by such an approach are felt both by the developer and the user. From a developer point of view, it allows the release features as gradual upgrades to the basic system. It also stimulates the third part development, i.e., the concurrency between different software developers. These two characteristics are felt on the user, who is free to add or remove features whenever it is necessary to do. Finally, it allows the user to chose which developer provides the best product or service.

But this method may also present some problems. More precisely, the superposition of features on top of the same basic system leads to *feature interaction*. Feature interaction is the situation where the behaviour of a feature is affected

* Supported by Spinoza Project

by the behaviour of another. The interaction can be either benign to the system (e.g., a patch fixing a security hole) or malign. If the interaction of features raises unexpected behaviour of the overall system, then the interaction is a problem. The problem of detecting or predicting feature interaction is called the *feature interaction detection problem.*

Although the problem is perceivable in a variety of systems, in this paper we consider the problem of detecting feature interaction in the setting of telecommunication. Specifically, we focus on telephony systems. In such systems, features are pieces of functionality that are usually designed to provide a new facility to the subscriber. However, features may also be engineered to provide facilities to the network administration. The current telecommunication scenario presents an ever growing number of features. For example, features are required to support to technologies such as *video-conferencing*, *web-tv* and *web-phone*.

Some "classical" examples of features are *Call Forward* (with a number of different flavours), *Terminating Call Screening* and *Call Waiting*. These features can interact in several ways. To mention but an example, in *Call Waiting* is a busy subscriber receives a call, the new incoming call is put on hold; yet, in *Call Forward*, every call to a subscriber is forwarded to another phone. Enabling these two features simultaneously leads to feature interaction: if one feature has priority, the other is disabled.

There is a number of aspects that make the detection feature interaction an interesting subject. We mention the following:

1. Everybody uses telephones and anyone can study this domain, even without access to private intellectual property. Moreover, telecommunications is bound to be one of the most important enterprises of the 21st century, both economically, politically, and technically [19].
2. Featured telephone systems present all the problems that any extended, long-lived, distributed, high-performance and concurrent software system presents. In particular, there is the highly combinatorial character of the interaction. The freedom that each subscriber has to add features or not generates a number of alternative scenarios. Thus, the search for interactions has to analyse all these combinations, and the addition of a single subscriber (or feature) creates exponentially many new possibilities to be verified. This combinatorial explosion can quickly lead to intractability.
3. Finally, an important reason that makes the problem an interesting one from a modelling or knowledge representation point of view is the non-monotonic character of the addition of features, see e.g., [17]. By definition, a feature modifies the behaviour of the basic system, altering its properties.

Approaches to the problem are mostly based in model checking [13, 14]; some approaches are based on satisfiability checking [4, 5, 9]. In essence, these methods attack the items 1 and 2 discussed above. However, the non-monotonicity aspect addressed in item 3 has always been relegated.

In this paper we present a new methodology for modelling and exploring the behaviour of a featured telephony systems. In particular, we will use *constraints*

to model non-monotonicity. The system is represented by a set of possible models constructed by means of constraint rules. Features will be then integrated as additional rules. The constraints corresponding to a given feature will let us both prune old models and generate new ones. Technically, we implement these ideas using *stable model semantics* for logic programming: it offers the versatility of the logic programming together with the strength of constraint programming.

Our approach builds a stepwise methodology for feature integration and analisys which relies on the capability of stable model semantics to handle non-monotonicity. We use `smodels` tool box for modelling and exploring the addition of features on top of the telephony system; `smodels` — one of the most efficient implementations of stable model semantics currently available — is particularly interesting because it offers a query-based search facility, which allows its application as verification tool. Furthermore, it is freely available; see [3].

The remainder of the paper is set out as follows. In Section 2 we provide some background on stable models and on the `smodels` tool. Section 3 describes the Basic Call Service (henceforth, BCS) and the modelling phase. Section 4 provides a number of different tests on the implementation and Section 5 presents the method to add features as new rules. In Section 6 we provide a glance on the related work and we conclude in Section 7.

2 Stable Model Semantics

Stable model semantics The stable model semantics is one of the main flavours of declarative semantics for logic programming. This approach radically differs from the standard logic programming used in Prolog: while in the later the aim is to evaluate a single query following a goal directed backward chaining strategy, the stable models semantics considers the rules as constraints that the models should satisfy.

The intuition behind logic programming with stable model semantics is to merge the advantages of logic programming knowledge base representation techniques with constraint programming. These techniques seem to be particularly useful in dynamic domains and for combinatorial problems such as intractable problems in complexity theory.

We briefly introduce syntax and semantics of this approach to logic programming. A solution set is a set of atoms and a logic programming rule of the form

$$A \leftarrow A_1, \ldots, A_n, not(B_1), \ldots, not(B_m)$$

is viewed as a constraint stating that if the atoms $A_1, \ldots, A_n$ are in the solution set and none of $B_1, \ldots, B_m$ is, then A must be included in the set.

For a ground (variable-free) program P, the stable models are defined as follows. The *reduct* of a program P with respect to a set of propositions S is the program obtained by:

1. Deleting each rule in P that has a *not(x)* in its body such that $x \in S$;
2. Deleting all *not* atoms in the remaining clauses.

A set of ground atoms S is a *stable model* of P if and only if S is the unique minimal model of the reduct of P with respect to S.

Example 1. Let P be the program

$$\{p \leftarrow r, not(q) \quad q \leftarrow not(p) \quad r \leftarrow not(s) \quad s \leftarrow not(p)\}.$$

Then $S_1 = \{r, p\}$ is a stable model because the reduct of P with respect to S_1 is $\{p \leftarrow r, \quad r \leftarrow\}$ and S_1 is its unique model. But, $S_2 = \{p, s\}$ is not a stable model of P, because its reduct is $p \leftarrow r$ and its unique minimal model is $\{\}$. However P does have another stable model $\{s, q\}$. Hence, a program may posses multiple stable models, one or none at all.

The problem of deciding whether a ground program has stable models is NP-complete [10]. Indeed, to build a stable model it is enough to guess which atoms will appear non-negated, and then verify uniqueness in polynomial time using the *deductive closure* of the reduct of the program with respect to this set.

Smodels. `smodels` [12] is a C++ implementation of logic programming for stable model semantics. The system includes two modules: (a) `smodels` which implements the stable model semantics for ground programs and (b) `lparse` which computes a grounded version of so-called range-restricted programs.

The implementation is based upon a bottom-up backtrack search where one of the underlying ideas is that stable models are characterised in terms of their *full sets*, i.e., their complements with respect to negative atoms in the program for which the positive atoms are not included in the stable model. The search space is drastically pruned by exploiting an approximation technique for stable models which is very similar to well-founded semantics.

The advantage of this implementation is the linear space requirement. This makes it possible to apply stable model semantics in problem areas where large numbers of stable models are generated. Moreover, `smodels` has proved to be significantly more efficient than other recent implementations of stable model semantics, see [11].

3 Modelling

In this section we provide a translation from the BCS defined in [4] into `smodels` code. The specification of BCS obtained in [4] uses description logics to characterise the set of states and actions that subscribers can take in the BCS model. Basically, the axioms constitute a declarative way of defining a transition system. The *declarative approach* is appealing because the full transition system corresponding to the BCS is enormous, growing exponentially with the number of subscribers considered.

The main idea we will use when encoding this transition system into `smodels` is that actually we don't need to encode piece by piece the complete transition system. Instead, we can consider each subscriber as an independent *dimension* of the n-dimensional transition system where n is the number of subscribers.

But before going into an explanation, we need to define the intended meaning of the different propositional symbols (or labels) we will use.

The following labels express the possible (mutually exclusive) states of a subscriber and the allowed actions.

$idle\_u$	the telephone of u has the receiver on hook and silent.
$ready\_u$	the receiver is off hook and emits a dial tone.
$rejecting\_u$	the telephone emits a busy tone which indicates a failed call attempt or a disconnected line.
$ringing\_u$	the phone is ringing and its receiver is on hook.
$calling\_uv$	the telephone of v is *ringing* and u is waiting for
$path\_uv$	u and v can communicate.

Now we specify the labels representing the possible actions of subscribers.

$offhook\_u$	u lifts the receiver.
$onhook\_u$	u places the receiver back to the phone.
$dial\_uv$	u dials v's number.

The description logic approach to BCS separates statements in four categories:

Interface statements. These axioms are expressions connecting the observable states of a telephone with the ones representing network states.
(e.g., $calling\_uv \sqsubseteq ringing\_v \sqcap ringback\_v$)

Assertional statements. They corresponds to initialisation states. In particular, every user is *idle* at the state s_0.
(e.g., $s_0 : \sqcap_{u \in \mathsf{SUBS}} idle_u$)

Frame statements. These rules specify that certain events do not influence the state of other parts of the system.
(e.g., $path\_uv \doteq \forall offhook\_z.path\_uv$)

Liveness statements. The *liveness statements* consist of a set of declarative transition rules for the BCS. These statements are responsible for describing how to go through the different phases of a call
(e.g., $idle\_u \sqsubseteq \exists offhook\_u.ready\_u$)

We refer to [4] for details. The liveness statements constitute the basic functionality of the BCS scheme. They follow the standard description logic semantics, and we provide below an intuitive reading.

1. $idle\_u$	$\sqsubseteq \exists offhook\_u.ready\_u$	If u is idle, she can go off hook and get ready to dial;
2. $ready\_u \sqcap idle\_v$	$\sqsubseteq \exists dial\_uv.calling\_uv$	If u is ready and v is idle, u can dial v's number and establish a call;
3. $ready\_u$	$\sqsubseteq \exists onhook\_u.idle\_u$	If u is ready, she can go on hook and return to idle;
4. $ready\_u \sqcap \neg idle\_v$	$\sqsubseteq \exists dial\_uv.rejecting\_u$	If u is ready and v is not idle, a dial leads to a busytone;
5. $rejecting\_u$	$\sqsubseteq \exists onhook\_u.idle\_u$	If u is being rejected, she can be idle by going on hook;
6. $calling\_uv$	$\sqsubseteq \exists offhook\_v.path\_uv$	If u is calling v and v goes off hook, u and v can talk;
7. $calling\_uv$	$\sqsubseteq \exists onhook\_u.(idle\_u \sqcap idle\_v)$	If u goes on hook while calling v, both return to idle;
8. $path\_uv$	$\sqsubseteq \exists onhook\_u.(idle\_u \sqcap rejecting\_v)$	If u goes on hook while talking to v, v receives a busytone and u goes to idle.

These rules provide us exactly with a formal definition of a transition system. Furthermore, the rules define the behaviour of the system *locally*, i.e., from the perspective of each subscriber. This will enable us to construct a dimensional view of BCS which will lead us directly to an implementation model in `smodels`.

3.1 The Dimensional View

The specification provided above suggests a graphical interpretation. Having such a graphical view is particularly useful in reasoning about further extensions of the system, such as *features* and updates to the basic service. Furthermore, it provides a concise and clear understanding of the whole system.

The description logic approach provides a clear distinction between states and actions. This separation is the first step towards a graphical interpretation. We define the labelled transition system BCS_{LTS} upon the following sets. Let $\mathbb{L}$ be the union of $\mathbb{A}$ and $\mathbb{S}$ where,

$$\mathbb{S} = \{idle\_u, ready\_u, rejecting\_u, ringing\_uv, calling\_uv, path\_uv\}$$
$$\mathbb{A} = \{offhook\_u, onhook\_u, dial\_uv\}$$

According to the usual interpretation of a transition system, change of states are triggered by actions. We provide the following set $\mathbb{T}$ of transition. These transitions are translated directly from the set of *liveness axioms* provided in the description logic approach.

Let us explain the notation: *states* are represented by italics, e.g., *idle_u*; actions are represented by sans serif font, e.g., offhook_u. The use of a down arrow ($\downarrow$) in front of a state or action means that the next label represents information of a different subscriber. For instance, the first line means "*an 'offhook' action in the state 'idle' moves the subscriber 'u' to the 'ready' state.*"

1. $idle\_u$ + offhook_u	$\rightsquigarrow ready\_u$
2. $ready\_u + \downarrow idle\_v$ + dial_uv	$\rightsquigarrow calling\_uv \quad (if\ u \neq v)$
4. $ready\_u + \downarrow not\ idle\_v$ + dial_uv	$\rightsquigarrow rejecting\_u$
5. $rejecting\_u$ + onhook_u	$\rightsquigarrow idle\_u$
6. $calling_uv + \downarrow$offhook_v	$\rightsquigarrow path\_uv$
7. $calling\_uv$ + onhook_u	$\rightsquigarrow idle\_u + \downarrow idle\_v$
8. $path\_uv$ + onhook_u	$\rightsquigarrow idle\_u + rejecting\_v$

The use of variables for subscribers in the statements suggests the allocation of a transition system to each subscriber. The advantage of such an approach lies in the size: a transition system for the whole set of subscribers would be enormous, growing exponentially as the number of subscribers increase. In our multi-dimensional approach, we view each subscriber as an independent dimension of an n-dimensional transition system, where $n = |\mathsf{SUBS}|$. Figure 1 contains the transition system for a subscriber u.

Each node corresponds to one of the mutually exclusive *states* of $\mathbb{S}$. These nodes are connected by arrows, which correspond to *actions* moving a subscriber to

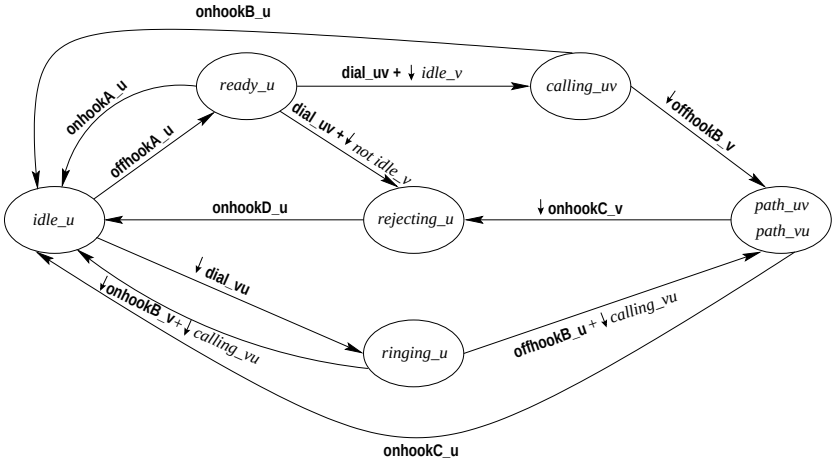

Fig. 1. Transition system for a subscriber

another state. The same action may be used in different states. To differentiate them, we assign different indexes to the actions, e.g., *onhookA*, *onhookB*.

This multi-dimensional view gives meaning to the $\downarrow$ we use in actions and states. The down arrow is supposed to convey the idea that action or states are defined in a different dimension. We can now proceed with the translation into `smodels`.

3.2 The `smodels` Encoding

Before describing implementation issues, we provide an intuitive overview of the computational process. We will implement the transition system in Figure 1 by encoding transitions as `smodels` rules into a program $\mathbb{P}$. Following these rules, the `smodels` toolbox will compute the possible interactions between subscribers and the network.

Each stable model of $\mathbb{P}$, i.e., each set of atoms which is coherent with the transitions of $\mathbb{T}$, will describe a complete set of interactions among subscribers, in one "run" or "possible scenario." Let's make this precise.

Definition 1 (Cycles and Runs). A *cycle* is a sequence of labels from the transition system corresponding to the BCS that represents a move of a subscriber u from the state *idle_u* back to *idle_u*. A *run* is a sequence of cycles.

On the one hand, encoding information about cycles allows us to differentiate among actions executed in different cycles; and on the other it gives us a bound on the number of possible interactions, thus ensuring termination. These concepts are depicted in the following example:

Example 2. The set of labels below represent a run constructed from two cycles. Note that "doubled" idle_u stands for the same state on both cycles.

$$\{\underbrace{idle\_u, offhookA\_u, ready\_u, onhookA\_u, \mathsf{idle\_u}}_{\text{First Cycle}}$$
$$\underbrace{\{\mathsf{idle\_u}, offhookA\_u, ready\_u, dial\_uv, rejecting\_u, onhookD\_u, idle\_u\}}_{\text{Second Cycle}}$$

Summing up, every state that a subscriber visits and every action she engages on is recorded in a stable model together with the actions and states of the other users, i.e., stable models reflect the behaviour of subscribers in the network. Eventually, a valid run for each subscriber in SUBS constitutes a stable model.

Some Changes. The method in which stable models are computed forces a number of changes in Figure 1. Now we aim at describing them and at justifying the changes leading to Figure 2.

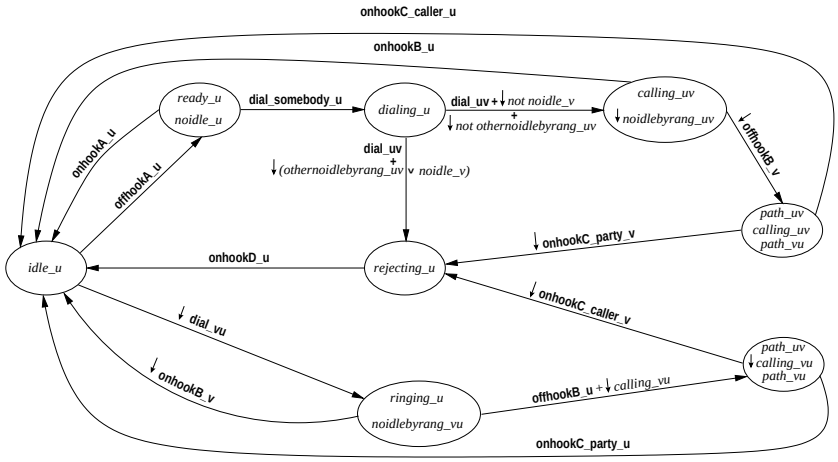

Fig. 2. Transition system for the encoding on BCS

The idle *state.* According to the description logic approach, the network is initialised with every subscriber in the *idle* state. Moreover, the subscribers return to the *idle* state after each run. Therefore, atoms representing these *idle* states are included in the stable model. But the transition from the state *ready* to *rejecting* checks for the presence or absence of the *idle* state of a subscriber v to decide its outcome.

To represent the information that a subscriber u has abandoned her *idle* state we add new labels. Each subscriber has two distinct ways of being active in the network:

– Going off hook and moving to *ready*;
– Having her number dialled by a subscriber and being taken to *ringing*.

To represent the first case we add the *noidle* label, for the second we use *noidlebyrung*. See Figure 2.

The dial *action.* Encoding the behaviour of the *dial* action is complex. For simplicity the action is split into two phases.

The first phase introduces the intermediate state *dialling*, which is reached when the subscriber is in the *ready* state and wants to establish a call with a party. This transition adds two new labels: an action label *dial_somebody* and the state label *dialing*.

The second phase determines the outcome of the *dial* action. Starting from the *dialling*, the *dial* action takes a subscriber to either a *rejecting* state or a *calling*, depending on an external checking of the state of the party. To decide the outcome of the *dial* action, `smodels` uses the information about the *noidle* and *noidlebyrung* states as we described above.

Constraints on the dial *action.* The *dial* action should generates all the possible calls for a given caller. Implementing this "random choice" in `smodels` is tricky, especially because synchronisation with the party is involved. Our approach is the following: on one hand we generate all possible calls, while on the other we impose constraints on the "coherent" calls. Some examples of the constraints are as follows:

– A subscriber cannot establish two calls in the same cycle.
– If a subscriber attempts to dial her number, she should synchronise with herself in the same cycle.
– If two dial actions between users u and v are established, first they should occur in different cycles of u and furthermore cycle counters should be consistent (both increasing).

Split of the path *state.* Figure 1 shows a single *path* state in which there is no distinction between caller and callee. The identification of caller and callee is relevant to determine which subscriber is taken to the *rejecting* state and which is taken to the *idle* state. Thus, the *path* state is split and the calling action is used to derive information about caller and callee. The separation on the *path* state forces the separation of the *onhookC* action. *onhookC_caller* represents an on hook action from the caller and the *onhookC_party* represents the same action when taken by the party.

Encoding into `smodels`. It is now fairly straightforward to encode Figure 2 in `smodels`. The full code is available on-line at [1]. We comment here only on the encoding of the *ready_u* state as an example.

```
1. false :- onhookA(ME,T), dial_somebody(ME,T), ready(ME,T), subs(ME), st(T).
2. onhookA(ME,T) :- not dial_somebody(ME,T), ready(ME,T), subs(ME), st(T).
3. dial_somebody(ME,T) :- not onhookA(ME,T), ready(ME,T), subs(ME), st(T).
4. idle(ME,plus(T,1)) :- onhookA(ME,T), ready(ME,T), subs(ME), st(T).
5. dialing(ME,T) :- dial_somebody(ME,T), ready(ME,T), subs(ME), st(T).
```

The first line encodes the fact that the `onhook` and the `dial_somebody` actions are mutually exclusive. The second and third lines say that if one of the two actions is not taken then the other is. Line 4 reflects the effect of taking the `onhook` action, i.e., it moves the subscriber back to the `idle` state. Action `dial_somebody` moves the user to the `dialing` state, in which the dialled number will be generated.

Given the code as input, `smodels` can compute all possible stable models satisfying the constraints we imposed. The number of models generated will be a function of the number of subscribers and the number of cycles allowed to each of them.

However, an interesting characteristic is that we are able to ask for explicit network properties by using the `compute` statement. The `compute` statement acts as a filter over the stable models that are calculated. This enables us to check for the existence of specific configurations, by indicating which atoms should (or should not) be in the model. We can also define the maximum number of models that are going to be computed. A typical example used in our encoding is `compute 0 {not false}`, which means compute all models that do not include the `false` atom which was used in the encoding to forbid certain configurations.

4 Tests

We are now ready to evaluate the model. The first set of tests is meant to check the parameters (number of subscribers, number of cycles). Tests were performed on a Sun ULTRA II (300MHz) with 1Gb of RAM, under Solaris 5.2.5.

Exhaustive search. The statement `compute 0 {not false}` is used to search for every valid configuration of BCS. The following table shows some of the results obtained.

Subs. \ Cycles	1	2	3
2	0:00.18 / 15	0:00.42 / 375	0:08.08 / 11173
3	0:00.22 / 136	0:45.36 / 82268	14:42:45.23 / 18262292
4	0:02.38 / 1633	4:25:37.11 / 14774656	> 36:00:00.00

For each configuration pair of (# of subscribers, # of cycles), the time and the number of models is shown. The configuration (4,3) was aborted after 36 hrs. of runtime.

Notice that exhaustive generation of models is extremely expensive, as the combinatorial nature of the problem would hint at. But exhaustive search is meaningless from a model checking perspective. What we are really interested in, instead, is the generation of models with certain specific properties.

Checking properties in the model. In Table 1 we provide some examples of specific queries concerning BCS. In each case we first describe the property to check, and then provide the `compute` scheme that will check the property of a specific configuration. The time shown corresponds to the verification of one instance of the scheme. To permit a comparison, all tests have been run on the 4 subscribers, 3 cycles configuration.

Existence of models: The `smodels` implementation of BCS has a model.	
`compute 1 {not false}`	Elapsed time 0:19.60
Self dialling: There is no model where a subscriber dial her own number and avoid the rejecting state.	
`compute 1 {dial(`$s1$`,`t`,`$s1$`,`t`), not rejecting(`$s1$`,`t`)}`	Elapsed time 0:05.00
Parallel calls: Parallel calls among different subscribers are possible.	
`compute 1 {path(`$s1$`,`$t1$`,`$s2$`,`$t1$`), path(`$s3$`,`$t1$`,`$s4$`,`$t1$`)}`	Elapsed time 0:14.9
Multiple callings: A user can establish three different calls in three cycles.	
`compute 1 {path(`$s1$`,1,_,_),path(`$s1$`,2,_,_), path(`$s1$`,3,_,_)}`	Elapsed time 0:11.10

Table 1. Examples

Taking stock, we can take advantage of `smodels`'s `compute` statement to apply constraints over specific atoms and generate only models that satisfy the constraint being issued. This way, checking system's properties turns into a very efficient task as the examples in Table 1 show. At the moment we are generating further test on more subscribers and cycles, and investigating more complex properties of BCS.

5 Features

As we commented in the introduction, we take full advantage of the non-monotonic behaviour of the stable models framework when introducing features as constraints that prune and enlarge the set of models of the BCS. We first list a number of points that guide the design and implementation of features. Then we turn to the concrete specification and implementation of three features (Terminated Call Screening, Originated Call Screening and Call Forwarding Unconditional) on top of the model for BCS described in the previous sections.

Clearly, the first step towards the design and implementation of a feature is to define its behaviour. In other words, we have to understand the expected behaviour of the system after the activation of the feature. Note that this step is independent of any particular implementation.

In the particular case of features, there are at least two tasks that we have to carry out when we try to pin down the properties of a new feature. On the one hand, we should define the enhanced functionality provided by the new feature,

and on the other hand, we should also specify in which ways the new feature *explicitly modifies the previous behaviour of the basic system*. Even though we could say that modifying the previous behaviour of the system is part of the new functionality provided by the feature, it pays off to differentiate between these two. In our approach, we carefully list the parts of the system that are affected by the feature, i.e., the atoms (states and actions) upon which the feature will act. In addition, there might be new states and actions which are characteristic of the new feature. For example, at least one atom has to be added — the *activation predicate* — signalling that the feature is activated for a given subscriber.

As we will see in the examples below, implementing a feature boils down to first eliminating a subset of the previous models of the system by providing new rules leading to the `false` atom. In addition, the newly introduced atoms that are characteristic for the feature create a "new space" in the set of all possible models. From this space the proper models are obtained by providing further rules governing the behaviour of the feature.

Let us turn to the implementations now. We describe implementations of TCS and CFU.

Terminate Call Screening (TCS). Inhibits calls to the subscriber's phone from any number on her screening list. Any dial from a screened subscriber takes the caller to the `rejecting` state. The following implements the TCS feature.

```
% Feature TCS - Terminating Call Screening.
1. rejecting(ME,U) :- dial(SHE,U,ME,T), tcs(ME,SHE), st(U), st(T).
2. false :- calling(SHE,U,ME,T), tcs(ME,SHE), st(U), st(T).
```

In line 1 the behaviour of the dial action is changed: whenever a screened caller dials a subscriber with the TCS feature she is unconditionally taken to the `rejecting` state. Models where the invalid call is established are pruned in line 2. A domain predicate `tcs(x,v)` is used to add v to the screening list of x.

As we did in Section 4, we can query BCS extended by TCS.

Call blocking: If subscribers are pairwise screened there are no callings.	
`tcs(s1,s2)` $\forall s1, s2 \in$ SUBS, $s1 \neq s2$ `compute 1 {not false, calling(`$s1$`, `$t1$`, `$s2$`, `$t2$`)}`	Elapsed time 0:5.1

Call Forward Unconditional (CFU). Diverts calls addressed to a given subscriber's phone to another phone. The CFU feature is implemented as follows.

To encode **CFU** it is necessary to completely re-implement some of the runs in the transition system. A naïve implementation leads to a situation where too many models are pruned (all runs of the party forwarding its phone are eliminated). We need to do some work to solve this problem.

```
% Feature CFU - Call Forward Unconditional.
1. false :- calling(SHE,U,ME,T), cfu(ME,ALTER),
            subs(SHE), st(U), st(T).
2. idle2(ME) :- cfu(ME,ALTER), calling(SHE,U,ME,T),
            subs(ME), subs(SHE), st(U), st(T).
3. offhookA2(ME,T) :- idle2(ME), subs(ME), st(T).
4. ready(ME,T) :- offhookA2(ME,T), subs(ME), st(T).
5. dial_forward(SHE,U,ALTER) :- not notdial(SHE,U,ME,T),
            cfu(ME,ALTER), subs(SHE), st(U), st(T).
6. nodialf(SHE,U,ALTER) :- not dial_forward(SHE,U,ALTER),
            cfu(ME,ALTER), subs(SHE), st(U).
7. false :- dial_forward(SHE,U,ALTER),
            nodialf(SHE,U,1), nodialf(SHE,U,2), nodialf(SHE,U,3),
            cfu(ME,ALTER), subs(SHE), st(U).
8. false :- 2{dial_forward(SHE,1,1), dial_forward(SHE,2,1),
            dial_forward(SHE,3,1), dial_forward(SHE,1,2),
            dial_forward(SHE,2,2), dial_forward(SHE,3,2),
            dial_forward(SHE,1,3), dial_forward(SHE,2,3),
            dial_forward(SHE,3,3)}, subs(SHE).
9. dial(SHE,U,ALTER,T) :- not nodialf(SHE,U,ALTER),
            cfu(ME,ALTER), subs(SHE), st(U), st(T).
```

Line 1 prunes models where the forwarding party is being called. Line 2 is activated whenever the forwarding happens. Given our implementation of the BCS, a call from `SHE` to `ME` will move `ME` to a non idle state. The new `idle2` and `offhookA2` are dummy states, introduced to move `ME` back to the `ready` state. This is done by lines 3 and 4, from `ready` the subscriber will be able to continue her cycle. Thus, the models which were lost are now recovered. The remaining code is mostly a re-encoding of the `dial` action, which is now called `dial_forward`, where a `dial(SHE,U,ME,T)` is interpreted as a dial from `SHE` to `ALTER`.

Of course, feature interaction can occur also when two or more features are switched on at the same time. Below we exemplify one particular interaction between **TCS** and **CFU**.

Interaction between TCS and CFU: If a subscriber `ALTER` screens `SHE` and every call directed to `ME` is forwarded to `ALTER` then `SHE` always obtains a busy tone when calling `ME`.	
`tcs(`$s3$`,`$s2$`)` $\wedge$ `cfu(`$s1$`,`$s3$`)` $\forall s1, s2, s3 \in$ `SUBS`, $s1 \neq s2, s1 \neq s3, s2 \neq s3$ `compute 1 {not false, calling(`$s1$`, `$t2$`, `$s2$`, `$t1$`)}`	Elapsed time 0:3.3

We should expect that the computation produces no model satisfying the condition. Indeed, the computation presents no model.

6 Related Work

Below we comment on some of the most significant works on detection of feature interaction. The reader is referred to [2] for an extensive bibliography.

Tabular approach. In [7] and [15] a graphical and tabular notation for specifying the functional behaviour of telephone services and its features is developed. The features are organised as layers which are assigned to both caller and callee. These layers are dynamically modified to reflect the current set of active features. Feature requests and information about the system is issued by means of tokens and system specifications are composed into reachability graphs. These graphs are analysed in order to reveal if any pathological behaviour of a certain feature or service is found.

The SFI tool. In [13] and [14] a model checking approach is exploited by means of an extension of the **SMV**[1] tool. The **SMV** tool is a **CTL**[2] model checker for unlabelled finite automata. The extension **SFI**[3] automates the integration of features into the system's description. The final code is checked and every interaction is revealed. One of the distinctive characteristics of this method lies in its versatility: it is not tied to any specific system but instead it is a general purpose tool designed for feature integration.

The Description Logic approach. [4] presents a formal model for the specification of telephone features using description logic. This approach fits in the satisfiability method that is mentioned above. It uses a decidable language ($\mathcal{ALC}$ with generic TBoxes) which provides a sound and complete inference mechanism. Hence, properties of the model can be established by automated deduction. However, the method presents a drawback in terms of complexity: $\mathcal{ALC}$ with generic TBoxes has an EXPTIME complete, worst case satisfiability problem. Nonetheless, it provides an exhaustive checking of the properties of the whole model.

Towards Intelligent Networks. Finally, we mention [18] where a new feature detection and resolution approach for intelligent networks is presented. The main idea is to monitor system behaviour and detect interactions as deviations from the *signature* behaviour, which can be seen as a tight feature description. The method also presents *generic resolution techniques* and *generic interaction resolution* to provide recovery from disruptions caused by feature interactions. These techniques were adapted from error recovery techniques of operating systems. Both are eventually integrated under a detection approach.

[1] SMV stands for **S**ymbolic **M**odel **V**erifier.
[2] CTL stands for **C**omputational **T**ree **L**ogic
[3] SFI stands for **S**MV **F**eature **I**ntegrator.

7 Conclusions and Future Work

In this work we have investigated an approach to feature interaction which differs in many aspects from the one studied in [4]. A complete but computationally expensive inference method (based on Description Logics) is traded for a fast, model checking methodology (based on stable models). Still, the two methods are similar enough to actually be able to work in collaboration. Model checking of particular instances on the model produces interesting general properties which can be fully tested by the inferential approach. On the other hand, whenever an inference turns too difficult for the deductive method, it is still possible to attempt to model check all its instances. Even though the number of instances can be exponential, the careful pruning algorithm used in `smodels` can significantly reduce the search space and turn it into a feasible method. The full extent of the interaction between the two approaches remains to be explored.

It is strickling how well the non-monotonic behaviour of `smodels` lends itself to feature integration. By definition, features alter the behaviour of the basic system, by modifying its functionality. In `smodels`, this translates into a set of rules which prunes the models where the old behaviour shows, and replaces them with those modified by the feature.

References

1. The feature interaction project. Url: `http://www.illc.uva.nl/~mdr/Projects/FI/`. Accessed December 04, 1999.
2. The uppsala feature interaction group. Url: `http://www.docs.uu.se/docs/fi/`. Accessed December 04, 1999.
3. Smodels home page. Url: `http://www.tcs.hut.fi/Software/smodels/`. Accessed December 04, 1999.
4. C. Areces, W. Bouma, and M. de Rijke. Description logics and feature interaction. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 28–32, 1999.
5. C. Areces, W. Bouma, and M. de Rijke. Feature interaction as a satisfiability problem. In *Proceedings of* MASCOTS'99, October 1999.
6. Bellcore. LATA switching systems generic requirements (LSSGR). Tech. Reference TR-TSY-000064, Bellcore, Piscataway, N.J., 1992.
7. K.H. Braithwaite and J.M. Atlee. Towards automated detection of feature interactions. In *Proceedings of the 2^{nd} International Workshop on Feature Interactions in Telecommunications Software Systems*, pages 36–59, May 1994.
8. P. Cholewiński, V. Marek, and M. Truszczyński. Default reasoning system DeReS. In *Proceedings of the 5^{th} International Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528, Cambridge, MA, USA, November 1996. Morgan Kaufmann.
9. A. Gammelgaard and J. Kristensen. Interaction detection, a logical approach. In L. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunication Systems*, pages 178–196, Amsterdam, Oxford, Washington DC, Tokyo, 1994. IOS Press.

10. V. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38(3):588–619, 1991.
11. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Workshop on Computational Aspects of Nonmonotonic Reasoning*, Trento, Italy, June 1998.
12. I. Niemelä and P. Simons. Implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, September 1996.
13. M. Plath and M. Ryan. Plug and play featues. In 5^{th} *International Workshop in Feature Interactions in Telecommunications and Software Systems*, 1998.
14. M. Plath and M. Ryan. SFI: a feature integration tool. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 201–216. Springer-Verlag, 1999.
15. K.P. Pomakis and J.M. Atlee. Reachability analysis of feature interactions: A progress report. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 216–223, January 1996.
16. K. Sagonas, T. Swift, and D. Warren. XSB as an efficient deductive database engine. In *Proceedings of SIGMOD 1994 Conference. ACM*, 1994.
17. H. Velthuijsen. Issues of non-monotonicity in feature-interaction detection. In K. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunication Networks, IV*, pages 31–42, Amsterdam Oxford Washington Tokyo, 1994. IOS Press.
18. T. Tsang and E.H. Magill. Behaviour based run-time feature interaction detection and resolution approaches for intelligent networks. In P. Dini, R. Boutaba, and L. Logrippo, editors, 4^{th} *International Workshop on Feature Interactions in Networks and Distributed Systems*, pages 254–270, Montreal, 1997. IOS Press.
19. P. Zave. 'Calls considered harmful' and other observations: a tutorial on telephony. In T. Margaria, editor, *Services and Visualization: Towards user-friendly design*, volume 1385 of *Lecture Notes in Computer Science*, pages 8–27. Springer-Verlag, 1998.