Features as Constraints

Rafael Accorsi[1] Carlos Areces[2] Wiet Bouma[3] Maarten de Rijke[2]

[1] Albert-Lüdwigs-Universität Freiburg, Freiburg, Germany
[2] ILLC, University of Amsterdam, Amsterdam, The Netherlands
[3] KPN Research, Leidschendam, The Netherlands

Abstract. We report on ongoing work on using a constraint-based approach towards feature interaction detection. Constraint programming is introduced as a natural way of handling the inherent non-monotonic aspects of feature integration. Starting from a logic specification of the Basic Call Service (**BCS**), we obtain a labeled 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 our method, features act as constraints on models of the original basic system. On the one hand they forbid some of the original behavior of the system (thus pruning some models), and on the other they give rise to new models, representing new behavior. Using this methodology, we have so far 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

An important approach towards software design that is advocated by modern software engineering is to consider a complex system as a combination of a basic system, which provides functions for stand-alone operation, and a set of modules of functionality, called features, which are added on top of the basic system. Viewing a complex system as a combination of a basic system and features is particularly useful for both system developer and user. From a software engineering point of view, it allows the system developer to release features as gradual upgrades to the basic system. These upgrades can be evaluated and integrated as part of the basic system. Additionally, it allows third parties to design and develop features. These two characteristics also affect the user, who is free to add or remove features whenever it is necessary or desirable to do so. Furthermore, it allows the consumer to decide which developer of features offers the best services.

However, the feature-oriented approach towards software design also presents problems. In particular, the presence of a number of features on top of the same basic system leads to *feature interaction*, i.e., to the situation where the behavior of one feature affects the behavior of another. If the interaction of features generates unexpected behavior of the overall system, then feature interaction becomes a problem.

In this paper we consider the problem of detecting feature interaction in the setting of telecommunication. In telephony systems, features are pieces of functionality that are usually designed to provide a new facility to a subscriber; however, they can also be developed to make the administration of the network easier. There are many features today, and their number continues to grow; [6] contains descriptions of hundreds of features.

There are various aspects that make the detection of feature interaction in telecommunications an attractive and challenging problem:

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 early 21st century, both economically, politically, and technically [16].

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 large number of alternative scenarios. The search for interactions has to analyze all these combinations, and the addition of a single 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 modeling or knowledge representation point of view is the non-monotonic character of the addition of features, see e.g., [15]. By definition, a feature modifies the behavior of the basic system, altering its properties.

There have been several formal approaches to the feature interaction problem. Most standard methodologies are based on model checking [12, 13], and some are based on satisfiability checking [8, 5, 4]. Both methods address items 1 and 2 discussed above: there exists freely available verification systems which can deal with reasonably complex instances of the feature interaction problem. But the issue of non-monotonicity raised in item 3 has always been neglected. We believe that the non-monotonic nature of feature addition is one of the central themes in feature interaction.

The field of non-monotonic logics, which includes branches like default logic, autoepistemic logic, circumscription, etc., flourished during the late 1970s and 1980s, and constitutes an example of a serious attempt to get to grips with non-monotonicity and ways of building expressive knowledge representation formalisms. But one of the clear outcomes of the work in this field, was the realization that accounting for non-monotonicity was computationally expensive.

The important questions, then, is whether we can reconcile the fact that non-monotonicity issues appears to be central to feature integration and feature interaction detection, with the staggering computational costs that logical approaches to non-monotonicity seem to imply.

In this paper, we use *constraints* as a natural way to model non-monotonic phenomena: constraint programming is an effective algorithmic approach to many combinatorially complex problems [3]. Our methodology is as follows. We start by representing a system by the set of its possible models defined by means of constraint rules. Feature integration will constitute the addition of new constraints, perhaps over an extended vocabulary. The new constraints corresponding to a given feature will let us both prune old models and generate new ones, altering thereby the behavior of the system. As the entire modeling effort takes place in a constraint programming framework, we can take advantage of efficient heuristics for two tasks that are of crucial importance in finding interactions: for model construction and querying.

The main methodological point of this paper, then, is to our attempt to verify intuitions that a constraint oriented approach to non-monotonicity in the feature interaction problem is feasible. To test these intuitions we set up an experimental environment based on the *stable model semantics* for logic programming. This approach offers the expressivity of a non-monotonic environment, the flexibility of the logic programming paradigm and the strength of constraint programming. Even thought at this early stage we will only model very simple cases of interaction, we will attempt a realistic verification of these by exhaustive testing.

We propose a stepwise methodology for feature integration and analysis which relies naturally on the capability of stable model semantics to handle non-monotonicity. We use the `smodels` tool box — one of the most efficient implementations of stable models semantics currently available — for modeling and exploring the addition of features in the setting of telecommunications; `smodels` is particularly interesting because it offers a query-based search facility, which allows us to use it as a verification tool. Furthermore, it is freely available; see [2].

The remainder of the paper is organized as follows. In Section 2 we provide some background on stable models semantics and on feature interaction in telephony systems. In Section 3 we model the Basic Call Service; we report on tests carried out with the **BCS** in Section 4. In Section 5 we show how to model feature interaction. Finally, in Section 6 we evaluate the results obtained so far, and we conclude in Section 7.

2 Modelling the Basic Call Service

Stable Model Semantics. Stable models constitute a declarative semantics for logic programming. This approach is radically different from the standard semantics used in Prolog: while in the later the aim is to evaluate a single query following a goal directed backward chaining strategy, stable models semantics considers program 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 the one we aim to model in this paper.

Let us briefly recall the syntax and semantics of stable model semantics. The vocabulary is required to be purely relational, i.e., there are no function symbols; moreover, we do have the negation symbol *not* in the language. A *solution set* is a set of atoms. A *program* is a set of rules of the form

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

Here, A is called the *head* of the rule, and the part to the right-hand side of the arrow the *body*. Such rules are viewed as constraints stating that if the atoms $A_1, \ldots, A_n$ are in a solution set and none of $B_1, \ldots, B_m$ is, then A must be included in the set. The stable models of a ground (variable-free) program P, are defined as follows. The *reduct* of a program P *with respect to* a set of atoms 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 negative literals $not(B)$ in the remaining clauses.

A set of ground atoms S is a *stable model* of P 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, namely $\{s, q\}$. Hence, a program may possess multiple stable models, one, or none at all.

The problem of deciding whether a ground program has stable models is NP-complete [9]. 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` is a C++ implementation of logic programming with stable model semantics [11]. 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 stable models are generated. Moreover, `smodels` has proved to be significantly more efficient than other recent implementations of stable model semantics, see [10].

Basic Call Service. The specification of **BCS** obtained in [4] uses a description logic to characterize the sets of states and actions available to subscribers 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 we don't actually need to encode the complete transition system piece by piece. Instead, we can consider each subscriber as an independent *dimension* of a many-dimensional transition system. But before going into an explanation, we need to define the intended meaning of the different atoms we will use.

Table 1 lists the atoms that express the possible (mutually exclusive) states of a subscriber and the allowed actions.
In [4], a set of rules providing a formal definition of a transition system for **BCS** is specified. Furthermore, the rules define the behavior of the system *locally*, i.e., from the perspective of each subscriber. This enables us to construct the dimensional view of **BCS** shown in Figure 1. Each node corresponds to one of the mutually exclusive *states* of the system. These nodes are connected by arrows, which correspond to *actions* moving a subscriber from state to state. Because the same action may be taken in different states we use subindexes to differentiate them, e.g., *onhookA*, *onhookB*. This is done to avoid the need to explicitly encode source and target states in the implementation.

A quick word about our notation: *states* are represented by italics (e.g., *idle_u*) and actions are represented by sans serif font (e.g., offhook_u). The use of a down arrow

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.
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.
offhook_u	u lifts the receiver.
onhook_u	u places the receiver back to the phone.
dial_uv	u dials v's number.

Table 1: Atoms used in the modeling

($\downarrow$) in front of a state or action means that the next label represents information of a different subscriber. For instance, the annotated arrow from *ready_u* to *calling_uv* represents a transition between these two states, triggered by the dial_uv action under the condition *idle_v* which should the checked for the user v.

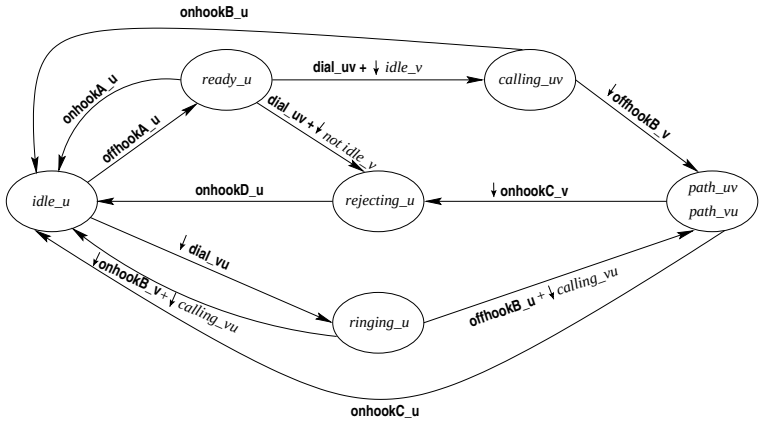

Figure 1: Transition system for a subscriber

The full transition system is obtained by the composition of the transition systems for each single subscriber. Clearly this notation helps us to succinctly represent a transition system which would otherwise be enormous.

3 Encoding the Basic Call System

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 a set $\mathbb{P}$ of `smodels` constraint rules. Using these rules, the `smodels` toolbox will compute the possible interactions between subscribers and the network. Each stable model of $\mathbb{P}$, will describe a complete set of interactions among subscribers, in one "run" or "possible scenario." Let us make this precise.

First, a *cycle* in the transition system in Figure 1 is a sequence of states and actions representing a move of a subscriber u from the state *idle_u* back to *idle_u*. A *run* is a sequence of cycles. As an example, the set of atoms below represents a run constructed from two cycles:

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

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

It is now fairly straightforward to encode Figure 1 in `smodels`. Table 2 illustrates the main intuition; it gives the encoding for a situation where we are in state S, with actions A_1 and A_2 available, leading to states S_1 and S_2, respectively.

`false :-` A_1`,` A_2`,` S	avoids both actions being taken simultaneously
A_1 `:- not` A_2`,` S	forces the A_1 action whenever A_2 is not taken
A_2 `:- not` A_1`,` S	idem for A_2
S_1 `:-` A_1`,` S	codes the transition to S_1 after action A_1 is taken
S_2 `:-` A_2`,` S	idem for A_2.

Table 2: A simple encoding

The full code is available on-line at our web site [1]. By way of example, we spell out the encoding of the *calling* state; see Figure 2. The atoms `subs` and `cycle` are used to parametrize the rules for each subscriber and for each cycle. Line 1 makes the actions `onhookB(ME,T)` and `offhookB(SHE,U)` mutually exclusive, notice that the second action corresponds to a different subscriber. Lines 2 and 3 encode the effects of taking the `onhookB(ME,T)` action. Finally, line 4 encodes the effect of the party taking the action `offhookB(SHE,U)`; notice that we don't need to encode the effect that this action causes in the party.

```
% The CALLING State.
1.  false                 :-   onhookB(ME,T), offhookB(SHE,U), calling(ME,T,SHE,U),
                               subs(ME), subs(SHE), cycle(T), cycle(U).
2.  onhookB(ME,T)         :-   not offhookB(SHE,U), calling(ME,T,SHE,U),
                               subs(ME), subs(SHE), cycle(T), cycle(U).
3.  idle(ME,plus(T,1))    :-   onhookB(ME,T), calling(ME,T,SHE,U),
                               subs(ME), subs(SHE), cycle(T), cycle(U).
4.  path(ME,T,SHE,U)      :-   offhookB(SHE,U), calling(ME,T,SHE,U),
                               subs(ME), subs(SHE), cycle(T), cycle(U).
```

Figure 2: Encoding the *calling* state

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 to the extent that this is necessary, i.e., stable models reflect the behavior of subscribers in the network. Eventually, a valid run for each subscriber constitutes a stable model.

3.1 Some Changes

The method used to compute stable models forces a number of changes in Figure 1. This subsection aims at describing them and at justifying the changes leading to Figure 3.

The idle *State.* The network is initialized 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

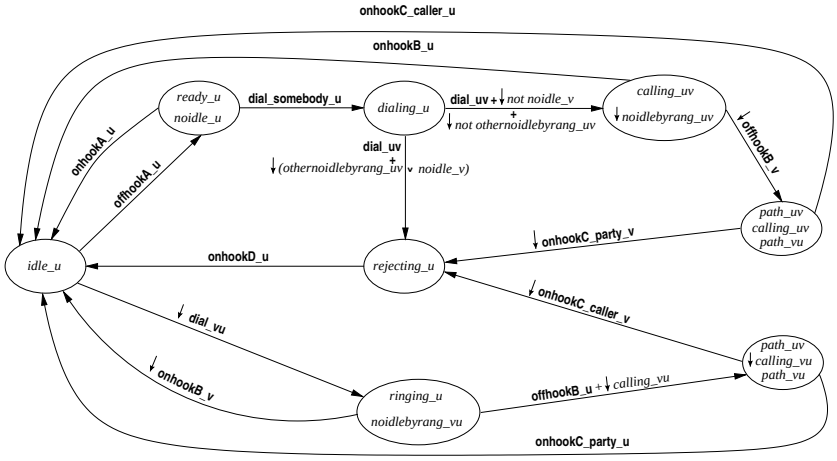

Figure 3: Transition system for the encoding of **BCS**

subscriber v to decide its outcome. To represent the information that a subscriber u has abandoned her *idle* state we add new atoms. 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 3.

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 atoms: 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 generate all the possible calls for a given caller. Implementing this "random choice" in `smodels` is tricky, especially because synchronization is involved. Our approach is the following: we first generate all possible calls, and we then impose constraints to eliminate those which cannot occur. 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 own number, she should synchronize with herself in the same cycle.
- If two dial actions between users u and v are established, they should occur in different cycles of u and the cycle counters should be consistent (both increasing).

Splitting of the path *State.* Figure 1 shows a single *path* state in which there is no distinction between caller and callee. But the identification of caller and callee is needed 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 of 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 other party.

Starting from Figure 3, and following the guidelines we have explained above we obtain the full encoding of the **BCS** as `smodels` constraint rules. Given this code as input, the `smodels` interpreter can compute all possible stable models satisfying the constraints we imposed, i.e. the valid runs. The number of models generated will be a function of the number of subscribers and the number of cycles allowed to each of them.

Interestingly, we can take advantage of `smodels`' `compute` statement to query the properties of our model. The `compute` statement acts as a filter over the stable models that are calculated, by indicating which atoms should (or should not) be in the model and the maximum number of models that are going to be computed. For example, running `smodels` with the `compute 0 {not false}` statement will search for all models of the program not containing the `false` atom, which we used in the encoding to forbid certain configurations. We investigate this issue further in the next section.

4 Testing the BCS Implementation

We are now ready to evaluate the model, and we do so by performing a number of tests. 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, with `smodels` version 2.9.

Exhaustive search. We aim at computing every valid network configuration that a finite set of subscribers can generate in a finite number of cycles. This kind of test is particularly useful for determining the behavior of the problem. By varying the number of subscribers and cycles we obtain different results. Thus we see how the number of models (and the time for computing them) is affected by changes on the network configuration. Additionally, we are able to test characteristics of the tool that we use for computing the models. The statement `compute 0 {not false}` is used to search for every valid configuration of our model of the **BCS**. Table 3 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	> 100:00:00.00

Table 3: Computing all models

For each configuration pair of (# of subscribers, # of cycles), the time and the number of generated models is shown. Note how hard it is to explore all the network configurations. For instance, the test with the 4 subscribers/3 cycles setting was aborted after more than 100 hours of computation. Furthermore, the number of models was enormous (over

60 million). However the linear space complexity of the `smodels` algorithms ensures that all models will be computed given enough time.

In conclusion, exhaustive generation of models is extremely expensive — a fact that is to be expected given the combinatorial nature of the problem. But the characteristics of our modeling and the tools we have chosen offer a much more interesting possibility: the generation of models with certain specific properties.

Checking properties in the model. We want to explore certain branches of the search tree, i.e., given a property, we are looking for specific models where this property is valid. In Table 4 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 over 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.

Q1. Existence of a model: The `smodels` implementation of **BCS** has a model.	
`compute 1 {not false}`	Elapsed time 0:19.60
Q2. Self dialling: 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
Q3. Parallel calls: Parallel calls among different subscribers are possible.	
`compute 1 {path(`$s1$`,`$t1$`,`$s2$`,`$t1$`), path(`$s3$`,`$t1$`,`$s4$`,`$t1$`) }`	Elapsed time 0:14.90
Q4. 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
Q5. Call chain: "cyclic" pairing of calls cannot occur.	
`compute 1 {not false, calling(`$s1$`, `$t1$`, `$s2$`, `$t1$`), calling(`$s2$`, `$t1$`, `$s3$`, `$t1$`), calling(`$s3$`, `$t1$`, `$s4$`, `$t1$`), calling(`$s4$`, `$t1$`, `$s1$`, `$t1$`),}`	Elapsed time 0:9.36
Q6. Dial Target: Checking a specific configuration.	
`compute 1 {not false, not noidle(`$s1$`, `$t1$`), dial(`$s4$`, `$t1$`, `$s1$`, `$t1$`), dial(`$s2$`, `$t1$`, `$s1$`, `$t1$`), dial(`$s3$`,`$t1$`,`$s1$`,`$t1$`), rejecting(`$s2$`, `$t1$`), rejecting(`$s3$`, `$t1$`)}`	Elapsed time 0:13.52

Table 4: Checking specific properties

We use the `compute` statement to query the implementation for specific properties. However, because of `smodels`' syntactic limitations, we are not allowed to query the system using variables, i.e., the `compute` statement must be grounded in a specific scenario. From this scenario we check whether there are stable models where this property holds.

In *Q1. Existence of a model* we check that our implementation of **BCS** does have a model. The expected outcome is that there is indeed a valid model. By using `compute 1 {not false}` as our query, we obtain the expected answer in 19.60 seconds. Table 5 represents the times needed to find a stable model in each setting of N subscribers, 3 cycles. Note the increase in time.

In *Q2. Self dialing* we check that there is no model in which a subscriber can dial her own number and avoid the rejecting state. We obtain the expected answer (no such model exists) in 5.00 seconds. In *Q3. Parallel calls* we verify that parallel calls among different subscribers are possible. The expected (positive) outcome is computed in 14.90 seconds. *Q4. Multiple callings* concerns a technical issue, namely that every user should be able to establish three different calls in three cycles. Feeding `smodels` with a query of the form `compute 1 {not false, path(`$s1$`, 1, _, _), path(`$s1$`, 2,`

Subscribers	1	2	3	4	5
Time	–	0:00.41	0:04.66	0:19.60	00:54.76
Subscribers	6	7	8	9	10
Time	2:14.68	19:21.97	3:25:44.75	29:17:34.88	> 72 hours

Table 5: Computing `compute 1 {not false}`

`_, _), path(`$s1$`,3,_,_)}` shows that this is indeed the case by producing a model in 11.10 seconds. Next, we want to confirm that it cannot be the case that there is a chain of subscribers, one calling the other, producing a cycle. Query *Q5. Call chain* obtains the expected answer in 9.36 seconds. Finally, in *Q6. Dial target*, we check for the possibility of a particular configuration in the network. If all the subscribers dial the same party and the party is idle, two callers must be rejected (recall that we're assuming the presence of 4 subscribers). The expected outcome is a model where two callers are rejected. The code that we can use to formulate this query is `compute 1 {not false, not noidle(`$s1$`, `$t1$`), dial(`$s4$`, `$t1$`, `$s1$`, `$t1$`), dial(`$s2$`, `$t1$`, `$s1$`, `$t1$`), dial(`$s3$`,`$t1$`,`$s1$`,`$t1$`), rejecting(`$s2$`, `$t1$`), rejecting(`$s3$`, `$t1$`)}`. We get our answer in 13.52 seconds.

Taking stock, we can take advantage of `smodels`' `compute` statement to apply constraints over specific atoms and generate only models that satisfy the constraint being issued. This way, checking a system's properties turns into a very efficient task as the examples in Table 4 show. We are currently generating further tests on more subscribers and cycles, and investigating more complex properties of **BCS**.

Our next step is to tackle the integration of features on top of the **BCS** implementation.

5 Integrating Features

As we commented in the introduction, we take full advantage of the non-monotonic behavior 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, Originating 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 behavior. In other words, we have to understand the expected behavior 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 behavior of the basic system.* Even though we could say that modifying the previous behavior 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* — signaling 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 behavior of the feature.

Let us turn to the implementations now. The full description of each feature is provided together with an interpretation of the implementation.

Terminating call screening. The **TCS** feature 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 atoms affected by the feature are the `dial` action and the states `rejecting` and `calling`. The activation predicate is `tcs(ME,SHE)` whose intended meaning is that `ME` is screening the subscriber `SHE`. Finally, the affected behavior in the original system is that given `tcs(ME,SHE)`, the atom `calling(SHE,U,ME,T)` cannot appear in a model. The **TCS** is implemented as follows:

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

Line 1 prunes the models which are invalidated by the activation of the feature, i.e., models where the calling action from `SHE` to `ME` is considered. In line 2 we represent the change of behavior. The rule stipulates that whenever `SHE` is dialing a party `ME` and `ME` has `SHE` on her screening list, `SHE` is taken to the `rejecting` state. Note that both rules 1 and 2 are only activated when the activation predicate is included in the model. In other words, the featured system allows for subscribers with and without the feature activated. We do not need to modify the code of the basic system.

Originating call screening. **OCS** is dual to **TCS**: it forbids calls from `ME`'s phone to any number `SHE` from a given list. Any attempt of `ME` to ring such a number takes her to the `rejecting` state. The code for **TCS** is similar to the one for **TCS**.

```
% Feature OCS - Originating Call Screening.
1. false :- calling(ME,T,SHE,U), ocs(ME,SHE), cycle(U), cycle(T).
2. rejecting(ME,U) :- dial(ME,T,SHE,U), ocs(ME,SHE), cycle(U), cycle(T).
```

Call forwarding unconditional. In the **CFU** feature every call addressed to a subscriber `ME` is unconditionally forwarded to the subscriber `ALTER`. The affected atom is the `dial` action and the activation predicate is the atom `cfu(ME,ALTER)`, meaning that `ME` is forwarding her calls to `ALTER`. The invalidated action in presence of `cfu(ME,ALTER)`, is that `ME` can never be called. The implementation of **CFU** is more involved than the previous two, and we will address the reasons for this below.

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

```
8. false :- 2{dial_forward(SHE,1,1), dial_forward(SHE,2,2), 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), cycle(U), cycle(T).
```

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 subscriber forwarding its phone are eliminated). We need to do some work to solve this problem.

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`

Now that we have provided implementations of the three features (**TCS**, **OCS**, and **CFU**) we are able to ask for specific properties of the model.

*5.1 Checking Properties on the Featured **BCS**.*

Implementing features on top of **BCS** changes the original characteristics of the basic system. Thus, one may want to re-run the tests that we performed in Section 3 to make sure that the featured system still satisfies desirable basic properties. We are not including the results of those tests here. Instead, we will use the `compute` statement to ask for specific properties of the featured **BCS**. Below we provide a number of tests involving the features implemented above.

We first ask for properties of the **BCS** together with a single feature.

***TCS** and Call Blocking.* If the subscribers are pairwise screened, there can be no callings. The following table provides the query scheme together with the time required for checking one instance of this property.

`tcs(`$s1$`,`$s2$`)` $\forall s1, s2 \in \mathsf{SUBS}, s1 \neq s2$ `compute 1 {not false, calling(`$s2$`, `$t2$`, `$s1$`, `$t1$`)}`	Elapsed time 0:2.80

As we expect, the implementation returns no model satisfying this property.

***OCS** and Call Blocking.* We run a similar test for the OCS feature: if the OCS feature is pairwise activated for every subscriber, there can be no calls. The query below returns no model, as we expected.

`ocs(`$s1$`,`$s2$`)` $\forall s1, s2 \in \mathsf{SUBS}, s1 \neq s2$ `compute 1 {not false, calling(`$s1$`, `$t1$`, `$s2$`, `$t2$`)}`	Elapsed time 0:2.80

***CFU** and Call Looping.* If two subscribers forward their phones to each other, no dial to either of them can be performed. The query below produces no model, as we expected.

`cfu(`$s1$`,`$s2$`)` $\wedge$ `cfu(`$s2$`,`$s1$`)` $s1, s2, s3 \in \mathsf{SUBS}, s1 \neq s2$ `compute 1 {not false, calling(`$s3$`, `$t1$`, `$s1$`, `$t2$`)}`	Elapsed time 0:4.50

More interestingly, though, the possibility of dial has also been eliminated: the query

`cfu(`$s1$`,`$s2$`)` $\wedge$ `cfu(`$s2$`,`$s1$`)` $s1, s2, s3 \in \mathsf{SUBS}, s1 \neq s2$ `compute 1 {not false, dial(`$s3$`, `$t1$`, `$s1$`, `$t2$`)}`	Elapsed time 0:4.50

returns no models. This shows a real interaction of **CFU** with itself.

Of course, feature interaction can occur also when two or more features are switched on at the same time. Below we exemplify some of these situations.

*Interaction between **TCS** and **CFU**.* If a subscriber `OTHER` screens `SHE` and every call directed to `ME` is forwarded to `OTHER` then `SHE` always obtains a busy tone when calling `ME`.

`tcs(`$s3$`,`$s2$`)` $\wedge$ `cfu(`$s1$`,`$s3$`)` $\forall s1, s2, s3 \in \mathsf{SUBS}, s1 \neq s2, s1 \neq s3, s2 \neq s3$ `compute 1 {not false, dial(`$s2$`, `$t2$`, `$s1$`, `$t1$`), not rejecting(`$s2$`,`$t2$`)}`	Elapsed time 0:3.30

As we expect, no model satisfying this property can be found.

*Interaction between **TCS** and **OCS**.* If a subscriber `ME` activates both `TCS` and `OCS` for the same subscriber `SHE`, then there is no model with a call between `ME` and `SHE`.

`tcs(`$s1$`,`$s2$`)` $\wedge$ `ocs(`$s1$`,`$s2$`)` $\forall s1, s2 \in \mathsf{SUBS}, s1 \neq s2$ `compute 1 {not false, calling(`$s1$`, `$t2$`, `$s2$`, `$t1$`)}`	Elapsed time 0:2.40

There are no models satisfying this configuration.

Although we have only reported on a small number of queries involving features on top of the basic call service here, we hope that it has provided ample illustration of our methodology.

6 Evaluation

As we have seen in our testing examples, in `smodels` it is necessary to specify a query and then check whether it holds. In terms of detection of feature interaction, this means that we have to detect a "harmful" [16] combination of features and query for its existence. This contrasts with the model checking approach where one provides the formal specification to a model checker, which then verifies the whole input for consistency. Thus, on the one hand we provide precision and efficiency of the query-based search to check for one particular kind of interaction. On the other hand, the model checking tool provides a robust, but costly, method to search for all possible interactions.

Exploiting the non-monotonic behavior of a feature is a natural way of viewing the process of feature integration. The methodology that we developed in this paper relies on this idea. The advantages of the approach are clear. In the design phase we are able to understand the feature without taking into account the basic system. This complies with the generality of feature descriptions, advocated by [12].

The tool that we use, the `smodels` tool box, has both good points and some drawbacks. The linear space algorithm implemented by `smodels` permits the computation of really big problems, but it offers nearly no support for developers. For instance, it is not possible to declare the atoms in the logic program. Thus, any typo is interpreted as a new atom. Also, some syntactic constructions do not work properly at present. This has forced us to hard-code some parts of the **BCS**.

7 Conclusion

In this paper we have reported on our ongoing work on using a constraint-based approach towards feature interaction detection. Starting from a logic specification of the **BCS**, we drew a labeled transition system encoding its behavior. This graphical interpretation was implemented in `smodels` and tested for consistency. The testing phase not only showed the complexity of the problem but also the efficiency of the tool in checking specific properties of the implementation. We also described a method for encoding features as changes to the system's behavior. Finally, we integrated features on top of the **BCS** implementation. We first checked the consistency of each feature individually; we then verified feature interaction by integrating multiple features on top of the system.

The `smodels` tool box has previously been used as a model checking tool (see [10]), but in the present paper we focus instead on `smodels` (and more generally on constraints) as a means to model non-monotonicity. Even though it is too early to make useful extrapolations from the results we have obtained thus far, the behavior we have observed is encouraging. The methodology seems to be able to obtain answers to specific queries in a matter of seconds, offering useful information on models witnessing the query, if such a model exists.

Of course, for the approach advocated here *not* to be classified as Yet Another Formal Approach to feature interaction, further features should be modelled and more testing should carried out. An obvious next step would be to apply our approach to the features described in the feature interaction contest accompanying the FIW'00 workshop.

A particularly attractive aspect of the approach we advocate is that it brings the field of Computational Logic with all its machinery to the Feature Interaction problem,

and allows standard approaches in Knowledge Engineering to be applied. We can take advantage of standard techniques from Knowledge Engineering to, for example, easily quantify the complexity of the approach and capture its exact expressive power.

To finish, we discuss some further questions about the general framework. First, it may be worthwhile to migrate to other implementations of stable model semantics. Even though we have opted for `smodels` in our first investigations, there is a number of alternatives such as XSB [14] and DeReS [7]. Encoding the **BCS** and features in those systems would permit a better understanding of the idea of "features as constraints," independently of the specific characteristics of a particular implementation of stable semantics. Second, as we already mentioned, our approach is close to model checking but the two differ in many aspects. It would be rewarding to compare both approaches in detail, in terms of, among others, generality, flexibility, and performance.

Acknowledgement. Maarten de Rijke was supported by the Spinoza project 'Logic in Action.'

References

[1] The feature interaction project. URL: `http://www.illc.uva.nl/~mdr/Projects/FI/`. Accessed November 22, 1999.

[2] Smodels home page. URL: `http://www.tcs.hut.fi/Software/smodels/`. Accessed November 22, 1999.

[3] K.R. Apt. *Programming with Constraints*, 2000. Manuscript; available from `http://www.cwi.nl/~apt/krzysztof.html`.

[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] P. Cholewiński, V. Marek, and M. Truszczyński. Default reasoning system DeReS. In *Proceedings of the 5^{th} Internatinal Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528, Cambridge, MA, USA, November 1996. Morgan Kaufmann.

[8] 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.

[9] V. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38(3):588–619, 1991.

[10] 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.

[11] 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.

[12] M. Plath and M. Ryan. Plug and play featues. In *5^{th} International Workshop in Feature Interactions in Telecommunications and Software Systems*, 1998.

[13] 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.

[14] K. Sagonas, T. Swift, and D. Warren. XSB as an efficient deductive database engine. In *Proceedings of SIGMOD 1994 Conference. ACM*, 1994.

[15] 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.

[16] 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.