Towards Feature Interaction via Stable Models

Rafael Accorsi Carlos Areces Maarten de Rijke

ILLC, University of Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam. The Netherlands
E-mail: {`accorsi,carlos,mdr`}`@wins.uva.nl`

Abstract. In this paper we propose a new approach to the problem of detecting feature interactions based on the stable models semantics for logical programs. Starting from a formal definition of a model of the Basic Call System, we encode the allowed actions and transitions by means of rules in the style of Reiter's defaults. We can then use tools like `smodels` to perform tasks like verification and testing.

1 Feature Interaction

In the telecommunication domain, a *feature* is characterized as the addition of new functionalities over an already established telephone system. In [2], an extensive list of features is cited, from which *Call Waiting*, *Call Forwarding* and *Automatic Call-Back* are some examples. More generally in a software system, a feature is an optional unit or an increment of functionality over the base system.

Feature interaction arises when the behavior of one feature influences the behavior of another, mostly in an undesirable or unpredicted way. One simple example is the combination of Call Waiting and Call Forwarding on Busy: in giving priority to one, the other is automatically disabled. The first attempts to approach the problem of detecting feature interactions using a software engineering framework started in the early 1980's [3], and since then many other proposals have been made in order to tackle the problem. Due to its general nature — almost anything can be considered a feature and the notion of interaction is vague — there have been several approaches to detect and solve interactions. Usually analytical methods based on formal verification techniques are used. The idea is to build a formal model of both systems and features, and to use these formal descriptions to detect interaction.

In [1] a full formal model of the Basic Call System (BCS) has been provided by means of Description Logics (the language used was $\mathcal{ALC}$ with generic TBoxes [4]). Because $\mathcal{ALC}$ is a decidable language with a sound and complete inference mechanism, properties of the model can be established by automated deduction. The drawback of this method is its complexity: $\mathcal{ALC}$ with generic TBoxes has an EXPTIME complete worst case satisfiability problem. The very generality and expressivity of the $\mathcal{ALC}$ language is its own pitfall.

In this paper we explore an alternative. Once we have obtained the desired model and perhaps checked some global constraints by means of description logics, we can actually *encode this model* and attempt further checking by other means. In particular, we will encode the BCS transition system using logic rules in the style of Reiter's default rules [9].

In recent years, extremely efficient implementations of the stable model semantics for this kind of rules have been devised, like for example the `smodels` compiler [8].

In Section 2 we briefly introduce the stable models semantics of logic programs, together with some notes on `smodels`. In Section 3 we present a *dimensional view* of the BCS system and provide its encoding into `smodels` code. In Section 4 we test the approach by verifying basic properties of BCS. Section 5 introduces features into BCS, in particular we provide the implementation of Terminating Call Screening and Call Forward Unconditional. Finally, in Section 6 we comment on future steps towards detecting feature interactions via stable models.

2 Stable Models Semantics

The stable model semantics is one of the main flavors 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 2.1 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 [7]. 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` [5] 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 characterized 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 [8].

3 Modeling

In this section we provide a translation from the BCS defined in [1] into `smodels` code. The specification of BCS obtained in [1] uses description logics to characterize 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., $\textit{calling\_uv} \sqsubseteq \textit{ringing\_v} \sqcap \textit{ringback\_v}$).

Assertional statements. They corresponds to initialization 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 [1] 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 behavior 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,

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

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.

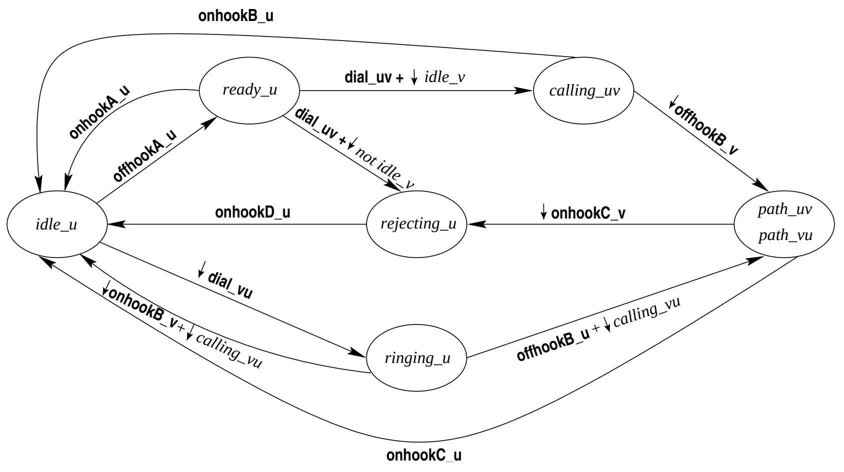

Figure 1: Transition system for a subscriber

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 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 3.1 (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 3.2 The set of labels below represent a run constructed from two cycles

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

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 behavior of subscribers in the network. Eventually, a valid run for each subscriber in SUBS constitutes a stable model.

3.2.1 Some Changes

The method in which stable models are computed forces a number of changes in Figure 1. This subsection aims at describing them and at justifying the changes leading to Figure 2.

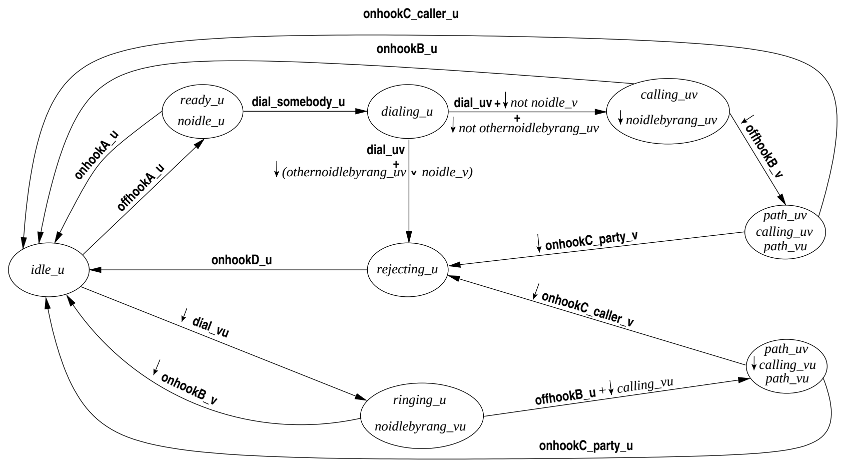

Figure 2: Transition system for the encoding on BCS

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 dialed 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 behavior of the *dial* action is complex. For simplicity the action is split into two phases.

The first phase introduces the intermediate state *dialing*, 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 *dialing*, 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 synchronization 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 attemps to dial her number, she should synchronize 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.

3.2.2 Encoding into `smodels`

It is now fairly straightforward to encode Figure 2 in `smodels`. The full code is available on-line at `http://www.wins.uva.nl/~accorsi/thesis`. We comment here only on the encoding of the *ready_u* state as an example.

```
% State READY.
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 dialed 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 verifiation of one instance of the scheme. To permit a comparison, all tests have been run on the 4 subscribers, 3 cycles configuration.

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

We now describe the process of feature integration, i.e., how to add features on top of the BCS implementation. Here we take full advantage of the non-monotonic behavior of

Existence of models: The `smodels` implementation of BCS has a model.	
`compute 1 {not false}`	Elapsed time 0:19.60
Self dialing: In no model a subscriber can dial her 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

`smodels`. Figure 2 is the starting point to reason about feature integration. To begin with, we should identify which part of the transition system is modified by the feature. The next step is to determine how the behavior of BCS is changed by the feature. Note that this step requires a precise specification of the behavior of the feature. The last step is to activate the feature, i.e., assign features to subscribers.

As an example, 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 behavior 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)` $\forall s1, s2 \in \mathsf{SUBS}, s1 \neq s2$ `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.

```
% Feature CFU - Call Forwarding Unconditional
1. dial_forward(SHE,U,ALTER) :- not notdial(SHE,U,ME,T), cfu(ME,ALTER), subs(SHE), st(U),
st(T).
2. dial(SHE,U,ALTER,T) :- not notdialf(SHE,U,ALTER,T), dialing(SHE,U), cfu(ME,ALTER),
subs(SHE), st(U), st(T).
3. nodialf(SHE,U,ALTER,T) :- not dial(SHE,U,ALTER,T), cfu(ME,ALTER), subs(SHE), st(U), st(T).
4. false :- dial_forward(SHE,U,ALTER), notdialf(SHE,U,ALTER,1), notdialf(SHE,U,ALTER,2),
notdialf(SHE,U,ALTER,3), cfu(ME,ALTER), subs(SHE), st(U).
5. false :- dial(SHE,U,ME,T), cfu(ME,ALTER), subs(SHE), st(U), st(T).
```

Line 1 exchanges the `dial` action by a request of forwarding. Lines 2, 3 and 4 implement the non-deterministic synchronization with the cycle of the forwarding subscriber. Finally, line 5 prunes the models where dials to the original phone exists.

Call looping: If two subscribers forward their phones to each other, no dial to either of them can be performed.	
`tcs(`$s1$`,`$s2$`), tcs(`$s2$`,`$s1$`)` $s1, s2 \in \mathsf{SUBS}, s1 \neq s2$ `compute 1 {not false, dial(`$s3$`, `$t1$`, `$s1$`, `$t2$`)}`	Elapsed time 0:5.1

6 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 [1]. 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 stricking how well the non-monotonic behavior of `smodels` lends itself to feature integration. By definition, features alter the behavior of the basic system, by modifying its functionality. In `smodels`, this translates into a set of rules which prunes the models where the old behavior shows, and replaces them with those modified by the feature.

References

[1] C. Areces, W. Bouma, and M. de Rijke. Description logics and feature interaction. In *Proceedings of DL'99. Linköping, Sweden*, 1999.

[2] Bellcore. LATA switching systems generic requirements (lssgr). Technical Report Tech. Reference TR-TSY-000064, Bellcore, Piscataway, N.J., 1992.

[3] L. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunication Systems*, Amsterdam, Oxford, Washington DC, Tokyo, 1994. IOS Press.

[4] Martin Buchheit, Francesco M. Donini, and Andrea Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. Artificial Intelligence Res.*, 1:109–138, 1993/94.

[5] Niemelä I. and P. Simons. Implementation of the well-founded and stable model semantics. In *Proc. Joint Int. Conf. and Symp. on Logic Programming*, Bonn, Germany, September 1996.

[6] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Found. Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann Publishers, Los Altos, 1988.

[7] W. Marek and M. Truszczyński. Autoepistemic logic. *J. Assoc. Comput. Mach.*, 38(3):588–619, 1991.

[8] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Workshop on Computational Aspects of Nonmonotonic Reasoning*, Trento, Italy, May , June 1998.

[9] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.