Master of Logic Thesis

Exploring the Behaviour of Feature Interaction via Stable Models

Rafael Accorsi ‡

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht, 24
1018 TV Amsterdam
The Netherlands
September, 1999

‡ This thesis is dedicated to my father *Valter*, my mother *Magda* and my brothers *Frederico* and *Felipe*.

Contents

1 Introduction

One the recent approaches advocated by software engineering is to consider a system as a combination of a basic system, which provides functions for stand-alone operation together with a set of modules of functionality, called features, which are added on top of the basic system. To see the system as a merging of basic system and features is particularly useful for both system developer and user. From the 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-party companies to design and develop features. These two characteristics also affect the user, who is free to add/remove features whenever it is necessary to. Furthermore, it allows the consumer to decide which feature developer offers the best services.

However, this approach 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., the situation where the behaviour of one feature affects the behaviour of another. If the interaction of features generates an unexpected system behaviour, then feature interaction becomes a problem. The problem of identifying feature interactions is called the *feature interaction detection problem.*

In telephony systems, features are pieces of functionality. In most cases these features are designed to add a new facility to a subscriber. However, they can also be developed to make the administration of the network easier. From a subscriber perspective, some “classical” examples of features are *Call waiting*, *Call forwarding* and *Voice mail.* There is a number of other features. For instance [3] contains hundreds of different feature descriptions.

The interaction of features can happen in different ways. For example, in *Call waiting*, if a busy subscriber receives a call, the new incoming call is put on hold. But in *Call forwarding*, every call to a subscriber is forwarded to another phone. The combination of these two features leads to feature interaction: if one of these two features has priority, the other is automatically disabled.

In practice, the network may offer a large number of possible scenarios. To realise that it suffices to understand that each subscriber can, at any time, modify her set of features by adding or removing them. Thus, the way that each subscriber reacts in the network changes upon the addition or removal of features. More precisely, the behaviour that a subscriber presents is tied to the features she has enabled.

The freedom that each subscriber has to add features generates several possible network configurations. Therefore, any attempt to analyse the behaviour of a telephone system has to go through all the combinations between features and subscribers and verify some properties. In the case of feature interaction, we should detect if any of the basic properties becomes invalid after the initialisation of a feature.

However, this combinatorial nature may impose problems to this task. Precisely, if the size of the network, i.e., the number of subscribers and its possible features, is high, then the task of generating all these configurations may explode in size. This combinatorial explosion could quickly lead to intractability. Hence, any computational approach should be carefully tuned with clever heuristics.

There is a number of different approaches to model feature interaction. Some are based on formal modelling of the system and features for a further property checking. This method, usually called *model checking*, has been applied with some success in most the cases. However, there is another stream using *satisfiability checking* to detect feature interactions. The idea in this case is to use some logic (mostly modal logic) to model the system and its features. In both cases the size of the problem requires an automated analysis procedure.

It is possible, however, to model features under a (still unexplored) non-monotonic perspective. The idea is to realise the feature as a "*a new piece of behaviour*." The advantages of such method appear in both the develop and the implementation phases. From a designing perspective, it provides an intuitive and modular understanding of features as bits of functionality. From an implementation aspect, it simplifies the task of identifying which behaviour is changed by the feature after its activation.

Intuitiveness is not a strong premise to measure one or the another method. A natural way of approaching the feature interaction detection problem should not only be able to provide an intuitive understanding of features. It also has to assure that we can safely compute with it. If one's choice is model checking, then a number of different tools equipped with a number of heuristics are found. If other's choice is satisfiability checking, then the option for different theorem provers is given.

Fortunately, one attempting to model features as "pieces of behaviour" can also find tools to compute with. One particular tools seems to collect the qualities that one looks for. Namely, the `smodels` tool box offers the ability to express non-monotonicity and moreover, compute with it. Formally, the `smodels` tool box is an implementation of stable model semantics for logic programs. Stable model semantics is an useful formalism to model knowledge representation. It merges the flexibility of the logic programming with the strength of constraint programming. Thus, it builds a proper framework for expressing non-monotonicity.

Technically, the `smodels` implementation of the framework is of special interest. The tool is not only able to model non-monotonic behaviour but also to compute instances of such problems. Even though there are other implementations of stable model semantics, `smodels` provides an extremely efficient algorithm. In fact, `smodels` is able to compute the stable models of a logic program using only linear space. In other words, provided the required amount of time, it can compute hard instances of combinatorial problems as the ones arising from the feature interaction detection.

Usually systems are described by means of labelled transition systems, which are simply graphs prescribing how the system changes its states along the possible interactions. In essence, this representation stands for the standard behaviour the system. Features, when active, redraw this graph in some predictable way, thus reflecting the change of behaviour. If one could realise a stepwise methodology strong enough to capture this informal process, then realise the integration of features would not present problems. Additionally, if one could reflect this facility on the feature implementation, then the feature deployment would be a seamless problem.

This thesis addresses the problem of modelling features and detecting its interactions. The stable model semantics and one of its implementations (`smodels`) provide the foundation over which we model the basic system and its corresponding features. In the course of the modelling, we develop a method to understand features as "pieces of behaviour" and, because the tool allows us to think under such perspective, we are able to reflect the heuristics in the implementation level.

Thesis Organisation

The remainder of this thesis is organised as follows.

In Section 2 we introduce the problem. We first provide conceptual definitions of *feature* and *feature integration*. The general definition of *feature interaction* is provided in Subsection 2.3 and its different classifications in Subsection 2.4. In the same subsection we describe the *causes of feature interaction* and *how to deal* with these interactions. Subsection 2.5 provides a description of a number of *approaches for feature interaction detection*. Subsection 2.6 focuses on *feature interaction in telephone systems*, which is the basic system we aim at. Finally, Subsection 2.7 closes the section surveying some of the work that has being made in feature interaction detection.

Section 3 introduce the stable model semantics for logic programs. We start by defining *basic Horn logic programming* in Subsection 3.1. In Subsection 3.2 we provide the *connection between standard logic programming and logic programming with stable model semantics*. In particular, this subsection addresses basic definitions about *syntax, semantics* and *complexity*. In Subsection 3.3 we introduce the `smodels` tool box by describing each of its modules and defining the computational engine. We close the section by tracing a parallel between `smodels` tool box and model checking.

Section 4 aims at describing the modelling of the basic system. We introduce the Basic Call System (**BCS**) in Subsection 4.1 and introduce an alternative approach to **BCS** in Subsection 4.2. In the same subsection we provide a rough introduction to description logic. The *dimensional view* of the system is provided in Subsection 4.3 and Subsection 4.4 provides relevant information about the encoding of the basic system into `smodels`. Subsection 4.5 develops *the* `smodels` *code* of **BCS**. Finally, Subsection 4.6 closes this section describing two kinds of tests on the **BCS** implementation.

In Section 5 we provide the implementation of features on top of the **BCS** implementation. Subsection 5.1 deepens the issues about *feature integration* which was first introduced in Section 2. Section 5.2 describes our *methodology for feature implementation* which leads to the *encoding of features*, described in Subsection 5.3. This section is closed by Section 5.4, where we query our implementation for properties, i.e., *feature interaction detection*.

Section 6 presents conclusions and directions to further work.

Acknowledgements

A thesis like this is always a major undertaking. Without the help of *many* other people, this work would have never become reality at all.

When comes to a *psychological* matter, I owe much to the tremendous fellows on the other side of the Atlantic: *Ana Paula*, *Betine*, *Camila*, *Christian*, *Daniel*, *Fábio*, *Fátima*, *Felipe*, *Fredy*, *Luís Fernando*, *Mariara*, *Marilton*, *Marta*, *Pedro Ivo*, *Renata*, *Roberta*, *Susana*, *Thaia* ... (this list would go on forever.) And my mates who shared Amsterdam with me: *Alex*, *Andreas*, *Antônio*, *Catarina*, *Clemens*, *Christopher*, *Fabíola*, *George*, *Giampaolo*, *Maria*, *Nick*, *Nikos*, *Stelyos*, *Theodoros*, *Valentina* ... (again, list everybody is impossible.) All of you have provided me an unimaginable amount of support and care. Finally, I shall express my deepest gratitude to *Katleen Schiepers* and her family for having (literally) adopted me.

Further, Nuffic and *Wendeline de Crook* have helped me quite a lot. Nuffic supported me with a scholarship, which allowed to stay in Netherlands and to follow the course. Specially Wendeline, for her expertise and experience with the subject, i.e., coordinating and guiding me in the application process and later, answering my (constant) set of e-mails with doubts and remarks. If there is someone responsible for my stay in Netherlands, then I should not deny Wendeline's hand on that.

But a scholarship is nothing if there is no place to study. For that I thank to *University of Amsterdam* for supporting my with *high tech* support and top rank classes and staff. *Ingrid van Loon* deserves a special mention though. She was much more than a course coordinator; indeed, Ingrid is an unforgettable friend, who helped from the they I arrived in Amsterdam, to the day I left it. Nonetheless, I cannot forget Professor's *Anne Troelstra* pieces of advice – these I will take with me for the whole life – and Professor *Dick de Jongh* for his efforts in teaching even outside the classes.

Also, a *HUGE* "thank you" goes to *Maarten de Rijke* and *Carlos Areces*. Without their help, I would probably never have delved so deeply in this thesis. Also, I would like to thank them for showing me their example of academical life and conduct. A special mention goes to *Carlos* and his willing to read through my drafts and suggest (innumerous) improvements, correct my (again, innumerous) spelling mistakes and suggest changes on the layout. Not only that though, also for those *midnight deadlines* when we ran down the stairs to not get stuck in the building.

In addition, I owe respectful thanks to my previous advisors Professors *Antônio Rocha Costa*, *Dalcidio Moraes Claudio*, *Graçaliz Dimuro*, *Paulo Blauth Menezes* and *Renata Reiser*. They have triggered this process and certainly, their never ending motivation was essential for me to reach this far.

For some reason which is unknown to me, authors always list their family last in these acknowledgement sections (that's why I dropped a footnote on the title page), but who am I to break this tradition? So, I cannot really say that my father has helped me to write this thesis provided he struggles to get the word processor running. Neither my mother, who learned how to e-mail because I came to Amsterdam or my brothers, who barely know what `Linux` is. However they showed me the most important *theorem* of life – no matter how far you are away from home, family's love, comprehension and care turn this distance to nothing. I am forever grateful.

2 Feature Interaction

Feature and feature interaction are usually introduced in a specific context. However, it is possible to place the subject in a more general setting. Indeed, much of the software and hardware systems currently available offer some sort of features. To be able to understand this characteristic one has to be able to distinguish the basic properties of a "featured" system

In this section we provide a general study of features to show its fundamental concepts and subtleties. Ou motivation is two-folded: firstly there is a noticeable gap in the literature covering the design of featured systems; and secondly, to fully understand our approach to feature interaction we shall provide enough background on the subject.

2.1 Services and Features

The first point that has to be well understood is the following: on the one hand there is a basic system; on the other hand, there is a number of possible features. Let's start by giving the following intuitive example:

Example 2.1 (Service and features). *An* Internet browser *can be seen as a specification of a basic system on top of which one adds pieces of functionality as required. Consider a* basic browsing system. *The only functionality that it has to provide is to understand the HTML code, load the pictures and show them.*[1] *To this basic behaviour a number of functionalities may be added, e.g., an e-mail composer, support to a number of different multimedia applications and data security facilities. However, there is an ever changing scenario; applications are designed much faster than the new version of the product. Therefore, the way to provide access to these new facilities is to release a specific module that is plugged into the basic system, i.e., add* features *to the system.*

From the example above we extract two of the main concepts from which we develop the ideas on this section.

Definition 2.2 (Service). A *service is a system that provides stand alone functionality.*

Definition 2.3 (Feature). A *feature is a piece of functionality which is unable to operate stand alone.*

There are two main approaches to conceive features. These may be studied by general software engineering, however below we provide a sketch on both.

1. Modifying the basic system's realisation to reflect the feature implementation. This appraches offers a number of weaknesses. For instance, it makes very complicated to extend the system with additional features. Additionally, the enabling (equivalently, disabling) of feature is nearly impossible.

[1]Assuming that a graphical interface is available, otherwise the functionality that has to be provided reduces to translate HTML.

2. Another, yet more clever, method is to conceive features under a modular perspective. The advantages on this approach relie on the intuitiveness and manageability that it provides: intuitive because it is easy to view a feature as an isolate piece of functionality which possibly alters the behaviour of the system. The system, then, turns to be easy to manage because activation (equivalently, deactivation) of the feature is just as simple as switch a button.

Additionally, modularity os a desirable characteristic, viz [22].

2.2 Feature Integration

Both basic system and features have to be described precisely. Literature usually separates the description of both entities, thus already providing conducting to a modular development of the system. We provide the formal definition of these notions below.

Definition 2.4 (System description). *A* system description *is a formal account of the basic properties and behaviour that must be satisfied by the basic system.*

Definition 2.5 (Feature description). *A* feature description *is a prescription of how the piece of functionality modifies the behaviour of the basic system. This description may also take into account the existence of another features.*

Features are usually designed off-line, i.e., they are engineered and, roughly speaking, tested in a "small proportion" scenario. After this phase, features are deployed, thus integrated to the real system. This phase is called *feature integration* in both formal and technical senses. We define it formally below.

Definition 2.6 (Feature integration). Feature integration *is the process in which a system description (containing features or not) is merged with a feature description. For instance, in Example 2.1 the process of merging basic browsing system description and a Postscript interpreter plug-in description is a feature integration.*

So far we have reached have defined what are the requirements to integrate a feature on top of a basic system. This then means that we are ready to implement features. It is desirable that the basic system is general enough to allow the implementation of several features. That is exactly the point where feature interaction araises. We now turn to the study of feature interaction.

2.3 Feature Interaction

One system may present one, several or no feature at all. It suffices to activate two features simultaneously to make feature interact. Let's us recapitualte some of the concepts introduced above.

On the one hand the system description defines a number of properties that the basic system must satisfy. On the other hand it specifies how the resources are allocated to treat the requests. But features cannot operate stand alone and must rely on the properties of the basic system and its resources. Thus, if a system has several features, these features have to share the same set of properties and resources. Features may also act together and reveal a compositional behaviour, i.e., features can *interact*. We define feature below.

Definition 2.7 (Feature interaction). Feature interaction *is a consequence of a number of feature integrations resulting in the scenario where one feature affects the behaviour of the other.*

Feature interactions may be characterised by the effect on the general behaviour of the system. To be more precise, they can be either *benign* or *malign* to the system. We provide a broader explanation below:

Definition 2.8. *An interaction is* benign *to the system when its compositional behaviour acts hormonically with the basic functionalities and other features. In other words, no property is violated and no unexpected behaviour is raised. Conversely, an interaction is called* malign *if it disrupts the functioning of the system.*

Let's provide an example to make these ideas more concrete and precise.

Example 2.9 (Feature interaction). *A simple example of malign feature interaction can happen in the basic browsing system (Example 2.1). Suppose that the browser has two plug-ins allowing the same multimedia capability. If none of these features has priority, the application will not be displayed. But note that the same basic system can also be an example of benign feacture interaction. Suppose that a new version of the basic browsing system is released. Additionally, suppose that a security hole is detected. A patch, i.e., a feature can be release to fix this problem. Surely the patch will result a benign interaction.*

As we note above, undesirable or malign interactions may disrupt the behaviour of the system as a whole; the system might turn unreliable and provide unpredictable answers. This characterises the feature interaction problem.

Definition 2.10 (Feature interaction problem). *The* feature interaction problem *occurs when a feature interaction violates any of the system's properties in unpredictable ways.*

Features can interact in various ways. In a complete account of every possible behaviour, one has to analyse every interaction. Even for small systems such a task is usually beyond human capabilities; the use of *automated analysis* is not only suggested but required. Automated analysis is also important when new features are designed by third-party developers since they may not be acquainted with the basic system and the already existing set of features [5].

2.4 Classifications of Feature Interaction

The feature interaction problem described above is rather general and the issue of classifying the various kinds of interaction is an important step towards comprehension of the problem. We follow the presentation of [2] which separates interactions according to their *cause*, i.e., reason of occurrence; the *phase* in the software-engineering life-cycle where they occur (for instance: requirements specifications, software design, implementation, distribution); the *number of users* involved (occurs within a multi-user environment); and lastly the *organisational hierarchy*, i.e., focusing on *who* in an organisation is responsible for dealing with the occurrence of interactions. We follow [24] to analyse the *cause* of feature interactions.

Causes of Feature Interaction In the following list we consider a *token* as a piece of information. The formal definition is dependent to context, therefore we avoid a formal definition.

• *Data consumption.* Happens when a piece of information, i.e., a *token* is consumed before arriving to the requester. The *lift system* [22] provides an example of *data consumption.* The lift system works with different floors and requests of attendance. Interactions can override the requests. Hence, data is lost.

• *Data modification.* Occurs when a token is unproperly modified before its consumption. Again, the lift system can provide an example. The requests in the lift system may be rescheduled by the activation of features. Hence, data is modified.

• *Resource contention.* Concerns the hardware involved in serving the feature. It occurs when a feature requires resources which are beyond the capacity of the resource server. The typical example of such interaction occurs with the *bridge* device. In *telephony systems*, the *bridge* is responsible for the telephone connections.

• *Data loss.* Happens when a token is received (emmited by a feature or base system in the party's side) but the system does not understand/know how to deal with it. This interaction can happen in the *browsing system* (Example 2.1) when the browser loads an image which it is unable to be shown. Possible reasons for that are the mismatch of versions or uncapability of displaying in the current hardware.

• *Control interaction.* Happens when a feature tries to control the behaviour of another feature; the problem arises when the feature is not expecting this control. LaTeX sytle files provide a suitable example of control interaction. The order in which the style files are loaded can modify the desired behaviour of one style.

Dealing with feature interaction According to [4], there are two main characteristics to be observed:

1. *When* interactions are dealt with.
    - *on-line*, when the features are deployed;
    - *off-line*, when the problem is treated beforehand.
2. *How* one deals with interactions.
    - *avoidance*, by designing features in such a way that either there is no interaction or the interaction is completely specified;
    - *detection*, by analysing the feature descriptions and tracing the interactions;

- *resolution*, by devising different methods to deal with interactions.

We have analised general concepts of feature interaction, i.e., its characterisation and different classifications. We are now ready to study view to understand methods to detect feature interactions. We turn to this point in the forthcoming section.

2.5 Approaches to Detection of Feature Interaction

Model checking approaches are the most employed methods to detect feature interaction. The idea in these approaches is to model the basic system and feature description using a formal language. Specifications are then provided as an input to a *model checking tool*, which verifies its correctness. Every possible behaviour is analysed, interactions revealed and eventually solved. Relevant work using the detection approach is given in [5, 24, 22, 23].

In contrast to model checking approaches, there are also contributions using the *satisfiability* method. The intuition behind this approach is to describe the system using a specific logic (modal, predicate or temporal in most contributions), add the features and check for the existence of inconsistencies in the resulting specification, as in [2] and [10]. Both model checking and satisfiability approaches are *detection methods.*

There are also works on the structural approach, method that fits in the *avoidance* category [32]. This method emphasises the modular specification of features and basic system. Therefore its intuition is to organise the specification in such a way that desirable properties are guaranteed by the specification's structure. Hence it is easy to add features without destroying its properties or increasing its complexity. [32] provides a thorough comparison between detection and modular methods.

In this work we aim at a specific area, i.e., we aim at studying feature interaction in the telecommunication setting. Thus, we now focus on the study of feature interaction on telephone systems.

2.6 Feature Interaction and Telephony Systems

The concepts and ideas described so far are applicable to any system where one perceives the distinction between a basic system and modules of functionality, i.e., features.

The telephony system matches this criteria. On the one hand there is a basic system (traditionally **POTS** ([5, 24, 22, 23]), alternatively **BCS** ([2] and [10]) and recently **INCM** ([31] and [33]) These acronyms stand for **P**lain **O**ld **T**elephony **S**ystem, **B**asic **C**all **S**ystem and **I**ntelligent **N**etwork **C**onceptual **M**odel respectively). On the other hand there are facilities that can be added on top of these systems aiming to provide new functionalities to both subscribers and network administration.

The addition of features can be seen as a natural extension of the basic system. These pieces behaviour aim at providing facilities to either the subscriber (alternatively, group of subscribers), or to the management of the network. Most of the features are designed taking into account the subscriber perspective and there is a simple reason for that: functionalities (in this contex) have a dynamic nature. In other words, a subscriber may want to activate a feature for a limited period of time and disable it afterwards.

Let's exemplify two "classical" examples of features.

Example 2.11 (Features).

- Call waiting. *when the subscriber is engaged in a call and there is a second incoming call, the subscriber is notified and the second call is put on hold. The subscriber has the choice of switching between the two calls at will.*
- Call forwarding. *All calls to a subscriber's phone are redirected to another phone.*

Feature interaction can occur even in the presence of a small number of features and subscribers. We mention an example below.

Example 2.12 (Feature interaction). *The interaction between* Call Waiting *and* Call Forwarding *leads to a situation where either feature can be active but not both. If the system gives priority to one of these features, the other will never be activated.*

In the telecommunication domain the expression *feature interaction* seems to have been introduced in 1983 in a publication by Bellcore containing a number of different features [3].

There are various aspects that make the detection of feature interaction in telecommunications into 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 21st century, both economically, politically, and technically [32].
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 analyze 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 modeling or knowledge representation point of view is the non-monotonic character of the addition of features. By definition, a feature modifies the behavior of the basic system, altering its properties.

Another fact that reveals the importance of the feature interaction analysis in the telecommunication domain is the impressive number of workshops and publications covering the topic.

As we mention in Section 2.5, there have been several formal approaches to the feature interaction problem. Most standard methodologies are based on model checking [22, 23], and some are based on on satisfiability checking [2]. Both methods fairly address the 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 relegated.

Next section addresses these methods in more detail.

2.7 Previous Work

Below we comment on some of the most significant works in feature interaction detection.

• *Tabular approach.* In [5] and [24] 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 are passed by means of tokens and system specifications are composed into reachability graphs. This graphs are analysed in order to reveal if any pathological behaviour of a certain feature or service is found.

• *The SFI tool.* In [22] and [23] a model checking approach is exploited by means of an extension of the **SMV**[2] tool. The **SMV** tool is a **CTL**[3] model checker for unlabelled finite automata. The extension **SFI**[4] 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.* [2] 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 [31] 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.

There is an extensive work on feature interaction and to exhaust this list goes

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

beyond the scope of this thesis. [33] provides an extensive bibliography. Furthermore there is plenty of electronic material available in the web. We point out the Uppsala Feature Interaction group (`http://www.docs.uu.se/docs/fi/`) as a basic reference to start with.

In this section we characterised feature interaction and its associated problem, i.e., the detection of feature interaction. As we note above (Section 2.6), there is no approach viewing the problem in a non-monotonic setting. That is exactly the gap that we aim at filling. However, to reach this objective we need both a formalism and a tool.[5]

Therefore, next section addresses one particular formalism (*stable model semantics*) and one implementation of this formalism (`smodels` tool box) which seems to be able to support our requirements.

[5] Recall that in Section 2.3 we recognised the need for automated analysis

3 Logic Programming with Standard and Stable Model Semantics

The idea is to be able to detect feature interactions (Section 2) using non-monotonicity. Thus, we need to select a formalism that both provides a formal background and a tool. However this may not be an easy task: firstly because the formalism itself should be intuitive and expressive; secondly, the tool supporting its computation should reflect the "intuitiveness" and expressivity while offering an optimised computational process.

One possible formalism is *stable model semantics*. The framework presented by stable model semantics evaluates positively for the aspects we mention above. Additionally, one particular implementation, the `smodels` tool box, turns to capture most of the required implementation aspects. Finally, the joint *stable model semantics &* `smodels` seems to build the framework we aim at. In this section we investigate them in order to understand how this joint can help us.

3.1 Basic Horn Logic Programming

Stable model semantics shares some of its aspects with standard logic programming. More precisely, the syntax is nearly *copied* from Horn Logic programming. Because we do not assume any previous knowledge on this formalism, we sketch the Horn Logic syntax below.

The logic programming paradigm substantially differs from the other programming paradigms. The distinction lies in the computing method: logic programs need to specify the logical part of the program, i.e., definitions and properties of objects and domains of interest. The way that the values are computed (namely, *control*) is provided by the resolution method.

The explicit separation between logic and control is often classified as *declarative programming*. Logic programming is used in Prolog and constraint programming implementations; recent contributions are works on reasoning representation and approaches to computational learning [1].

3.1.1 Syntax of Horn Logic Programs

Syntax conventions are important in any programming paradigm. Logic programming is not an exception.

Definition 3.1 (Logic programming alphabet). *We start by introducing an* alphabet *which is divided according to the following disjoint sets of symbols:*

- function symbols*;*
- relation *or* predicate symbols*;*
- variables*;*
- parentheses*, "(" and ")";*
- comma*, ",".*

Each function symbol or predicate symbol has a number of arguments associated to it. This number of arguments defines their arity*. The 0-ary function symbols are called* constants.

Now it is possible to define the language of Horn programs. We provide the following recursive definition.

Definition 3.2. *We use the following terminology:*

- *Constants or a variable are* terms*, furthermore, if f is a n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*
- *If p is a n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an* atom*. A variable-free atom is called* ground atom*;*
- *A* query *is a finite sequence of atoms;*
- *A* definite Horn clause*, or simply* clause *is a construct of the form $H \leftarrow B$ where H is an atom and B is a query. Specifically, H is called the* head *of clause and B is the body of the clause. According to the logic programming terminology, a definite Horn clause is called a* definite logic programming rule*;*
- *A* definite Horn theory *is a set of definite Horn clauses. Equivalently, a* definite logic program *is a set of definite logic programming rules.*

The following terminology is usually associated to logic programming. It plays an important role in further definitions and we introduce it here.

Definition 3.3. *Consider a Horn logic program P, we specify the following terminology:*

- *The* Herbrand universe *$\mathcal{HU}(P)$ of a logic program P is the set of all ground terms that can be constructed using the constants and function symbols in P. When P is a constant-free program, a special symbol representing this information has to be introduced. If P is function-free the $\mathcal{HU}(P)$ is finite, otherwise it is infinite [16].*
- *The* Herbrand base *$\mathcal{HB}(P)$ is the set of all ground atoms that can be constructed using the terms in $\mathcal{HU}(P)$ and the predicates symbols of a program P.*
- *The* Herbrand instantiation *$\mathcal{HI}(P)$ is the set of all ground instances of the rules of P. These instances are constructed by means of the terms in $\mathcal{HU}(P)$.*
- *An* interpretation *of P is a subset of the Herbrand base of P. An interpretation assigns a truth value to each element of the Herbrand base. If an atom is in the instantiation, then its valuation is true, otherwise, false.*
- *A (possible empty) set of atoms of the Herbrand base is a* model *for a logic program if its interpretation makes every rule of $\mathcal{HI}(P)$ true when the rules are seen as logical implications. A* least Herbrand model *of a program P ($\mathcal{LHM}(P)$) is the least set of atoms that satisfies P.*

The presence of function symbols in the language is crucial for Horn logic programming. It allows the encoding of hereditarily finite higher-order objects as terms of the language.[6] Most of the expressive power of logic programs is due to the use of recursive definitions [16]. If the program is function-free, then the Herbrand universe is finite and the ability of representing higher-order objects is significantly restricted.

3.1.2 The Procedural Interpretation

We use the following terminology: variables are denoted by a upper-case letters, e.g., X, Y. Overlined capital letter denote sets of variables, e.g., $\overline{X}, \overline{Y}$. Finally, atoms are denoted by lower-case letters q, r, s.

A query $q_1(\overline{X}), \ldots, q_n(\overline{X})$ should be viewed as a request for finding values for variables $x_1 \ldots, x_k \in \overline{X}$ such that the conjunction $q_1(\overline{X}) \wedge \ldots \wedge q_n(\overline{X})$ becomes true. Thus we can read the clause $H \leftarrow B_1, \ldots, B_n$ as the statement "*prove H by providing a proof for each B_i.*"

In logic programming, variables stand for unknown values, similarly to mathematics. This interpretation differs from the imperative approach where variables represent known but varying values.

Up to now we provided a theoretical view of logic programming. However, this approach has being used in a number of computational applications. There are methods supporting the implementation of this paradigm. Below we outline the transition from theory to computation.

One of the basic constructions behind the current automated deduction and logic programming is the *unification algorithm*, devised by Herbrand [13]. The unification algorithm is responsible for assigning values to variables by means of automatically generated substitutions. These substitutions are called *most general unifiers* and the process of applying the substitutions to a program is called *unification*.

Another fundamental construction is the resolution method, which is considered to be one of the most successful approaches for automated reasoning. Roughly, resolution is a forward search procedure for deriving logical consequences using the inference rules. Together, unification and resolution are responsible for the control part of most logic programming implementations.

3.2 Logic Programming with Stable Model Semantics

Stable model semantics appeared on the logic programming scene in the late 1980s. The idea was to provide an understanding to logic programs with negation. Standard logic programming could not be used since its methods are unable to represent logic programs enriched with the negation operator. Therefore, an alternative approach involving constraint programming was proposed.

Constraint programming offers a new perspective to programming. The intuition is that programs should specify requirements (constraints) which are applied over domains. These requirements are solved by means of general or

[6] The list operator "`[·]`" in a classical example.

specific domain methods. The goal of constraint programming is to solve *constraint satisfaction problems*, i.e., finite set of relations over a certain domain.

Logic Programming with Stable Model Semantics ($\mathbf{LP_{SM}}$, according to [19]) merges the advantages of the logic programming knowledge representation with constraint programming. This amalgam seems to be useful in reasoning about combinatorial problems, which frequently emerge on the computational scene.

The stable models semantics does not fit into the standard paradigm of logic programming languages. These approaches differ in the following aspects:

- The standard logic programming approach does not impose any restriction on the input program. The stable model programming requires a function-free input program.
- While standard logic programs approaches assign a single model to a logic program, stable models semantics assigns a (possible empty) set of models;
- In the presence of function symbols, Horn logic programs can specify any recursively enumerable set [28]. The stable model semantics requires a function-free logic program. The use of function-free program decreases the complexity and guarantees a finite Herbrand universe of finite programs. However, it disables the use of hereditary types.
- The resolution method used in standard logic programming is inadequate for logic programs with stable model semantics. This motivates the use of constraint programming (Definition 3.2).

While logic programming with stable model semantics is in many aspects different from Horn logic programming, the approaches share the idea of separating the logic and the control. Consequently, $\mathbf{LP_{SM}}$ is declarative [16]. The differences between the approaches rest on the syntax and semantics, affecting the control mechanism and the corresponding programming methodology.

3.2.1 Syntax and Semantics of $\mathbf{LP_{SM}}$

In this subsection we sketch both the syntax and the semantics of $\mathbf{LP_{SM}}$. As we shall see, the syntax resembles at a large the standard logic programming. It is not the cause though for the semantics, which offers different structures. Below we first focus on the syntax and then turn to the semantics.

The alphabet used in $\mathbf{LP_{SM}}$ is similar to the alphabet used in standard Horn logic programming (Definition 3.1). However we have to consider two significant changes:

- Forbid the use of functions;
- Include the negation symbol *not* in the language.

The modifications above reflect on the terminology we use. We then add the following standard concepts.

Definition 3.4 (Literal). *A* literal *is either an atom or the negation of an atom. A* query *is a finite sequence of literals.*

Now we define the shape of an $\mathbf{LP_{SM}}$ rule.

Definition 3.5 (Rules of stable model programming). *The rules for* $\mathbf{LP_{SM}}$ *have the same form as in Horn logic programming. However it is restricted to function-free clauses possibly enriched with negative literals. The rules have the following form:*

$$\underbrace{A}_{H} \leftarrow \underbrace{B_1(\overline{X}), \ldots, B_n(\overline{X})}_{\mathbb{P}},\ \underbrace{not\ C_1(\overline{X}), \ldots, not\ C_n(\overline{X})}_{\mathbb{N}}$$

where:

H *is the* head *of the rule;*

$\mathbb{P}$ *is the set of* positive body literals *of the rule;*

$\mathbb{N}$ *is the set of* negative body literals *of the rule.*

The intuitive reading attached to a rule r is sketched as follows:

- If every atom B_i in the *positive literal body* of r is in a solution and
- If none atom C_i in the *negative literal body* of r is in a solution then
- The head of r, H, is included in the solution.

The set of constraints of a rule r, denoted by $\mathbb{CR}(r)$, is obtained by joining the sets H, $\mathbb{P}$ and $\mathbb{N}$. We define $\mathbb{CR}(P)$ as the set of all constraints in a program P.

Now we formally define the concept of solution set.

Definition 3.6 (Solution set). *Given a set of atoms S and a logic program P, S is a* solution set *of P if S satisfies the set of constraints of P. A solution set S is* minimal *of a logic program P when no proper subset of S is a model to P.*

The sketch above together with the definitions on Section 3.1.1 provide enough syntactical background for the remaining of the work. We then address the semantics.

The idea is to formalise the intuitive constraint interpretation of rules by giving a declarative semantics to solution sets. We start by providing an example which motivates the representation of solution sets as stable models.

The intuitive reading of the rules does not capture all the relevant properties of solution sets. It is desired that solution sets are minimal, i.e., a subset of a solution set is no longer a solution. This means that a program $\mathtt{r(X)} \leftarrow \mathtt{r(X)}$ has the empty set as its unique solution set and the set $\{\mathtt{r(X)}\}$ is not a valid solution.

Example 3.7. *Consider the following program P.*

```
% Example code #1.
1.  r(X) ← r(X)
2.  c(X) ← not r(X)
```

According to the constraint interpretation, $\{\mathtt{r(X)}\}$ *and* $\{\mathtt{c(X)}\}$ *are minimal solution sets of* P. *However, the inclusion of* $\{\mathtt{r(X)}\}$ *is not properly justified. Intuitively, the inclusion of any atom in the solution set has to be supported by atoms that are already in the model, i.e., the solution has to be grounded.*

We now define the concept of groundedness of a solution set.

Definition 3.8 (Ground solution set). *A solution set* S *of a logic program* P *is* grounded *when every atom* $a \in S$ *is a logical consequence of a subset* $X \subset S$ *when the logic rules of* P *are understood as logical inferences.*

It is clear that the constraint interpretation does not capture minimality and groundedness. Thus a more expressive semantics has to used.

The stable model semantics is expressive enough to obtain minimal and ground models of logic program. Hence, we present the stable model semantics by means of a stepwise process. Remember that the concept of definite logic program and model were defined in Definition 3.2 and in Definition 3.3 respectively.

The process of determining the stable models for a logic program P is stepwise described as follows:

Definition 3.9 (Reduct). *Given a logic program* P *and a set of atoms* S, *the reduct* $\mathcal{R}$ *of the logic program* P *with respect to the set of atoms* S *is the definite logic program* $\mathcal{R}_S(P)$ *obtained by deleting:*

- *each rule that has a negative body literal not* C *such that* $C \in S$
- *all the negative body literals on the bodies of the remaining rules.*

The definite logic program $\mathcal{R}_S(P)$ can be seen as the set of applicable rules of P with respect to the set of atoms S, i.e., as the set of rules where the negative body literals are satisfied by the model.

To compute the stable models of a logic program P we start by fixing a set of atoms S as a candidate (stable) model for the program P. By computing the reduct $\mathcal{R}_S(P)$ of P we obtain the set of potentially applicable rules. The idea is fairly clear: capture minimality and groundedness of a (stable) model by requiring that *every atom* in the model is a consequence of the set of applicable rules given by the set S. Additionally, it is required that *every consequence* of the potentially applicable rules is included in the model.

Definition 3.10 (Closure of a definite logic program). *The closure* $Cl(Q)$ *of the definite logic program* Q *is the unique set of consequences obtained from* Q *when its rules are interpreted as inference rules. The closure can be computed in a forward chaining manner: starting from the empty set and including the head of a rule* r *as long as the atoms in the body of* r *are already in the model. The process stops when no atom can be added to the set.*

Notice that $\mathcal{R}_S(P)$ is a definite logic program and then we are able to compute the closure $Cl(\mathcal{R}_S(P))$

Definition 3.11 (Stable model). *Let* P *be a ground (variable-free) logic program and let* $\mathcal{R}_S(P)$ *be the reduct of* P *with respect to the set of atoms* S. *Then the set of ground atoms* S *is a* stable model *of* P *if* $S = Cl(\mathcal{R}_S(P))$.

We show the process by means of an example.

Example 3.12 (Computing stable models). *Given the following logic program P and a set of atoms $S = \{\mathtt{r(X)}, \mathtt{p(X)}\}$:*

```
% Example code #.2
1.  p(X) ← not q(X), r(X)
2.  q(X) ← not p(X)
3.  r(X) ← not s(X)
4.  s(X) ← not p(X)
```

The reduct $\mathcal{R}_S(P)$ of P is the following:

```
1.  p(X) ← r(X)
2.  r(X) ←
```

Clearly $Cl(\mathcal{R}_S(P)) = \{r(X), p(X)\}$. Therefore S is a stable model of P. The $\mathbf{LP_{SM}}$ *allows multiple models: the set $S' = \{q(X), s(X)\}$ is also a stable model of P.*

Before addressing the complexity of the process, we comment on the relation between stable model semantics and non-monotonic reasoning. Stable model semantics is closely related to a number of knowledge representation formalisms:

- Stable model semantics relates with *autoepistemic logic* [15]. A stable model of a program can be interpreted as "*a possible set of beliefs that a rational agent might hold*" when the agent's behaviour is defined by program's rules.
- Stable model semantics relates to Reiter's *default logics* [25]. Within default logic, stable models are *extensions* to default theories [16].

3.2.2 Complexity of Computing Stable Models

Several decision problems can be associated with stable models. We mention the following two:

- Determine whether a program P has stable models;
- Given an element of the Herbrand base, determine if there is any model containing this atom.

The method used to determine the existence of stable models (Example 3.12) is not constructive, i.e., considering a logic program P, for every set of atoms S, we compute the reduct and check whether S matches with the closure of the reduct. However, once a candidate set of atoms is provided, it is possible to check if it consists in a stable model in quadratic time. From this, satisfiability is easily seen to be decidable in NP. Moreover a matching lower-bound was proved in [15].

Theorem 3.13. The problem to decide whether a finite propositional logic program has a stable model is NP-complete.

But we can ask for the computation of models satisfying certain conditions. In this case we provide a subset of the solution set and the computation searches stable models in which these atoms are included. Even though the problem remains in NP, searching for a particular model can considerably prune the search space.

3.3 The `smodels` Tool Box

We have provided the formal framework in which we aim at developing our idea. Remember that we emphasised the need for both a formalism and a mechanism to compute with it. We now introduce one particular implementation of stable models semantics — the `smodels` tool box.

`smodels` is a C++ implementation of stable model semantics. The implementation has being developed at the Department of Computer Science of the Helsinki University of Technology and it is supervised by Niemelä [27].

The `smodels` tool box presents two modules:

- `lparse` which is responsible for the grounding procedure;
- `smodels` which is responsible for the stable models computation.

Before providing an outline of these two modules, it is worth noting that the `smodels` tool box seems to be the current state-of-art in stable model semantics. A comparison with the `SLG` system (which is another implementation of stable model semantics for logic programs [7]), is provided in [20]. This comparison, reveals that `smodels` outperforms the `SLG` system.

3.3.1 The `lparse` Module

According to the Theorem 3.13, the problem of determining whether a logic program has stable models is NP-complete. Usually problems in this particular class are difficult to compute with. The reason for this hardness is not the process itself (e.g., the algorithm to compute stable models is fairly intuitive), but the size that this computation may require. In most of the cases, the computation triggers an explosion in the size, leading then to intractability. Hence, any attempt to compute stable models (and in general, problems in NP) should equip the process with a clever representation heuristics.

The `smodels`tool box uses a linear space algorithm for computing the stable models for logic programs.[7] This characteristic is deeply connected with the grounding process. In the `smodels` tool box, the task of grounding a function-free logic program is executed by the `lparse` module. We shall first define what is ground process.

Definition 3.14 (Grounding process). *The* grounding process *transforms a logic program P into an equivalent program P_G such that every rule in P is variable-free. The grounding process has to ensure that* no *stable model is lost.*

[7] We analyse the implications of this characteristic below.

Remark 3.15. *Given a function-free logic program, its Herbrand universe is always finite. Thus, the ground instantiation of a program is finite as well. However, the instantiation can be very large in the case where the whole Herbrand universe is taken into account and the size problem mentioned above may arise. Additionally, this method can lead to unacceptable performance.*

We first provide an example.

Example 3.16. *Consider the following logic program P.*

```
% Example code #3.
1.  da(1)  ←
2.  da(2)  ←
3.  r(X,Y) ← da(X), db(Y), not db(X)
4.  r(X,X) ← da(X)
```

The Herbrand universe of P is the set $\{1, 2\}$ and the Herbrand base is

$$\{\mathtt{d_a}(1), \mathtt{d_a}(2), \mathtt{d_b}(1), \mathtt{d_b}(2), \mathtt{r}(1,1), \mathtt{r}(1,2), \mathtt{r}(2,1), \mathtt{r}(2,2)\}.$$

Given the Herbrand base, we can provide the full ground instantiation of P.

```
% Example code #4.
1.  da(1)  ←
2.  da(2)  ←
3.  r(1,1) ← da(1), db(1), not db(1)
4.  r(1,2) ← da(1), db(2), not db(2)
5.  r(2,1) ← da(2), db(1), not db(1)
6.  r(2,2) ← da(2), db(2), not db(2)
7.  r(1,1) ← da(1)
8.  r(2,2) ← da(2)
```

According to the process for computing stable models that we defined, this program has an unique stable model:

$$\{\mathtt{d_a}(1), \mathtt{d_a}(2), \mathtt{r}(1,1), \mathtt{r}(2,2)\}.$$

It is clear that from the third line up to the sixth there is no contribution to the stable model computation, i.e., these lines can be suppressed from the grounding process.

The idea then is to consider a subset of the Herbrand instantiation in such a way that *no* stable model is lost. Thus, we have to apply certain restrictions in the input of the program. This restrictions allows the distinction of which rules have to be considered and which have not. We start by providing the following definition.

Definition 3.17 (Domain predicate). *A* domain predicate *is a predicate which is not defined recursively or in terms of recursively defined predicated. Given a logic program P, the domain predicates of P are responsible for assigning values to variables.*

Example 3.18. *Consider the program in Example 3.16. The set of domain predicates of domain predicates is the following:*

$$\{\mathtt{d_a}\}$$

Restricting the program by requiring that every variable in a rule also appears in some domain predicate seems to be the most efficient heuristic to optimise the grounding process. With domain predicates it is easy to determine which rules are relevant for computing stable models and which are not.

Definition 3.19 (Domain restricted program). *A program P is called domain restricted iff it holds that for every rule r of P, if a variable v appears in r, then it also appears in some domain predicate of r.*

Summing up, to obtain an efficient grounding process, the `smodels` tool box requires a function-free, domain restricted logic program as input. Given this restriction, the process of computing stable models requires linear space. We provide a final example before addressing the `smodels` module.

Example 3.20. *Consider the program in Example 3.16. The set of ground rules is the following.*

```
% Example code #5.
1.  da(1)  ←
2.  da(2)  ←
3.  r(1,1) ← da(1)
4.  r(2,2) ← da(2)
```

It is clear that the program offers the same stable models.

3.3.2 The `smodels` Module

The `smodels` module provides an algorithm for implementing stable models semantics for function-free domain restricted logic programs. A program is first grounded by the `lparse` module and then piped into the `smodels` module. The `smodels` module computes the solution sets of the program. We provide the most relevant properties of `smodels` below.

Properties of `smodels`

Flexibility `smodels` is able to solve a number of decision problems concerning stable models. `smodels` is also able to execute a query-based search upon the set of stable models. The query-based search is particularly useful when the `smodels` tool box is used for model checking purposes;

Efficiency `smodels` is based upon a bottom-up backtracking algorithm that drastically prunes the search space. This algorithm is based on an approximation technique for stable model semantics and presents strong connections with well-founded semantics [19];

Time complexity `smodels` has a quadratic time algorithm for checking whether a set of atoms S is the well-founded stable model of a ground logic program.

Space complexity `smodels` runs in linear space.

These properties open the possibility of using the stable model semantics in areas where the resulting ground programs presents a large set of ground instances and/or a large set of stable models. The linear space complexity also enables `smodels` to solve hard instances of combinatorial problems, provided the allocation of the necessary amount of time.

3.3.3 Model Checking and the `smodels` Tool Box

The `smodels` tool box is not a native model checker. However, it can be adapted to execute model checking. Work relating the `smodels` tool box and model checking for distributed systems is given in [19]. In [12], an algorithm for verification of deadlock in Petri-nets is provided.

Recall that model checking is a verification procedure for properties of a system. The system is first described using a formal language. The formal description of the system is provided to a model checking tool, which investigates its properties.

The `smodels` syntax provides a statement that enables the model checking with the `smodels` tool box. The `compute` statement is a reserved word in `smodels` syntax that specifies the number of stable models to be computed and the characteristics they should satisfy. This statement can be suppressed from the code. In that case a default configuration is assumed and `smodels` searches for a single stable model in which the atom `false` is not present.

Example 3.21. *Consider the following `compute` command.*

$$\texttt{compute}\ 4\{\texttt{not r}(1),\ \texttt{u}(1,6)\}$$

Given a program P, this command specifies the search for at most 4 models of P in which the ground atom `r(1)` is not *present and the ground atom `u(1,6)` is present.*

Hence, the `smodels` tool box can be used as a model checking tool. The idea is to define a system using $\mathbf{LP_{SM}}$ and, by means of the `compute` statement, check for specific properties.

One of the advantages of the `compute` statement is its efficiency. To ask for properties is to provide explicit constraints on the solution sets. The use of such constraints results in a considerable gain in time since these constraints shorten the search space. However, in terms of complexity, the problem is kept within NP.

Taking stock, on the one hand we have the problem of detecting feature interaction and one still unexplored facet, i.e., to approach the problem by a non-monotonic perspective (Section 2). On the other hand, we have seen that stable model semantics is an useful formalism to model non-monotonicty. In addition to that, it also provides one implementation that suits to the requirements we have emphasised.

The further step is to merge the problem (detection of feature interaction) and one possible solution (stable model semantics and `smodels`) and eventually model the problem using the framework. We take the first step towards this objective in the next section.

4 The BCS Implementation

We are now ready to apply the framework we presented in the last section to the problem of detecting feature interactions in telecommunication. However, as we note in Section 2.6, every telephony system is built by a basic system and features, which are added on top of this system. Hence an attempt to model feature interaction has first to address the modelling of the basic system and then move to features.

Another fairly common heuristic to describe the behaviour of a system is to draw a graphical interpretation of it. This idea is to provide a concrete picture of the possible states that the system may be as well as the actions triggering the changes of state. Additionally, if an extension to the system is required (e.g., add features or to implement a new state), it is more intuitive to reason about it with the graphical view.

More importantly though is that the graphical view makes the encoding of the system much easier. Together with the formal description, which has to be provided clarify details about data and functions, it helps to choose the heuristic to implement the system.

In this section we address these issues in detail. In particular, we move towards the implementation of the basic system in the formalism of choice, i.e., the stable model semantics

4.1 Introducing the BCS

The **BCS** (Basic Call Service) stands for a formal description of a telephone network. As **POTS** and **INCM** (Subsection 2.6), the **BCS** describes how subscribers act in a telephony network. The **BCS** also offers a high-level account of the physical consequences of the actions on the system, e.g., allocation of hardware resources, detection of states of the network, etc.

The **BCS** assumes that all the telephones in the network have a set of basic characteristics to use the system. Further, the **BCS** has to keep an account of every resource of the network to guarantee that no resource is allocated beyond its maximum capacity. The set of network properties, i.e., the least set of conditions to assure the functionality of the service, is defined taking into account both subscriber's actions and hardware properties.

Consider a set SUBS of subscribers. The **BCS** defines that at each time every subscriber finds herself in an unique state in the network. These states may or may not be perceivable to the user. On the one hand there is a finite set of perceivable states, e.g., *idle_u*, which defines that the telephone of the subscriber u is on hook and silent. On the other hand there is a finite set of *internal states* of the network, which may not be directly perceivable. For instance, the state *ready_u* defines that the telephone of the subscriber u is accepting a dialling. Further, a finite set of actions is provided. These actions are responsible for taking the subscribers from one state to another on the network.

This completes the outline of the **BCS**. A complete formal description is provided in [10]. However, we focus on another, more abstract formal description provided in [2], which is based on description logic.

4.2 The Description Logic Approach

A formal description of the **BCS** is provided in [2]. This approach uses the *description logic* $\mathcal{ALC}$ with generic T-Boxes. The work is the starting point of our approach and we provide an overview of its ideas in the next section.

4.2.1 Basic Description Logic

Description logics aim at representing structured concepts and objects. Objects may be related to other object and may satisfy some of the concepts. To preserve the structural properties of the data being represented, description logics descending from the KL-ONE system [6] split information into two sorts:

- **T-Boxes** which provide *terminological knowledge*, i.e., definitions of derived notions.
- **A-Boxes** which contains *assertional information*, i.e., facts concerning the individuals.

Example 4.1. *Consider the following:*

T-Box	***A-Box***
$\mathit{computer} \sqsubseteq \mathit{OS\text{-}controlled} \sqcap \mathit{electronic\text{-}device}$ $\mathit{e\text{-}mail} \doteq \mathit{computer} \sqcap \exists\ \mathit{network}.\top$	*c:electronic-device* *c:OS-controlled* *(c,n) network*

The first formula in the T-Box defines the concept of a computer *as a subset of those electronic devices which are controlled by an operating system. This definition is not strict because further conditions like* to have an input device *are not specified. On the other hand, the second formula strictly defines the concept of* providing e-mail services *as the set of all computers with access to network facilities.*

The formulas in the A-Box assert that a certain c *is both an electronic device and controlled by an operating system. Furthermore, there exists an element* n *which stands in the* network *relation with* c*. Notice that the available information is not enough to characterise* c *as being able of using e-mail since we do not know if* c *satisfies the conditions which are left open in the definition of computer.*

Summing up, *concepts* and *roles* are the basic constructions of *description logics* and each description logic is characterised by the method in which both concepts and roles can be combined with A-Boxes and T-Boxes. More formally, the signature of a description logic is given by specifying three disjoint sets C, R, and A of atomic concepts, roles and constants, respectively.

We now describe the formal description of **BCS** provided in [2].

4.2.2 The Description Logic Approach to BCS

We provided a sketch of description logic above. As we mentioned before, [2] models the **BCS** using a description logic approach. Hence, intuition is to provide a formal model of **BCS** by means of terminologies and assertions.

Definition 4.2 (Knowledge base). *A* knowledge base *is a pair $\langle T, A\rangle$ where T is a subset of the set of terminologies and A is a subset of the set of assertions.*

The **BCS** is modelled as a knowledge base. The first step is to fix the set of atomic concepts, relations and constants, i.e., the *signature*. It is assumed that u and v stand for elements of a finite set SUBS of subscribers.

Definition 4.3 (Atomic concepts). *The atomic concepts express that a subscriber u is in a given state in the network.*

idle_u	*the telephone of* u *has the receiver on hook and silent.*
dialtone_u	*the receiver is off hook and emits a dial tone.*
busytone_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.*
ringback_u	*the receiver is off hook and emits a ring back tone. The called party is ringing.*
engaged_u	*there is a speech path between* u *and another subscriber.*

Concepts to express a (minimal) presence of the network are also required. These concepts represent the states that are not directly perceivable to the subscribers.

ready_u	*the telephone of* u *is ready to receive a subscriber number.*
calling_uv	*the telephone of* v *is ringing and* u *is waiting for* v*'s off hook to accept the call.*
rejecting_u	u *called a party that is already busy or has put the receiver back to the phone.*
path_uv	u *and* v *can communicate.*

Now we specify the set of roles.

Definition 4.4 (Atomic roles). *Roles represent 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.*

Now that we have definitions of concept and role names, the **BCS** can be specified. The description logic approach to **BCS** separates the statements in four categories, which are provided below.

Interface statements. These axioms are T-Box expressions connecting the observable states of a telephone with the ones representing network states. **Example**. $calling\_uv \sqsubseteq ringing\_v \sqcap ringback\_v$

Assertional statement. The **BCS** requires a single A-Box statement corresponding to the initialisation state, i.e., it is required that every user are in *idle* at the moment s_0.
Example. $s_0 : \prod_{u \in \mathsf{SUBS}} idle_u$

Frame statements. These rules specify that certain events do not influence the state of other parts of the system.

Example. $path\_uv \doteq \forall offhook\_z.path\_uv$

Liveness statements. The *liveness statements* consist on 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.

Example. $idle\_u \sqsubseteq \exists offhook\_u.ready\_u$

We refer to [2] for details on the statements above. Now we focus on the liveness statements. The liveness statements constitute the basic functionality of the **BCS** scheme and therefore the complete set is provided below.

Definition 4.5 (Liveness statements). *The liveness statements are defined as follows.*

1. $idle\_u$	$\sqsubseteq \exists offhook\_u.ready\_u$
2. $ready\_u \sqcap idle\_v$	$\sqsubseteq \exists dial\_uv.calling\_uv$
3. $ready\_u$	$\sqsubseteq \exists onhook\_u.idle\_u$
4. $ready\_u \sqcap \neg idle\_v$	$\sqsubseteq \exists dial\_uv.rejecting\_u$
5. $rejecting\_u$	$\sqsubseteq \exists onhook\_u.idle\_u$
6. $calling\_uv$	$\sqsubseteq \exists offhook\_v.path\_uv$
7. $calling\_uv$	$\sqsubseteq \exists onhook\_u.(idle\_u \sqcap idle\_v)$
8. $path\_uv$	$\sqsubseteq \exists onhook\_u.(idle\_u \sqcap rejecting\_v)$

The statements follow the standard description logic reading of [2]. The next table defines the meaning of each of the above statement.

1. If *u* is idle, she can go off hook and get ready to initiate a dial;
2. If *u* is ready to dial and *v* is idle, *u* can dial *v*'s number and establish a call;
3. If *u* is ready to dial, she can decide to go on hook and return to idle;
4. If *u* is ready to dial and *v* is *not* idle, a dial from *u* to *v* results in a busytone;
5. If *u* is being rejected, she can be idle by going on hook;
6. If *u* is calling *v* and *v* goes off hook, *u* and *v* can talk;
7. If *u* goes on hook while calling *v*, both return to idle;
8. If *u* goes on hook while talking to *v*, *v* receives a busytone and *u* goes to idle.

As we have been advocating, a graphical view of a system is desirable. We now describe the graphical interpretation of **BCS**.

4.3 The Dimensional View

The specification provided in Subsection 4.2.2 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.

Labels. The description logic approach provides a clear distinction between states and actions. This separation is the first step towards a graphical interpretation. *States* and *actions* are the basic constructions, both in description logic (where they stand for concepts and roles, respectively) and in a transition system. We define the labelled transition system $\mathbf{BCS}_{LTS}$ upon the following sets.

Definition 4.6 (Set of labels). *We select a proper subset $\mathbb{S}$ of the set C of concepts to represent the states of the transition system. The set $\mathbb{A}$ of actions consists of the set R of roles. Finally, the set $\mathbb{L}$ of labels is $\mathbb{S} \cup \mathbb{A}$.*

$$\begin{aligned}
\mathbb{S} &= \{idle_u, ready_u, rejecting_u, ringing_uv, calling_uv, path_uv\} \\
\mathbb{A} &= \{offhook_u, onhook_u, dial_uv\} \\
\mathbb{L} &= \mathbb{A} \cup \mathbb{S}
\end{aligned}$$

We now focus on the transitions.

Transitions. According to the usual interpretation of a transition system, changes of states are triggered by actions. More formally, the set of transitions is a relation $R_{\mathbb{T}} \subseteq \mathcal{P}(\mathbb{S}) \times \mathbb{A} \times \mathbb{S}$.

Definition 4.7 (Transitions). *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.*

The notation is described as follows: 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 (↓) in front of a state or action means that the next label represents information of a different subscriber. Finally, "$\rightsquigarrow$" means* "move." *Hence, the statements in the box below become fairly easy to read. For instance, the first axiom means that* "an 'offhook' action in the state 'idle' moves the subscriber 'u' to the 'ready' state."

1.	$idle\_u + \mathsf{offhook\_u}$	$\rightsquigarrow ready\_u$
2.	$ready\_u + \downarrow idle\_v + \mathsf{dial\_uv}$	$\rightsquigarrow calling\_uv \quad (if\ u \neq v)$
4.	$ready\_u + \downarrow not\ idle\_v + \mathsf{dial\_uv}$	$\rightsquigarrow rejecting\_u$
5.	$rejecting\_u + \mathsf{onhook\_u}$	$\rightsquigarrow idle\_u$
6.	$calling\_uv + \downarrow \mathsf{offhook\_v}$	$\rightsquigarrow path\_uv$
7.	$calling\_uv + \mathsf{onhook\_u}$	$\rightsquigarrow idle\_u + \downarrow idle\_v$
8.	$path\_uv + \mathsf{onhook\_u}$	$\rightsquigarrow idle\_u + rejecting\_v$

It is not difficult to impose a declarative flavour on the transitions. The use of variables 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 increases.

The set of statements allows us to consider each subscriber as an independent dimension of the n-dimensional transition system, where n stands for the cardinality of SUBS. Figure 1 contains the transition system for a single 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 another state. The same action may be used in different states. To differentiate between the uses in different states, we assign different indexes to the actions, e.g., *onhookA*, *onhookB*.

This "multi-dimensional" view gives meaning to the down arrow operator $\downarrow$ that we use in actions and states. The down arrow is supposed to convey the idea that actions or states are defined in a dimension that is different from the current one. It is possible to identify four distinct contexts involving actions and subscribers: (see Figure 1 for the examples).

Context 1. Internal actions with effect inside. When the subscriber takes an action which does not involve any other subscriber but herself.
Example. Movement from the *idle_u* state to *ready_u* by an off hook.

Context 2. Internal actions with effect outside. When the subscriber takes an action which moves herself and her party from one state to another.
Example. Movement from *path_uv* to *rejecting_u* of party by an on hook of the caller.

Context 3. External actions with effect inside. When the subscriber sees her state affected by an action taken by a different subscriber, who is acting in a different dimension of the graph.
Example. The transition from *idle_u* to *ringing_uv*.

Context 4. External actions with effect outside. These actions correspond to the frame actions, i.e., actions which do not affect the subscriber on which we are focusing.
Example. Movement of a subscriber v from the *idle_v* state to *ready_v* by an off hook.

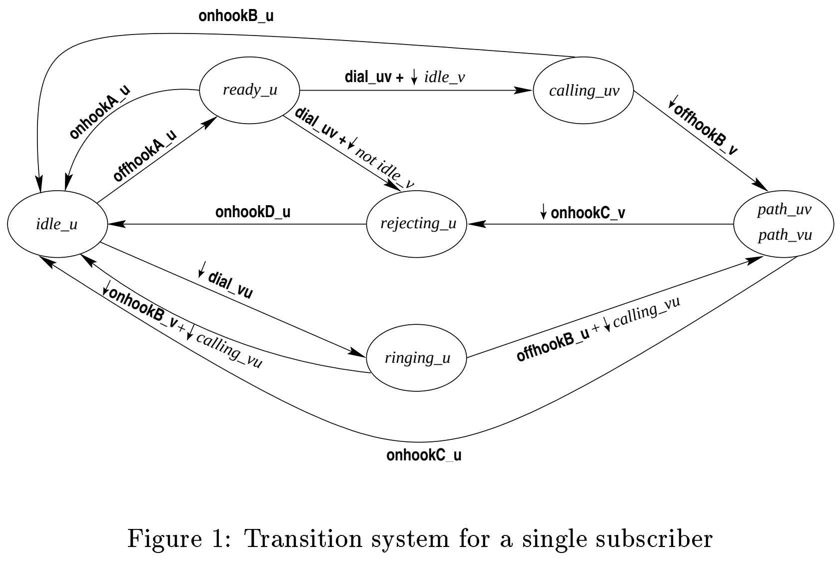

Figure 1: Transition system for a single subscriber

We now turn to the translation of $\mathbb{T}$ (Definition 4.7) into `smodels`.

4.4 The `smodels` Encoding

The `smodels` tool box (Subsection 3.3) provides enough expressive power to encode the transition system presented above. The **BCS** is defined by means of transition schemes, and hence our implementation calls for a declarative programming language. Indeed, he `smodels` tool box is declarative [16]. Additionally, one of `smodels`' characteristics is the efficiency in computing combinatorial problems. To exploit and explore **BCS** configurations is a combinatorial problem and to encode the **BCS** in `smodels` allows us to automate this process.

Lastly, the encoding allows us to use `smodels` as a model checking tool (Subsection 3.3.3). The model checking approach seems to be the most appropriate method to analyse the properties of systems like **BCS**. Checking properties via the `compute` statement (page 23) is straightforward and efficient.

Before going into details about the implementation of **BCS**, we describe some characteristics of `smodels`. These characteristics cause changes to Figure 1. Below we describe the changes and present the new picture.

4.4.1 Intuitions

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 4.8 (Cycle). *A* cycle *is a sequence of labels from* $\boldsymbol{BCS}_{LTS}$ *that represents a move of a subscriber from the* idle *state back to* idle.

Remark 4.9. *Encoding information about cycles allows us to:*

- *Differentiate between actions executed in different cycles;*
- *Provide a bound on the number of possible interactions, thus assuring termination.*

The limit on the number of cycles suggests the following definition.

Definition 4.10 (Run). *A* run *is a set of cycles starting from the initial cycle and finishing when the limit on the number of cycles is reached.*

These concepts are depicted in the following example:

Example 4.11. *The set of labels below represent a valid run for a two cycles setting.*

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

Summing up, every state that a subscriber visits and every action she engages in, is recorded in a stable model, i.e., stable models reflect the behaviour of subscribers in the network. Eventually each valid run is a stable model.

We now analyse some consequences of this method of computation.

4.4.2 Changes in the $\mathbf{LTS}_{BCS}$

The specific method in which the stable models are computed by `smodels` forces a number of changes in Figure 1. This subsection aims to describe the reason of their occurence and, to justify the changes leading to Figure 2.

The `idle` state. According to the description logic approach, the network is initialised with every subscriber in the `idle` state (page 26). Moreover, the subscribers return to the `idle` state after any interaction with the network. Therefore, atoms representing these `idle` states are included in the stable model. However, one of the transitions of $\mathbb{T}$ works upon the condition that a subscriber does not have an `idle` state recorded in the stable model.

More specifically, the transition from the state `ready` to the state `rejecting` checks the `idle` state of a subscriber v. If the atom representing the idle state is not an element of the stable model, then she moves to `rejecting`. However, it will never be the case that she has no `idle` recorded. Thus we have to encode additional information to enable this transition.

To represent the information that a subscriber u is not in her `idle` state we add new labels. These labels act as boolean variables which are activated, i.e., are recorded in the stable model, whenever a subscriber u leaves the `idle` state. However Figure 1 shows that each subscriber has two distinct ways of being active in the network:

- Going off hook and moving to `ready`, i.e., a potential *caller*;
- Having the number dialed by a subscriber and being taken to `ringing`, i.e., a potential *callee.*

When a subscribers u dials v's number, there are three possible outcomes: either v is a potential caller, which is represented by the `noidle` label; or v is a potential callee, and then an `othernoidlebyrang` is recorded in the model; or finally v can be `idle`, and then the `noidlebyrang` is included in the model. This information is combined with the dial action, which is described below.

The `dial` action. The `dial` action works upon a number of conditions. To understand the mechanism, we split the action in two phases.

The first phase is introduced to make the encoding more transparent. The idea is to introduce an intermediate state (namely, `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 `dialing`, the `dial` action takes a subscriber to either a `rejecting` state or a `calling`, depending on a external checking of the properties of the party. To decide the outcome of the `dial` action `smodels` uses the information about the `idle` state which is encoded in the first item above. Now we describe how this information affects the `dial` action.

- *The* `rejecting` *state.* The subscriber is moved to a `rejecting` state whenever her party is engaged on another interaction. In the discussion of the `idle` state we presented the separation of this information by means of new atoms. Thus, the `dial` action has to test whether one of the atoms `noidle` or `othernoidlebyrang` is in the model. If the inclusion holds, the subscriber is taken from the `dialing` state to `rejecting`.

- *The* `calling` *state.* The same external checking is made in this transition. But unlike the previous transition, the subscriber is taken to the `calling` state when none of the conditions `noidle` or `othernoidlebyrang` holds in the stable model. Thus, it suffices to use the negation operator *not* in the external checking. Note that the callee is moved to a `ringing` state where the atom `noidlebyrang` is recorded in the stable model.

Constraints on the `dial` ***action.*** The `dial` action generates all the possible callees to a given caller. The range of values which is generated specifies settings that are inconsistent with the **BCS** framework. Thus, these values cannot be considered as valid calls attempts and to define explicit constraints over this action seems the most efficient and succint method to select the proper set of values. We return to this topic later in this section.

Splitting 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 uppermost `path` stands for the case where the subscriber u calls v. The lowermost represents the situation where u is the callee, i.e., she had her number dialled by a subscriber v.

The separation on the `path` state forces the separation of the `onhookC` action. The `onhookC_caller` part represents an on hook action from the caller and the `onhookC_party` represents the same action however when the party takes it.

Figure 1 has to be modified to deal with the changes above. Figure 2 depicts the final design of the **BCS** according to the implementation requirements.

4.4.3 Basic Atoms

This section aims at defining the necessary background for implementing the **BCS** code. We use atoms to encode the set of states and actions ($\mathbb{S}$ and $\mathbb{A}$, respectively). The idea is to capture the intuitive meaning that a subscriber is either in one of the **BCS** states or she is moving from one state to the other by means of an action. We start by fixing the domain predicates.

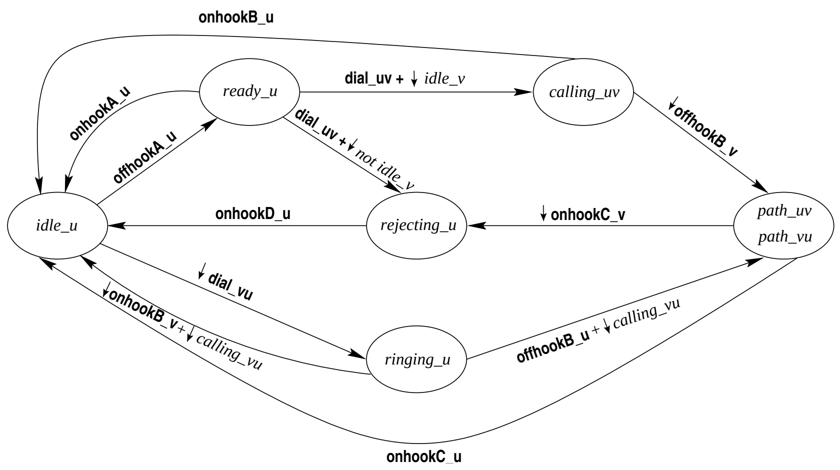

Figure 2: Transition system for the encoding on **BCS**

Definition 4.12 (Domain predicates of BCS). *The set of* domain predicates *(Definition 3.17) is responsible for defining the number of subscribers in the network (`subs`) as well as the number of cycles they are allowed to execute (`cycles`). We define the set $\mathbb{D}$ of domain predicates as follows:*

$$\mathbb{D} = \{subs, cycle\}$$

Definition 4.13 (Implementation atoms). *We include the set $\mathbb{I}$ of* implementation atoms.

$$\mathbb{E} = \{noidle, dialing, noidlebyrang, othernoidlebyrang, onhook\_caller, onhook\_party\}$$

These atoms represent additional information required in the implementation, see Figure 2.

Definition 4.14 (Atom names). *The set* ATOM *of atom names used in the* ***smodels*** *encoding of* ***BCS*** *is obtained by merging the set $\mathbb{L}$ of* labels*, the set $\mathbb{D}$ of* domain predicates *and the set $\mathbb{I}$ of implementation atoms.*

$$\mathsf{ATOM} = \mathbb{L} \cup \mathbb{D} \cup \mathbb{E}$$

To be able to carry the cycle information along the computation requires that every atom stores a *cycle counter*, which has to be updated every time a subscriber returns to her `idle` state. Additionally, the atoms have to be associated to some subscriber.

Definition 4.15 (Atoms of BCS). *The set of* atoms of the ***BCS*** *is the set of atom names* ATOM *where each atom is enhanced with information about the subscribers involved and their respective cycles. We define the following set of atoms:*

idle(ME,T)	*ready(ME,T)*	*calling(ME,T, SHE, U)*
rejecting(ME,T)	*ringing(ME,T)*	*path(ME,T, SHE,U)*
offhook(ME,T)	*onhook(ME,T)*	*dial(ME,T, SHE,U)*
subs(ME)	*cycle(T)*	*noidle(ME,T)*
dialing(ME,T)	*onhook_party(ME,T)*	*othernoidlebyrang(ME, SHE, T)*
noidlebyrang(ME,T, SHE,U)	*onhook_caller(ME,T)*	

*We use **ME** and **SHE** to represent variables over the set of subscribers and **T** and **U** are variables over the number of cycles.*

Now that we have provided these basic building blocks, we can focus on the encoding of rules. Note that, basically, there are two entities to be encoded: *transitions* and *states*. We make this distinction precise below.

4.4.4 Encoding of Rules

Figure 2 provides an intuitive understanding of how the state transitions are activated. Encoding such transitions in `smodels` is straightforward. We provide an example before we give the full explanation.

Example 4.16. *Suppose we want to encode the transition from the **idle** state to **ready**. We start by selecting the proper set of atoms from* ATOM *that matches the transition in $\mathbb{T}$, i.e., the atoms representing the states **idle** and **ready** together with the enabling action **offhook**. Now we simply rewrite the transition using the **smodels** syntax (Definition 3.5) and insert the predicate domains. The resulting rule has the following form:*

```
ready(ME,T) :- idle(ME,T), offhook(ME,T), subs(ME), cycle(T).
```

4.4.5 Encoding of States

To encode the state we focus on the *outgoing arrows*, i.e., the possible set of actions starting from the current state and leading to a new state. Figure 2 depicts a number of nodes where there is a choice of different *exclusive* actions. A standard mechanism to express these exclusive choices is the inclusion of the `false` atom in the case where both actions are satisfied by certain model. We exemplify this situation below.

Example 4.17. *Suppose that the subscriber **ME** is in the state **idle** in a certain cycle **U**. According to Figure 2, there are two possibilities: either **ME** goes offhook and reaches the state **ready** or **ME** has her number dialled by another subscriber, which moves **ME** to **ringing**. However, no valid stable model can contain both actions in the same cycle. The following line of **smodels** code expresses this choice:*

```
false :- offhookA(ME,T), ringing(ME,T), idle(ME,T),
         subs(ME), cycle(T).
```

*Thus, any model having both **offhookA(ME,T)** and **ringing(ME,T)**, will include the atom **false** in it. To invalidate this model we simply apply the **compute** statement (Definition 3.3.3) to ask for stable models in which there is no **false** as element.*

Now that we know how to encode rules and to express mutually exclusive actions we can encode of the whole **BCS**.

4.5 The `smodels` Code of BCS

We now describe the encoding of the **BCS**. We split the code into logical pieces: each bit encodes a relevant state of Figure 2. Because some are not automatically generated (e.g., constraints on `dial`), they were hard coded. Hence, the code below represents the code for the *four subscribers* and *three cycles* configuration. We comment on the constraints on the `dial` action separately.

Remark 4.18. *Remember that Figure 2 represents one dimension of the multidimensional.* ***`smodels`*** *capacity to to deal with variables and grounding suits well in this framework. We need only to specify the behaviour for one subscriber and then* quantify *over all subscriber in SUBS by means of domain predicates. Particularly, only half of the synchronised actions need to be specified because the behaviour of the party will be obtained from the general behaviour of the current subscriber when* ***SHE*** *is acting as called party.*

4.5.1 The BCS Code

Initialisation of the network. It is necessary to specify the allowed number of subscribers and cycles. Additionally, it is necessary to initialise the network. We encode this information below.

```
% Domain Predicates.
1.  subs(1).  subs(2).  subs(3).  subs(4).
2.  cycle(1).  cycle(2).  cycle(3).
% Network Initialisation.
3.  idle(1,1).  idle(2,1).  idle(3,1).  idle(4,1).
```

In line 1 we define the domain predicate `subs`, ranging over the set SUBS of subscribers. Line 2 defines the domain predicate `cycle`, which specifies the number of cycles that each of the subscribers is allowed to execute. Thus the setting described above specifies four subscribers with three possible cycles. Line 3 initialises the network with every subscriber in her `idle` state.

The `idle` state. The following box contains the code for the `idle` state.

```
% The IDLE State.
4.  false :- offhookA(ME,T), ringing(ME,T), idle(ME,T), subs(ME), cycle(T).
5.  offhookA(ME,T) :- not ringing(ME,T), idle(ME,T), subs(ME), cycle(T).
6.  ready(ME,T) :- offhookA(ME,T), idle(ME,T), subs(ME), cycle(T).
7.  noidle(ME,T) :- offhookA(ME,T), idle(ME,T), subs(ME), cycle(T).
```

Line 4 expresses a mutually exclusive choice: the subscriber `ME` either takes the `offhookA` action or she has her number dialed by some subscriber. Note that this choice determines whether the subscriber is a potential caller or a callee. The `dial` action of the party takes her to the `ringing` state. We encode the `ringing` behaviour further in line 16. In line 5 the subscriber takes the `offhookA` action. Lines 6 and 7 set the current state of `ME`, i.e., she is in the `ready` state and not idle, which is expressed by the atom `noidle`.

The ready state. We describe the `ready` state below.

```
% The READY State.
8.  false :- onhookA(ME,T), dial_somebody(ME,T),
          ready(ME,T), subs(ME), cycle(T).
9.  onhookA(ME,T) :- not dial_somebody(ME,T), ready(ME,T),
          subs(ME), cycle(T).
10.  idle(ME,plus(T,1)) :- onhookA(ME,T), ready(ME,T),
          subs(ME), cycle(T).
11.  dial_somebody(ME,T) :- not onhookA(ME,T), ready(ME,T),
          subs(ME), cycle(T).
12.  dialing(ME,T) :- dial_somebody(ME,T), ready(ME,T), subs(ME),
          cycle(T).
```

Line 8 presents a mutually exclusive choice: `ME` can either take the `onhookA` action or she can decide to dial some subscriber, i.e., take the `dial_somebody` action. Lines 9 and 10 stands for the case where the `onhookA` is taken. In Line 9 the `onhookA` is selected and in Line 10 the subscriber is taken back to `idle` state. According to the definition of cycle (Definition 4.8), the cycle counter is updated every time that the subscriber returns to the `idle` state. Line 10 takes care of this case by means of the `plus(.,.)` predicate. In Line 11 the `dial_somebody` action is taken. This action takes the subscriber to the `dialing` state, which is represented in Line 12.

The dial state. The `dial` action is one of the most important actions of the **BCS**. It depends on the state of the called party: if the callled party is `idle` it moves her to the `ringing` state while the caller is taken to the `calling` state. If the called party is not `idle` it moves the caller to the `rejecting` state. We encode this behaviour with the following code.

```
% The DIALING State.
13.  dial(ME,T,SHE,U) :- not notdial(ME,T,SHE,U), dialing(ME,T),
          subs(ME), subs(SHE), cycle(T), cycle(U).
14.  notdial(A,N1,B,N2) :- not dial(A,N1,B,N2), subs(A), subs(B),
          cycle(N1), cycle(N2).
15.  false:- dial_somebody(A,N1),
          notdial(A,N1,1,1), notdial(A,N1,1,2), notdial(A,N1,1,3),
          notdial(A,N1,2,1), notdial(A,N1,2,2), notdial(A,N1,2,3),
          notdial(A,N1,3,1), notdial(A,N1,3,2), notdial(A,N1,3,3),
          notdial(A,N1,4,1), notdial(A,N1,4,2), notdial(A,N1,4,3),
          subs(A), cycle(N1).
```

The `dial` action has to synchronize the caller and her party. To implement the synchronization we first generate all the possible `notdial` actions (Line 14). Line 15 represents a constraint which expresses that if a subscriber `ME` dials some subscriber, then it cannot be the case that she is not dialing any of the subscribers. Because of `smodels` syntax limitations we had to hardcode line 15.

Finally, Line 13 generates the possible parties to whom `ME` can possibly dial to. From a number of possible parties just one can be taken in each cycle. By applying a constraint on this action we get an unique `dial`. We comment on the constraints on the `dial` action in Subsection 4.5.2. Below we explore the different outcomes that a dial can have.

The effects of the* `dial` *action. A `dial` action from `ME` can have two possible outcomes: either the party `SHE` is idle or she is not idle. Additionally, there are different outcomes to both the caller and to the called party. The modules below cover this possibilities. We first encode the effect of a `dial` action when the party is idle.

```
% Effect of taking the dial on party (ONLY ACTION ON PARTY).
16.  ringing(SHE,U) :- dial(ME,T,SHE,U), dialing(ME,T),
          not noidle(SHE,U), noidlebyrang(ME,T,SHE,U),
          subs(ME), subs(SHE), cycle(T), cycle(U).
17.  noidlebyrang(ME,T,SHE,U) :- dial(ME,T,SHE,U), dialing(ME,T),
          not noidle(SHE,U), not othernoidlebyrang(ME,SHE,U),
          subs(ME), subs(SHE), cycle(T), cycle(U).
18.  othernoidlebyrang(ME,SHE,U) :- noidlebyrang(A,N,SHE,U),
          neq(ME,A),cycle(N), cycle(U), subs(A), subs(SHE), subs(ME).
```

This module takes the party to the `ringing` state. Note the connection with line 4 where `ringing` was one of the mutually exclusive options. Thus, we are encoding the scenario where `ME` is affected by a `dial` action, i.e., `ME` is the called party. However, we have to check some properties before taking `ME` to `ringing`. We first define the meaning of the atoms.

- `noidlebyrang(ME,T,SHE,U)`. We use the atom `noidlebyrang` to express the fact that the subscriber `SHE` (at cycle `U`) is not idle because the subscriber `ME` (at cycle `T`) has successfully dialled `SHE`'s number.
- `othernoidlebyrang(ME,SHE,U)`. The atom `othernoidlebyrang` means that the subscriber `SHE` (at cycle `U`) is not idle because she is already engaged with another subscriber `A`. Note that `A` is a subscriber different from `ME`. It is worth noting the case of construction `neq`. It represents the inequality symbol "$\neq$". Thus, `A` is not equal to `ME`.

Once the intuitive reading of the atoms is provided, the meaning of the rules gets clear. Line 18 will record the atom `othernoidlebyrang(ME,SHE,U)` in the model only if the party `SHE` is already engaged with some other subscriber different from `ME`. Thus, if `SHE` is not idle by any dial other than the one from `ME` (`not othernoidlebyrang(ME,SHE,U)`) and if `SHE` has not taken the `offhookA` action (`not noidle` atom, Line 7), then `SHE` is not idle by a dial of `ME`, Line 17. Finally, that party is taken to the `ringing` in Line 16.

Now we should determine the affect on the caller. On the one hand the caller `ME` can find her called party on idle. This case takes `ME` to the `calling` state. On the other hand, the called party can be no idle and then `ME` is taken to the `rejecting` state. The following module encodes these transitions.

```
% Effect of taking the dial action when party is idle.
19.  calling(ME,T,SHE,U) :- dial(ME,T,SHE,U), dialing(ME,T),
          subs(ME), subs(SHE), cycle(T), cycle(U).
% Effect of taking the dial action when party is not idle.
20.  rejecting(ME,T) :- dial(ME,T,SHE,U), dialing(ME,T),
          othernoidlebyrang(ME,SHE,U), subs(ME), subs(SHE),
          cycle(T), cycle(U).
21.  rejecting(ME,T) :- dial(ME,T,SHE,U), dialing(ME,T), noidle(SHE,U),
          subs(ME), subs(SHE), cycle(T), cycle(U).
```

Line 19 covers the case where the party is idle. The caller is then taken to the `calling` state (remember that the party is taken to the `ringing` state, Line 16). Lines 20 and 21 represent the fact that the party is not idle. In Line 20 the reason is that `SHE` has taken the `offhookA` action (atom `noidle`, Line 7). In Line 21 `SHE` is engaged in another call (atom `othernoidlebyrang`, Line 18).

The calling state. We encode the `calling` state below.

```
% The CALLING State.
22.  false :- onhookB(ME,T), offhookB(SHE,U), calling(ME,T,SHE,U),
            subs(ME), subs(SHE), cycle(T), cycle(U).
23.  onhookB(ME,T) :- not offhookB(SHE,U), calling(ME,T,SHE,U),
            subs(ME), subs(SHE), cycle(T), cycle(U).
24.  idle(ME,plus(T,1)) :- onhookB(ME,T), calling(ME,T,SHE,U),
            subs(ME), subs(SHE), cycle(T), cycle(U).
25.  path(ME,T,SHE,U) :- offhookB(SHE,U), calling(ME,T,SHE,U),
            subs(ME), subs(SHE), cycle(T), cycle(U).
```

Line 22 represents the mutually exclusive action: either the subscriber `ME` takes the `onhhokB` action or the subscriber `SHE` takes the `offhookB` action. In Line 23 the `onhookB` action os taken and Line 24 complements this action by taking `ME` back to the `idle` state. Note that similarly to Line 10, the cycle counter has to be updated. Line 25 establishes a speech path between `ME` and `SHE` provided the party `SHE` takes the `offhookB` action. Note that it is not required to encode the `offhookB` of the party. Remember that Remark 4.18 commented on the symmetry of some actions. Thus we encode the `onhookB` just in the party side, in line 27.

The ringing state. The encode the `ringing` state uses the following module.

```
% The RINGING State.
26.  false :- onhookB(SHE,U),offhookB(ME,T), ringing(ME,T),
            calling(SHE,U,ME,T), subs(ME), subs(SHE), cycle(T), cycle(U).
27.  offhookB(ME,T) :- not onhookB(SHE,U), ringing(ME,T),
            calling(SHE,U,ME,T), subs(ME), subs(SHE), cycle(T), cycle(U).
28.  path(ME,U,SHE,T) :- offhookB(ME,U), ringing(ME,U),
            calling(SHE,T,ME,U), subs(ME), subs(SHE), cycle(T), cycle(U).
29.  idle(ME,plus(U,1)) :- onhookB(SHE,T), ringing(ME,U),
            calling(SHE,T,ME,U), subs(ME), subs(SHE), cycle(T), cycle(U).
```

Line 26 encodes the mutually exclusive action: provided `ME` is ringing, there can be either an `offhookB` of the party `ME` or there is an `onhookB` of the caller `SHE`. In Line 27 the `offhookB` action is taken, which establishes a speech path between `ME` and `SHE` in Line 28. However, if the caller takes `onhookB`, then `SHE` is taken to the `idle` state (Line 29). Note that it is not required to encode the `onhookB` of the party (Remark 4.18). The `onhookB` is encoded in line 23, on the caller side.

The rejecting state. As we can see in Figure 2, the `rejecting` state consists in a single deterministic action. Thus its encoding is trivial.

```
% The REJECTING State.
30.  onhookD(ME,T):- rejecting(ME,T), subs(ME), cycle(T).
31.  idle(ME,plus(T,1)) :- onhookD(ME,T), rejecting(ME,T),
           subs(ME), cycle(T).
```

Line 30 stands for the situation where a subscriber `ME` takes the `onhookD` action. The effect of taking this action is encoded in Line 31, i.e., `ME` is taken to the `idle` state and her counter cycle is incremented.

The path state. Subsection 4.4.2 comments on the splitting of the `path` state. Below we provide the description of the `path` by separating its code in comprehesive blocks.

```
% The PATH State.
32.  false :- onhookC_party(ME,T), onhookC_caller(SHE,U),
           path(ME,T,SHE,U), calling(SHE,U,ME,T),
           subs(ME), subs(SHE), cycle(T), cycle(U).
```

Line 32 represents a mutually exclusive action: provided a calling from `SHE` to `ME` and a speech path between these two subscribers, there can be either an `onhookC_caller` or an `onhookC_party`. We analyse the outcome of each action below. We first describe the case where the `onhookC_party` is taken.

```
34.  onhookC_party(ME,T) :- not onhookC_caller(SHE,U), path(ME,T,SHE,U),
           calling(SHE,U,ME,T), subs(ME), subs(SHE), cycle(T), cycle(U).
35.  idle(ME,plus(T,1)) :- onhookC_caller(ME,T), path(ME,T,SHE,U),
           calling(ME,T,SHE,U), subs(ME), subs(SHE), cycle(T), cycle(U).
36.  rejecting(ME,T) :- onhookC_party(SHE,U), path(ME,T,SHE,U),
           calling(ME,T,SHE,U), subs(ME), subs(SHE),
           cycle(T), cycle(U).
```

Line 34 represents the case where the `onhookC_party` is taken. The effect of taking this is encoded in Line 35, i.e., the party (`ME`) is taken to the `idle` state. The same action `onhookC_party` takes the caller (`SHE`) to the `rejecting` state, where the Line 30 is activated. However, the mutually exclusive choice can result in `onhookC_caller`. We describe this case below.

```
37.  onhookC_caller(ME,T) :- not onhookC_party(SHE,U), path(ME,T,SHE,U),
           calling(ME,T,SHE,U), subs(ME), subs(SHE), cycle(T), cycle(U).
38.  idle(ME,plus(U,1)) :- onhookC_caller(ME,U), path(ME,U,SHE,T),
           calling(ME,U,SHE,T), subs(ME), subs(SHE), cycle(T), cycle(U).
39.  rejecting(ME,T) :- onhookC_caller(SHE,U), path(ME,T,SHE,U),
           calling(SHE,U,ME,T), subs(ME), subs(SHE), cycle(T), cycle(U).
```

Line 37 encodes the case where the action `onhookC_caller` is taken. The effect of taking this action on the caller is encoded in Line 37, i.e., the caller is taken to back to `idle` state. However, according to Line 39, the party is taken to the `rejecting` state, activating Line 30.

We now turn to explain the contraints on the `dial` action.

4.5.2 Constraints on the `dial` Action

Recall that the underlying idea of stable model semantics. We first generate models and then prune the unwanted models by means of constraints. The dial actions is an example of such method.

In the dial action we generate every possible party for a caller. However, a run (Definition 4.10) has to be cycle consistent, i.e., the dial and the cycles have to obey a progressive time line. To tame this action we have to demand a number of constraints, which are provided below.

Constraint 1.

```
% Constraint nr. 1.
% You cannot dial two people in the same cycle.
false :- dial(ME,T,SHE,U), dial(ME,T,HE,Z), neq(SHE,HE), neq(U,Z),
          subs(ME), subs(SHE), subs(HE), cycle(T), cycle(U), cycle(Z).
```

The constraint above includes the atom false whenever there are two dial actions from ME in the same cycle.

Constraint 2.

```
% Constraint nr. 2.
% You cannot dial yourself in different cycles.
false :- dial(ME,U,ME,T), neq(U,T), subs(ME), cycle(U), cycle(T).
```

This constraint aims at avoiding cases where ME intends to call herself in different cycles. Note that the subscriber ME can dial herself. But, because she is noidle, the resulting state has to be rejecting.

Constraint 3.

```
% Constraint nr. 3.
% You cannot dial in different cycles a person in her same cycle.
false :- dial(ME,U,SHE,T), dial(ME,Z,SHE,T), neq(U,Z)
          subs(ME), subs(SHE), cycle(U), cycle(T), cycle(Z).
```

The constraint above works in the following manner: if ME dials SHE, both are acting in the network. Therefore both will return to the idle state regardless of the outcome of the dial action. By definition, the cycle counter of both subscribers is updated. Therefore the case where ME dials SHE's number while SHE continues in the same cycle is inconsistent. The following contraint represent this information.

Constraint 4.

```
% Constraint nr. 4.
% If you dial the same number twice the cycles should
% have increased with yours.
false :- dial(ME,U,SHE,T), dial(ME,Z,SHE,W), lt(U,Z), gt(T,W)
          subs(ME), subs(SHE), cycle(U), cycle(T), cycle(Z), cycle(W).
```

The last constraint includes a false atom if ME has increased her cycle counter and SHE has not. Both SHE and ME have to increase. The smodels constructions "lt" and "gt" stand for the binary relations "$<$" and "$>$" respectively.

This sums up the implementation of **BCS** on smodels. The **BCS** encoding allows us to test several configurations and properties of the BCS. To test for specific properties seems to be the most straightforward method to check system correctedness. We address the testing of **BCS** in the next section.

4.6 Testing the BCS Implementation

We now turn to test the **BCS** implementation. There are two different ways of testing the system. On the one hand we can explore every branch of the search tree, i.e., to explore every valid network configuration. On the other hand, it is possible to ask for specific configurations on the network.

These tests are not only different in the method in which they traverse the search space. They also carry a distinct flavour. We start by exploring the whole search space and then move to check for specific properties of the model. The tests below were performed on a Sun ULTRA II (300MHz) with 1Gb of RAM, under Solaris 5.2.5.

4.6.1 Exhaustive Search

Here we focus on tests over the entire search space. Precisely, 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 behaviour of the problem. By varying the number of subscribers and cycles we obtain different results. Thus we can realise how the number of models (and the time for computing them) is affected by these changes on the network configuration. Additionally, we are able to test characteristics of the tool that we use for computing the models. We elaborate on this issues after showing the results.

The implementation of **BCS** can be easily queried for an exhaustive search. We use the `compute` statement with the following syntax: `compute 0 {not false}`. This statement queries for every valid (stable) model. The table below shows that results for exhaustive searches on different **BCS** configurations. Each cell represents the time required for the whole computation together with the number of valid stable models.

	1 Cycle	2 Cycles	3 Cycles
2 Subs.	0:00.18 / 15	0:00.42 / 375	0:08.08 / 11173
3 Subs.	0:00.22 / 136	0:45.36 / 82268	14:42:45.23 / 18262292
4 Subs.	0:02.38 / 1633	4:25:37.11 / 14774656	$>$ 100 Hours / $>$ 60 Million

It is clear that the exhaustive generation of the entire search space is extremely expensive. Indeed, the non-deterministic nature of finding the stable models already indicates that the growth occurs fast. The tests above confirm this characteristic.

Note how hard is to explore all the network configurations. E.g., the 4 subscribers / 3 cycles configuration was aborted after more than 100 hours of runtime. Furthermore the number of models was enormous. But note that `smodels` is able to compute it. Remember that one of the `smodels` properties (Subsection 3.3.2) is to run in linear space. Thus, provided enough memory to ground the program, `smodels` can compute every network configuration of a **BCS** setting.

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 property. We address this kind of testing below.

4.6.2 Checking Properties in the Model

We are now interested in exploring certain branches of the search tree, i.e., provided a property, we are looking for specific models where this property is valid. This kind of test has two objectives. On the one hand it aims at checking the consistency of the **BCS** implementation. On the other hand, it aims at showing that the `smodels` query based search is an efficient method for model checking.

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 some scenario. From this instantiation we check whether there are (stable) models where this property holds. Let's show these two possibilities by an example.

Example 4.19. *The table provides the property, the compute scheme that computes it and finally, the a specific scenario where the property holds.*

Property: *The* ***BCS*** *implementation must allow calls (provided* $|\mathsf{SUBS}| > 1$*).*	
Compute scheme	*Specific scenario*
`compute 1 {not false, calling(`$s1$`, `$t1$`, `$s2$`, `$t2$`)}`	`compute 1 {not false, calling(1,1,2,1)}`

Note that it is possible to obtain every network configuration satysfying a `compute` scheme. It suffices to generate all the n instances of the scheme and execute n successive runs of the program. However, we are interested in detecting a specific property and not exhausting every possible scenario where it happens. Remember that exhaustive searches can be very expensive.

Below we present a number of queries for basic properties of **BCS**. In each case we provide the aim of the query together the desired outcome, the `compute` scheme that checks the property of a specific configuration. The time shown corresponds to the verification of one instance of the scheme including the time required for grounding the program. To permit a comparison, the tests have been run on the 4 subscribers, 3 cycles setting.

Computing one model.

- *Description.* The `smodels` implementation of **BCS** has a model.
- *Expected outcome.* One valid model.
- **`Compute`** *scheme.*
  `compute 0 {not false}`
- *Elapsed time.* 0:19.60

The table below represent the time to provide a valid stable model in each setting of X subscribers, 3 cycles. Note the increase on time. However, `smodels` generated one model in each case.

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

Self dialing.

- *Description.* In no model a subscriber can dial her number and avoid the rejecting state.
- *Expected outcome.* No model, i.e., `False` answer.
- `Compute` *scheme.*
  `compute 1 {dial(`$s1$`,`t`,`$s1$`,`t`), not rejecting(`$s1$`,`t`)}`
- *Elapsed time.* 0:05.00

Void cycle.

- *Description.* A user has to act in the network.
- *Expected outcome.* No model.
- `Compute` *scheme.*
  `compute 1 {not false,not ringing(`$s1$`,`$t1$`),ready(`$s1$`,`$t1$`)}`
- *Elapsed time.* 0:12.24

Call chain.

- *Description.* The subscribers establish a chain of callings.
- *Expected outcome.* No model.
- `Compute` *scheme.*
  `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.* 0:09.36

Dial target.

- *Description.* If all the subscribers dial the same party and the party is idle, two callers must be rejecting.
- *Expected outcome.* One model where two callers are rejecting.
- `Compute` *scheme.*
  `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

Parallel calls.

- *Description.* Parallel calls among different subscribers are possible.
- *Expected outcome.* One model where there is a parallel call.
- `Compute` *scheme.*
  `compute 1 {not false,path(`$s1$`,`$t1$`,`$s2$`,`$t1$`),path(`$s3$`,`$t1$`,`$s4$`,`$t1$`)}`
- *Elapsed time.* 0:14.9

Multiple callings.

• *Description.* A user can establish three different calls in three cycles.
• *Expected outcome.* One model where a subscriber engages herself in three.
• `Compute` *scheme.*
`compute 1 {not false, path(s1,1,_,_), path(s1,2,_,_),path(s1,3,_,_)}`
• *Elapsed time.* 0:11.10

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 system's properties turns into a very efficient task as the examples above show. Additionally, the queries above reflect some of the **BCS** properties.

We have successfully encoded the **BCS**. Hence, the we can follow to the next step – the *implementation of features.* Note, however, that we have produced a graphical interpretation of the **BCS** (Section 4.3). In essence, this graphical view (Figure 2 shows the behaviour of the system. In addition, we could implement this behaviour with *minor* changes (Section 4.5). Hence, the implementation of features has to take advantage of this scenario, i.e., has to be realised as a change on the behaviour of the system.

In next section we show how to use the framework that was built to realise features and its implementation.

5 Implementation of Features

Because we are exploring a new approach to detection of feature interaction, we cannot use any of the previous methods as guideline. We can, however, devise a methodology that is general enough to both in the non-monotonic scenario and in general approaches to feature interaction.

Last section gives us the the graphical view of the system and its encoding. In other words, we have the behaviour of the system encoded in both abstractly and concretely. Hence, it is easy to realise an extension of the system (and it effects) by manipulating the graphical interpretation. Additionally, the concrete implementation enables us to actually encode these extensions.

Thus, in the method we devise below we take advantage of the abstract interpretation and the concrete one to approach the implementation of features and detection of feature interaction.

5.1 Feature Integration

The starting point on adding features is to specify their characteristics. This stage is called *feature description* (Definition 2.5). On the one hand a feature description has to be precise enough to cover every possible aspect concerning the implementation of the feature. An imprecise or incomplete feature description may generate unexpected results after the implementation. On the other hand, the feature description should be generic enough to abstract from implementation details, e.g., computer language, hardware architectures, etc.

Ideally, a feature description should contain the following information

- The feature behaviour;
- Affected parts of the basic system;
- Elements involved in the feature;
- The condition upon which the feature is activated.

Provided a feature description, we can turn to its implementation. Basically there are two approaches for feature implementation:

- *Built-in*, i.e., changing the basic system to cope with the new functionality;
- *Plug-in*, i.e., encoding features as independent modules of code.

The built-in approach is a naïve method for feature implementation. It leads to a number of drawbacks from which we emphasise the following:

- Hard maintainance;
- No (visible) distinction between basic system and features;
- Difficulty on extending the basic system and isolating implementation problems;

- The implementation of features by third-party developers is nearly impossible.

The plug-in approach defines a natural way of extending the basic system via features. To see the features as separate modules of code brings a number of advantages. We emphasise the following.

- Features can be easily activated and deactivated;
- The distinction between original behaviour and the behaviour introduced by the feature is clear;
- The code is modular and compact.

Additionally, the plug-in method solves the drawbacks presented by the built-in approach. Thus, it is easy to conclude that the plug-in method is the most appropriate way of adding feature on top of a basic system.

We aim at encoding features as independent modules which can be easily merged with the basic system. Note that the introduction of features changes the (expected) behaviour of the basic system. Thus, it is clear that the module where the obsolete behaviour was encoded is overridden when the feature is active. We present an example before going into details about feature implementation.

Example 5.1. *The example below outlines a methodology for feature integration. The method and the full implementation are formally defined later in this section. However it should be clear that every implementation must address this scheme, independently from the nature of the basic system or feature being implemented. We now proceed with the TCS (Terminating Call Screening) implementation.*

TCS. *This feature inhibits calls to the subscriber's phone from any number on the screening list she has chosen. The screened caller is rejecting when attempting to establish a call.*

We first isolate the behaviour of the feature. The feature description should be complete to provide every detail concerning the new functionality. For instance, in the presence TCS some dials are (unconditionally) rejected. More precisely, the dials from a screened caller are detoured to the **`rejecting`**.

Now we can isolate the elements which are involved in new functionality. Additionally, it is possible identify structures which are not native from the system. The TCS clearly deals with the **`dial`** *action. But note that the screened caller may also be* **`rejecting`**, *which adds the respective state in the list. Additionally, TCS requests a screening list, which is not native of* ***BCS***.

Finally it is possible to identify the part of the basic system where the new functionality acts in. Moreover, it is necessary to determine the consequences of adding the feature. Considering the TCS scenario, a screened caller can never call the party who screened her.

In the example above we split the feature description in three relevant parts: the *behaviour of the feature*, the *affected elements* of the system and the *consequences of adding the feature.* Provided these aspects we can safely design the new behaviour in the system being considered. We now formally define the methodology.

5.2 Approach for Feature Implementation

This subsection aims at defining our methodology for feature integration. The method is based on a search for specific information which, if not precisely defined, can lead to an incomplete implementation. Provided this information, we can safely turn to implementation details, i.e., details concerning basic system. The methodology splits the feature specification in three sets:

Design phase. The first step is to define the behaviour of the feature. Thus, this phase focus on *how* the feature works. From the behaviour of a feature we can extract a number of details. We emphasise the following.

- *Feature activation.* It is possible to define the conditions where the feature is invoked, i.e., *when* the new functionality is triggered. The system is not aware of the feature until its activation. Thus, it is important to define this condition.
- *Feature algorithm.* We have to specify the sequence of actions that is taken after the feature's activation, i.e., *what* to do when the feature is triggered. Note that here we are addressing the feature as an isolated functionality. Thus, we do not consider any aspect of the basic system at this stage.

This builds the first phase of our approach. Now we turn to identifying the tokens of the feature.

Token identification phase. The second step is to identify the tokens involved in the new functionality. We define this data according to the categorisation below.

- *Internal tokens.* These are the elements of the basic system which are involved in the feature, i.e., *who* is acting when the feature is activated. A graphical view of the basic system helps in capturing the proper set of tokens.
- *Support tokens.* These elements are not part of the basic system. However, its inclusion is required for the proper feature's operation. The support tokens can be retrieved either from the basic architecture (e.g., from the internal clock, in case of synchronisation) or from special purpose counters (e.g., number of trunks available in the network).

Merging phase. At this stage we comprehend the impact of the feature on the basic system. We split this phase in two parts.

- *Affected perimeter.* Identify *where* the feature works on.
- *Affected behaviour.* It is necessary to realise the basic system's behaviour which is overridden by the new functionality. From that we can derive the *consequences* and properties of the system when the feature is active.

Note the subtle difference between the *affected behaviour* and the *feature's algorithm*. When we address the feature's algorithm we abstract from details of the basic system. However, details about the basic system are highly relevant in defining the affected behaviour. Finally, the three phases provide enough information to implement the feature. We provide an example of such methodology below.

Example 5.2. *We split the TCS feature specification (Example 5.1) as follows.*

Designing phase.
- How. *If a screened caller attempts to dial, she unconditionally rejects.*
- When. *The feature is activated where is a dial from a screened caller.*
- What. *In the presence of a* screened dial*, the caller is rejecting.*

Token identification phase.
- Internal tokens. *The dial and the calling action and the rejecting state.*
- Support tokens. *A screening list.*

Merging phase.
- Where. *The feature works upon a dial action.*
- Consequences *A call from a screened caller is no longer possible.*

Now we turn to feature implementation. Note that throughout next section we follow this methodology.

5.3 Feature Implementation

We now explain how to implement features using the `smodels`. We first provide the intuitive process of implementing feature on the `smodels` implementation of the **BCS**. We then turn the implementation of features on top of **BCS**.

5.3.1 Implementing Feature on `smodels`

In this subsection we explain how to apply the methodology introduced in Subsection 5.2 to implement features on `smodels`, i.e., we encode and deploy features using the information that the methodology provides us.

The internal and support tokens are the building blocks of `smodels` rules. Therefore, we start by defining the atoms of the program. This information is obtained on the token identification phase. Particularly, there is one support token that must be provided. We call this token *activation token*. We make the idea precise.

Definition 5.3 (Activation token). *An* activation token *is a domain predicate used for activating the code of a feature.*

The next step is to encode the new functionality in terms of rules. Subsection 4.4.4 shows how to encode the transitions in `smodels`. On the one hand the feature's algorithm is provided in the designing phase. But note that these rules should be activated only when the feature is. We obtain this synchronisation by adding activation tokens in the body of the feature's rules.

Finally, we should prune the models where the old behaviour is developed. The information of which behaviour is no longer valid is provided by the merging phase. We override the old functionality by applying a constraint on the transition that is now forbidden.

Note that we take full advantage of the plug-in approach. Firstly, the new behaviour is implemented as a separate module. Secondly, to activate/deactivate the feature is straightforward. We activate the feature by adding its respective grounded predicates (activation tokens) to the code. To deactivate a feature is similarly easy. We delete the respective activation tokens. Lastly, the `smodels` tool box provides the non-monotonic framework to prune the old behaviour and implement the new functionality. We now turn to encoding the features.

5.3.2 Practical Implementation

This subsection aims at showing the implementation of feature on top of **BCS**. So far we introduced the methodology of implementing features. We now present the code of three features.

***Terminating Call Screening* – TCS.**

- *Feature description.* 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.
- *Affected atoms.* `dial`, `rejecting` and `calling`.
- *Activation predicate.* The atom `tcs(ME,SHE)`. Meaning: `ME` screened the subscriber `SHE`.
- *Affected action.* Provided the atom `tcs(ME,SHE)`, she cannot call `ME`.

Feature implementation.

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

Description of the implementation. In Line 1 we represent the change of behaviour. The rule stipulates that whenever `SHE` is dialling a party `ME` and `ME` has `SHE` in her screening list, `SHE` is taken to the `rejecting` state. Line 2 is prunes the (now invalid) models, i.e., models where the calling action from `SHE` to `ME` is considered. Note that both rules 1 and 2 are only activated when there is at least one screening list (the `tcs` atom). This allows us to leave both basic system code and features code together without any interference.

***Originating Call Screening* – OCS.**

- *Feature description.* The OCS feature inhibits call from `ME`'s phone to any number `SHE` from a set chosen by `ME`. Any attempt of `ME` to ring such a number takes her to the `rejecting` state.
- *Affected atoms.* The `dial` action and the states `rejecting` and `calling`.

• *Activation predicate.* The atom `ocs(ME,SHE)` which means that `ME` screened the subscriber `SHE`.

• *Affected action.* Provided the atom `ocs(ME,SHE)`, `SHE` cannot be called by `ME`.

Feature implementation.

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

Description of the implementation. Line 1 expresses the change of behaviour. Now, whenever `ME` is dialling a party `SHE` and `ME` has `SHE` in her screening list, `ME` is taken to the `rejecting` state. Line 2 prunes the models where the calling action from `ME` to `SHE` is considered valid. Note that this feature is similar to TCS. We simply exchange the parameters of the `dial` action.

***Call Forward Unconditional* – CFU.**

• *Feature description.* In the CFU feature every call addressed to a subscriber `ME` is unconditionally forwarded to the subscriber `ALTER`.

• *Affected atoms.* The `dial` action.

• *Activation predicate.* The atom `cfu(ME,ALTER)`, standing for the case where `ME` forward her calls to `ALTER`.

• *Affected action.* In presence of `cfu(ME,ALTER)`, `ME` can never be called.

Feature implementation.

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), 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,2,1),
          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), cycle(U), cycle(T).
```

Description of the implementation. 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`.

5.4 Checking Properties of the BCS + Features

The implementation of features on top of **BCS** changes the original characteristics of the basic system. This subsection explores the most significant properties by means of tests on the **BCS** implementation.

As we did in Section 4, we use different queries to ask for specific network configurations. However, the properties we now check are inferred by the features.

TCS and Call Blocking. If the subscribers are pairwise screened, there can be no calling. 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 0:5.1

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

OCS and Call Blocking. We can use a similar test to the case of the OCS feature. Thus, if the OCS feature is pairwise activated to every subscriber, there can be no calls. It is not s surprise to detect that the query requires the same amount of time. The following table provides time and query information.

`ocs(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

The query returns no model, as we expected.

CFU and Call Looping. If two subscribers forward their phones to each other, no dial to either of them can be performed.

`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 0:5.2

The query produces no dial, as we expected.

We may also ask for interactions by activating more features.

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.

The list of possible test could go further. However we take stock and evaluate what has been achieved.

As we have seen in the examples above, 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" [32] 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 behaviour of a feature is a natural way of viewing the process of feature integration. The methodology that we developed in this section 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 [22]. Already in the feature implementation, we can detect which part of the system is being changed and encode just that specific module.

In the next section we provide some conclusions of the overall work and fix these ideas in a more precise setting.

6 Conclusion

In this thesis we have considered the problem of detecting feature interaction in telephony systems. Our method took into account two aspects: the nonmonotonic behaviour of integrating features on top of a basic system and the difficulty of detecting feature interactions.

Starting from a description logic specification of the **BCS**, we drew a labelled transition system representing the **BCS**. 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 properties of the implementation.

We devised a stepwise methodology to approach feature integration. Additionally, we described a method to encode features as changes on the system's behaviour. Finally, we integrated features on top of the **BCS** implementation. We first checked the consistency of each feature individually. We then turned to verify feature interaction by integrating multiple features on top of the system.

It is possible to claim that our approach is a novel approach to detection of feature interaction. On the one hand there are works where `smodels` is used as a model checking tool [19]. On the other hand, the model checking approach is oftenly used on detection of feature interaction. However, there was no approach addressing both detection of feature interaction and stable models semantics as a model checking tool. Additionally, our approach focused on the nonmonotonic behaviour of features.

As we saw in Subsection 4.6, in `smodels` it is necessary to specify a query and then check whether it holds. In terms of detection of feature interaction, it means that we have to detect a "harmful" [32] combination of features and query for its existence. This contrasts with the model checking approach. In a model checking framework, we provide the formal specification to a model checker, which 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 interaction.

Exploiting the nonmonotonic behaviour of a feature is a natural way of viewing the process of feature integration. The methodology that we developed in Subection 5.1 relies deeply on this idea. The advantages of this approach are clear. On the design phase we were able to understand the feature without taking into account the basic system. This complies with the generality of feature descriptions, advocated by [22]. Already in the feature implementation, we could detect which piece of the system was changed and encode just that specific module. And finally, we could encode all these characteristics in `smodels`.

The tool we used, the `smodels` tool box, has some drawbacks. `smodels` offer nearly no support for developers. For instance, it is not possible to declare which are the atoms in the logic program. Thus, any mistype is interpreted as a new atom. Additionally, some syntactic constructions do not work properly. As we can see in the encoding of the `dial` action (Subsection 4.5), this forced us to hardcode some lines of code. However, some other constructions are very handy, as the `hide` statement, that allowed us to hide some atoms from the output.

We close by discussing some topic for further research. We first address some specific improvements to our implementation.

- Recall that in Subsection 4.4.1 we defined the concept of a cycle to ensure termination. This introduced the need for temporal consistency to the model, which were modelled by means of constraints over the set of stable models. A drawback of our modelling is that temporal constraints have to be reimplemented each time that the number of cycles were extended. Thus, one valuable contribution would be to automate this process.

- The tests on Subsection 4.6 showed not only efficiency in querying for specific properties but also that `smodels` can cope with hard instances of our implementation. Provided the automated extension of the code, it is possible to explore different settings.

- To encode the dial action (Subsection 4.5.2), we applied several constraints on it. This tight implementation affected the feature integration. Specifically, when we implement the CFU feature in Subsection 5.3.2, these constraints worked against us and forced a non-modular implementation. To redesign the `dial` action using a more flexible approach is required for further feature implementation and suggested as further work.

We now emphasise some further research on the general framework.

- Studies on complexity of the model are also possible. In particular, on which is the relation between the number of subscribers and network configurations they generate.

- Another idea is to migrate to other implementations of stable model semantics. Even though we opt for `smodels`, there is a number of alternative such as **XSB** and DeReS. Encoding the **BCS** in those systems permits the comparison between these different implementations.

- As we mention above, our approach is very close to model checking. Thus it is possible to compare both approaches, e.g., in terms of performance.

- Finally, our approach also revealed certain shortcomings on `smodels`, which could be improved. In particular, the poor support for system development. For instance, a construction allowing the declaration of atoms would improve the implementation considerably.

References

[1] K. Apt. Programming with constraint programming. Lecture Notes, 1999. University of Amsterdam.

[2] C. Areces, W. Bouma, and M. de Rijke. Description logics and feature interaction. In *Proceedings of the 1999 International Workshop on Description Logics (DL'98)*, Linköping, Sweden, July 1999.

[3] Bellcore. LATA switching systems generic requirements (LSSGR). Technical Report TR-TSY-000064, Bellcore, Piscataway, N.J., 1992.

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

[5] 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.

[6] M. Buchheit, F.M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993/94.

[7] W. Chen and D.S. Warren. The SLG system. Available for download at `seas.smu.edu/pub`, 1993.

[8] P. Cholewińsky, V.W. 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.

[9] W.F. Dowling and J.H. Gallier. Linear time algorithms for testing the satisfiability of propositional Horn-formulae. *Journal of Symbolic Logic*, 3:267–284, 1984.

[10] A. Gammelgaard and J.E. Kristensen. Interaction detection, a logical approach. In L. Bouma, editor, *Proceedings of Feature Interactions in Telecommunication Systems*, pages 178–196, 1994.

[11] M. Gelfond and V. Lifschitz. Stable model semantics for logic programming. In *Proceedings of the 5^{th} International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[12] K. Heljanko. Deadlock checking for complete finite prefixes using logic programs with stable model semantics. In *Proceedings of the Workshop Concurrency, Specification & Programming 1998 (CS&P' 98)*, pages 106–115, Berlin, September 1998. Humboldt-University.

[13] J. Herbrand. *Recherches sur la théorie de la démonstrations.* PhD thesis, Paris, 1930.

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

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

[16] V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, September 1998.

[17] I. Niemelä. Towards efficient default reasoning. In *Proceedings of the 14^{th} International Joint Conference on Artificial Inteligence*, pages 312–318, Montreal, Canada, August 1995. Morgan Kaufmann Publishers.

[18] I. Niemelä. Formalizing configuration knowledge using rules with choices. Technical Report TKO-B142, Laboratory of Information Processing Science, Helsinki University of Technology, 1998.

[19] 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.

[20] I. Niemelä and P. Simons. Efficient implementation of well-founded and stable model semantics. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, September 1996.

[21] 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.

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

[23] M.C. Plath and M.D. 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.

[24] 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.

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

[26] P. Simons. *Towards constraint satisfaction through logic programs and the stable model semantics.* PhD thesis, Helsinki University of Technology, Helsinki, Finland, August 1997.

[27] P. Simons and I. Niemelä. Smodels – an implementation of the stable model and well-founded semantics for logic programs. In *Proceedings of 4^{th} International Conference on Logic, Programming and Nonmonotonic Reasoning*, volume 1265, pages 420–429, Dagstuhl, July 1997. Lecture Notes in Artificial Intelligence.

[28] R. M. Smullyan. *First-order logic.* Springer Verlag, Berlin, 1968.

[29] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319, San Antonio, Texas, May 1999. Springer-Verlag.

[30] T. Syrjänen. Implementation of local grounding for logic programs with stable models semantics. Technical Report B18, Digital Systems Laboratory Helsinki University of Technology, October 1998.

[31] 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.

[32] 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.

[33] P. Zave. FAQ sheet on feature interaction, 1999. Manuscript.