# Logic and Abstraction, Verification and Falsification

Kumulative Habilitationsschrift

vorgelegt zur Erlangung der

Lehrbefugnis für Informatik

an der

Fakultät für angewandte Wissenschaften

der

Albert-Ludwigs-Universität Freiburg i. Br.

von

**Jan-Georg Smaus**

Freiburg, November 2008

Für Corinna

# Summary for Cumulative Habilitation

**Abstract**

The works collected in this habilitation are concerned with the use of logic and abstraction techniques for the purpose of verifying and falsifying transition systems. The works have been structured into two main themes.

For the first theme, the transition systems in question are programs, in particular *logic* programs. Various aspects of correctness of such programs are considered. One aspect is *termination*; several contributions concerning methods for proving termination are contained in this habilitation. Another aspect is *type safety*, where this habilitation contains results extending the type systems for which type safety can be guaranteed.

For the second theme, the transition systems in question are timed and hybrid systems. Heuristic methods for detecting error paths in such systems are presented, which is important for supporting the design process of the systems. Another contribution is a method for representing propositional formulas compactly as a set of *linear pseudo-Boolean constraints*, which is useful, among other reasons, because propositional logic plays a prominent role in the analysis of transition systems, namely in the field of *bounded model checking*.

## Contents of the Summary

# 1 Introduction

## 1.1 Transition Systems and Logic

In one way or the other, all my work is concerned with the correctness of *transition systems*. Such a system has a set of states and a transition relation between these states, both defined in some suitable mathematical formalism. We thus have a graph whose nodes are states and whose edges represent the transition relation, and the graph represents the possible *behaviours* of the system.

The transition systems studied in my work are logic and functional programs on the one hand, and timed and hybrid automata on the other hand. This can be seen as a rather arbitrary choice of a number of formalisms out of a spectrum of formalisms for defining transition systems. However, in the study of correctness of transition systems, there are commonalities across many different formalisms, both concerning the aims and the techniques used. It is common for new techniques developed in an exemplary way for one formalism to be later adopted to other formalisms.

*Transition system* is thus the first important keyword of my work; *logic* is the second. On the one hand, logic formulae can be given an operational interpretation, i. e., be used as programs, which is what *logic programming* is about. As any program is a transition system, we can say that in this case a logic formula defines a transition system. On the other hand, logic allows one to describe the behaviour of any transition system in an *abstract* or *concise* or *summary* manner, and in particular, to *specify* the desired behaviour. Examples of such desired behaviour are:

- *termination*: every computation is finite;
- *safety*: no error state is ever reached.

Correctness then means that the specification and the behaviour of the system agree.

I have chosen these examples of desired behaviour because they are the most relevant in my work. Analysis of program termination is one of my main topics. Concerning safety analysis, I have mainly studied *falsification*, i. e., error detection and diagnosis.

The "study" of transition systems and logic conducted in my work has aspects of *synthesis* and *analysis*. On the one hand, we have a logic formula (program) and want to synthesise an operational behaviour (a program execution) from it. More specifically, a concern particularly for logic programs is to execute them in such a way that they terminate. On the other hand, we have a system (automaton etc.) with a given behaviour, which we analyse using logic. More specifically, we want to detect the errors of such a system.

## 1.2 The Themes of this Habilitation

I have just introduced the research area of this habilitation in broad terms. I will now go to a more concrete level, just specific enough to explain what the main contributions are, but without going into much detail. The works included in this habilitation have been structured into several themes and subthemes, and I will present the contributions at the subtheme level. In Section 1.3 I shall return to a high-level view, while in Sections 2 and 3, I go into depth and discuss the contributions of the single articles included.

### 1.2.1 Logic Programming and Functional Programming

**Termination and Selection Rules** Logic programming has been praised for being *declarative*: the programmer should state *what* the problem to be solved is, rather than *how* it is solved [145]. More specifically, a logic program is a formula from a certain fragment of first-order logic stating the relationships between the objects of interest.

The *declarative* semantics of a logic program is usually defined as the set of all the atomic consequences of the program — the basic relationships implied by the program. However, logic programs also have an *operational* semantics: the execution of a program is a sequence of logical inference steps, which computes answers to so-called *queries*. Referring to Section 1.1, we might say that an operational behaviour is *synthesised* from the logic program. There are theoretical results [159] stating the equivalence between these semantics: the atomic consequences of the program are exactly the computed answers to atomic queries.

However, in practice, without any concern for how exactly the computation of a logic program is performed, there are considerable problems about the efficiency and termination of the computation. Therefore, logic programs often make use of certain language constructs that go beyond the "pure paradigm" and destroy the correspondence between the declarative and the operational semantics, for the sake of efficiency and termination. One such construct is the well-known *cut* [210], which removes some of the logically possible inference sequences.

Put simply, an important aim in the research on logic programming is to close the gap between the declarative and the operational semantics, between theory and practice so to speak.

One aspect of the operational semantics of logic programs is the so-called *selection rule*. It determines the order in which the logical inferences are performed, whenever several inferences are logically possible. There are some specific non-trivial termination problems related to the selection rule [29, 161, 163, 164, 167, 180]. My contributions in this area can be summarised as follows:

- I define on a theoretical level what a selection rule must look like in order to allow for a program to be used "in many ways" (what this means precisely will be explained in Section 2.1.2 on page 11), and explain how such a selection rule can be implemented in existing programming languages.

- I provide techniques for proving formally that programs supplied with such a selection rule terminate.

- The selection rule is implemented in such a way that the declarative semantics of a program is preserved. Thus, my work is a contribution to closing the "gap between theory and practice" mentioned above.

**Typed Programs** This subtheme is mainly concerned with logic programming but also touches some aspects of functional programming.

*Types* serve the correctness of programs. They restrict the syntax of programs so that meaningless operations, such as computing the cosine of a string, are disallowed. This allows for many programming mistakes to be detected by the compiler.

The type systems I consider here are based on the typed $\lambda$-calculus [216]. They feature the so-called *parametric polymorphism*. Examples of (functional) programming languages using such type systems are ML [183] or Haskell [218]. In the design of type systems for programming languages, the conflicting aims are:

- The type system should be flexible and expressive. One desirable feature, for instance, would be *subtyping*.

- The type system should give strong correctness guarantees and be easy to handle from the point of view of the compiler. In particular, the type system should allow for automatic type inference, so that the user does not have to write type declarations. Moreover, the type system should ensure static typing, that is to say, once the program has passed the compiler as being correctly typed, it should be guaranteed that there cannot be any type errors at runtime.

Although type systems with parametric polymorphism are widely established in functional programming languages, there are some aspects and features, related to the conflict just explained, that have sparked a considerable amount of research [49, 95, 126, 129, 137, 141, 142, 152, 160, 177, 179, 182].

My contributions in this area are rather diverse and difficult to pinpoint without becoming too technical, but they address the conflicting aims of type system design. I have several results stating under which conditions, or more precisely, for which language features, automatic type inference and static typing can be achieved, thereby extending the applications of types in logic and functional programming.

**"Formulas as Programs"** Krzysztof Apt has proposed a new approach to using first order logic as a programming language [24]. Put very abstractly, the approach takes a logic formula and synthesises a computation which is independent of the interpretation of the logic, that is to say, the computation refrains from relying on (e. g., arithmetic) symbols such as "7" or "+" to have their usual meanings. My contribution in this area addresses the question of whether this can be done in an optimal way, i. e., whether there is a theoretical limit to the amount of information that can be extracted from a formula without making any specific assumptions about the interpretation of the logic. The answer I give is negative, i. e., there is no such theoretical limit.

### 1.2.2 Logic Approaches to Model Checking of Timed and Hybrid Systems

**Finding Error Paths in Timed and Hybrid Systems** This subtheme is related to safety-critical systems such as trams, airplanes, chemical plants etc. Such systems usually feature an electronic, discrete control component and a "physical" component. It is desirable to design such systems so that they are safe and to prove the safety rigorously, i. e., to *verify* a system. Various mathematical modellings of such systems have been proposed, for example *communicating sequential processes* [192], *timed automata* [21], and *hybrid automata* [130].

Verification of systems is often done using *model checking*[1] [74]. This notion summarises certain techniques that more or less explicitly enumerate the state space of a system and check whether all transition sequences of the system satisfy a desired property, i. e., whether the graph representing the state space *is a model of* the property to be checked [120]. The property is often specified using some *temporal logic* [74].

However, my research is more focused on *falsification.* Falsification is of interest because systems are mostly not correct during their design process, and so it is important to diagnose errors. More specifically, we want to identify error trajectories of the system, i. e., executions of the system that lead from an initial to some unsafe state. I have contributed methods for doing this, both for timed and for hybrid automata. Although the methods are technically quite different, they share one remarkable aspect, namely that they aim at integrating verification and falsification methods. More specifically:

- I have developed a method for finding error paths in hybrid automata and integrated this method into a verification tool for hybrid automata. The tool is based on abstractions, and the remarkable aspect of this work is that the same abstractions useful for proving the absence of an error can be used for finding an error.

- I have developed heuristics for error search in timed automata, in the spirit of *directed model checking* [101, 102]. This allows one to find errors in timed automata where blind search is hopeless. Apart from being competitive with other heuristics proposed for this purpose, the heuristics are interesting and novel because they use

[1] I would like to draw the reader's attention to the index at the end of this summary, which points to the definitions or informal explanations of the most important concepts used in this habilitation.

a logic-based technique called *predicate abstraction*, which has previously been used for verification.

**Linear Pseudo-Boolean Constraints** This subtheme is related to propositional logic and is motivated by the use of propositional logic in *symbolic* or *bounded model checking* [55]. The approach of bounded model checking is to encode a model checking problem as a (propositional or other) logic formula. To solve the problem, propositional satisfiability (SAT) solving is then important [165, 173, 226].

Linear pseudo-Boolean constraints are a representation of propositional formulae that is often very compact. While this has inspired some authors to adapt SAT solvers to linear pseudo-Boolean constraints [18, 71, 96, 110, 111, 138], the full potential of this representation has not been exploited yet because of a lack of (practical) methods for encoding arbitrary propositional formulas as linear pseudo-Boolean constraints.

My main contributions in this area consist of two results, one "negative" and one "positive":

- In some cases, there is nothing to gain in terms of compactness using linear pseudo-Boolean constraints.
- There is a combinatorial algorithm for converting a formula into a single linear pseudo-Boolean constraint, if this is possible. This has been an open problem for 30 years [83].

## 1.3 Common Aims and Techniques

After having introduced the main contributions of this habilitation at the subtheme level, I now mention several common aims and techniques in my work. Throughout the rest of this summary, I will frequently refer back to this list to substantiate these points.

1. The main common aim of my work is verification or falsification (as applies) of transition systems (see Sections 2 and 3.1).
2. An operational behaviour is *synthesised* from logic formulae (see Sections 2.1, 2.3, and 3.1).
3. A "real" system (e. g., an automaton) has an operational behaviour by definition. I specify and *analyse* the desired behaviour using logic (see Section 3.1).
4. On the technical side, *abstraction* techniques are often used for verification or falsification. An abstraction of a system, overapproximation to be precise, is a simplification allowing for more behaviours. Thus, if the abstraction is free from errors, then the original system is a-fortiori free from errors (see Sections 2.2 and 3.1).
5. Statical correctness conditions such as types or modes are used for verification. By "statical" I mean that these conditions are imposed on the single (and finitely many) components defining a transition system, and that the conditions ensure certain desirable correctness properties as invariants of any execution of the system (see Sections 2.1 and 2.2).
6. In some of my works a *predicative* view on transition systems plays a role, i. e., the behaviour of a system is encoded as logical formula (see Sections 2.2 and 3).

### 1.4 Structure of this Summary

We will now proceed with Sections 2 and 3, corresponding to the two main themes of this habilitation work. Likewise, each subsection corresponds to a subtheme. In each subsection, I first very briefly explain the main, distinctive contribution of each included article. Afterwards, I explain the most important concepts used in each subtheme and illustrate the main problems using examples.

Of course, I am not the first to recognise that there are various connections between the themes considered here, most generally speaking, between model checking and logic programming. In Section 4 we will discuss the related literature, in particular the more recent works.

## 2 Logic Programming and Functional Programming

### 2.1 Termination and Selection Rules

#### 2.1.1 Contributions on this Subtheme

In logic programming, *dynamic scheduling* refers to a situation where the selection of the atom in each resolution (computation) step is determined at runtime, as opposed to a fixed selection rule such as the left-to-right one of Prolog. This has applications e. g. in parallel programming. A mechanism to control dynamic scheduling is provided in existing languages in the form of *delay declarations.* From the point of view of declarative, model-theoretic semantics [31, 159], programs using dynamic scheduling are more difficult to reason about than programs using left-to-right execution. In particular, giving a characterisation of terminating logic programs is more difficult in the presence of dynamic scheduling.

In the work I did together with Bossi, Etalle and Rossi [1], we made several contributions to this area. In order to provide a characterisation of dynamic scheduling that is reasonably abstract and hence amenable to semantic analysis, we consider *input-consuming derivations.* In an input-consuming derivation, only atoms whose input arguments do not get instantiated through the unification step may be selected. We demonstrate that under some statically verifiable conditions, input-consuming derivations are exactly the ones satisfying the (natural) delay declarations of programs. We then define a model-theoretic semantics for input-consuming programs. Based on this semantics, we present a result which fully characterises termination of input-consuming programs.

The subsequent journal version [2] contains all the proofs, more detailed explanations, and a more extensive comparison of related work. However, the journal version does not formally analyse the relationship between input-consuming derivations and delay declarations, and it omits a certain variant of our semantics which was not directly relevant for our termination results.

Later, I investigated input-consuming derivations with additional assumptions on the selection rule [3]. Certain additional assumptions can be formalised as a property that the selected atoms must have, e. g., the property of being ground. This can be done generically by parametrising the method of proving termination with the set $\mathcal{P}$ of atoms that may be selected. In analogy to the work above [1, 2], I provide a full characterisation of termination for input-consuming derivations where the selected atoms must have property $\mathcal{P}$. In addition, I give a sufficient criterion for termination for *local* selection rules. Local selection rules specify that an atom $A$ must be resolved away completely before any of $A$'s siblings can be selected.[2]

[2]Moreover, there is a technical report version [202] which extends the sufficient criterion for termination for local selection rules to a full characterisation, i. e., a criterion that is sufficient and *necessary.*

```
permute([],[]).
permute([U|X],Y) :-
  permute(X,Z),
  insert(Z,U,Y).
insert(Z,X,[X|Z]).
insert([U|Z],X,[U|Y]) :-
  insert(Z,X,Y).

        PERMUTE
```

```
permute([],[]).
permute([U|X],Y) :-
  insert(Z,U,Y),
  permute(X,Z).
insert(Z,X,[X|Z]).
insert([U|Z],X,[U|Y]) :-
  insert(Z,X,Y).

        PERMUTE2
```

```
append([],Y,Y).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).

        APPEND
```

Figure 1: Example programs

Together with Dino Pedreschi and Salvatore Ruggieri, I have written a survey on termination of logic programs depending on the selection rule [4]. Generally, one can say that the stronger assumptions one makes about the selection rule, the bigger is the class of programs that will terminate under those assumptions (i. e., terminate for *all* selection rules meeting those assumptions). We study six classes, ranging from programs that terminate for *all* selection rules to programs that terminate only in the weak sense that there are finitely many computed answers. The contribution of this work lies in the unification of different formalisms, allowing us to establish a formal hierarchy between the classes. This work was later invited to become part of a special volume [5], where we extend the hierarchy by the class of programs studied in my own previous work [3], and where we extend the discussion of related work.

### 2.1.2 More Details on the Concepts and Problems of this Subtheme

*Termination and Selection Rules* has been a theme of my research both before [199, 204, 206, 208] and after [1, 2, 3, 4, 5] my PhD. The main aim of this subsection is to provide the reader with some background on the theme, and thus I will often refer to the work of my PhD. However, I will also point out the contributions of this habilitation.

**Selection Rules, Modes, and Termination** The paradigm of logic programming is based on giving a computational interpretation to a certain fragment of first order logic. Kowalski [145] advocates the separation of the *logic* and *control* aspects of a logic program and has coined the famous formula

Algorithm = Logic + Control.

One aspect of control in logic programs is the *selection rule.* This is a rule stating which atom in a query is selected in each derivation step. Finding the right selection rule is the most important aspect of synthesising an operational behaviour from a logic program (see point 2 in Section 1.3).

The standard selection rule is the *LD* selection rule: in each derivation step, the leftmost atom in a query is selected for resolution.

**Example 2.1** Consider the `PERMUTE` program in Figure 1 and the following derivation, where the selected atom is underlined in each query:

$$\begin{array}{l} \underline{\mathtt{permute([1],As)}} \quad \Longrightarrow \\ \underline{\mathtt{permute([],Z')}},\ \mathtt{insert(Z',1,As)} \overset{\mathtt{Z'/[]}}{\Longrightarrow} \\ \underline{\mathtt{insert([],1,As)}} \quad \overset{\mathtt{As/[1]}}{\Longrightarrow} \quad \Box \end{array} \qquad (1)$$

To explain this derivation, it is helpful to use the concept of *modes*: In the second line, $\mathtt{Z'}$ is an *output* argument of $\mathtt{permute([],Z')}$. The process of resolving this atom instantiates $\mathtt{Z'}$ to `[]`, which is used by the atom $\mathtt{insert(Z',1,As)}$ as *input.*

For the moment, we just assume that each argument position of each predicate is assigned a mode (input or output) according to the intentions of the programmer. We denote a mode by writing, e. g., $\texttt{permute}(I, O)$, $\texttt{insert}(I, I, O)$.

Intuitively, it should be clear that each piece of data must be produced before it can be consumed. The LD selection rule implies that "before" means "to the left of". Thus the convention is to write logic programs in such a way that output (i. e., producing) occurrences of each variable are to the left of input (i. e., consuming) occurrences.

A different way of saying that producing should come before consuming is to say: an atom should only be selected when it has a sufficient degree of instantiation. The following example shows that this is crucial for termination.

**Example 2.2** Consider the following derivation for `PERMUTE`, where the *right*most atom is always selected:

$$\begin{array}{l}
\underline{\texttt{permute([1],As)}} \quad \Longrightarrow \\
\texttt{permute([],Z}'\texttt{)},\ \underline{\texttt{insert(Z}',\texttt{1,As)}} \overset{\texttt{Z}'\texttt{/[U}''\texttt{|Z}''\texttt{]}}{\Longrightarrow} \\
\texttt{permute([],[U}''\texttt{|Z}''\texttt{])},\ \underline{\texttt{insert(Z}'',\texttt{1,Y}''\texttt{)}} \overset{\texttt{Z}''\texttt{/[U}'''\texttt{|Z}'''\texttt{]}}{\Longrightarrow} \quad \ldots
\end{array} \tag{2}$$

The derivation is infinite although there exists only one computed answer to the query, namely `As/[1]`.

Using the notion of modes, it is possible to state precisely a certain minimal requirement for a "sufficient" degree of instantiation: In each derivation step, the input arguments of the selected atom cannot become instantiated. In other words, an atom in a query can only be selected when it is sufficiently instantiated so that the most general unifier with the clause head does not bind the input arguments of the atom. We call derivations which meet this requirement *input-consuming*, a notion I developed during my PhD work [198, 199, 200].

In most cases, the LD selection rule is adequate and in particular, ensures input-consuming derivations. However, there are at least four purposes for which other selection rules are useful: using predicates in multiple modes [204], parallel execution [29], the test-and-generate paradigm [180], and some programs using accumulators [105]. These purposes, but most importantly the first one, is what I meant by "using a program in many ways" (see page 6).

To define selection rules that are more flexible than just stating that the leftmost atom should be selected in each step, several logic programming languages provide *delay declarations* [136, 214, 209]. Using delay declarations, the user can specify a degree to which an atom must be instantiated in order to be selected.

In the literature, the need for sufficient instantiation of the selected atom and hence the purpose of delay declarations is usually explained as "ensuring termination" and "preventing runtime errors related to built-in predicates" [29, 161, 163, 164, 167, 180], both of which are aspects of verification of logic programs. Using the concepts introduced above, we can be more to the point: The minimal and most important purpose of delay declarations is to ensure input-consuming derivations in situations where the LD selection rule is not adequate.

In a work included in this habilitation [1], I show that under some statically verifiable conditions, input-consuming derivations are exactly the ones satisfying the (natural) delay declarations of programs.

**Modedness** For verification of logic programs, modes are useful. However, it is not sufficient just to assign a mode to each argument position in an arbitrary way. Rather, the useful results usually rest on certain correctness properties concerning those modes [28, 29, 30, 33, 59, 60, 65, 104, 105]. In my own work, I have adopted such correctness properties [199, 200, 204, 205, 206], in particular in the works contained in this habilitation work [1, 2, 3, 4, 5] (see point 5 in Section 1.3). In previous work, I have also introduced some new properties [198, 204, 208]. The following example gives a flavour of the properties.

**Example 2.3** Consider $\texttt{append}(I, I, O)$ in Figure 1. The query

$$\texttt{append([1],[2],Xs)},\ \texttt{append([3],[4],Ys)},\ \texttt{append(Xs,Ys,Zs)}$$

is "well-behaved" in that it meets all correctness properties I introduce.

In particular, note that the third atom has variables `Xs` and `Ys` in input positions, and that these variables occur elsewhere in output positions. In other words, every variable has a *producer*. Moreover, `Xs` and `Ys` occur each only once in an output position. In other words, every variable has *at most* one producer. Finally, for each variable, the output occurrence precedes any input occurrence.

Having *at most* one producer is the main aspect of a well-known correctness property called *nicely-modedness*, and having *at least* one producer is the main aspect of a correctness property called *well-modedness*.

**The Difficulty of Achieving and Showing Termination for Dynamic Scheduling**
In the 1990s, several authors have observed that for programs using dynamic scheduling, it is difficult to understand under which conditions they terminate [29, 161, 163, 164, 180]. That is, it is difficult to design the program and selection rule in such a way that the program terminates, and even if the program does terminate, it is difficult to *prove* that it does. It is helpful to separate these two aspects.

We have seen in Example 2.2 that if we do not at least require that derivations are input-consuming, we easily get non-termination. However, requiring input-consuming derivations is not enough, as the following example demonstrates. It shows a well-known termination problem of programs with dynamic scheduling, namely *circular modes* [180].

**Example 2.4** Consider the `APPEND` program in mode $\texttt{append}(I, I, O)$ (Figure 1). The following is an infinite *input-consuming* derivation:

$$\begin{array}{l}
\texttt{append([],[],As)},\ \underline{\texttt{append([1|As],[],Bs)}},\ \texttt{append(Bs,[],As)} \Longrightarrow \\
\texttt{append([],[],As)},\ \texttt{append(As,[],Bs}'\texttt{)},\ \underline{\texttt{append([1|Bs}'\texttt{],[],As)}} \Longrightarrow \\
\texttt{append([],[],[1|As}'\texttt{])},\ \underline{\texttt{append([1|As}'\texttt{],[],Bs}'\texttt{)}},\ \texttt{append(Bs}',\texttt{[],As}'\texttt{)} \Longrightarrow \ldots
\end{array}$$

The derivation starts in a query that is well moded but not nicely moded (see Example 2.3). It turns out that requiring programs to be nicely moded eliminates the problem seen in this example.

Even if we assume input-consuming derivations and correctness properties as shown in Example 2.3, there is another prominent termination problem for programs with dynamic scheduling.

**Example 2.5** Consider the `PERMUTE2` program in Figure 1 in mode $\texttt{permute}(O, I)$, $\texttt{insert}(O, O, I)$. Compared to `PERMUTE` we have swapped two body atoms for the sake of preserving the left-to-right dataflow in this mode. We have the following infinite input-consuming derivation:

$$\begin{array}{l}
\underline{\texttt{permute(W,[1])}} \Longrightarrow \\
\underline{\texttt{insert(Z}',\texttt{U}',\texttt{[1])}},\ \texttt{permute(X}',\texttt{Z}'\texttt{)} \xRightarrow{\texttt{Z}'\texttt{/[1|Z}''\texttt{]}} \\
\texttt{insert(Z}'',\texttt{U}',\texttt{[])},\ \underline{\texttt{permute(X}',\texttt{[1|Z}''\texttt{])}} \Longrightarrow \\
\texttt{insert(Z}'',\texttt{U}',\texttt{[])},\ \underline{\texttt{insert(Z}''',\texttt{U}'',\texttt{[1|Z}''\texttt{])}},\ \texttt{permute(X}'',\texttt{Z}'''\texttt{)} \Longrightarrow \\
\texttt{insert(Z}'',\texttt{U}',\texttt{[])},\ \texttt{insert(Z}'''',\texttt{U}'',\texttt{Z}''\texttt{)},\ \underline{\texttt{permute(X}'',\texttt{[1|Z}''''\texttt{])}} \Longrightarrow \ldots
\end{array}$$

The example is at the heart of the difficulty of showing termination for programs using dynamic scheduling. During my PhD work I have developed a method for showing termination for such programs assuming derivations that choose the leftmost atom among those atoms that can be chosen without violating the requirement that derivations should be input-consuming [198, 204]. This can be used to prove termination for programs that use the so-called test-and-generate paradigm [180]. The paradigm states that solution candidates are tested for being solutions even before they are completely generated so that non-solutions can be detected as early as possible.

Several methods for showing termination of logic programs assuming LD-derivations use *models*[3] and *computed answer substitutions* [31, 88, 104]. There is a criterion for clauses, called *acceptability*, based on a model $M$ of the program, essentially defined as follows: given a clause $h \leftarrow a_1, \ldots, a_n$, for each $i$ and substitution $\theta$ where $M \models (a_1, \ldots, a_{i-1})\theta$, it is required that $|h\theta| > |a_i\theta|$. Here, $|.|$ is a *level mapping*, i. e., a function assigning a natural number to each atom. The argument is that for LD-derivations, $a_1, \ldots, a_{i-1}$ must be completely resolved before $a_i$ is selected. By the correctness of LD-resolution [159], the accumulated answer substitution $\theta$, just before $a_i$ is selected, is such that $M \models (a_1, \ldots, a_{i-1})\theta$.

This argument does not apply for derivations that are merely required to be input-consuming. This is illustrated in Example 2.5. In the third line of the derivation, `permute(X′,[1|Z″])` is selected, although there is no instance of `insert(Z″,U′,[])` in the model of the program. This problem has been described by saying that `insert` makes a *speculative output binding* [180], here `Z′/[1|Z″]`.

Now of course, the derivation in Example 2.5 *does not* terminate, so it is perfectly correct that an argument for proving termination does not apply! However, even for programs that do terminate the argument does not apply:

**Example 2.6** The following is a terminating input-consuming derivation for `APPEND` of Figure 1 on page 10:

$$
\begin{array}{l}
\underline{\texttt{append([1,2],[],As)}},\ \texttt{append(As,[3],Bs)} \xrightarrow{\texttt{As/[1|As}'\texttt{]}} \\
\texttt{append([2],[],As}'\texttt{)},\ \underline{\texttt{append([1|As}'\texttt{],[3],Bs)}} \xrightarrow{\texttt{Bs/[1|Bs}'\texttt{]}} \\
\underline{\texttt{append([2],[],As}'\texttt{)}},\ \texttt{append(As}'\texttt{,[3],Bs}'\texttt{)} \xrightarrow{\texttt{As}'\texttt{/[2|As}''\texttt{]}} \\
\texttt{append([],[],As}''\texttt{)},\ \underline{\texttt{append([2|As}''\texttt{],[3],Bs}'\texttt{)}} \xrightarrow{\texttt{Bs}'\texttt{/[2|Bs}''\texttt{]}} \\
\underline{\texttt{append([],[],As}''\texttt{)}},\ \texttt{append(As}''\texttt{,[3],Bs}''\texttt{)} \xrightarrow{\texttt{As}''\texttt{/[]}} \\
\underline{\texttt{append([],[3],Bs}''\texttt{)}} \xrightarrow{\texttt{Bs}''\texttt{/[3]}} \quad \Box
\end{array}
$$

The above examples show the difficulties of achieving and proving termination for programs using dynamic scheduling. My contribution in this area [1, 2] lies in overcoming those difficulties and defining a notion of model that is adequate for dynamic scheduling, under carefully specified conditions. This model allows us to give a criterion for termination of input-consuming derivations that is sufficient and necessary, explaining the termination and non-termination of the above examples.

There is a class of recursive clauses, using a natural pattern of programming, that narrowly misses the property of termination for input-consuming derivations. Put simply, theses clauses have the form $p(\texttt{X})\texttt{:-}q(\texttt{X},\texttt{Y}),\ p(\texttt{Y})$, where the mode is $p(I),\ q(I,O)$. Due to the variable in the head, it follows that an atom using $p$ may always be selected, and hence

[3] A model of a logic program is a set of atoms $M$ such that for every program clause $h \leftarrow a_1, \ldots, a_n$, whenever $a_1, \ldots, a_n \in M$, then $h \in M$. That is, a model contains all atoms implied by the program [23]. There are generally many models and in any case there are variations of this definition, but for these intuitive explanations we do not have to worry about this.

we have non-termination. Sometimes, just requiring the argument of $p$ to be at least non-variable is enough to ensure termination. Showing this is the main contribution of my subsequent work on input-consuming derivations [3, 202]. The next example illustrates this point.

**Example 2.7** Consider the program obtained from `PERMUTE2` in Figure 1 on page 10 by replacing the recursive clause for `insert` by its *most specific variant* [180]:

```
insert([U|Z],X,[U,H|T]) :- insert(Z,X,[H|T]).
```

Assume the mode `permute`$(O, I)$, `insert`$(O, O, I)$ as in Example 2.5.

This program does not terminate for all input-consuming derivations. However, we can make the additional requirement that all selected atoms must be at least non-variable in their input positions. Under this assumption, I have shown that this program terminates [3].

Several authors including myself have written a number of articles about termination of input-consuming derivations, where the progress was mostly in extending the class for which termination could be proven, or in specifying minimal additional assumptions on the selection rule to ensure termination for programs that would otherwise not terminate [1, 2, 3, 60, 62, 63, 64, 65, 199], or in proposing a simple decidable sufficient criterion for termination [147].

**Selection Rules in General** There is an enormous body of literature on termination of logic programs, and a considerable number of works that study the impact of various selection rules on termination ([25, 29, 31, 54, 89, 146, 161, 163, 164, 167, 180, 184, 193, 194] and citations above). Due to the inhomogeneity of the approaches, formal comparisons among those works are non-trivial, and are the contribution of the surveys [4, 5], establishing the following hierarchy of termination:

1. Programs that terminate for all selection rules: *strong termination* [54];
2. Programs that terminate for input-consuming derivations: *input termination* [1, 2, 60, 62, 63, 64, 65, 199];
3. Programs that terminate for input-consuming derivations with an additional assumption about the selection rule, formalised as a set $\mathcal{P}$: *input* $\mathcal{P}$*-termination* [3];
4. Programs that terminate for *local* selection rules (see the paragraph about [3] on page 9) controlled by delay declarations: *local delay termination* [163, 164];
5. Programs that terminate for LD-derivations: *left-termination* (considered by the vast majority of works on termination [86]);
6. Programs for which a selection rule *exists* so that they terminate: $\exists$*-termination* [193, 194];
7. Programs that do not terminate at all, but for which there are only finitely many computed answers: *bounded nondeterminism.* These programs can be converted into terminating programs without compromising the logical meaning [184].

The hierarchy is shown in Figure 2. Solid arrows correspond to implications that hold without any restrictions, whereas dashed arrows correspond to implications that only hold under side conditions that are natural but must be stated very carefully — these implications are the main contribution of the work [4, 5].

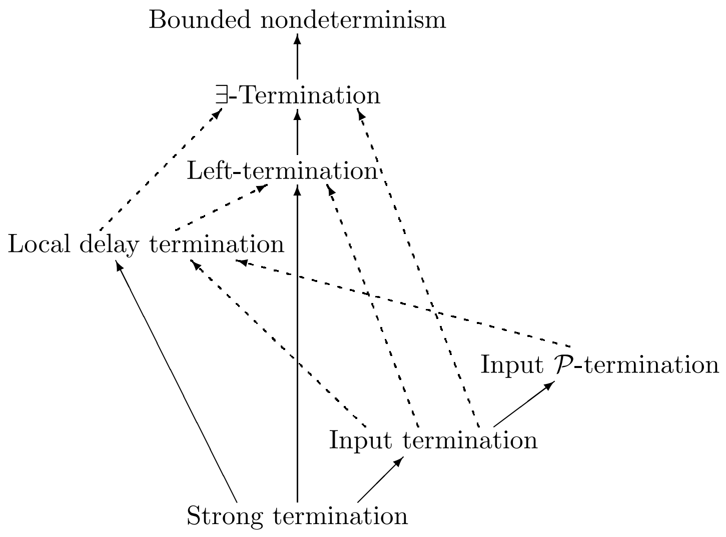

Figure 2: The termination hierarchy. Solid arrows stand for implication, dashed arrows for implication under additional assumptions.

**Termination for other Paradigms** Of course, termination has been studied for all programming paradigms and under all conceivable aspects, and some of the work on termination is either independent of the programming paradigm or can be applied to several paradigms with some adaptations. I will discuss one particular approach to showing termination of imperative programs [186] because it establishes a link to model checking (see the paragraph on predicate abstraction for termination on page 26). To show termination of an imperative program, the authors propose to use a *transition invariant*, which is a binary relation holding between any pair of program states in the transitive closure of the transition relation. The transition relation should be decomposable into several parts such that for each part, it is easy to establish well-foundedness. One might say that the work provides a technique for modularising termination proofs, an aspect also studied for logic programs [32, 104].

While the notion of transition invariant would certainly translate to logic programming, and some interesting insights might be derived from that, this would concern aspects of the termination problem quite different from the ones considered in this habilitation, which are related to the selection rule.

## 2.2 Typed Programs

### 2.2.1 Contributions on this Subtheme

I have worked on *typed* logic programming languages (such as Gödel or Mercury). Typed logic (as well as functional) programs usually have the *subject reduction* property, which means that only well-typed queries can occur during the execution of a program.

Together with François Fages and Pierre Deransart, I have studied subject reduction for typed logic programs allowing for subtyping à la `int` < `real`.[4] For such type systems, subject reduction depends on a clear notion of dataflow: wherever a `real` is expected, an `int` can also be accepted, but not vice versa. This is problematic in logic programming where, contrary to functional programming, an argument of a procedure (predicate) can serve both as input and output. We have given syntactic conditions that ensure subject

[4]`int` stands for `integer`.

reduction also in the presence of general subtyping relations between type constructors [6, 203]. The idea is to consider logic programs with a fixed dataflow, given by modes.

Together with Pierre Deransart, I have phrased subject reduction as a property of the *proof-theoretic* semantics of a program, thus abstracting from the usual operational (top-down) semantics [7]. This proof-theoretic view led us to questioning a condition which is usually considered necessary for subject reduction, namely the *head condition.* It states that the head of each clause must have a type which is a variant (and not a proper instance) of the declared type. We study more general conditions, thus reestablishing a certain symmetry between heads and body atoms.

I investigated the relationship between typing and recursion in both logic and functional programming [8]. The *head condition* was just mentioned [7], and is a concept known in the logic programming context. In typed functional programming, *polymorphic recursion* means that in a recursive definition of a function $f$, the recursive call to $f$ uses a type which is a proper instance of the declared type of $f$. I have shown that both notions are also meaningful in the respectively other paradigm, and observed a symmetry between the head condition and polymorphic recursion. This leads to an investigation of arguments for and against the head condition and polymorphic recursion, in both paradigms.

In logic programming, analysing which arguments of a predicate are (partially) ground when the predicate is called is an important application of abstract interpretation, because such information is useful for compiler optimisations. For typed languages, the types can be used to characterise the degree of instantiation of a term in a precise and yet inherently finite way, e. g. by speaking about a list that is instantiated but whose elements are not instantiated. I have developed a formalism for expressing the degree of instantiation (an *abstract domain* for each type) that works for polymorphically typed programs [9, 201]. The analysis of the program is performed by running the program with the usual terms being replaced by *abstract* terms (members of the abstract domain), and the usual unification being replaced by unification modulo a theory including certain associativity and commutativity axioms.

### 2.2.2 More Details on the Concepts and Problems of this Subtheme

**Type System with Parametric Polymorphism** In logic programming, types have been used as a tool for program analysis even for untyped programming languages (*descriptive* types). However, these are not the kind of types considered in this subtheme. Here, I consider types that are a part of the programming language [91, 136, 209], sometimes called "prescriptive" types for emphasis. In functional programming, types are more wide-spread [183, 217, 218] and although not all functional programming languages are typed (LISP [224] is not), when one speaks of types in functional programming one means prescriptive types.

My research on typed languages has addressed rather diverse questions, but in all cases, I assumed a type system with *parametric polymorphism.* Let us explain the central notions of such a type system using an example that applies to logic and functional programming.

**Example 2.8** Let $\mathtt{int}/0, \mathtt{list}/1$ be some *type constructors*, $\mathtt{0}_{\rightarrow\mathtt{int}},$, $\ldots$, $\mathtt{nil}_{\rightarrow\mathtt{list(U)}}$, $\mathtt{cons}_{\mathtt{U},\mathtt{list(U)}\rightarrow\mathtt{list(U)}}$ be some *term constructors*, , $\mathtt{append}_{\mathtt{list(U)},\mathtt{list(U)},\mathtt{list(U)}}$ be a (logic programming) *predicate*, and $\mathtt{append}_{\mathtt{list(U)},\mathtt{list(U)}\rightarrow\mathtt{list(U)}}$ be a (functional programming) *function.* Then, $\mathtt{U}$ is a *(type) parameter*, $\mathtt{U}$, $\mathtt{list(U)}$, and $\mathtt{list(list(int))}$ are examples of *types*, `[7]` is an example of a *constructor term*[5], and $\mathtt{append(nil,[7],X)}$ is an example of a (logic programming) *atom.*

[5]We use the usual list notation `[7]` instead of $\mathtt{cons(7,nil)}$.

In the example, we have written the *declared type* of each symbol in the term language as a subscript. Typed programming languages provide some syntax for declaring which type constructors there are and what the type of each symbol in the term language is. Moreover, the programming language has fixed rules for specifying well-typed syntactic objects. These rules would, for instance, rule out an atom $\mathtt{append}(\mathtt{5},[\mathtt{7},\mathtt{3}],\mathtt{X})$ since $\mathtt{5}$ is of type $\mathtt{int}$ and not of type $\mathtt{list}(\mathtt{int})$ as required.

**Subject Reduction** A type system as just described allows for many programming errors to be detected by the compiler. Moreover, it ensures that once a program has passed the compiler, the types of arguments of predicates can be ignored at runtime, since it is guaranteed that they will be of correct type (see point 5 in Section 1.3). This has been turned into the famous slogan: *Well-typed programs cannot go wrong* [171, 178]. Adopting the terminology from the theory of the $\lambda$-calculus [216], this property of a typed program is called *subject reduction.* There, subject reduction states that the type of a $\lambda$-term is invariant under reduction. Translated to logic programming, resolving a "well-typed" query with a clause will always result in a "well-typed" query, and so the successive queries obtained during a derivation are all "well-typed".

The subject reduction property depends on the details of the formalisation of the type system. For example, one condition discussed in the literature is the *transparency condition* [135, 137]. It states that for the declared type $\tau_1, \ldots, \tau_n \rightarrow \tau$ of a term constructor $f$, each type parameter occurring in $\tau_1, \ldots, \tau_n$ must also occur in $\tau$. The condition ensures that if we have two terms $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ of identical type, then the two terms have identical types in all matching subterms, which is crucial when we attempt to unify the two terms. In the first famous article advocating typed logic programming [178], this condition was ignored, leading to an erroneous claim of subject reduction, as explained by Hill [135].

Subject reduction is particularly problematic for logic programs with subtyping. This is illustrated by the following example.

**Example 2.9** Consider a typed language consisting of the predicates $\mathtt{sqrt}_{\mathtt{real},\mathtt{real}}$ and $\mathtt{fact}_{\mathtt{int},\mathtt{int}}$, and assume that $\mathtt{int}$ is a subtype of $\mathtt{real}$. Consider the program

```
fact(3,6).
sqrt(6,2.449).
```

and the derivations

$$\begin{array}{l}\mathtt{fact}(\mathtt{3},\mathtt{x}),\ \mathtt{sqrt}(\mathtt{x},\mathtt{y}) \Longrightarrow \mathtt{sqrt}(\mathtt{6},\mathtt{y}) \Longrightarrow \Box \\ \mathtt{sqrt}(\mathtt{6},\mathtt{x}),\ \mathtt{fact}(\mathtt{x},\mathtt{y}) \Longrightarrow \mathtt{fact}(\mathtt{2.449},\mathtt{y})\end{array}$$

In the first derivation, all arguments always have a type that is less than or equal to the declared type, and so we have subject reduction. In the second derivation, the argument $\mathtt{2.449}$ to $\mathtt{fact}$ has type $\mathtt{real}$, which is strictly greater than the declared type. The atom $\mathtt{fact}(\mathtt{2.449},\mathtt{y})$ is illegal, and so we do not have subject reduction.

The problem for subject reduction can be attributed to the fact that in logic programs, there is a-priori no fixed direction of dataflow. In the work with Fages and Deransart [6], we introduce the notion of *nicely typed program*, which is an adaptation of *nicely moded programs*, used in the works discussed above [1, 2, 3, 4, 5] (see Ex. 2.3 on page 12), to the typed context. For such programs, we can guarantee subject reduction even in the presence of subtyping.

The *head condition*, also called *definitional genericity* [152], is also an important property for subject reduction. It states that the head of each clause must have a type which is a variant (and not a proper instance) of the declared type. For example, if the predicate $\mathtt{append}$ has declared type $\mathtt{list}(\mathtt{U}), \mathtt{list}(\mathtt{U}), \mathtt{list}(\mathtt{U})$, then it would be forbidden

```
h(X):-q(X),p(X).
q([]).
p(X):-r(X).
```

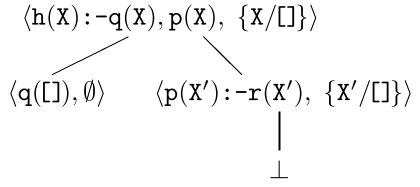

Figure 3: A program and a derivation tree

to have a clause head append([1],nil,[1]), as its type is list(int),list(int),list(int) and thus more specific than the declared type. The condition ensures that whenever there is an atom append($s,r,t$) in a query of a derivation of a logic program, then the type of every clause for append is always consistent with that atom.

**Proof-theoretic Semantics** In logic programming, one is interested in the logic or *proof-theoretic* semantics of a logic program, as opposed to the operational semantics given by SLD resolution [159]. Deransart and Małuszyński established what they call a *grammatical view of logic programming* [93].[6] The execution of a program is viewed as a *derivation tree*, composed of the clauses used, together with the substitution being applied to each clause. Figure 3 illustrates this. This view abstracts away from the order in which atoms are selected, in fact from any notion of *past*, *current*, and *future* queries, which in turn suggests that there should be a symmetry between the head and the body atoms of a query, i. e., one would intuitively expect that any atom that is legal as a body atom is also legal as a head and vice versa.

In the work with Deransart [7], we applied this proof-theoretic view to the subject reduction property. We defined subject reduction as stating that every derivation tree that can be constructed consists of well-typed nodes, i. e., the substitution of the node applied to the clause of the node yields a well-typed instance of the clause. Since the head condition contradicts the desired symmetry of heads and body atoms, we questioned this condition. We showed that when lifting this condition, subject reduction is an undecidable property. We then introduced sufficient conditions for subject reduction that generalise the head condition.

**The Head Condition and Polymorphic Recursion** In the work with Deransart [7], it has already been observed that the head condition, or better, *violations* thereof, is somehow related to a phenomenon discussed in functional programming, namely *polymorphic recursion* [56, 129, 141, 142], which refers to the situation that in a recursive definition of a function $f$, the recursive call to $f$ uses a type which is a proper polymorphic instance of the declared (generic) type of $f$. A (contrived) example is the following Miranda program defining list length:

```
len :: [*] -> num
len []    = 0
len (a:l) = 1 + len (lift l)
```

The first line is a type declaration stating that len is a function from the (built-in) parametric list type to the (built-in) type of numbers. The other two lines form a recursive definition. Here we assume that lift is the function that takes a list [$t_1,\dots,t_n$] and returns the list [[$t_1$],…,[$t_n$]]. When len is called with the list, say, [1,2,3] of type [num]

[6]This is somewhat similar to a *predicative* view of transition systems (see point 6 in Section 1.3), in that one abstracts away from a transition sequence by turning it into something "statical", here a derivation tree.

(Miranda notation for a list of numbers), the recursive calls will be with types `[[num]]`, `[[[num]]]`, and `[[[[num]]]]`.

I have written an article establishing a strong relationship between the head condition and polymorphic recursion [8]. Adjusting the terminology, one can speak of a symmetry: the head condition is symmetric to *monomorphic* recursion, meaning absence of polymorphic recursion. The symmetry can be illustrated with the logic programs $P_{pr} = \{$`p(X).`, `p(X):-p([X]).`$\}$ and $P_{hc} = \{$`p([]).`, `p([X]):-p(X).`$\}$. Letting `list(U)` be the declared type of `p`, $P_{pr}$ uses polymorphic recursion, and $P_{hc}$ violates the head condition. The symmetry also shows up in the operational semantics: when $P_{pr}$ is called with `p([])`, an infinite sequence of calls `p([])`, `p([[]])`, $\ldots$ arises, whereas $P_{hc}$ has infinitely many computed answers `p([])`, `p([[]])`, $\ldots$.

Both the head condition and polymorphic recursion have been debated in the literature [56, 126, 129, 137, 141, 142, 152, 160, 177, 179, 182]. However, due to the fact that the former was mainly discussed in logic programming and the latter in functional programming, the formalisations of the type systems were quite different. My contribution [8] to this area lies in a uniform formalisation of the type systems that allows for comparisons between functional and logic programming and comparisons between the head condition and polymorphic recursion. I give an example of a term language for which one finds that the head condition and monomorphic recursion create (almost) unacceptable obstacles for programming, but I also discuss the advantages of these conditions. Furthermore, violations of the head condition, as well as polymorphic recursion, may be responsible for an infinite number of different types occurring in the semantics of a program. This phenomenon is undesirable in certain contexts of program analysis, an issue which I actually took up in later work [9], discussed below.

**Groundness Analysis** Analysis of (partial) groundness [36, 37, 51, 75, 76, 77, 78, 79, 80, 112, 127, 128, 143, 166, 215] is an important application of abstract interpretation [82] (see point 4 in Section 1.3). There have been several proposals for improving the precision of such an analysis by exploiting type information [70, 78, 81, 113, 114, 140, 151, 195], including also some work I did for my PhD [198, 207]. For typed languages, the types can be used to characterise the degree of instantiation of a term in a precise and yet inherently finite way. For example, one might infer that the call `append(list(any), list(any), any)` leads to the answer `append(list(any), list(any), list(any))`, that is to say: when the `APPEND` program (see Figure 1 on page 10) is called with the first two arguments being instantiated to lists (but the list elements are not necessarily instantiated), then on successful completion of the run, the third argument will also be instantiated to a list (but the list elements are not guaranteed to be instantiated) [78]. The expressions `list(any)` and `any` are called *abstract terms* from an *abstract domain*. In this context, an abstract domain is a language that can be used to express the degree of instantiation of a term. E. g., `list(any)` represents a higher degree of instantiation than `any`.

The question is: how can the types give rise to a generic definition of an abstract domain, i. e., how can the types determine the granularity with which the instantiation of the corresponding terms can be characterised? What I mean by "generic" is best illustrated by a counterexample: I might propose an abstract domain that can express that a list is ground in its 2nd, 3rd, 5th, 7th, 11th etc. element, i. e., in all positions corresponding to prime numbers. The key concept of a generic definition [9] is the *subterm type* relation (written $\lhd$), subdivided into the *recursive type* relation (written $\bowtie$) and the *non-recursive subterm type* relation (written $\lll$).

**Example 2.10** The relations can be illustrated using a graph with the types as nodes. Consider Figure 4. We have (assuming type declarations as in Example 2.8) `int` $\lhd$ `list(int)` because a term of type `int`, say `7`, can be a *subterm* of a term of type `list(int)`, say `[5,7,2]`. Since the converse does not hold, we have `int` $\lll$ `list(int)`.

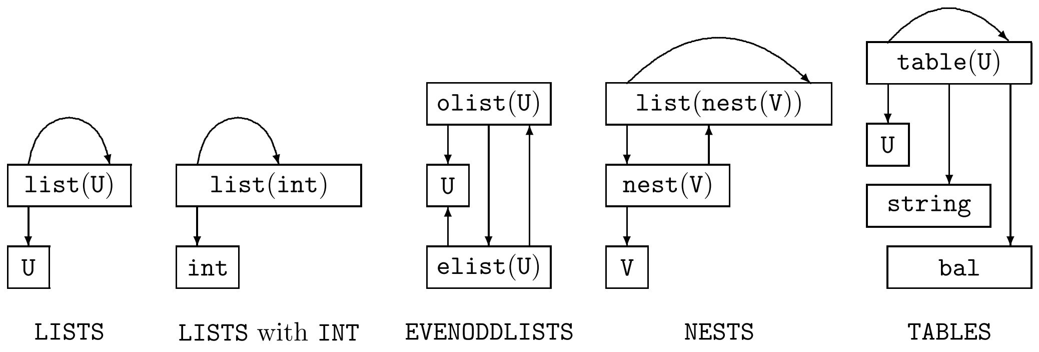

Figure 4: Illustrating the subterm type relation

Next, consider a typed language where we have $\mathtt{nil}_{\to\mathtt{elist(U)}}$, $\mathtt{econs}_{\mathtt{U,olist(U)}\to\mathtt{elist(U)}}$, $\mathtt{ocons}_{\mathtt{U,elist(U)}\to\mathtt{olist(U)}}$, i. e., a somewhat unusual pair of list types, one for lists of even length, one for lists of odd length. We have $\mathtt{U} \lll \mathtt{elist(U)}$, $\mathtt{U} \lll \mathtt{olist(U)}$, and $\mathtt{elist(U)} \bowtie \mathtt{olist(U)}$.

Next, consider a typed language where we have $\mathtt{e}_{\mathtt{V}\to\mathtt{nest(V)}}$, $\mathtt{n}_{\mathtt{list(nest(V))}\to\mathtt{nest(V)}}$, plus the list constructors. This language implements *rose trees* [56, 170], i. e., trees where the number of children of each node is not fixed. This example illustrates a phenomenon that accounts for a lot of complication in the formalism I developed [9]: the $\lll$-relation is not closed under type instantiation! We have $\mathtt{U} \lll \mathtt{list(U)}$, but $\mathtt{nest(V)} \bowtie \mathtt{list(nest(V))}$.

A *table* is a data structure containing an ordered collection of nodes, each of which has two components, a key of type `string`, and a value, of arbitrary type. The typed language consists of $\mathtt{lh}_{\to\mathtt{bal}}, \mathtt{rh}_{\to\mathtt{bal}}, \mathtt{eq}_{\to\mathtt{bal}}, \mathtt{null}_{\to\mathtt{table(U)}}$, $\mathtt{node}_{\mathtt{table(U),string,U,bal,table(U)}\to\mathtt{table(U)}}$. The type `bal` contains three constants representing balancing information. We have $\mathtt{table(U)} \bowtie \mathtt{table(U)}$ and $\mathtt{U} \lll \mathtt{table(U)}$, $\mathtt{bal} \lll \mathtt{table(U)}$, $\mathtt{string} \lll \mathtt{table(U)}$. This is an example where one type has more than just one non-recursive subterm type, a possibility ignored by previous work [78].

In devising an analysis for polymorphically typed programs, one main problem is to achieve a construction of an abstract domain for $c(\tau_1, \ldots, \tau_n)$, where $c$ is a type constructor, that is truly parametric in $\tau_1, \ldots, \tau_n$, that is to say, the abstract domain for `list(int)`, say, should relate to `int` in exactly the same way as the abstract domain for `list(nest(int))` relates to `nest(int)`, in spite of the fact that $\mathtt{int} \lll \mathtt{list(int)}$ but $\mathtt{nest(int)} \bowtie \mathtt{list(nest(int))}$. Another problem is that abstract domains should be finite for a given program and query. These requirements are non-trivial, as suggested by the `NESTS` example, but they are fulfilled by my approach [9]. Previous works [78, 81] only deal with a specific set of types including integers, lists, difference lists, and trees, while the follow-up work [151] does not consider polymorphism at all.

For types that have no non-recursive subterm types (such as `int`), the abstract terms in my approach simply characterise whether a term is ground. For types that do have non-recursive subterm types, the abstract terms are constructed in a recursive way so that they can represent the degree of instantiation of the subterms, for each non-recursive subterm type. For example, the abstract term $\mathtt{table}^{\mathcal{A}}(\mathtt{int}^{\mathcal{A}} \oplus \mathtt{X}, \mathtt{bal}^{\mathcal{A}}, \mathtt{str}^{\mathcal{A}})$ represents any table whose values might be ground (represented by $\mathtt{int}^{\mathcal{A}}$) or variables (represented by the variable `X`), and whose balancing labels and keys are ground.

The analysis of programs follows the *abstract compilation* [78, 79, 81, 87, 134, 151] approach, i. e., an abstract program is obtained by replacing each term in the program by its abstraction, and then the abstract program is "run". However, there is one difference to the way logic programs are usually run, namely that the usual syntactic unification is replaced with unification modulo a theory that includes associativity, commutativity, and idempotence. In this theory, $\mathtt{table}^{\mathcal{A}}(\mathtt{int}^{\mathcal{A}} \oplus \mathtt{X}, \mathtt{bal}^{\mathcal{A}}, \mathtt{str}^{\mathcal{A}})$ and

$\mathtt{table}^{\mathcal{A}}(\mathtt{int}^{\mathcal{A}}, \mathtt{bal}^{\mathcal{A}}, \mathtt{Y})$ would have the unifier $\{\mathtt{X}/\mathtt{int}^{\mathcal{A}}, \mathtt{Y}/\mathtt{str}^{\mathcal{A}}\}$ and the common instance $\mathtt{table}^{\mathcal{A}}(\mathtt{int}^{\mathcal{A}}, \mathtt{bal}^{\mathcal{A}}, \mathtt{str}^{\mathcal{A}})$.

As an example of the information that might be inferred by such an analysis, suppose there is a predicate `insert`/4 whose arguments represent: a table $t$, a key $k$, a value $v$, and a table obtained from $t$ by inserting the node whose key is $k$ and whose value is $v$. From the abstract semantics of the program, it is possible to read that a query whose abstraction is

$$\mathtt{insert}(\mathtt{table}^{\mathcal{A}}(\mathtt{int}^{\mathcal{A}}, \mathtt{bal}^{\mathcal{A}}, \mathtt{str}^{\mathcal{A}}), \mathtt{str}^{\mathcal{A}}, \mathtt{V2}, \mathtt{T}),$$

i. e., a query to insert an *uninstantiated* value into a ground table, yields an answer whose abstraction is

$$\mathtt{insert}(\mathtt{table}^{\mathcal{A}}(\mathtt{int}^{\mathcal{A}}, \mathtt{bal}^{\mathcal{A}}, \mathtt{str}^{\mathcal{A}}), \mathtt{str}^{\mathcal{A}}, \mathtt{V2}, \mathtt{table}^{\mathcal{A}}(\mathtt{int}^{\mathcal{A}} \oplus \mathtt{V2}, \mathtt{bal}^{\mathcal{A}}, \mathtt{str}^{\mathcal{A}})),$$

i. e., the result is a table whose values may be uninstantiated.

**Types and Termination Analysis** To relate the subthemes considered in Sections 2.1 and 2.2, let us mention that several authors have used types to improve the precision of termination analyses [29, 61, 68, 69, 150, 168].

## 2.3 "Formulas as Programs"

Krzysztof Apt et al. have proposed a new approach to using first order logic as a programming language, based on analogies between logic concepts and features in imperative programming languages [24, 26, 27, 35]. Together we have written an article comparing this approach with logic programming [34]. The approach might be seen as a "fresh look" at how to synthesise an operational behaviour from a logic formula (see point 2 in Section 1.3). One motivation was to gain a better understanding of the distinction between the "purely logic-based" aspects of a computation and the aspects involving domain-specific knowledge.

At the heart of this approach is a semantics, i. e., a set of solutions, for an equation $s = t$. In full generality, the problem of giving such a set of solutions cannot be addressed properly. For example, $x^2 - 1 = 0$ has one solution $\{x/1\}$ if we assume the set of arithmetic expressions interpreted over the natural numbers, but two solutions $\{x/1\}$ and $\{x/-1\}$ if we assume the integers. Apt's objective was to define a semantics *for an arbitrary algebra*. This means that the definition must not refer to any particular algebra (or language) or use any knowledge about it other than how to evaluate a ground term. Apt has given such a semantics and showed that it generalises syntactic unification, and moreover argued informally that this semantics is optimal in the sense that it is the best one can do without specific knowledge of the algebra [24].

I have investigated the optimality issue [10]. I proposed an improvement of the semantics and showed that this improved semantics is nonetheless *not* optimal in any formal sense, since there seems to be no satisfactory formalisation of what it means to have "no specific knowledge of the algebra". It is always possible to "improve" a semantics in a formal sense, but such semantics become more and more contrived. For example, we might propose a semantics that considers as solution candidates all substitutions defined by the replacement of each variable $x$ occurring in the equation by a ground subterm $r$ also occurring in the equation. This is an unambiguous description independent of any particular language. For $x + 1 = 2$, it would compute the solution $\{x/1\}$ since the equation contains the candidate 1, but for $x + 1 = 3$ it would not work because the equation does not contain any occurrence of 2. Such a semantics would be "better" than the one I propose, but it would be unacceptably contrived.

# 3 Logic Approaches to Model Checking of Timed and Hybrid Systems

## 3.1 Finding Error Paths in Timed and Hybrid Systems

### 3.1.1 Contributions on this Subtheme

*Hybrid systems* are systems featuring the interaction between a continuous part (often modelling some physical entities such as temperature or speed), and a discrete control part. The behaviour (flow) of real-valued variables over time is usually described by differential equations. Given a hybrid system with a suitably specified error state, one is interested in verification, i. e., proving that the error state is unreachable (safety). However, if the system is actually not safe, one is also interested in falsification, i. e., one looks for an actual trajectory over time that will lead to an error state. Falsification is usually difficult since the solutions to the differential equations are trigonometric and similar functions. However, for nondeterministic systems, i. e., systems where the flow is described by differential inequalities rather than equations, falsification is often easier because one can find error paths where the flows are described by polynomials or even piecewise linear functions. Together with Stefan Ratschan, I have devised a method for coupling safety verification algorithms for nondeterministic hybrid systems (implemented in a tool called HSolver) with the ability of finding concrete error trajectories, i. e., with falsification [11]. Such a tight integration of verification with falsification has the advantage that verification attempts guide the search for concrete error trajectories, and endless attempts to verify unsafe systems or to falsify safe systems can often be avoided.

Together with Jörg Hoffmann, Andrey Rybalchenko, Sebastian Kupferschmid and Andreas Podelski, I have worked on the design of heuristic functions for speeding up *directed model checking*, i. e., error search, of timed automata [12]. These heuristic functions can be plugged into the well-known Uppaal system. The heuristic functions are defined based on an over-approximation of the timed automaton, more precisely a predicate abstraction, which is a technique previously used for verification. Here, a *predicate* is a logic formula that speaks about the automaton; e. g., a predicate might state: the automaton is in location $\ell_5$. Fixing a list of such predicates, the abstraction of a timed automaton state is just a bitvector stating which of the predicates are true and which ones are false in this state. The abstract state space is then defined as an over-approximation of the concrete state space, and for each concrete state, the heuristic value is given by the length of a path leading to an error state, within the abstraction. The abstract state space is exhaustively built in a pre-process, and used as a lookup table (*pattern database*) during search. We have empirically explored the behaviour of the resulting family of heuristics, showing that one can easily obtain heuristic functions that are competitive with the state-of-the-art in directed model checking. As a follow-up on this work, we studied how predicates for predicate abstraction can be generated using specialised variants of *abstraction refinement* [13].

### 3.1.2 More Details on the Concepts and Problems of this Subtheme

**Hybrid Automata and Timed Automata** One formalisation of hybrid systems is that of *hybrid automata* [20, 130, 133, 212]. A hybrid automaton is a finite automaton enhanced with real-valued variables. Figure 5 shows an example of a hybrid automaton, modelling a car (with the variables denoting the angle of the steering wheel, the speed, etc.) trying to steer clear of a canal. Each box depicts a *location* (*state*, according to the usual terminology of finite automata) with a name (e. g., "go_ahead"). In each location, the behaviour of the variables is governed by logic formulae built from differential (in)equations, i. e., equations using the variables of the hybrid automaton as well as their derivatives, as in $\dot{x} = -r \ \sin\gamma$. In the example, all locations except "in_canal" have

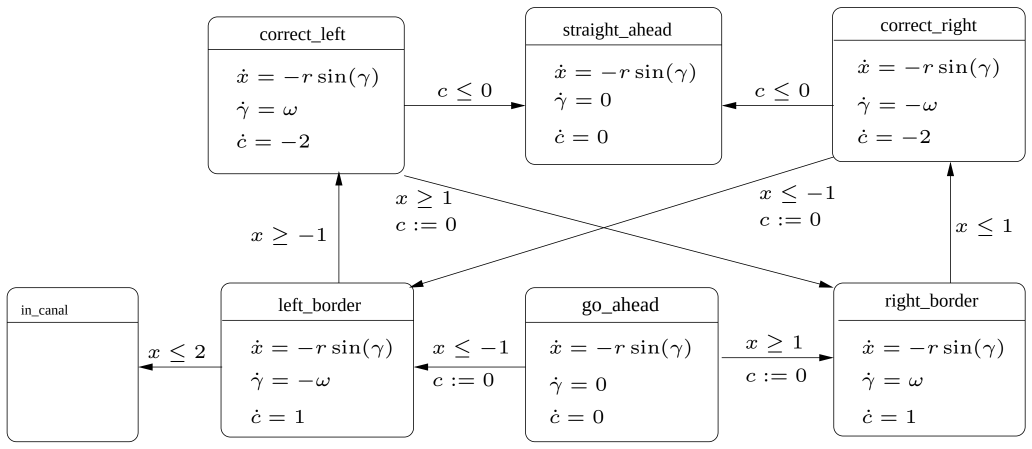

Figure 5: Model of a car steering clear of a canal

a conjunction of three differential equations. We say that these (in)equations describe the possible *flows* of the variables. The (in)equations might also just use the automaton variables without the derivatives, e. g. $x \leq 5$ (not in the car example above), in which case we speak of an *invariant* of that location. For transitions, there can be *guards*, which are also logic formulae, restricting the possibilities of taking the transition. In the example, $c \leq 0$ or $x \geq 1$ are guards. Moreover, there are updates, e. g. $c := 0$, which are executed when a transition is taken.

The semantics of a hybrid automaton is given by its *trajectories*. Intuitively, the trajectory starts in some location marked as initial, remains there for some time $l$ during which the variables change according to some differentiable function $f : [0, l] \to \mathbb{R}^n$ (where $n$ is the number of variables) that solves the differential equations mentioned above, then a transition is taken (if the guards permit it), and so on. A formal definition can be found in the work by Henzinger [130], or in the according contribution in this habilitation [11].

*Timed automata* are a special case of hybrid automata, essentially obtained by restricting the behaviour of each real-valued variable $x$ in each location using the differential equation $\dot{x} = 1$ [21, 50, 225]. That is, each variable changes at the same rate as time and is thus a *clock*. When a transition is taken, a clock may be reset to 0, but afterwards, it immediately starts running again. Figure 6 shows a timed automaton. Actually, it is the product of two automata. In the parlance of Uppaal [48, 153], which is a model checker for timed automata, we have a *system* composed of two *processes*. The initial location of each process is drawn as a double circle. The labels "**study**", "**idle**" etc. denote the names of the locations. The symbol "press" is a *channel*; two transitions labelled "press!" and "press?" must be taken simultaneously. Apart from that the figure should be read similarly as Figure 5.

**Reachability and Safety** There are a number of (model checking) problems for timed and hybrid automata, which have been studied for numerous variations of the two formalisms (typically, restrictions of hybrid automata or generalisations of timed automata) [19, 20, 50, 131, 133, 212]. This gives rise to plethora of decidability issues and results.

In my work, and in other work conducted within the AVACS project[7] (e. g. [16, 17, 94, 98, 109, 148, 149, 189, 190]), the focus was on proving or disproving safety. A

[7] Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB TR14/AVACS) of the Deutsche Forschungsgemeinschaft

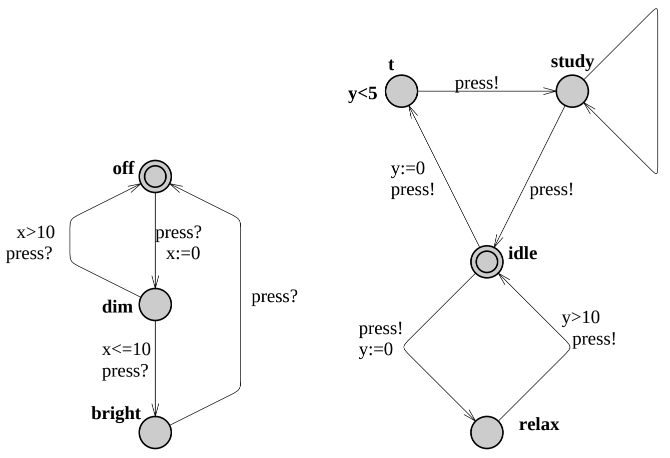

Figure 6: A timed automaton system composed of two processes

system is safe if and only if no unsafe state (specified in a suitable way) is reachable from an initial state. Otherwise we have an *error trajectory*. Reachability is decidable for timed automata. The algorithm is based on the insight that it is not necessary to distinguish infinitely many system states, but that the infinite continuous state space can be partitioned into so-called *regions*, so that the state space becomes effectively finite [50].

However, consider an extension of the clock notion of timed automata, namely stopwatches. A *stopwatch* is a clock that is allowed to stop running, i. e., its behaviour may be specified by $\dot{x} = 0$ *or* $\dot{x} = 1$. For timed automata with one single stopwatch, the reachability problem is undecidable [130]. Not surprisingly, for general hybrid automata the reachability problem is undecidable [133].

The works included in this habilitation are concerned with reachability for timed [12, 13] and for hybrid [11] automata.

**Directed Model Checking** For timed automata, the model checker UPPAAL can be used to find an error trajectory, but since the state spaces of timed automata can be huge, the search for an error can benefit greatly from *heuristics*. This approach has been called *directed model checking* [101, 102] and has been applied to timed automata within the AVACS project [98, 148, 149], including two works contained in this habilitation [12, 13].

**Nondeterminism** For hybrid automata, we distinguish continuous nondeterminism and discrete nondeterminism. By *continuous* nondeterminism, we mean that the flow of the variables is not uniquely determined. This could be caused by having disjunctions such as $\dot{x} = 2 \vee \dot{x} = -2$ or by having differential inequalities such as $2 \leq \dot{x} \leq 3$. The continuous nondeterminism caused by inequalities is the one assumed in my work [11]. By *discrete* nondeterminism, we mean that it is not specified uniquely at which point in time a *jump* from one location to another should take place, and that there could be several possibilities of where to jump to. In timed automata, we only have the latter type of nondeterminism, but usually to a very big extent, which makes the reachability problem non-trivial.

**Verification and (Predicate) Abstraction** In our context, *verification* means to show that a system is safe, i. e., that no error trajectory exists. Verification can be done by defining a suitable *abstraction* or *over-approximation* of the system. This is a system

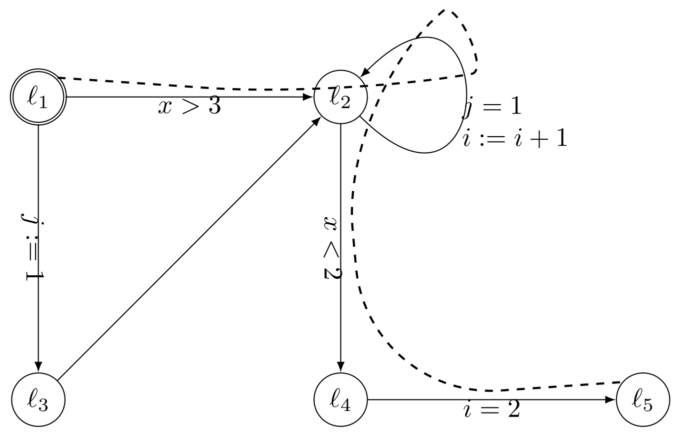

Figure 7: A timed automaton with spurious abstract error path ($\ell_5$ is the error state)

which exhibits all the behaviours of the concrete system and possibly more, and which is easier to handle than the original system. If in this abstract system no error trajectory exists, then surely there exists no error trajectory in the concrete system (see point 4 in Section 1.3).

One particular abstraction method is that of *predicate* abstraction [42, 72, 132, 188]. Here, a system is abstracted by specifying a finite set of logic formulas, called *predicates*, each of which formulates some property of a system state. Fixing a list of such predicates, the abstraction of a system state is just a bitvector stating which of the predicates are true and which ones are false in this state. Figure 7 shows a simple timed automaton used for illustrating this idea. Suppose we have abstraction predicates $loc = 2, loc = 5, i > 0$. The first two state that the automaton is in location 2 or 5, respectively, and the third makes a statement about the integer variable $i$.

In the abstraction, there is the following error path:

$$\begin{pmatrix} \neg loc = 2 \\ \neg loc = 5 \\ \neg i > 0 \end{pmatrix} \to \begin{pmatrix} loc = 2 \\ \neg loc = 5 \\ \neg i > 0 \end{pmatrix} \to \begin{pmatrix} loc = 2 \\ \neg loc = 5 \\ i > 0 \end{pmatrix} \to \begin{pmatrix} \neg loc = 2 \\ \neg loc = 5 \\ i > 0 \end{pmatrix} \to \begin{pmatrix} \neg loc = 2 \\ loc = 5 \\ i > 0 \end{pmatrix}.$$

This error path uses the transitions $\ell_1 \to \ell_2$, $\ell_2 \to \ell_2$, $\ell_2 \to \ell_4$, and $\ell_4 \to \ell_5$, as shown by the dashed curve in the figure. Note that the abstraction is too coarse to check whether the guard of $\ell_2 \to \ell_2$ is fulfilled, since the abstraction does not speak about $j$ — this is where the abstraction loses information compared to the concrete system, i. e., it is an over-approximation. In fact, this path is *spurious*, i. e., it does not correspond to a concrete path. Thus, the abstraction is not (yet) fine enough for verifying this system. This can often be resolved by *abstraction refinement* [43, 73, 188], but not in this example, because this timed automaton is actually unsafe! The example was chosen deliberately because in our work [12, 13], predicate abstraction is not used for verification as in previous works, but for defining heuristic functions.

Note that logic is used here for *analysing* the operational behaviour of the timed automaton (see point 3 in Section 1.3). However, the use of logic for defining a heuristic might also be seen as a *synthesis* of operational behaviour (of the search algorithm) from logic (see point 2 in Section 1.3).

Compared to other kinds of abstraction (consider for example HSolver discussed below), predicate abstraction is an approach that explicitly rests on *logic*, and in the ARMC model checker [188], this approach has been implemented using *logic* programming. The implementation makes use of features such as the automatic renaming of clause instances built into logic programming, but also of add-on features such as the possibility to assert and retract clauses during the execution, which are sometimes frowned upon because they destroy the declarative semantics of logic programs discussed in Sections 1.2.1 and 2.1. However, it is the use of these features that make this implementation a very interesting application of logic programming.

**Predicate Abstraction for Termination Analysis** Predicate abstraction has also been used for termination analysis [187], thus linking the present theme of this habilitation to the theme discussed in Section 2.1. The work is on termination of (imperative) *programs*, but in fact the modelling of programs is such that one can interpret them as an UPPAAL system composed from possibly several processes, but without clocks (see also the paragraph on termination for imperative programs on page 15).

*Transition predicates* are predicates which do not formulate a property of a system *state*, but of a system *transition*, typically by using unprimed and primed variables for the predecessor and successor state, e. g. $x' \geq x + 1$ which states that the transition increases the value of $x$ by at least 1. Thus depending on the choice of transition predicates, one may or may not be able to describe certain behaviours of transitions. If the predicates are well chosen, then it is possible to show termination of the program by showing a decrease of some well-founded order for each of the finitely many *abstract* transitions of the program.

**HSOLVER** HSOLVER is a tool for *verification* of hybrid systems [189, 190]. It works by partitioning the continuous state space into finitely many boxes and using interval arithmetic [172] for determining for any box pair $A, B$ whether there is a trajectory from some point in box $A$ to some point in box $B$, in which case there should be an edge connecting $A$ and $B$ (see point 4 in Section 1.3). As usual with abstractions (see above), one aims at an abstraction that is fine enough so that it contains no error trajectories, in which case the system has been proven safe. In HSOLVER, abstractions are successively refined for this purpose. The refinement is done by splitting a box in two. My work [11] interleaves the HSOLVER verification procedure with a falsification procedure.

**Bounded Model Checking** Bounded model checking is an approach for proving unsafety, or a limited degree of safety, for a system, by translating the behaviour of the system into a logic formula [16, 55, 111, 169, 174, 175, 211] (see point 6 in Section 1.3). Given some bound $k$, the formula encodes the statement "there exists an error path of length up to $k$ in this system". Clearly, if this formula is satisfiable, then there exists an error path; otherwise, there exists no error path for path lengths up to $k$. To decide (or at least: *attempt* to decide) the satisfiability, one uses a suitable logic solver. Depending on several parameters (the kind of system considered, the logic used etc.), it may be possible to read an actual error path off the satisfying instance computed by the solver.

In my approach for finding error paths of hybrid systems [11], I use a formula that is similar to the one used in bounded model checking. The search for error paths is guided by the HSOLVER abstraction [189, 190]. The formula states "there exists an error path which closely corresponds to an abstract error path in the current abstraction" (see also point 3 in Section 1.3). This restriction to error paths that closely correspond to a certain abstract error path provides good guidance for the search.

One important method that bounded model checking rests on is *SAT solving*. In fact, several authors have studied adaptations of general SAT solving for the bounded model checking context [16, 111, 211], which is an important topic in the AVACS project[7]. For SAT solving, concise representations of propositional formulae are an important issue, which leads us to Section 3.2.

## 3.2 Linear Pseudo-Boolean Constraints

### 3.2.1 Contributions on this Subtheme

A *linear pseudo-Boolean constraint* (LPB) is an expression of the form $a_1 \cdot l_1 + \ldots + a_m \cdot l_m \geq d$, where each $l_i$ is a *literal* (it assumes the value 1 or 0 depending on whether a propositional variable $x_i$ is true or false) and the $a_1, \ldots, a_m, d$ are natural numbers. The formalism can be viewed as a generalisation of a propositional clause, on the other hand it is a restriction of *integer linear programming.* It has been said that LPBs can be used to represent Boolean functions more compactly than the well-known *conjunctive* or *disjunctive* normal forms (CNF or DNF). I have studied the question: *how much* more compactly? In a first work, I have compared the expressiveness of a single LPB to that of related formalisms. Moreover, there is a formal statement that outlines how the problem of computing an LPB representation of a given CNF or DNF might be solved recursively. On the other hand, I have shown that there is a class of *monotone* Boolean functions for which the LPB representation saves nothing compared to the CNF or DNF [14]. In subsequent work, I have completed the above outline and given a full recursive algorithm for computing an LPB representation of a given formula if this is possible [15].

### 3.2.2 More Details on the Concepts and Problems of this Subtheme

**Monotone Boolean Functions** An *m-dimensional Boolean function* is a function $Bool^m \rightarrow Bool$. It can be represented as a propositional formula. A Boolean function is **monotone** if it can be written as $\vee, \wedge$-combination of literals, where each variable occurs in only one polarity [219]. The class of monotone Boolean functions is a superset of the functions that can be represented as a single LPB, the so-called *threshold functions* [83]. One interesting problem, unsolved until today, is to give a closed form expression for the number of monotone Boolean functions of a given dimension. This is known as Dedekind's problem [144, 219].

**SAT Solving** *SAT solving* is the problem of deciding whether a propositional formula is satisfiable [165, 173, 226]. This problem has applications in verification and design automation concerning finite state systems. Several authors [18, 71, 96, 110, 111, 138] have focused on generalising techniques applied in CNF-based propositional satisfiability solving to LPBs, emphasising that this is beneficial because of the compactness of LPB representations: solving a problem based on such a compact representation can often be more efficient than based on a CNF or DNF encoding [45]. The following example illustrates the compactness.

**Example 3.1** The Boolean function represented by the DNF $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_1 \wedge x_5) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_3 \wedge x_4 \wedge x_5)$ can be represented more compactly by a single LPB: $4x_1 + 3x_2 + 2x_3 + 2x_4 + x_5 \geq 5$.

My work on LPBs was motivated by the application of SAT solving in bounded model checking within the AVACS project[7], discussed in Section 3.1.2 on page 26 (see also point 6 in Section 1.3). Via the application of SAT solving in bounded model checking, we can establish a link between linear pseudo-Boolean constraints and transition systems (see point 1 in Section 1.3).

**(Integer) Linear Programming** Linear programming is the problem of optimising a linear objective function subject to linear equality and inequality constraints [196]. A linear program is a problem that can be expressed in the following form:

$$\begin{array}{ll} \text{Maximize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ \text{where} & \mathbf{x} \geq 0 \end{array}$$

Here, $\mathbf{x}$ represents the vector of variables, while $\mathbf{c}$ and $\mathbf{b}$ are vectors of coefficients and $\mathbf{A}$ is a matrix of coefficients. Tools for solving linear programs include Cassowary [40] and CPLEX (by the ILOG company).

Whenever $\mathbf{x}$ is restricted to be a vector of *integers* rather than real numbers, one speaks of *integer linear programming.* This restriction makes the problem $\mathcal{NP}$-hard [57].

Whenever $\mathbf{x}$ is restricted further to be a vector of either 0 or 1, one speaks of 0-1 integer linear programming [18, 71]. A 0-1 linear constraint (i. e., one line of $\mathbf{Ax} \leq \mathbf{b}$) can be rewritten to an *LPB*, i. e. an expression of the form $a_1 \cdot l_1 + \ldots + a_m \cdot l_m \geq d$, where the $a_1, \ldots, a_m, d$ are natural numbers [45, 110].

The relationships between SAT Solving, pseudo-Boolean solving and (integer) linear programming have been studied by several authors [41, 45, 83, 96, 111, 162].

**Binary Decision Diagrams** A *binary decision diagram* (BDD) [99] is a directed acyclic graph where each node is labelled with a propositional variable and has two outgoing edges, the *0-edge* and the *1-edge.* Moreover, there are two nodes labelled 0 and 1, respectively, the so-called *value nodes.* BDDs are a formalism for representing Boolean functions, and they are often very concise. It is interesting to compare BDDs to the class of threshold functions. Hosaka et al. [139] have shown that there is a class of threshold functions for which the optimal BDD representation is of exponential size in the number of variables.

**"Typical" Linear Pseudo-Boolean Constraint Problems** The (not very explicit) assumption underlying the works advocating the LPB encoding [18, 71, 96, 110, 111] instead of the CNF encoding is that the problems, as they arise in an application domain, have a natural encoding as a set of LPBs, and that the CNF encoding would be larger. Linear pseudo-Boolean constraints are suited for expressing implications, which is useful for bounded model checking applications [110, 111]. Aloul et al. [18] mention the problems Min-Cover, Max-SAT, and MAX-ONEs. For example, Max-SAT is the problem of finding a variable assignment that maximises the number of satisfied clauses of an unsatisfiable SAT instance. Furthermore, applications from design automation [71], the pigeonhole problem [96], and gate level netlists [110] are mentioned as applications. Barth [45] mentions that LPBs arise in artificial intelligence applications [46].

However, apart from possibly the last work and one example given in the context of bounded model checking [111], in all these applications the LPBs are actually *cardinality constraints*, which are a restricted form of linear pseudo-Boolean constraints, namely $l_1 + \ldots + l_m \geq d$. Thus all those works do not clarify what the additional expressiveness of LPBs compared to cardinality constraints is useful for.

**The Threshold Recognition Problem** In my work [14, 15], I have suggested the possibility that LPBs could be used to encode arbitrary propositional formulae, not just the ones where the application dictates such an encoding in a straightforward way.

For this, any formula must be converted into a set of linear pseudo-Boolean constraints. This raises the fundamental question of whether a formula can be represented as a single linear pseudo-Boolean constraint, which is the so-called *threshold recognition problem* [83]. While this problem can be solved naïvely by formulating it as a linear programming problem (where the coefficients one searches for become the variables of the linear programming problem), it has been an open question studied for 30 years whether there is a more direct, combinatorial procedure for solving this problem. I have designed such a procedure [15].

# 4 Conclusion and Outlook

Since the research on logic programming presented in this habilitation (Section 2) has been conducted several years ago, I would now like to give an overview of some recent developments in logic programming. I am pleased to state that several of these developments touch upon topics related to the second main theme of this habilitation, treated in Section 3. That is to say, those recent works establish various links between, broadly speaking, logic programming and model checking.

## 4.1 Implementations of Model Checking in Logic Programming

Several authors have used the logic programming paradigm for implementing model checking tools. We have already discussed ARMC in Section 3.1.2. Nonnengart [181] has developed an approach to verifying hybrid systems, i. e., to showing that no unsafe state is reachable. He argues that explicit *forward reachability analysis*, while not computing redundant *information*, may perform redundant computations, namely when dealing with a hybrid system where trajectories pass through a location very often. In such cases, the author proposes to compute the behaviour of such a location once and for all and express it as a formula. The method is implemented (mainly) in Sicstus Prolog [214] with constraints.

Recently, *coinductive* logic programming has been proposed as a paradigm that can directly represent and verify properties of $\omega$-automata [125], i. e., automata that accept infinite words, a formalism which is in fact the basis for defining timed [21] and hybrid automata [130]. In coinductive logic programming, proofs (computations) are infinite in theory, which means that results of computations can be output only if those results exhibit a certain regularity. Just as finite automata can be programmed using standard logic programming, $\omega$-automata can be programmed using coinductive logic programming. Properties such as *safety* (a certain location is never visited) or *liveness* (a certain location is visited infinitely often) can then be checked by the execution of the coinductive logic program in a natural way. The authors have applied this approach also to timed automata, where the clocks are implemented using real numbers and arithmetic constraint programming in addition to logic programming. The authors conjecture that adding *negation* to coinductive logic programming, one would obtain something resembling answer set programming, see Section 4.3.

## 4.2 The Web

In the last years, logic programming applications concerning the World Wide Web, in particular, the *Semantic Web* [52, 53, 108], have received much attention [123, 223]. Wielemaker et al. [223] have proposed an approach where an entire HTTP server is implemented in SWI-Prolog. The features of logic programming that prove most valuable in this context are: the safe semantics and automatic memory management (absence of pointers), and the nondeterminism. However, the application to Web services also points to some desirable extensions to the ISO-Prolog standard [92]. One extension is support for concurrency, necessary mainly due to possible network delays causing a single transaction to take a long time. Another extension is the use of arbitrary Unicode [222] characters in Prolog atoms, to enable Prolog atoms to represent arbitrary text. It should also be pointed out that for certain subtasks that turned out to be performance bottlenecks, the HTTP server uses an implementation in C rather than SWI-Prolog. The applications of this HTTP server include a Prolog-powered system for ontology management and reasoning based on *description logics* [38]. The description logic reasoning services are provided using a combination of reasoning implemented in Prolog and performed on Prolog terms, and calls to external description logic servers.

The Semantic Web [52, 53, 108] develops standards and technologies to help machines understand the information on the Web to support richer discovery, data integration, navigation, and automation [103]. It has several layers, the highest being the Ontology layer in the form of the OWL language based on description logics [38]. To encode reasoning tasks in description logics for use in the Semantic Web, *answer set programming* has been used [44], leading us to the next important topic of recent logic programming research.

## 4.3 Answer Set Programming

In recent years, *answer set programming* (ASP) has become one of the most popular topics in the field of logic programming. ASP has its root in logic programming and non-monotonic reasoning [117, 118, 155]. It is well-suited for modelling and solving problems involving common-sense reasoning [103]. The ASP paradigm constitutes a shift from theorem proving (proof-finding) to constraint programming (model-finding) [67, 220].

The ASP semantics deals with negation in clause (*rule* in ASP terminology) bodies of logic programs. The semantics is based on the Gelfond-Lifschitz transformation [117, 118]: assuming a literal set $A$, one removes all rules having an atom from $A$ negated in their body. Afterwards one removes all negated atoms from all rule bodies. The set $A$ is an *answer set* if the smallest model (see footnote 3 on page 13) of this transformed program is exactly $A$. The ASP semantics is the definitive solution to the negation-in-bodies problem within the class of 2-valued semantics [220]. The definition of answer sets was motivated by the aim of providing a declarative semantics of negation-as-failure [155].

Unlike usual logic programming, most ASP implementations do not support function symbols in the logic language. *Finitary* and *finitely recursive programs* were introduced in order to allow function symbols (hence infinite domains) and recursion in ASP to some extent. Restrictions are imposed that ensure that there are only finitely many potential sources of inconsistency [47, 58].

### 4.3.1 ASP Solvers

Several ASP solvers have been proposed, e. g. Smodels [197], DLV [154], nomore++ [22], noMoRe [158], ASSAT [156], and Cmodels [121]. In order to provide a formalism for systematic comparison between ASP solvers, a *tableau calculus* has been developed [115] and later generalised [116]. One of the key technicalities of this formalism is to define an assignment not only for single propositional atoms, but also for entire clause bodies. Existing ASP solvers can then be characterised by specifying a subset of the tableau rules.

Some ASP solvers such as ASSAT [156] and Cmodels [122] are based on SAT solvers, see Section 3.2. The SAT solver is used as a model generator. It is then checked whether the model contains an unfounded loop. Interleaving these phases is an important aspect of improving the performance [157].

Smodels [197] is an *atom-based* solver, i. e., it does not talk about the truth or falsity of an entire body explicitly, but only about the truth or falsity of single atoms.

The essential difference between ASP solvers lies in the use of the *cut*, i. e., an explicit case distinction for the assignment of a variable. One can formally show that applying the cut exclusively to single atoms (as in Smodels) is not superior to applying it exclusively to entire rule bodies (as in noMoRe), and vice versa, while a combination of both (as in nomore++) is exponentially stronger.

As pointed out by Demoen [90], ASP has not yet reached a high level of maturity when it comes to the implementation of the solvers, and he has referred to the famous formula by Kowalski (see Section 2.1.2), stressing that it is the “Control” part that still needs improvement. Recall that the "Control" issue is also considered in this habilitation, however in the context of logic programming (see Section 2.1).

### 4.3.2 Applications

ASP provides a form of declarative programming well-suited to difficult combinatorial search problems [66, 155]. Applications include data integration, configuration, diagnosis, text mining, reasoning about actions and change [103].

Arguably, ASP is still lacking *large-scale* applications. However, one of the first large-scale applications, as claimed by the authors of the application, is generating optimal code using superoptimisation [66].

Another application is planning [155, 221]. I emphasise this application here because planning is a search problem whose solution techniques have inspired heuristics for directed model checking [148, 149], one of the themes of this habilitation work, see Section 3.1. Let us review the approach by Lifschitz [155]. The non-monotonic character of negation-as-failure, a logic programming feature, makes logic programming particularly suited for representing properties of actions. When modelling planning tasks in ASP, it is helpful to view the rules under the aspect of whether they generate *multiple* answer sets, or whether they *eliminate* some of the answer sets. The approach is demonstrated using *Blocksworld*, a standard planning domain where the task is to stack building blocks using a robot arm. The ASP rules are structured into those that describe the effects of an action, those that state certain side conditions, and those formulating a particular problem instance. There is a notion of *generating* some action sequences and then *testing* them in order to eliminate the ones that are impossible or do not lead to the goal state. For efficient execution of such an ASP program, the scheduling of the generating and testing is an important issue (see the paragraph at the end of Section 4.3.1).

ASP has also been applied in bioinformatics. We will now turn to this field.

## 4.4 Bioinformatics

Bioinformatics deals with biological systems simulations and with prediction of the spatial conformation of a biological polymer, among others.

Modelling of biological networks has been approached using ASP [100, 124], see Section 4.3. While ASP has been related to *action* languages [119], in the biological context the notion of *re*action is more adequate, but this can be addressed by adaptations to the language.

### 4.4.1 Systems Biology

The work by Fages and Soliman on *systems biology* [106, 107] ties together several concepts and topics that play important roles in this habilitation work as well. Systems biology aims at elucidating the high-level functions of the cell from their biochemical basis at the molecular level. A central issue in systems biology is studying the behaviour of biochemical systems at different levels of abstraction.

The biochemical basis at the molecular level is given in the form of chemical reaction rules à la $H_2O + CO_2 \rightarrow H_2CO_3$, which are written in a particular format called *SBML* (systems biology markup language). Given a set of such rules, the behaviour of a biological system can be interpreted using at least three different semantics, which are all interpreted using the *BIOCHAM* abstract machine, implemented in Prolog.

One semantics is the *stochastical semantics* where the transitions connect discrete states representing the concentrations of each molecule in the system, and each transition is labelled with a probability. Next there is the discrete semantics where one abstracts from the probabilities and just keeps the information of whether a transition is possible at all. Finally there is the Boolean semantics where one further abstracts from the precise quantities of the molecules and just keeps the information of whether a molecule is present at all. It has been shown that these semantics correspond to different levels of abstraction in the sense of *abstract interpretation* [82, 106], a technique I have used for groundness analysis of logic programs [9], see Section 2.2 on pages 16 and 19.

Biologists assign *functions*, i. e., high-level behaviour descriptions, to reaction rules, such as *kinase* or *phosphotase*. Fages and Soliman propose to interpret these functions as the programming-language concept of a *type*, which plays a prominent role in this habilitation work, see Section 2.2. Analysis of the function of a reaction rule then boils down to type inference. In order to refine the functions, the authors even use subtyping in the sense of my joint work with Fages [6], see pages 15 and 17.

In another work [107], the authors use *temporal logic* to formalise biological properties known from experiments. The question is similar as in model checking, see Section 1.2.2: Given a set of reaction rules as above, does this set describe, i. e., *is it a model of*, the experimental observations? If not, the model is *revised*, using *inductive logic programming* [176] techniques. Inductive logic programming is an approach for generating logical formulae (here: reaction rules) from data (here: the temporal formulae).

### 4.4.2 The Protein Structure Prediction Problem

The *primary structure* of a protein is a lined sequence of aminoacids. Ignoring the secondary structure here, the *tertiary structure* is a three-dimensional conformation associated to the primary structure. The *protein structure prediction* problem is the problem of determining foldings with minimal energy. For this one uses *lattice models*, which are three-dimensional graphs with repeated patterns, e. g., the so-called *face-centred cube*.

Backofen and Will [39] solved the problem using constraint programming for proteins of length 160 and more on the face-centred cube. Efficiency is obtained by sophisticated symmetry detection.

Dovier and his coauthors have also approached this problem using logic and constraint programming wherever possible [84, 85, 97].

They use a CLP(FD) (constraint logic programming over finite domains) encoding [84]. For efficiency reasons, the approach of *constrain-and-generate* is used, much like the test-and-generate paradigm [180] mentioned in Example 2.5. In addition, the efficiency can be enhanced using an ad-hoc labelling search with biologically motivated heuristics.

Dovier et al. have also analysed the folding process using model checking results [85] and planning [97], concepts which are relevant for this habilitation work as well, see Section 3.1.

## 4.5 Concluding Remarks and Outlook

This habilitation work contains contributions in quite diverse areas of computer science. Yet, these contributions are not isolated from each other. There are some common aims and techniques, which I listed in Section 1.3 and pointed out throughout Sections 2 and 3 as they became manifest. The above study of the literature has shown some further connections and combinations of ideas and techniques from different fields. In particular, I would like to emphasise the use of predicate abstraction for showing termination by Podelski and Rybalchenko [186, 187], and the links established between model checking and planning on the one hand and logic programming and type systems on the other [85, 97, 106, 107], in the context of bioinformatics.

I conclude this summary with some directions for future work:

- Many open questions remain in the study of linear pseudo-Boolean constraints: What exactly is the relationship to binary decision diagrams? What is the complexity of the algorithm for threshold recognition [15]? How can one generate a *minimal* linear pseudo-Boolean constraint for a threshold function? How can an arbitrary Boolean function be represented compactly as a set of linear pseudo-Boolean constraints?

- Concerning the problem of finding error trajectories of hybrid systems [11], I have recently done some work pursuing the same aim, but for *deterministic* systems [191]. In this work, the problem of finding an error trajectory is viewed as the problem of optimising a numerical quality function for trajectories. There are some relationships to heuristic search (hence to the work I have done for timed automata) and to reinforcement learning [213], but these need to be developed further.

- Concerning directed model checking of timed automata, it would be interesting to compare the heuristic definition formalisms proposed within the AVACS project [12, 13, 98, 148, 149] with each other on a theoretical level, and to generalise those formalisms to systems other than timed automata. The hope is that such a study would reveal some fundamental results about the strengths and weaknesses of heuristic definition formalisms, possibly revealing that some formalism is strictly more powerful than some other formalism.

- On the more visionary side, there is one issue which at present I can only describe at a high level of abstraction but which in my view would be highly interesting to explore further:

  My works on hybrid automata on the one hand [11, 191] and timed automata on the other hand [12, 13] are very different on the technical level, both regarding the exact formulation of the problem and regarding the techniques used for solving the problem, yet they have one aspect in common: in both cases, an abstraction technique originally conceived for the purpose of verification is used for providing guidance in the search for an error trajectory, i.e., for falsification (see Section 1.2.2 on page 7, Section 3.1.1, and page 25). I am not aware of any other works that have integrated verification and falsification under this aspect.

  However, at present, I cannot describe a general principle underlying the use of verification-inspired abstraction techniques for falsification. Under which circumstances are such techniques useful for falsification? In which ways should abstractions used for falsification be different from abstractions used for verification?[8] What does this mean for approaches that interleave verification and falsification attempts: is it good to use the same abstraction for both tasks [191], or is there some better approach? Giving answers to these questions might be one key in designing better abstractions for falsification of transition systems, thus enhancing the performance of falsification algorithms.

[8]A first, preliminary investigation of this question [13] followed the hypothesis that they should be different in a certain respect, but found no strong support for this hypothesis.

# References for the Selected Articles

## Logic Programming and Functional Programming

### Termination and Selection Rules

[1] Annalisa Bossi, Sandro Etalle, Sabina Rossi, and Jan-Georg Smaus. Semantics and termination of simply-moded logic programs with dynamic scheduling. In David Sands, editor, *Proceedings of the European Symposium on Programming*, volume 2028 of *LNCS*, pages 402–416. Springer-Verlag, 2001.

[2] Annalisa Bossi, Sandro Etalle, Sabina Rossi, and Jan-Georg Smaus. Termination of simply moded logic programs with dynamic scheduling. *Transactions on Computational Logic*, 5(3):470–507, 2004.

[3] Jan-Georg Smaus. Termination of logic programs using various dynamic selection rules. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *LNCS*, pages 43–57. Springer-Verlag, 2004.

[4] Dino Pedreschi, Salvatore Ruggieri, and Jan-Georg Smaus. Classes of terminating logic programs. *Theory and Practice of Logic Programming*, 2(3):369–418, 2002.

[5] Dino Pedreschi, Salvatore Ruggieri, and Jan-Georg Smaus. Characterisations of termination in logic programming. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, volume 3049 of *LNCS*, pages 376–431. Springer-Verlag, 2004.

### Typed Programs

[6] Jan-Georg Smaus, François Fages, and Pierre Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In Sanjiv Kapoor and Sanjiva Prasad, editors, *Proceedings of the 20th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *LNCS*, pages 214–226. Springer-Verlag, 2000.

[7] Pierre Deransart and Jan-Georg Smaus. Subject reduction of logic programs as proof-theoretic property. *Journal of Functional and Logic Programming (electronic journal)*, 2002(2), 2002.

[8] Jan-Georg Smaus. The head condition and polymorphic recursion. In Zhenjiang Hu and Mario Rodríguez-Artalejo, editors, *Proceedings of the 6th International Symposium on Functional and Logic Programming*, volume 2441 of *LNCS*, pages 259–274. Springer-Verlag, 2002.

[9] Jan-Georg Smaus. Analysis of polymorphically typed logic programs using ACI-unification. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *LNAI*, pages 280–295. Springer-Verlag, 2001.

### "Formulas as Programs"

[10] Jan-Georg Smaus. Is there an optimal generic semantics for first-order equations? In Catuscia Palamidessi, editor, *Proceedings of the 19th International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 438–450. Springer-Verlag, 2003.

## Logic Approaches to Model Checking of Timed and Hybrid Systems

### Finding Error Paths in Timed and Hybrid Systems

[11] Stefan Ratschan and Jan-Georg Smaus. Verification-integrated falsification of non-deterministic hybrid systems. In Christos G. Cassandras, Alessandro Giua, Carla Seatzu, and Janan Zaytoon, editors, *Proceedings of the 2nd (2006) IFAC Conference on Analysis and Design of Hybrid Systems*, pages 371–376. Elsevier Science Inc., 2007.

[12] Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in UPPAAL. In Stefan Edelkamp and Alessio Lomuscio, editors, *Post-Proceedings of the 4th (2006) Workshop on Model Checking and Artificial Intelligence*, volume 4428 of *LNCS*, pages 51–66. Springer-Verlag, 2007.

[13] Jan-Georg Smaus and Jörg Hoffmann. Relaxation refinement: A new method to generate heuristic functions. In Doron Peled and Michael Wooldridge, editors, *Post-Proceedings of the 5th (2008) Workshop on Model Checking and Artificial Intelligence*, LNCS. Springer-Verlag, 2009. To appear.

### Linear Pseudo-Boolean Constraints

[14] Jan-Georg Smaus. Representing Boolean functions as linear pseudo-Boolean constraints. In Youssef Hamadi, editor, *Proceedings of the CP 2006 Workshop on the Integration of SAT and CP techniques*, pages 49–63, 2006.

[15] Jan-Georg Smaus. On Boolean functions encodable as a single linear pseudo-Boolean constraint. In Pascal Van Hentenryck and Laurence Wolsey, editors, *Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 4510 of *LNCS*, pages 288–302. Springer-Verlag, 2007.

## Further References

[16] E. Ábrahám, B. Becker, F. Klaedtke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. In R. Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 396–412. Springer-Verlag, 2005.

[17] H. Aljazzar, H. Hermanns, and S. Leue. Counterexamples for timed probabilistic reachability. In P. Pettersson and W. Yi, editors, *Proceedings of the 3rd International Conference on Formal Modeling and Analysis of Timed System*, volume 3829 of *LNCS*, pages 177–195. Springer-Verlag, 2005.

[18] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In L. T. Pileggi and A. Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 450–457. ACM, 2002.

[19] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[20] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Proceedings of the 1992 Hybrid Systems Conference*, volume 736 of *LNCS*, pages 209–229. Springer, 1993.

[21] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[22] C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The nomore++ approach to answer set solving. In G. Sutcliffe and A. Voronkov, editors, *Proceeding of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *LNCS*, pages 95–109. Springer-Verlag, 2005.

[23] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[24] K. R. Apt. A denotational semantics for first-order logic. In J. Lloyd et al., editors, *Proceedings of the 1st International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 53–69. Springer-Verlag, 2000.

[25] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 29(3):335–363, 1991.

[26] K. R. Apt and M. Bezem. Formulas as programs. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25 Years Perspective*, pages 75–107. Springer-Verlag, 1999.

[27] K. R. Apt, J. Brunekreef, A. Schaerf, and V. Partington. Alma-0: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems*, 20(5):1014–1066, 1998.

[28] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, pages 1–19. Springer-Verlag, 1993.

[29] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology*, volume 936 of *LNCS*, pages 66–90. Springer-Verlag, 1995. Invited Lecture.

[30] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[31] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[32] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[33] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Transactions on Programming Languages and Systems*, 16(3):687–726, 1994.

[34] K. R. Apt and J.-G. Smaus. Rule-based versus procedure-based view of logic programming. *Joint Bulletin of the Novosibirsk Computing Center and Institute of Informatics Systems; Series: Computer Science*, 16:75–97, 2001.

[35] K. R. Apt and C. F. M. Vermeulen. First-order logic as a constraint programming language. In M. Baaz and A. Voronkov, editors, *Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2514 of *LNCS*, pages 19–35. Springer-Verlag, 2002.

[36] T. Armstrong, K. Marriott, P. Schachte, and H.Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[37] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proceedings of the 1st Static Analysis Symposium*, volume 864 of *LNCS*, pages 266–280. Springer-Verlag, 1994.

[38] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[39] R. Backofen and S. Will. A constraint-based aproach to fast and exact structure prediction in three-dimensional protein models. *Constraints*, 11(1):5–30, 2006.

[40] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction*, 8(4):267–306, 2001.

[41] O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of pseudo Boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:191–200, 2006.

[42] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 36 of *SIGPLAN Notices*, pages 203–213, 2001.

[43] T. Ball, A. Podelski, and S. K. Rajamani. Completeness of abstraction refinement for software model checking. In J.-P. Katoen and P. Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*. Springer-Verlag, 2002.

[44] C. Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.

[45] P. Barth. Linear 0-1 inequalities and extended clauses. In A. Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 40–51. Springer-Verlag, 1993.

[46] P. Barth and A. Bockmayr. Solving 0-1 problems in CLP(PB). In *Proceedings of the 9th Conference on Artificial Intelligence for Applications*. IEEE, 1993.

[47] S. Baselice, P. A. Bonatti, and G. Criscuolo. On finitely recursive programs. In V. Dahl and I. Niemelä, editors, *Proceedings pf the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, pages 89–103. Springer-Verlag, 2007.

[48] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Revised Lectures on Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 200–236, 2004.

[49] C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 765–779. MIT Press, 1995.

[50] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets 2003*, volume 3098 of *LNCS*, pages 87–124. Springer-Verlag, 2004.

[51] F. Benoy, M. Codish, A. Heaton, and A. M. King. Widening *Pos* for Efficient and Scalable Groundness Analysis. Technical Report 515, University of Kent at Canterbury, 1997. Available at `http://www.cs.ukc.ac.uk/pubs/1997/515/index.html`.

[52] T. Berners-Lee. *Weaving the Web.* Harper, 1999.

[53] T. Berners-Lee, J. A. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284:34–43, 2001.

[54] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–97, 1993.

[55] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[56] R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings of the Conference on Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.

[57] A. Bockmayr and V. Weispfenning. *Handbook of Automated Reasoning*, chapter Chapter 12: Solving Numerical Constraints. Elsevier, 2001.

[58] P. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.

[59] A. Bossi and N. Cocco. Successes in logic programs. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic Program Synthesis and Transformation*, volume 1559 of *LNCS*, pages 219–239. Springer-Verlag, 1999.

[60] A. Bossi, N. Cocco, S. Etalle, and S. Rossi. Declarative semantics of input consuming logic programs. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, volume 3049 of *LNCS*, pages 90–114. Springer-Verlag, 2004.

[61] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Comput. Sci.*, 124(2):297–328, 1994.

[62] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. In S. Etalle and J.-G. Smaus, editors, *Proc. of the ICLP Workshop on Verification of Logic Programs*, volume 30(1) of *Electronic Notes in Theoretical Computer Science*, 1999.

[63] A. Bossi, S. Etalle, and S. Rossi. Semantics of input-consuming logic programs. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Proceedings of the 1st International Conference on Computational Logic*, volume 1861 of *LNCS*, pages 194–208. Springer-Verlag, 2000.

[64] A. Bossi, S. Etalle, and S. Rossi. Semantics of well-moded input-consuming logic programs. *Computer Languages*, 26(1):1–25, 2000.

[65] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming*, 2(2):125–154, 2002.

[66] M. Brain, T. Crick, M. D. Vos, and J. Fitch. TOAST: Applying answer set programming to superoptimisation. In S. Etalle and M. Truszczyński, editors, *Proceedings of the 22nd International Conference on Logic Programming*, volume 4079 of *LNCS*, pages 270–284. Springer-Verlag, 2006.

[67] G. Brewka. Preferences, contexts and answer sets. In V. Dahl and I. Niemelä, editors, *Proceedings pf the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, page 22. Springer-Verlag, 2007.

[68] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. R. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[69] M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. In M. V. Hermenegildo and G. Puebla, editors, *Proceedings of the 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, pages 477–492. Springer-Verlag, 2002.

[70] M. Bruynooghe, W. Vanhoof, and M. Codish. Pos(T): Analyzing dependencies in typed logic programs. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Proceedings og the 4th International Andrei Ershov Memorial Conference, Revised Papers*, volume 2244 of *LNCS*, pages 406–420. Springer-Verlag, 2001.

[71] D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. In *Proceedings of the 40th Design Automation Conference*, pages 830–835. ACM, 2003.

[72] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering*, pages 385–395. IEEE Computer Society, 2003.

[73] E. Clarke, A. Gupta, and O. Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer Aided Design*, 23(7):1113–1123, 2004.

[74] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2002. 5th print.

[75] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. In L. Naish, editor, *Proceedings of the 14th Joint International Conference and Symposium on Logic Programming.* MIT Press, 1997. Presented as poster.

[76] M. Codish, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Exploiting goal independence in the analysis of logic programs. *Journal of Logic Programming*, 32(3):247–261, 1997.

[77] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–125, 1994.

[78] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of *Prop*. In B. Le Charlier, editor, *Proceedings of the 1st Static Analysis Symposium*, volume 864 of *LNCS*, pages 281–296. Springer-Verlag, 1994.

[79] M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.

[80] M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. Goal dependent versus goal independent analysis of logic programs. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *LNCS*, pages 305–319. Springer-Verlag, 1994.

[81] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238(1–2):131–159, 2000.

[82] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[83] Y. Crama and P. L. Hammer. *Boolean Functions – Theory, Algorithms, and Applications.* Cambridge University Press, 2008. To appear.

[84] A. Dal Palù, A. Dovier, and F. Fogolari. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics*, 5:186, 2004.

[85] E. De Maria, A. Dovier, A. Montanari, and C. Piazza. Exploiting model checking in constraint-based approaches to the protein folding. In *Proceedings of the Workshop on Constraint Based Methods for Bioinformatics*, pages 46–54, 2006.

[86] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

[87] S. K. Debray and D. S. Warren. Detection and optimization of functional computations in Prolog. In E. Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, LNCS, pages 490–504. Springer-Verlag, 1986.

[88] S. Decorte and D. De Schreye. Termination analysis: Some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proceedings of the 15th Joint International Conference and Symposium on Logic Programming*, pages 235–249. MIT Press, 1998.

[89] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. F. Sagonas. Termination analysis for tabled logic programming. In N. E. Fuchs, editor, *Proceedings of the 7th International Workshop on Logic Programming Synthesis and Transformation*, volume 1463 of *LNCS*, pages 111–127. Springer-Verlag, 1998.

[90] B. Demoen. My life as a Prolog implementor. In *Newsletter*. Association for Logic Programming, February 2008. Available from `http://www.logicprogramming.org/newsletter/`.

[91] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer-Verlag, 1999.

[92] P. Deransart, A. A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[93] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* The MIT Press, 1993.

[94] H. Dierks, S. Kupferschmid, and K. G. Larsen. Automatic abstraction refinement for timed automata. In J.-F. Raskin and P. S. Thiagarajan, editors, *Proceedings of the 5th International Conference on Formal Modelling and Analysis of Timed Systems*, volume 4763 of *LNCS*, pages 114–129. Springer-Verlag, 2007.

[95] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming*, volume 300 of *LNCS*, pages 79–93. Springer-Verlag, 1988.

[96] H. E. Dixon and M. L. Ginsberg. Combining satisfiability techniques from AI and OR. *The Knowledge Engineering Review*, 15:31–45, 2000.

[97] A. Dovier, A. Formisano, and E. Pontelli. Multivalued action languages with constraints in CLP(FD). In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, pages 255–270. Springer-Verlag, 2007.

[98] K. Dräger, B. Finkbeiner, and A. Podelski. Directed model checking with distance-preserving abstractions. In A. Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software*, volume 3925 of *LNCS*, pages 19–34. Springer-Verlag, 2006.

[99] R. Drechsler and B. Becker. *Binary Decision Diagrams - Theory and Implementation.* Kluwer Academic Publishers, 1998.

[100] S. Dworschak, S. Grell, V. J. Nikiforova, T. Schaub, and J. Selbig. Modeling biological networks by action languages via answer set programming. *Constraints*, 13(1-2):21–65, 2008.

[101] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2-3):247–267, 2004.

[102] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology*, 6(4):277–301, 2004.

[103] T. Eiter. Answer set programming for the Semantic Web. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, pages 23–26. Springer-Verlag, 2007.

[104] S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.

[105] S. Etalle and M. Gabbrielli. Layered modes. *Journal of Logic Programming*, 39:225–244, 1999.

[106] F. Fages and S. Soliman. Abstract interpretation and types for systems biology. *Theoretical Computer Science*, 403(1):52–70, 2008.

[107] F. Fages and S. Soliman. Model revision from temporal logic properties in computational systems biology. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *LNCS*, pages 287–304. Springer-Verlag, 2008.

[108] D. Fensel, W. Wahlster, H. Lieberman, and J. A. Hendler, editors. *Spinning the Semantic Web: Bringing the World Wide Web to its Full Potential.* MIT Press, 2002.

[109] M. Fränzle and M. R. Hansen. Deciding an interval logic with accumulated durations. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 201–215. Springer-Verlag, 2007.

[110] M. Fränzle and C. Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In M. Y. Vardi and A. Voronkov, editors, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850 of *LNCS*, pages 302–316. Springer-Verlag, 2003.

[111] M. Fränzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.

[112] T. Gabric, K. Glynn, and H. Søndergaard. Strictness analysis as finite-domain constraint solving. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation*, volume 1559 of *LNCS*, pages 255–270. Springer-Verlag, 1998.

[113] J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *LNCS*, pages 27–42. Springer-Verlag, 2004.

[114] J. P. Gallagher and A. d. Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[115] M. Gebser and T. Schaub. Tableau calculi for answer set programming. In S. Etalle and M. Truszczyński, editors, *Proceedings of the 22nd International Conference on Logic Programming*, volume 4079 of *LNCS*, pages 11–25. Springer-Verlag, 2006.

[116] M. Gebser and T. Schaub. Generic tableaux for answer set programming. In V. Dahl and I. Niemelä, editors, *Proceedings pf the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, pages 119–133. Springer-Verlag, 2007.

[117] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[118] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[119] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2/3&4):301–321, 1993.

[120] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Proceedings of the 15th International Symposium on Protocol Specification, Testing, and Verification*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1996.

[121] E. Giunchiglia, Y. Lierler, and M. Maratea. A SAT-based polynomial space algorithm for answer set programming. In J. P. Delgrande and T. Schaub, editors, *Proceedings of the 10th 10th International Workshop on Non-Monotonic Reasoning*, pages 189–196, 2004.

[122] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[123] D. C. Gras and M. V. Hermenegildo. Distributed WWW programming using (Ciao-)Prolog and the PiLLoW library. *Theory and Practice of Logic Programming*, 1(3):251–282, 2001.

[124] S. Grell, T. Schaub, and J. Selbig. Modelling biological networks by action languages via answer set programming. In S. Etalle and M. Truszczyński, editors, *Proceedings of the 22nd International Conference on Logic Programming*, volume 4079 of *LNCS*, pages 285–299. Springer-Verlag, 2006.

[125] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, pages 27–44. Springer-Verlag, 2007.

[126] M. Hanus. *Logic Programming with Type Specifications*, chapter 3, pages 91–140. MIT Press, 1992. In [185].

[127] A. Heaton, M. Abo-Zaed, M. Codish, and A. King. A simple polynomial groundness analysis for logic programs. *Journal of Logic Programming*, 45(1-3):143–156, 2000.

[128] A. Heaton, P. M. Hill, and A. King. Analysing logic programs with delay. In N. E. Fuchs, editor, *Proceedings of the 7th International Workshop on Logic Program Synthesis and Transformation*, volume 1463 of *LNCS*. Springer-Verlag, 1998.

[129] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[130] T. A. Henzinger. The theory of hybrid automata. *Verification of Digital and Hybrid Systems*, 170:265–292, 2000.

[131] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):540–554, 1998.

[132] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st Symposium on Principles of Programming Languages*, pages 232–244. ACM, 2004.

[133] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[134] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(1-4):349–366, 1992.

[135] P. M. Hill. The completion of typed logic programs and SLDNF-resolution. In A. Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 182–193. Springer-Verlag, 1993.

[136] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[137] P. M. Hill and R. W. Topor. *A Semantics for Typed Logic Programs*, chapter 1, pages 1–61. MIT Press, 1992. In [185].

[138] J. N. Hooker. Generalized resolution for 0-1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, 6(1-3):271–286, 1992.

[139] K. Hosaka, Y. Takenaga, T. Kaneda, and S. Yajima. Size of ordered binary decision diagrams representing threshold functions. *Theoretical Computer Science*, 180(1-2):47–60, 1997.

[140] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992. First author name erroneously spelt "Janssen".

[141] S. Kahrs. Limits of ML-definability. In H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th Symposium on Programming Language Implementations and Logic Programming*, volume 1140 of *LNCS*, pages 17–31. Springer-Verlag, 1996.

[142] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993. Title wrongly given in table of contents: Type *recursion* in the presence of polymorphic recursion.

[143] A. King, J.-G. Smaus, and P. M. Hill. Quotienting *Share* for dependency analysis. In D. Swierstra, editor, *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *LNCS*, pages 59–73. Springer-Verlag, 1999.

[144] D. Kleitman and G. Markowsky. On Dedekind's problem: the number of isotone Boolean functions. II. *Transactions of the American Mathematical Society*, 213:373–390, 1975.

[145] R. A. Kowalski. Algorithm = Logic + Control. *Commun. ACM*, 22(7):424–436, 1979.

[146] M. Krishna Rao, D. Kapur, and R. K. Shyamasundar. Proving termination of GHC programs. *New Generation Computing*, 15(3):293–338, 1997.

[147] M. R. K. Krishna Rao. Input-termination of logic programs. In S. Etalle, editor, *Proceedings of the 14th International Symposium on Logic Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 215–230. Springer-Verlag, 2005.

[148] S. Kupferschmid, K. Dräger, J. Hoffmann, B. Finkbeiner, H. Dierks, A. Podelski, and G. Behrmann. Uppaal/DMC – Abstraction-based heuristics for directed model checking. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 679–682. Springer-Verlag, 2007.

[149] S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In A. Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software*, volume 3925 of *LNCS*, pages 35–52. Springer-Verlag, 2006.

[150] V. Lagoon, F. Mesnard, and P. Stuckey. Termination analysis with types is more accurate. In C. Palamidessi, editor, *Proceedings of the 19th International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 254–268. Springer-Verlag, 2003.

[151] V. Lagoon and P. J. Stuckey. A framework for analysis of typed logic programs. In H. Kuchen and K. Ueda, editors, *Proceedings of the 5th International Symposium on Functional and Logic Programming*, volume 2024 of *LNCS*, pages 296–310. Springer-Verlag, 2001.

[152] T. K. Lakshman and U. S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[153] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[154] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[155] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[156] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[157] Z. Lin, Y. Zhang, and H. Hernandez. Fast SAT-based answer set solver. In *Proceedings on the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference*. AAAI Press, 2006.

[158] T. Linke, C. Anger, and K. Konczak. More on noMoRe. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the European Conference on Logics in Artificial Intelligence*, volume 2424 of *LNCS*, pages 468–480. Springer-Verlag, 2002.

[159] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[160] P. Louvet and O. Ridoux. Parametric polymorphism for Typed Prolog and $\lambda$Prolog. In H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th Symposium on Programming Language Implementations and Logic Programming*, volume 1140 of *LNCS*, pages 47–61. Springer-Verlag, 1996.

[161] S. Lüttringhaus-Kappel. Control generation for logic programs. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 478–495. MIT Press, 1993.

[162] V. M. Manquinho and O. Roussel. The first evaluation of pseudo-Boolean solvers (PB'05). *Journal on Satisfiability, Boolean Modeling and Computation*, 2:103–143, 2006.

[163] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. W. Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 447–461. MIT Press, 1995.

[164] E. Marchiori and F. Teusink. On termination of logic programs with delay declarations. *Journal of Logic Programming*, 39(1-3):95–124, 1999.

[165] J. P. Marques Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[166] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[167] J. C. Martin and A. King. Generating efficient, terminating logic programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of the 7th International Joint Conference on Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 273–284. Springer-Verlag, 1997.

[168] J. C. Martin, A. M. King, and P. Soper. Typed norms for typed logic programs. In J. P. Gallagher, editor, *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, LNCS, pages 224–238. Springer-Verlag, 1997.

[169] K. L. McMillan. Applications of Craig interpolants in model checking. In N. Halbwachs and L. D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 1–12. Springer-Verlag, 2005.

[170] L. Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.

[171] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[172] R. E. Moore. *Interval Analysis*. Prentice Hall, 1966.

[173] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM, 2001.

[174] L. M. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 438–455. Springer-Verlag, 2002.

[175] L. M. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category A). In W. Hunt, Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification*, volume 2725 of *LNCS*, pages 14–26. Springer-Verlag, 2003.

[176] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[177] A. Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the 6th International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer-Verlag, 1984.

[178] A. Mycroft and R. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[179] G. Nadathur and F. Pfenning. *Types in Higher-Order Logic Programming*, chapter 9, pages 245–283. MIT Press, 1992. In [185].

[180] L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.

[181] A. Nonnengart. Hybrid systems verification by location elimination. In N. A. Lynch and B. H. Krogh, editors, *Proceedings of the 3rd International Workshop on Hybrid Systems*, volume 1790 of *LNCS*, pages 352–365. Springer-Verlag, 2000.

[182] C. Okasaki. Catenable double-ended queues. In *Proceedings of the International Conference on Functional Programming*, volume 32(8) of *SIGPLAN Notices*, pages 66–74. ACM Press, 1997.

[183] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[184] D. Pedreschi and S. Ruggieri. Bounded nondeterminism of logic programs. *Annals of Mathematics and Artificial Intelligence*, 42(4):313–343, 2004.

[185] F. Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.

[186] A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings of the 19th Symposium on Logic in Computer Science*, pages 32–41. IEEE, 2004.

[187] A. Podelski and A. Rybalchenko. Transition predicates and fair termination. In M. Abadi, editor, *Conference Record of the 32nd ACM Symposium on Principles of Programming Languages*, pages 132–144, 2005.

[188] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In M. Hanus, editor, *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 245–259. Springer-Verlag, 2007.

[189] S. Ratschan and Z. She. Constraints for continuous reachability in the verification of hybrid systems. In *Proceedings of the 8th International Conference on Artificial Intelligence and Symbolic Computation*, volume 4120 of *LNCS*, pages 196–210. Springer-Verlag, 2006.

[190] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Transactions on Embedded Computing Systems*, 6(1), 2007.

[191] S. Ratschan and J.-G. Smaus. Finding errors of hybrid systems by optimising an abstraction-based quality estimate. In T. Uustalu and J. Vain, editors, *Proceedings of the 20th Nordic Workshop on Programming Theory*, 2009. To appear.

[192] A. W. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall, 1997.

[193] S. Ruggieri. *Verification and Validation of Logic Programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[194] S. Ruggieri. ∃-universal termination of logic programs. *Theoretical Comput. Sci.*, 254(1-2):273–296, 2001.

[195] H. Sağlam and J. P. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. Technical Report CSTR-95-017, University of Bristol, 1995. Presented as a poster at the 7th Symposium on Programming Language Implementations and Logic Programming.

[196] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[197] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[198] J.-G. Smaus. *Modes and Types in Logic Programming.* PhD thesis, University of Kent at Canterbury, 1999.

[199] J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 335–349. MIT Press, 1999.

[200] J.-G. Smaus. Proving termination of input-consuming logic programs. Technical Report 10-99, Computing Laboratory, University of Kent at Canterbury, United Kingdom, 1999. Long version of [199].

[201] J.-G. Smaus. Analysis of polymorphically typed logic programs using ACI-unification. Long version of [9], available via CoRR: `http://arXiv.org/archive/cs/intro.html`, 2001.

[202] J.-G. Smaus. Termination of logic programs using various dynamic selection rules. Technical Report 203, Institut für Informatik, Universität Freiburg, 2004. Long version of [3].

[203] J.-G. Smaus, F. Fages, and P. Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. Technical Report RR-4020, INRIA, 2000. Long version of [6], available via CoRR: `http://arXiv.org/archive/cs/intro.html`.

[204] J.-G. Smaus, P. M. Hill, and A. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, volume 1490 of *LNCS*, pages 73–88. Springer-Verlag, 1998.

[205] J.-G. Smaus, P. M. Hill, and A. King. Verification of logic programs with `block` declarations running in several modes. Technical Report 7-98, University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom, July 1998. Contains proofs for [204, 206]; see also [208].

[206] J.-G. Smaus, P. M. Hill, and A. King. Preventing instantiation errors and loops for logic programs with multiple modes using `block` declarations. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic-based Program Synthesis and Transformation*, volume 1559 of *LNCS*, pages 289–307. Springer-Verlag, 1999.

[207] J.-G. Smaus, P. M. Hill, and A. King. Mode analysis domains for typed logic programs. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation*, volume 1817 of *LNCS*, pages 83–102, 2000.

[208] J.-G. Smaus, P. M. Hill, and A. King. Verifying termination and error-freedom of logic programs with `block` declarations. *Theory and Practice of Logic Programming*, 1(4):447–486, 2001.

[209] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[210] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[211] O. Strichman. Tuning SAT checkers for bounded model checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 480–494. Springer-Verlag, 2000.

[212] O. Stursberg, S. Kowalewski, I. Hoffmann, and J. Preußig. Comparing timed and hybrid automata as approximations of continuous systems. In P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems IV*, volume 1273 of *LNCS*, pages 361–377. Springer-Verlag, 1997.

[213] R. S. Sutton and A. G. Barto. *Reinforcement Learning.* The MIT Press, 1998.

[214] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 2003. `http://www.sics.se/isl/sicstuswww/site/documentation.html`.

[215] J. Tan and I. Lin. Recursive modes for precise analysis of logic programs. In J. Małuszyński, editor, *Proceedings of the 14th International Logic Programming Symposium*, pages 277–290. MIT Press, 1997.

[216] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[217] S. Thompson. *Miranda: The Craft of Functional Programming.* Addison-Wesley, 1995.

[218] S. Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1999. Second Edition.

[219] V. I. Torvik and E. Trintaphyllou. Inference of monotone Boolean functions. In C. A. Floudas and P. M. Pardalos, editors, *Encyclopedia of Optimization*, pages 472–480. Kluwer Academic Publishers, 2001.

[220] M. Truszczyński. Logic programming for knowledge representation. In V. Dahl and I. Niemelä, editors, *Proceedings pf the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, pages 76–88. Springer-Verlag, 2007.

[221] P. H. Tu, T. C. Son, and C. Baral. Reasoning and planning with sensing actions, incomplete information, and static causal laws using answer set programming. *Theory and Practice of Logic Programming*, 7(4):377–450, 2007.

[222] The Unicode Consortium. *The Unicode Standard, Version 5.0.* Addison-Wesley Professional, 2006.

[223] J. Wielemaker, Z. Huang, and L. van der Meij. SWI-Prolog and the Web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.

[224] P. H. Winston and B. K. P. Horn. *LISP.* Addison Wesley, 1987.

[225] S. Yovine. Model checking timed automata. In G. Rozenberg and F. W. Vaandrager, editors, *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems, Veldhoven, The Netherlands, November 25-29, 1996*, volume 1494 of *LNCS*, pages 114–152. Springer, 1998.

[226] H. Zhang. SATO: An efficient propositional prover. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *LNCS*, pages 272–275. Springer-Verlag, 1997.

# Index