Agenda Control for Heterogeneous Reasoners

Reinhard Stolle*
PARC
Palo Alto, California
stolle@parc.com

and
Apollo Hogan
Department of Mathematics, University of California
Berkeley, California
apollo@math.berkeley.edu

and
Elizabeth Bradley†
Department of Computer Science, University of Colorado
Boulder, Colorado
lizb@cs.colorado.edu

Preprint, July 2003
to appear in The Journal of Logic and Algebraic Programming

*Corresponding author: Reinhard Stolle, PARC, 3333 Coyote Hill Road, Palo Alto, California, 94304. Email: stolle@parc.com. Phone: +1-650-812-4346.
†Supported by NSF NYI #CCR-9357740, ONR #N00014-96-1-0720, and a Packard Fellowship in Science and Engineering from the David and Lucile Packard Foundation. Much of this research was carried out while the first two authors were research assistants at the University of Colorado,
Abstract

As artificial intelligence techniques are maturing and being deployed in large applications, the problem of specifying control and reasoning strategies is regaining attention. Complex AI systems tend to comprise a suite of modules, each of which is capable of solving a different aspect of the overall problem, and each of which may incorporate a different reasoning paradigm. The orchestration of such heterogeneous problem solvers can be divided into two sub-problems: 1. When and how are various reasoning modes invoked?, and 2. How is information passed between various reasoning modes? In this paper, we explore some solutions to this problem. In particular, we describe a logic programming system that is based on three ideas: equivalence of declarative and operational semantics, declarative specification of control information, and smoothness of interaction with non-logic-based programs. Meta-level predicates are used to specify control information declaratively, compensating for the absence of procedural constructs that usually facilitate formulation of efficient programs. Knowledge that has been derived in the course of the current inference process can at any time be passed to non-logic-based program modules. Traditional SLD inference engines maintain only the linear path to the current state in the SLD search tree: formulae that have been proved on this path are implicitly represented in a stack of recursive calls to the inference engine, and formulae that have been proved on previous, unsuccessful paths are lost altogether. In our system, previously proved formulae are maintained explicitly and therefore can be passed to other reasoning modules. As an application example, we show how this inference system acts as the knowledge representation and reasoning framework of PRET—a program that automates system identification.

Keywords: automated reasoning; reasoning architectures; meta architectures; meta control; meta programming; reasoning strategies; logic programming; declarative programming.

1 Introduction

As artificial intelligence techniques are maturing and being used in large applications, the problem of specifying control and reasoning strategies is regaining attention. Fur-
1 INTRODUCTION

thermore, as the complexity of applications is scaling up, it is becoming less feasible to capture all aspects of an AI system's functionality in a single reasoning paradigm. Rather, complex AI systems tend to comprise a suite of modules, each of which is capable of solving a different aspect of the overall problem and each of which may incorporate a different reasoning paradigm. There seems to be an emerging sense in the AI research community that the orchestration of such heterogeneous problem solvers is in itself a difficult problem that deserves to be solved using AI techniques (e.g., [13, 67]). The orchestration problem can be divided into two subproblems: 1. When and how are various reasoning modes invoked?, and 2. How is information passed between various reasoning modes? In this paper we present some ideas about how to solve this problem, along with an application example.

Many languages that are designed for the declarative representation of domain knowledge are variants of first-order logic. One of the major advantages of logical representations is their clearly defined semantics: the domain knowledge can be interpreted as a logical theory. Logic programs can also be executed. Ideally, a logic program's declarative semantics (when interpreted as a logical theory) are equivalent to its operational semantics (when executed with respect to queries). In practice, the equivalence of declarative and operational semantics is often sacrificed for various reasons. Purely procedural constructs like the PROLOG cut, for example, are useful in the construction of efficient programs; however, their semantics cannot be described declaratively. Furthermore, control information is typically encoded implicitly in the static ordering of rules and goals. Finally, the commonly used principle of negation as failure confuses existential with universal quantification of non-ground goals.

This paper presents a logic system that accomplishes three important goals:

1. Declarative and operational semantics are equivalent.

2. Control information is represented explicitly, declaratively, and separately from domain knowledge.

3. Interaction with other programs is facilitated by an explicit representation of the theorem prover's state.

The first two goals are achieved by implementation of concepts developed as part of the "RISC" project (Reason Maintenance Based Inference System for Generalized
1 INTRODUCTION

Horn Clause Logic) at the University of Erlangen [4, 5, 6]. The third goal was accomplished by allowing non-logic-based reasoning modules access to the current state of the theorem prover. This feature is particularly important for the design of heterogeneous systems that integrate and orchestrate a variety of domain-specific reasoning techniques. For example, the logic system presented in this paper is currently used as the knowledge representation and reasoning framework of PRET, an automated modeling tool that finds ordinary differential equations (ODEs) that model black-box dynamical systems [10, 12, 59, 61]. The achievement of the three goals listed above is crucial to the success of this modeling task, but the contributions described here generalize well beyond this particular application domain. The third goal, in particular, is significant for any automated reasoning system that integrates several different reasoning modes. The various modules of such a hybrid reasoner typically must be able to access knowledge that has been generated before (either by themselves or by other modules). In our SLD-based system,\(^1\) invocation of different modules is triggered by the evaluation of subgoals of the currently active goal. Traditional SLD inference engines maintain only the linear path to the current state in the SLD search tree [44]. Formulae that have been proved on this path are typically implicitly represented in a stack of recursive calls to the inference engine, and formulae that have been proved on previous, unsuccessful paths are lost altogether. In our system, previously proved formulae are maintained explicitly and therefore can be passed to other reasoning modules.

We use the term “agenda control” to emphasize that the goal of the logic system presented in this paper is not the control of theorem proving per se. Rather, the purpose is the control and orchestration of a suite of heterogeneous, higher-level, and possibly domain-specific, reasoners. In our application domain of dynamic systems modeling, an agenda may contain items such as “find out if the differential equation is linear” or “see whether the target system exhibits oscillation.” In a different domain, such as robot planning\(^2\) for example, agenda items may be “figure out where I am” or “take sensor reading from the right arm sensor.” The purpose of this paper is to describe an implemented knowledge representation design that is appropriate for

\(^1\)The acronym SLD stands for Selecting a literal, using a Linear strategy, restricted to Definite clauses [58].

\(^2\)The automated planning community uses the term “agenda” in much the same way.
encoding a domain theory, associated agenda items and agenda control, and that lends itself to intuitive specification and manipulation by domain experts (rather than theorem proving experts). In the internal representation, domain knowledge turns into logical axioms, and agenda items become goals in the logic programming sense. Correspondingly, the question of agenda control turns into the task of defining appropriate strategies and tactics for theorem proving.

The language of the logic system presented in this paper is that of Generalized Horn Clause Intuitionistic Logic (GHCIL) [45, 46]. The inference engine can be briefly characterized as a GHCIL reasoner with declarative meta-level control and explicit representation of previously derived knowledge. The next three sections describe the GHCIL language, the meta-level control, and the explicit representation of previously derived formulae. As an application example, we show how this inference system acts as PRET's knowledge representation and reasoning framework. We conclude the paper with some pointers to related work.

# 2 The Language

## 2.1 GHCIL Clauses

General Horn Clause Intuitionistic Logic (GHCIL) clauses\(^3\) the corresponding are (implicitly) universally quantified implications of the following form.

1. Every definite Horn clause is a GHCIL clause.\(^4\)

2. If \(A\) is an atomic formula and \(B_1, \ldots, B_n\) are GHCIL clauses, then \(A \leftarrow B_1, \ldots, B_n\) is a GHCIL clause.

That is, GHCIL clauses are generalizations of Horn clauses that also allow embedded implications (other GHCIL clauses) in the body. For example,

\[
\text{dedicated}(P) \leftarrow (\text{working}(P) \leftarrow \text{assigned}(W, P), \text{unfinished}(W))
\]

---

\(^3\)This section is a recapitulation of the corresponding section in [4]. It is included here in order to make this paper more self-contained.

\(^4\)Recall that a definite Horn clause is a clause of the form \(A \leftarrow B_1, \ldots, B_n \ (n \geq 0)\) where \(A\) and \(B_i\) are all atomic formulae.
is a GHCIL clause that is not a Horn clause. Informally, its meaning is:

For all people $P$: $P$ is considered a dedicated person if $P$ is working under the assumption that there is some unfinished work $W$ that is assigned to $P$.

Thus, embedded implications can be seen as hypothetical statements. For a more detailed discussion of clausal intuitionistic logic, see [45, 46].

In our system, there are several distinguished predicates that may occur in GHCIL clauses. One of them is $\text{false}$: GHCIL clauses having $\text{false}$ as their head indicate contradictory situations. Negation as failure is not suitable for our purposes because it destroys the equivalence of declarative and operational semantics. Instead, our intuitionistic semantics uses negation as inconsistency [30] and interprets $\text{not}(p)$ as an abbreviation for $\text{false} \leftarrow p$. For example, consider the following rulebase:

1: $\text{false} \leftarrow \text{male}(X), \text{female}(X)$.
2: $\text{male}(\text{john})$.
3: $\text{female}(\text{betty})$.
4: $\text{male}(\text{pat})$.

The query $\text{not(\text{male}(X))}$ succeeds with $X$ bound to $\text{betty}$, consistent with the interpretation of the query: “is there an $X$ such that $X$ is not male?” With negation as failure, on the other hand, this query would fail; the interpretation in that case would be: “is it not the case that there is an $X$ such that $X$ is male,” or, in other words, “is it the case that, for every $X$, $X$ is not male?” This behavior would be inconsistent with the usual existential quantification of free variables in queries.

### 2.2 Evaluation of Queries in GHCIL Programs

The evaluation of queries is similar to SLD resolution in Horn clause queries. The main loop of the prover consists of the following steps. Let $P$ be the program and $G_1, \ldots, G_m$ the current goal.

A1. Select a subgoal $G_i$. 
A2. Search \( P \) for a clause \( C = (A \leftarrow B_1, \ldots, B_n) \) such that \( A \) and \( G_i \) can be unified by \( \theta \) with \( \text{mgu}(\theta, A, G_i) \).

A3. Set the new current goal to \( (G_1, \ldots, G_{i-1}, B_1, \ldots, B_n, G_{i+1}, \ldots, G_m)\theta \).

An SLD prover searches only the logic program for unifying clauses. Our GHCIL prover also looks up clauses in the so-called current assumptions. The current assumptions are a set of unquantified GHCIL clauses. Initially, the current assumptions are empty. But whenever, in Step A1, a goal \( G_i \) of the form \( D \leftarrow H_1, \ldots, H_k \) is selected, the prover

B1. adds \( \{H_1, \ldots, H_k\} \) to the current assumptions,\(^5\)

B2. tries to prove \( D \) (w.r.t. the program plus the extended current assumptions), and

B3. eventually removes \( \{H_1, \ldots, H_k\} \) from the current assumptions.

In the next section, we describe the extensions of this core GHCIL language that allow the implementer of the logic program to declaratively control the way in which GHCIL theories are processed.

3 Expressing Control Information

Traditionally, the control flow of a logic program is specified by the static ordering of rules and goals: the programmer expresses control knowledge implicitly by taking advantage of the inference machine’s properties, e.g., its depth-first-left-right strategy [58]. This approach conflicts with our goal of expressing all information in a declarative way. A program that relies on a certain evaluation strategy of the inference engine contains information—control information—that is not reflected by a purely logical interpretation of the program.

\(^5\)The hypotheses \( H_i \) must be added to the database as clause instances, i.e., it is not possible to use several different variants of the same hypothesis with different bindings in the same proof. For details see [59].
3 EXPRESSING CONTROL INFORMATION

Other common non-logical programming means of achieving efficient control of the deduction process include the PROLOG cut or the "predicates" assert, retract, and if-then-else. Such procedural constructs have declarative semantics—if any—that are different from their operational semantics. They result in a more or less imperative programming style and destroy the equivalence of procedural and declarative semantics, which is one of the main reasons for logic programming in the first place.

Meta control is a much better solution. It allows specification of control without interfering with the declarative representation of knowledge. For example, suppose we have the following declarative knowledge about a small initial segment of the ordinal numbers:

1: \[ \text{ord(succ}(X)) \leftarrow \text{ord}(X) \]  
2: \[ \text{ord}(0) \]  
3: \[ \text{ord}(\omega) \]  

If we were to use this knowledge in a PROLOG system, we would have to reorder the rules so that Rules 2 and 3 occurred before Rule 1 in order to avoid infinite loops for existential queries such as \[ ?\text{ord(succ}(Z)) \]. When adding new rules (e.g., \[ \text{ord}(\omega_1) \]), a programmer must pay close attention to how they interact with the rest of the rules—in this case ensuring that the new rule is added before Rule 1. That is, in addition to the declarative knowledge that 0, \( \omega \), and their successors are ordinals, we must also keep in mind the correct order for the rules and the control strategy of the inference engine. In other words, object-level information (in this case, knowledge about the structure of ordinal numbers) is intertwined with information about how to use object-level information (e.g., which rule should be used first). We call the latter control-level information.

If, instead, we separate control-level information from object-level information, we can specify the logical theory of ordinal numbers without worrying about the operational interpretation, or execution, of the theory as a logic program. In a separate set of meta control rules, we can then—again declaratively—specify the control information. The set of control rules, together with the object rules, represents a logical theory about the control of the logic program; we need only specify that Rule 1 is examined after any other rules, or we might specify that ground clauses are always to be preferred
3 EXPRESSING CONTROL INFORMATION

over non-ground clauses for the predicate \texttt{ord/1}.^6

3.1 Static Control: Abstraction Levels

One method for specifying meta control in a declarative fashion is \textit{abstraction levels}: static numeric annotations that describe the order in which clauses are considered to construct proofs, enforcing the preference of abstract proofs over less-abstract ones. These impose static \textit{global} constraints on the search for a proof. To every rule, the programmer assigns an abstraction level. For example, suppose that there are only two abstraction levels, \textit{low} and \textit{high}. Then any proof that uses only clauses with \textit{high} abstraction levels will be preferred to any proof that uses a clause with a \textit{low} abstraction level, even if the latter proof is much shorter.^7

The implementation of this scheme is straightforward; the inference engine proceeds to a less-abstract level only if the search for a proof at the more-abstract level fails. (This means that bad choices for abstraction levels affect only speed, and not correctness or completeness.)

Abstraction levels are a crude form of meta control. They are static and, though global, have a granularity at the clause level. Because of this, abstraction levels are often not general enough. The next section presents an example that calls for dynamic meta control.

3.2 Dynamic Control: Meta Rules

In \textsc{prolog}, as in many other logic-based knowledge representation systems, control information interferes with logical statements in order to achieve an efficient evaluation of huge sets of unit clauses [58]. Consider the following example (adapted from Example 8 of [4]).

\[
\text{grandparent}(X, Y) \leftarrow \text{var}(Y), \downarrow, \text{parent}(X, Z), \text{parent}(Z, Y).
\]

\[
\text{grandparent}(X, Y) \leftarrow \text{parent}(Z, Y), \text{parent}(X, Z).
\]

^6 A predicate \texttt{p} of arity \texttt{n} is denoted \texttt{p/n}.

^7 However, the programmer will typically assign abstraction levels to the rules in such a way that short proofs are also abstract proofs.
This example shows how efficiency considerations that have nothing to do with the declarative meaning of the logic program complicate the code. Expressing efficient control strategies for logical theories that are more complex than \textit{grandparent} requires increasingly baroque and hard-to-understand coding.

In our system, this kind of implicit control information is not necessary. We simply express the logical fact by the clause

\[
\text{grandparent}(X, Y) \leftarrow \text{parent}(X, Z), \text{parent}(Z, Y).
\]

In order to ensure an efficient evaluation, we specify that the subgoal that contains the ground argument must be evaluated before the subgoal that contains the variable:

\[
\begin{align*}
\text{before}(L_1, L_2) & \leftarrow \text{goal}(L_1, \text{parent}(X, Y)), \\
& \quad \text{goal}(L_2, \text{parent}(Y, Z)), \\
& \quad \text{ground}(X), \text{var}(Z).
\end{align*}
\]

\[
\begin{align*}
\text{before}(L_2, L_1) & \leftarrow \text{goal}(L_1, \text{parent}(X, Y)), \\
& \quad \text{goal}(L_2, \text{parent}(Y, Z)), \\
& \quad \text{ground}(Z), \text{var}(X).
\end{align*}
\]

At first sight, the Prolog formulation seems shorter and simpler. We argue that the number of characters needed is not a good measure of complexity. The order of clauses and goals and—more importantly—the cut in the Prolog program implicitly contain critical, complex information that is made explicit in our meta program. Furthermore, in our solution, the meta theory is conceptually and literally separated from the object-level theory. Moreover, operational semantics of the program are equivalent to the declarative semantics of the object-level theory.

The meta predicate \textit{before/2} allows us to specify control information in a clean fashion, separately from the logical theory about parents and grandparents. Other control predicates that our system makes available to the programmer are \textit{notready/1} and \textit{hot/1} for the selection of subgoals to be resolved and \textit{clauseorder/2} for the selection of the resolving clause. When the inference engine chooses the next subgoal to be resolved, it determines the minimal elements of the partial order defined by \textit{before/2}. Subgoals that are proved to be \textit{notready/1} may not be chosen; within these
constraints, hot/1 subgoals receive priority. The rule

\[ \text{clauseorder}(H, [N_1, \ldots, N_m]) \leftarrow B_1, \ldots, B_n. \]

states that clauses whose names belong to \(N_1, \ldots, N_m\) must be selected in that order for the next inference step if the selected subgoal is an instance of \(H\). The meta predicates \(\text{clause/2}\) and \(\text{goal/2}\) establish names for clauses and currently active subgoals.\(^8\)

If the meta rules do not completely specify the control decisions, the default control is from left to right.

A useful and informative way of describing the semantics of a logic programming system is to provide a meta-interpreter; [4] follows this approach by showing logic programs that define the semantics of the control predicates \(\text{before/2, notready/1, hot/1 and clauseorder/2}\). A different—but not less challenging problem—is to turn such a neatly defined meta logic system into an implementation that integrates a collection of heterogeneous reasoning modules, provides a caching mechanism for expensively derived partial solutions, and offers the meta control predicates to the knowledge engineer as a practical tool for the orchestration of the various reasoning modules. This is the topic of this paper and, in particular, of the next section. The semantics of all the meta predicates used in this system follows that of [4]. We shall not repeat these definitions here; instead, we refer the reader to that paper for the details of the declarative and procedural specification of the meta predicates.\(^9\)

4 Explicit Representation

If an inference engine is integrated in a multi-modal reasoning system, other—non-logic-based—reasoning modules must have access to previously derived knowledge: everything that has been successfully inferred so far. Resolution provers that do not remember previously derived knowledge only maintain the current root path of the

\(^8\)The meta predicates \(\text{var/1}\) and \(\text{ground/1}\) have the usual meaning. Since they have no first-order declarative semantics, they can, in fact, destroy the equivalence of declarative and operational semantics of the program if they appear outside the meta level and should therefore be used with care. See the discussion in the Related Work section.

search tree, which represents the (partial) proof tree of the current proof attempt. The advantage of maintaining only the root path is its linear space requirement; the disadvantage is that already-proven results must be rederived every time they occur on different root paths. Trading time for space, many problem solvers use some kind of caching of inferences in order to avoid duplication of effort, to generate explanations, and to guide backtracking or control [29]. This approach becomes particularly important when—as in the application example in Section 6 of this paper—the derivations of some formulae require the invocation of other reasoning modules and are therefore very expensive.

In this section, we describe, in some detail, the form of caching used in our logic system and the explicit representation that is necessary to achieve it. Since our approach can be viewed as a very simplified form of truth maintenance, we briefly discuss the similarities and differences between this approach and traditional truth maintenance systems (TMSs) and the motivation behind our choices. Finally, we describe how caching and explicit representation of the inference state are integrated with abstraction levels and dynamic meta control.

4.1 Implementation

The system described in this paper is implemented in Scheme. The state of the inference engine is encapsulated in a stack. The elements of this inference stack are either “choice points” or inference stacks themselves. A choice point is a branching node in the search tree. When choosing one of multiple branches, the choice point records the remaining choices in order to allow the alternative choices to be explored later in the course of backtracking.\(^{10}\) For example, assume we are trying to prove the subgoal \(p(X)\). Furthermore, assume the following two clauses in the database are the only ones whose heads unify with \(p(X)\).

\[
\begin{align*}
1: & \quad p(Y) & \leftarrow & \quad q(Y), r(Y). \\
2: & \quad p(a) & \leftarrow & \quad q(a).
\end{align*}
\]

\(^{10}\)Note that a choice point is a node in the search tree; it does not correspond to any single point in the flow chart of Fig. 1.
Then the system will create a choice point (abbreviated \( cp \)) for the goal \( p(X) \) that contains the matching clauses 1 and 2:

\[
cp = \ \{ \ (p(Y) \leftarrow q(Y), r(Y)), \\
(p(a) \leftarrow q(a)) \ \}
\]

Choice points also keep track of the corresponding bindings.

The typical (and elegant) way of programming a resolution inference engine is to call the engine recursively to resolve the subgoals of the current goal. However, if we were to actually do a recursive SCHEME call, the explicit representation of the inference engine state would be lost, as it would be embedded in the SCHEME call stack. Instead, to keep the state explicit, we do a “pseudo-recursive” call by pushing a new inference stack onto the old one and then using only the new inference stack until this simulated recursive call succeeds or fails. In the latter case, we throw away the new stack; if the simulated recursive call succeeds, we keep the new inference stack on the old one so we can backtrack into the call if necessary. We then continue the inference process, using the old inference stack and pushing new choice points (or inference stacks) above the other inference stack.\(^{11}\)

Our inference engine handles normal clauses in a straightforward PROLOG fashion. To handle embedded implications, we do a pseudo-recursive call to the inference engine, adding the formulae in the body of the embedded implication to the current assumptions and setting the current goal to be the head of the embedded implication. Since the state of the inference engine is encapsulated in a single explicit data structure, it is trivial to interrupt and resume the inference task: if the inference engine is interrupted, it simply returns the inference stack. To restart, it is only necessary to call the inference engine and pass the inference stack back in.

\(^{11}\)There are other methods for handling the need for recursive calls. Another embedding of PROLOG into SCHEME [37] used the fact that SCHEME has first-class continuations to enable backtracking through recursive calls, and used continuations for non-blind backtracking or “lateral” control transfers. We avoided using continuations because, although they do provide a handle into the SCHEME control stack, they are still not explicit enough—continuations are opaque. Our approach of reifying the control explicitly also allows for non-blind backtracking, though we did not implement it in our system.
4 \textit{EXPLICIT REPRESENTATION}

4.2 State Transitions

A flow chart of the inference process is shown in Figure 1. The symbols \(1\), \(2\), \(3\), \(4\), and \(5\) identify the important internal states of the inference engine. To show how the inference engine operates, we first describe the various state transitions algebraically and explain how they relate to the description of query evaluation in Section 2.2. Then, we illustrate the process by stepping through a concrete example.

The internal state of the theorem prover keeps track of various data structures: \(G\) is the current goal, \(Sel\) is the currently selected subgoal, \(S\) is the stack of choice points (corresponding to the root path of a traditional SLD prover), and \(Asm\) is the set of current assumptions.

\textit{Parent stacks} is the stack of stacks that keeps track of the states of the pseudo-recursively invoked prover instances, as described in the previous subsection. In our actual implementation, we maintain only one stack, which contains all information that is represented here by both \(S\) and \textit{Parent stacks}: the elements of the stack may be either choice points or whole inference stacks themselves. For the algebraic specification of the state transitions, however, it is more convenient and elegant to have two variables: \(S\) for the choice points of the current prover instance, and \textit{Parent stacks} for the outer prover instances in which the current prover instance is embedded.

In the following paragraphs, we define the state transitions by providing algebraic equations that relate the internal state of the engine after the transition to the internal state before the transition. We use subscripts to refer to the states. For example, in the description of the transition from State \(4\) to State \(2\), \(S_4\) refers to the stack before the transition and \(S_2\) refers to the stack after the transition. \(P\) always refers to the set of clauses that constitute the program; \(P\) is a constant.

In order to keep this presentation succinct and clear, we omit the description of the manipulation of the variable bindings that correspond to the transitions.

\textbf{Initial state.} The inference engine starts in State \(1\). The stack is empty, there are no assumptions, the goal is the original query, and no subgoal has been selected yet.

\[ S_1 = \text{empty\_stack} \]
Figure 1: Flow-chart of the inference engine.
\begin{align*}
Asm_1 &= \emptyset \\
G_1 &= \text{the query} \\
Sel_1 &= \text{undefined} \\
Parent_{\text{stacks}}_1 &= \text{empty stack}
\end{align*}

\textbf{From ① to ②: Selecting an atomic subgoal.} This transition corresponds to Steps A1 and A2 from Section 2.2. When moving from State ① to State ②, the engine invokes the dynamic meta control, which selects the current subgoal from the goal \( G_1 \). Assume the selected subgoal is the atomic formula \( A \). (For the case where the subgoal is not atomic, see the transition from ① to ③ below.)

\begin{align*}
Sel_2 &= A \\
G_2 &= G_1 \setminus \{ A \}
\end{align*}

Let \( cp \) be the set of matching clauses in \( P \cup Asm_1 \):

\[ cp = \{(H \leftarrow B) \mid (H \leftarrow B) \in P \cup Asm_1 \land H \text{ and } A \text{ unify}\} \]

If such matching clauses exist, the meta control is invoked to determine the order in which these clauses should be examined. The set of clauses is ordered accordingly (not shown in these equations) and added as a choice point to the stack:

\[ cp \neq \emptyset \Rightarrow S_2 = \text{push}(cp, S_1) \]

If no matching clauses exist, then no choice points are added to the stack:

\[ cp = \emptyset \Rightarrow S_2 = S_1 \]

The current assumptions and the stack of parent stacks are not affected by this step:

\begin{align*}
Asm_2 &= Asm_1 \\
Parent_{\text{stacks}}_2 &= Parent_{\text{stacks}}_1
\end{align*}
4 EXPLICIT REPRESENTATION

From ② to ①: Resolving an atomic subgoal. This transition corresponds to Step A3 from Section 2.2. Assume that the stack \( S_2 \) is not empty. (If it is empty, the engine moves to State ④ without any manipulation of the data structures.) In this step, the engine examines the first of the remaining clauses that resolve the currently selected subgoal. Let \( cp \) be the top choice point on the stack:

\[
cp = \top(S_2)
\]

We select the first clause from that choice point, push the remainder back on the stack (if non-empty), and add the body of the selected clause to the current goal:

\[
(H \leftarrow B) \in cp
\]

\[
\begin{align*}
cp' &= cp \setminus \{(H \leftarrow B)\} \\
G_1 &= G_2 \cup B
\end{align*}
\]

\[
\begin{align*}
\text{if } cp' \neq \emptyset &\Rightarrow S_1 = \text{push}(cp', \text{pop}(S_2)) \\
\text{if } cp' = \emptyset &\Rightarrow S_1 = \text{pop}(S_2)
\end{align*}
\]

The selected subgoal, the current assumptions and the stack of parent stacks are not affected by this step:

\[
\begin{align*}
Sel_1 &= Sel_2 \\
Asm_1 &= Asm_2 \\
Parent_{\text{stacks}}_1 &= Parent_{\text{stacks}}_2
\end{align*}
\]

From ① to ③: Selecting a non-atomic subgoal. This transition corresponds to Step A1 from Section 2.2. When moving from State ① to State ③, the engine invokes the dynamic meta control, which selects the current subgoal from the goal \( G_1 \). As opposed to the transition from ① to ②, in which the selected subgoal was atomic, we now assume that the selected subgoal has the form of an implication \((H_G \leftarrow B_G)\) where \( B_G \neq \emptyset \).

\[
\begin{align*}
Sel_3 &= (H_G \leftarrow B_G) \\
G_3 &= G_1 \setminus \{(H_G \leftarrow B_G)\}
\end{align*}
\]
The stack $S$, the assumptions $Asm$ and the stack of parent stacks are not affected by this transition; they will be updated in the transition from $3$ to $1$, as explained in the next paragraph.

\[
S_3 = S_1 \\
Asm_3 = Asm_1 \\
Parent\_stacks_3 = Parent\_stacks_1
\]

From $3$ to $1$: Setting up an embedded instance of the prover. This transition corresponds to Step B1 from Section 2.2. When an implication is selected from the current goal, the engine puts the current proof “on hold” and starts up a new instance of the prover. In our explicit representation this is achieved by creating a fresh stack and saving the current stack as a parent stack. Moreover, the engine saves the current goal as a “goal continuation,” which will be used to continue the current proof when the engine returns from the proof of the implication. For the duration of the embedded proof, the body of the implication is added to the set of current assumptions.

\[
Sel_3 = (H_G \leftarrow B_G) \text{ where } B_G \neq \emptyset \\
Parent\_stacks_1 = \text{push}(\langle S_3, G_3, Asm_3 \rangle, Parent\_stacks_3) \\
S_1 = empty\_stack \\
Asm_1 = Asm_3 \cup B_G \\
G_1 = H_G \\
Sel_1 = Sel_3
\]

The engine returns to State $1$ and tries to prove the new goal, $G_1$, relative to the program $P$ plus the extended current assumptions. This corresponds to Step B2 from Section 2.2.

From $5$ to $1$: Returning from a successful embedded proof. This step corresponds to Step B3 from Section 2.2. Once all subgoals of the goal have been resolved, the current goal becomes empty: $G_5 = \emptyset$. This means that the current instance of the prover has proved its initial goal. In this case, we discard the current
instance of the prover and continue the proof of the parent instance.\footnote{As described in Section 4.1, the actual implementation does not discard the embedded stack, so we can backtrack into it if necessary. For the algebraic description here, in which we ignore variable bindings, we can discard the embedded stack.} (In this transition, the parent instance is assumed to exist. The case where no parent stack exists is described in the next paragraph.)

\begin{align*}
(S_1, G_1, Asm_1) &= \text{top}(Parent\_stacks_5) \\
Parent\_stacks_1 &= \text{pop}(Parent\_stacks_5) \\
Sel_1 &= Sel_5
\end{align*}

\textbf{From (5) to SUCCESS: Succeeding the overall proof.} In this transition, the goal of the outermost prover has become empty. This means the overall proof has succeeded.

\textbf{From (4) to (2): Returning from a failed embedded proof attempt.} The stack $S$ contains the choice points that encode the remaining alternative choices for previous clause selections. Once $S$ becomes empty, all choices have been exhausted without having been able to finish the proof. This means the current proof attempt has failed. Therefore, the current instance of the prover is discarded, signalling a failure to the parent prover. The parent prover continues from State (2), exploring possible alternatives for that clause that contained the embedded (failed) subgoal. (In this transition, we assume that the current proof is an embedded proof. The case where the current proof is the outermost proof is described in the next paragraph.)

\begin{align*}
(S_2, G_2, Asm_2) &= \text{top}(Parent\_stacks_4) \\
Parent\_stacks_2 &= \text{pop}(Parent\_stacks_4) \\
Sel_2 &= Sel_4
\end{align*}

\textbf{From (4) to FAILURE: Failing the overall proof attempt.} In this transition, the choice points of the outermost prover have been exhausted without having been able to completely resolve the original goal. This means the overall proof has failed.
4 EXPLICIT REPRESENTATION

\[
\begin{align*}
G &= \{A\} \\
\hline
\leftarrow S
\end{align*}
\]

Figure 2: Initial conditions of the inference engine.

\[
\begin{align*}
A &\vdash (B \vdash C), D. \\
A &\vdash X. \\
G &= \{\}
\end{align*}
\]

Figure 3: Inference engine, state=2

4.3 Sample Inference

To illustrate this more concretely, let us step through the inference process for a simple query. Let the database of clauses be given by

1: \[ A \leftarrow (B \leftarrow C), D. \]
2: \[ A \leftarrow X. \]
3: \[ B \leftarrow F. \]
4: \[ B \leftarrow X. \]
5: \[ D \leftarrow E. \]
6: \[ E. \]
7: \[ F \leftarrow C. \]

Suppose the query is \( ?A \). The initial conditions of the inference engine are shown in Figure 2. The top of the stack, \( S \), is empty and the goal set \( G \) contains only the query formula: \( \{ A \} \). The engine starts in State 1. In this state, the meta control\(^{13}\) for the engine selects \( A \) (the only choice in this case) as the next goal to prove. A choice point (denoted c.p. in the figure) is created from all clauses in the database and all assumptions in the current assumption set that unify with the selected formula, \( A \), and this choice point is pushed onto \( S \). The state becomes 2.

In Figure 3 the inference engine is in State 2, so a choice point is popped off the stack and a clause is selected to try (again, the meta control makes this decision).

\(^{13}\)The meta control module selects goals and clauses according to the programmer’s meta control rules, which are described in Section 3.2 of this paper.
In this case, the chosen clause is $A \leftarrow (B \leftarrow C), D$; the remainder of the choices are pushed back onto $S$, the state is changed back to 1, and the goal set $G$ becomes $\{(B \leftarrow C), D\}$. Again, the meta control selects a formula from the goal set, namely $B \leftarrow C$. Since this is an embedded implication, the engine proceeds to State 3.

In State 3, a new stack is pushed onto the old stack. The old goal set (called “goal continuation” in the figure), $\{D\}$, and a pointer to the old stack top are saved. The head of the embedded implication becomes the (only) goal in the new goal set $G$, and the formulae in the body of the implication (called “assumptions” in the figure) are temporarily added to the rule base. Finally $S$ is set to the top of the new stack and the state becomes 1.

Next (Figure 5), the (only) formula from the goal set is selected, a choice point is pushed and the current state becomes 2. Notice that this choice point is pushed onto the inner stack that was created in State 3 above.

In Figures 6–8, the inference engine progresses through States 1, 2, and 4 until $G$ becomes empty (Figure 9). Then, the inference engine proceeds to State 5. (This means that the subgoal $B \leftarrow C$, which caused the creation of the second inner stack, was successful.) Then $S$ is set back to the parent stack and $G$ is reset to the old goal continuation $\{D\}$. The current state becomes 1 as shown in Figure 10.

The meta control selects a goal from $G$ and a new choice point is pushed onto $S$. Note that the current stack is now equivalent to the original, outer-most one (Figure 11).
Figure 6: Inference engine, state=2

Figure 7: Inference engine, state=1

Figure 8: Inference engine, state=2
Figure 9: Inference engine, state=1

Figure 10: Inference engine, state=1

Figure 11: Inference engine, state=2
The inference engine continues this process until either State 4 or State 5 is reached (failure or success, respectively), with $S$ pointing to the original outer-most stack.

The explicit representation of embedded call stacks described and illustrated in this section is important because it allows previously derived knowledge to be reused and passed around to other (possibly non-logical) reasoning modules. This feature is of crucial importance for a multi-modal reasoning system, an example of which we describe in Section 6 of this paper.\textsuperscript{14} The traditional scheme of recursively calling the inference engine would hide the current proof tree in the interpreter's implicit call stack.

In the next section we describe how available knowledge is reused in the inference process and how it is passed to non-logical reasoning modules.

### 4.4 Making Derived Knowledge Explicit

We maintain previously derived knowledge for two reasons. First, we want to be able to pass knowledge explicitly to other reasoning modules. Second, we want to avoid duplication of effort. Formulae that have been proved are stored in a database for reuse in later proofs or subproofs. The second reason is particularly important where the proof of a formula involves calls to other modules; these calls are typically expensive and should not be done more often than necessary.

In order to attain both of these goals, we maintain a database (implemented as a hash table) of previously derived formulae. This database contains formulae that have been proved in the current inference process, even if these formulae are not in the current proof tree, i.e., even if they are on a branch of the search tree that failed. However, because it is impractical to store everything that has been proved, we only cache predicates that the programmer declares as \textit{relevant}, using the meta predicate \textit{relevant/1} \textsuperscript{5}. (This would typically include those in which multiple modules are interested and those that are expensive to evaluate.)

\textsuperscript{14}An additional advantage of an explicit representation is that the meta control can choose between all subgoals in the call stack rather than just the subgoals of the inner-most stack. In this case, nested implications are not necessarily evaluated before the goal in which they are \textit{embedded}. However, our implementation does not take advantage of this possibility. Currently, the reasoner always finishes \textit{embedded} subgoals before returning to the \textit{embedding} goal.
Every time the proof of a relevant formula is completed, the database is updated. If there are no active assumptions, the proven relevant (atomic) formula is simply added to the database as is. If, however, we are currently in the middle of the proof of an embedded implication (which means that the set of current assumptions is not empty), the proven formula might be true only relative to some of the active assumptions. Therefore, we collect the assumptions that have been used since the start of the inference process for the relevant goal. If this set of used assumptions is empty, the relevant formula is stored as an atomic formula in the database. If the set of used assumptions is non-empty, we store a non-atomic formula—an implication—built from the relevant formula and the used assumptions.

These cached formulae are then used to speed up calls to the same subgoals in later proof attempts. They are used as if they were program clauses whenever a resolving clause must be chosen for a given subgoal (State 1). They are added at the front of the rule base, i.e., they receive priority unless the meta control decides otherwise.

In the database, we store only the most-general forms proved so far: if we prove $A$ and $B$ and $A\theta = B$ for some substitution $\theta$, then we store only $A$. This amounts to $\theta$-subsumption [65] in the case of atomic formulae. In the case of embedded goals (implications) this is only a crude form of caching; handling full $\theta$-subsumption in this general case is NP-complete [33]. However, our (seemingly ad-hoc) form of caching does exactly the right thing: since calls to expensive modules typically appear—statically—in only a few rules, rarely is an expensive call subsumed by previous calls but not detected by a purely syntactic check for generalization or specialization. A full subsumption check would add much complexity with little gain.

A similar complexity trade-off motivated our decision not to use a full truth maintenance system [21, 29]. In many problem solvers, TMSs provide an elegant solution to reasoning using beliefs, assumptions, and contexts. Maintaining labels (minimal sets of sufficient assumptions, in the case of an ATMS) brings complexity that is unnecessary for our purposes. Instead, we provide the programmer with the meta-predicate relevant/1, which is an appropriate tool to maintain just enough information to be able to pass all relevant current knowledge to other modules while avoiding duplicated work in evaluating time-intensive predicates.\(^{15}\)

\(^{15}\)For the case of logic-based truth maintenance systems (LTMS), Everett and Forbus [28] have
The following example illustrates how the "caching technique" described in this section facilitates efficient interaction with non-logical reasoning modules. Consider the following program fragment from the domain of ODE theory.

\[
\begin{align*}
\text{falsum} & \leftarrow \text{time\_series}(T), \text{chaotic}(T), \text{periodic}(T) . \\
\text{falsum} & \leftarrow \text{time\_series}(T), \text{chaotic}(T), \text{linear}(T) . \\
\text{chaotic}(T) & \leftarrow \text{time\_series}(T), \text{expensive\_test}(T, \text{chaotic}) . \\
\text{periodic}(T) & \leftarrow \text{time\_series}(T), \text{expensive\_test}(T, \text{periodic}) . \\
\text{linear}(T) & \leftarrow \text{time\_series}(T), \text{expensive\_test}(T, \text{linear}) . \\
\text{time\_series}(ts) .
\end{align*}
\]

Suppose that \( ts \) is an experimental time series that happens to be chaotic (hence non-periodic). Consider the query \( \text{falsum} \leftarrow \text{linear}(ts) \) whose interpretation is: "is the time-series non-linear?"\(^{16} \) If we assume a depth-first-left-right strategy, the system evaluates the formulae in the following order:

\[
\begin{align*}
\text{not}(\text{linear}(ts)) \\
\text{falsum} & \leftarrow \text{linear}(ts) \\
\text{falsum} \\
\text{chaotic}(ts) \\
\text{expensive\_test}(ts, \text{chaotic}) \\
\text{periodic}(ts) \\
\text{expensive\_test}(ts, \text{periodic}) & \quad \text{succeeds} \\
\text{falsum} \\
\text{chaotic}(ts) \\
\text{expensive\_test}(ts, \text{chaotic}) \\
\text{linear}(ts) & \quad \text{succeeds}
\end{align*}
\]

The system does not do the numeric test for linearity because we are assuming

\(^{16}\)The formula \( \text{linear}(ts) \) represents the fact that all data points of the time series \( ts \) lie on a line (modulo some specified resolution). The ODE rule used in this example is: a linear behavior is neither a periodic behavior nor a chaotic behavior. This ODE rule is much narrower and much more limited than the more general rule that a linear ODE system (represented by the formula \( \text{linear\_system}(\text{current\_model}) \)) cannot be chaotic.
linear(ts) in the query. Notice that the numeric test for chaoticity is evaluated twice, even though it only needs to be done once. For efficiency, we need to cache the result of this evaluation after the first call so that on the second call the inference system can simply report failure or success without actually doing the expensive numeric test a second time.

Caching intermediate results is crucial in order to avoid duplication of effort if a formula appears multiple times in a search tree. The overhead of the cache is negligible compared to the saved computation time. Storing or retrieving a formula from a hash table takes a fraction of a second, but the computation that establishes such a formula (e.g., an expensive numerical test) may take several seconds or even minutes.

This mechanism is critical to the efficiency of any system that uses this framework. The program in which we have tested this inference system, for example, incorporates a large variety of heterogeneous reasoning modes: symbolic reasoning, geometric reasoning, qualitative simulation, parameter estimation, and numerical simulation. Geometric reasoning and qualitative simulation are orders of magnitude more expensive than simple symbolic checks, and parameter estimation and numerical simulation are even more expensive. Therefore, the term “caching” may be misleading for the inference engine’s technique of storing and reusing previously derived formulae. The caching mechanism is not merely a matter of making the program more efficient by a small percentage. It makes heterogeneous reasoning feasible.

4.5 Integration of the Three Goals

The previous section explains how our implementation maintains derived knowledge, thereby allowing that knowledge to be passed to other modules. In this section, we describe where and how the solutions that achieve the other goals of the work described in this paper—equivalence of declarative and operational semantics, and declarative representation of control information—fit into this picture.

Relevant formulae are handled by the inference engine in the same way as embedded implications are. Conceptually, a new incarnation of an inference process tries to finish a proof of the relevant subgoal before other subgoals receive attention.\footnote{Gallaire and Lasserre [32] achieve a similar effect using the predicate finish.} In
State ① of Figure 1, for example, if the selected formula is deemed relevant, the inference engine passes to a State ③ (similar to State ③), where a new stack is pushed onto the old stack. The new goal set contains only the relevant formula, and the engine goes back to State ①. Later, when State ⑤ or ④ is reached (success or failure in proving the relevant formula, respectively), the engine will, before resuming the inference, store the result of the pseudo-recursion, as described in Section 4.1. This means that declaration of relevance takes priority over control decisions that are specified by meta rules. The advantage of this approach is that it is easy to keep track of when a relevant formula has been proved.

The inference engine handles the abstraction levels by iterating from the most-abstract level to less-abstract levels. Abstraction levels are identified by the programmer, who assigns a natural number (an “abstraction level number”) to each clause. For example, in the domain of ODE modeling, the abstraction levels are used to express static control knowledge of the type: “In general, try to build proofs involving qualitative properties of candidate ODE models before building proofs involving numeric properties.” First, only the clauses on the most-abstract level are considered. If this proof attempt fails, the clauses from the next abstraction level are added, and so on, until the proof succeeds or all levels are exhausted. Maintaining the database of derived knowledge reduces duplication of effort that would occur when knowledge has to be rederived in later iterations. Avoiding duplication of effort is, in general, crucial to all inference tasks that involve expensive proofs or repeated calls to time-consuming reasoning modules.

The meta-level control strategy is integrated into the inference engine at two points: when a subgoal is selected to be resolved (in State ①) and when a resolving clause is selected (in State ②). These two points are marked “Meta-control invoked here” in the inference engine flow-chart (Figure 1). In order to select a subgoal, the inference engine is called recursively\(^\text{18}\) to evaluate all notready-, before-, and hot-rules (see Section 3.2) that apply to the current situation. We call the facts that are proved by these evaluations of meta rules the current control facts. From the current goal, the meta control chooses a subgoal that meets all constraints imposed by the current

\(^{18}\text{A recursive call to the inference engine allows the full generality of the theorem prover to be used for meta control in a simple and elegant fashion.}\)
control facts. In order to select a clause, the meta control is only consulted the first time the inference engine reaches this choice point. Again, the meta control evaluates all clauseorder-rules that apply to the current situation in order to derive the current control facts. The meta control then determines an ordering of all matching clauses that meets all constraints that are expressed by the current control facts. The first clause in this ordering is chosen to resolve the current subgoal of the object-level proof. If the same choice point is reached again later via backtracking, the other clauses can be used in the already-determined order; meta control need not be invoked again. Please consult [4] for a formal description of the semantics of the control predicates.

5 Correctness and Completeness

Generalized Horn Clause Logic is intuitionistically equivalent to a certain subset of McCarty’s Clausal Intuitionistic Logic [45, 46]. According to Tobermann [63], the calculus of generalized Horn clauses upon which our theorem prover is based is logically sound and complete. Since the prover performs depth-first search, it is combinatorially incomplete in the same way as PROLOG is: it cannot effectively find a proof for a logical consequence of the theory represented by the program if its derivation is hidden by an infinite path in the search tree. The introduction of control rules into generalized Horn clause logic does not affect the soundness of the proof procedure. Control rules cannot “generate” new solutions that are not logical consequences of the logic program.

Control rules for the selection of subgoals preserve not only correctness but also completeness. Tobermann [63] has also shown that the selection function for a RISC-type prover may perform arbitrary computations. The only condition that the selection function has to meet in order to preserve completeness is that it must be a total function that selects one of the current subgoals. Ordering of clauses does not affect the logical completeness.\(^\text{19}\) It does, however, affect combinatorial completeness; a different order may make the prover follow an infinite path before it finds some

\(^\text{19}\)In our system, clause ordering only decides when a clause is applied, not whether it is applied. This stands in contrast to other information prioritization systems, in which prioritization amounts to an exclusive choice between possibly conflicting pieces of information [54].
5 \textit{CORRECTNESS AND COMPLETENESS}

Logical consequence of the program. One of the intended usages of the meta predicate \textit{clauseorder}/2 is—in addition to efficiency considerations—to (dynamically) determine a combinatorially complete clause order. Given the meta control predicates described in Section 3.2, this can be effected by the programmer in an easy and intuitive way.

The calculus that results from adding the abstraction level mechanism to the RISC-type prover is also correct and complete. First, consider correctness. More-abstract reasoning only takes away solutions of the program; it never adds new solutions. Thus, the resulting calculus is correct. Completeness is somewhat more subtle. The completeness of the underlying inference engine implies that the reasoning process is complete relative to the set of rules that are in use. However, reasoning performed at a more-abstract level is typically incomplete with respect to a less-abstract level. This is exactly our intention: to mask out logical consequences of the program that lead to too-detailed reasoning too early. Since queries ultimately fail only after the inference engine has considered all rules at all abstraction levels, the overall process is complete.

Both correctness and completeness are also preserved by the caching mechanism. A formula is only stored if it has been proved. Since the knowledge base does not change during the evaluation of a query, a stored formula remains true for the whole evaluation process and can thus be reused. A formula that is true only with respect to an extended context (that is, a set of assumptions) is stored as an implication whose body consists of those assumptions. These conditional formulae are also valid and do not affect correctness. Likewise, completeness remains unaffected by the caching mechanism since no rules are removed from the logic program; rather, the cache is \textit{added} in front of the logic program. Solutions may be found in a different order, however; currently, they are also found multiple times if the theorem prover first uses cached results and later also uses the corresponding original rules. This only poses a problem if the user asks for several proofs of a query, or if excessive backtracking occurs within a proof. A version of the cache manager that avoids even these duplication problems is currently under construction. In that version, every cached formula will maintain a pointer to the rules from which it was derived, along with some other book-keeping information.
In the application example described in the next section, the inference engine's task is to find the first proof of the query *falsum*.

## 6 An Example

The logic system presented in this paper has successfully been used as a knowledge representation and reasoning framework in the domain of ODE theory. The program PRET [10, 61] automates system identification [43]: given hypotheses, observations, and specifications, it constructs an ODE model of a black-box dynamical system. PRET uses the given hypotheses to construct a sequence of candidate models and checks each candidate against the observations. The first candidate that passes this check is returned as the answer. In this section, we describe how PRET employs our logic system to perform this model check.

PRET’s knowledge base encodes ODE theory in GHCIL clauses. The person who implements or maintains this knowledge base will presumably be an expert in engineering—not logic programming—so the declarative representation of knowledge without the use of “hack type” efficiency side effects is crucial. The concept of negation as inconsistency is ideal for this application: the candidate model checker combines the observations about the target system, the observations about the candidate model, and the ODE theory into one set of clauses and then checks that set for consistency, i.e., tries to derive *falsum* from it. This instantiates PRET’s opportunistic paradigm: a candidate that provides no reason for an inconsistency is considered a good model.

Checking candidate models against the given observations poses a difficult reasoning control problem, but one that can be solved elegantly using the framework described in this paper. The model checker makes use of several non-logic-based modules, e.g., the commercial symbolic algebra package Maple [18], a simple qualitative envisioning module, a nonlinear numerical parameter estimator [11], and a geometric reasoner for intelligent data analysis [9]. Calls to these modules require knowledge to be passed to them explicitly. By declaring the appropriate predicates as “relevant,” the PRET knowledge engineer instructs the inference engine to make the appropriate pieces of knowledge available. Different reasoning techniques vary considerably in their cost. Symbolic techniques are usually quick and cheap; the order of an ODE, for example,
can be established within a fraction of a second. Semi-numeric and numeric techniques take much longer. The time taken by a call to \textsc{pret}'s parameter estimation module, for example, ranges between a couple of seconds and several minutes. What \textsc{pret} needs in order to manage the complexity of its task—finding an ODE model for a given dynamic system—is the ability to dynamically orchestrate application of its ODE rules and the various reasoning modes that are triggered by the resulting evaluations, all in a manner that leads to the quickest possible test of a given model.

The three techniques described in this paper—abstraction levels, dynamic meta control, and reuse of previously derived formulae—achieve exactly this intelligent orchestration of reasoning modes. We use the concept of abstraction levels (see Section 3.1) to direct the search for an inconsistency toward a quick, abstract proof. For example, qualitative reasoning rules are assigned a more abstract level than rules that encode numerical reasoning. As a result, \textsc{pret} tries to discard models by purely qualitative means before resorting to numerical techniques. In other qualitative reasoning systems that work with different abstraction levels (e.g., [52, 70]), the levels are implicitly defined by the system's architecture and data structures. In \textsc{pret}, every rule is explicitly assigned an abstraction level number. Similarly, the logic engine's dynamic control is used in \textsc{pret} to guide the search toward a cheap and quick proof of \textit{falsum}. Rules that are likely to lead to a contradiction are chosen before other rules, and subgoals that are likely to fail quickly are evaluated before other subgoals.

As an example, consider the following (simplified) program.

\[
\begin{align*}
\text{stable} & \leftarrow \text{linear, all roots in left half plane.} \\
\text{stable} & \leftarrow \text{non linear, stable in all basins.} \\
\text{hot}(L) & \leftarrow \text{linear, goal}(L, \text{stable}).
\end{align*}
\]

In this example, the control rule specifies that reasoning about the system's stability should be done early on if that reasoning is known to be cheap, e.g., if the system is known to be linear. Stability reasoning does not get priority, however, in the nonlinear—expensive—case. The domain-specific reasoning behind this control flow is as follows: A linear dynamical system has a unique equilibrium point, and the stability of that point—and therefore of the system as a whole—can be determined by examining the system's eigenvalues, a simple symbolic manipulation of the coefficients of the equation. \textit{Nonlinear} systems can have arbitrary numbers of equilibrium sets.
These attractors are expensive to find and evaluate. Thus, if a system is known to
be linear, its overall stability is easy to establish, whereas evaluating the stability of
a nonlinear system is far more complicated and expensive. The framework described
in this paper not only makes it easy for a domain expert to specify this kind of
knowledge, but also turns that knowledge to advantage in an elegant and powerful
way. PRET’s meta theory, which captures key concepts in differential equations and
dynamical systems, allows the inference system to take advantage of the dynamic
dependencies described above. The major advantage of this approach is that PRET’s
control knowledge is separated from the ODE theory and does not interfere with the
ODE theory’s declarative semantics. This example illustrates how control information
that originates in a domain expert’s understanding of the application domain can be
expressed cleanly and intuitively using the logic system described in this paper.

Embedded implications are also a useful tool in the automated modeling domain.
As an example consider the following (simplified) rule, which expresses the simplest
input/output stability notion for control systems, called “bounded-input, bounded-
output stability” (BIBO) [43].

\[
\text{stable} \leftarrow (\text{bounded\_output} \leftarrow \text{bounded\_input}).
\]

Furthermore, embedded implication is at work every time a negated goal is evalu-
ated. The reason for this is that—as described in Section 2—our system interprets
not/1 as negation as inconsistency: every subgoal not(p) is replaced by the embedded
implication (falsum \leftarrow p).

Even though the computational complexity of PRET’s model checker has not yet been
formally analyzed, experiments (e.g., [10, 27]) show that it performs well on engineer-
ing textbook problems. The recursive call of the inference engine that evaluates the
bodies of control rules may be viewed as a potential source of complexity, or even
infinite loops. In practice, however, the proofs of control rules bottom out quickly.

Recently, there has been an interesting discussion in the AI community about the
need for domain-dependent control information in any application. Theoretically,
there is no need for domain-dependent control because control knowledge can be
factorized into domain-independent control information and domain-dependent modal
information [49] that encodes the structure of the search space [34]. While this elegant
result is true for logic programming in general, the PRET project (and others projects as well, e.g., [51]) is a prime example of an application that requires a different approach. Having to think about control in terms of the structure of the search space is exactly what we want to avoid. The implementer of the knowledge base should instead approach it from the viewpoint of his/her domain: which rules are more abstract than others, which rules or goals trigger expensive calls to other packages, and so on.

7 Context and Related Work

The work described in this paper draws upon ideas and techniques from several areas of mathematics, engineering, and computer science; citing more than the few most important and/or most closely related publications in each of these areas would yield an excessive bibliography. In this section, we mention only the most closely related publications from the large body of literature on meta-level systems and control. References to related work that appear in the body of the paper will not be repeated here.

Some of the earliest work on meta control includes [20, 24, 25, 31, 32]. More recently, implemented logic programming languages (e.g., [4, 36, 38]) have been influenced by these ideas. Furthermore, automated planning systems (e.g., [2, 16, 57]) typically employ meta-level decision-making. The planning system TLPan uses temporal logic to express control information [1]. The constraint-based framework of Satplan and Graphplan allows such rules to be compiled into these planners [39]. An answer-set programming approach to the domain-dependent control of planners was presented in [56]. Moreover, the notion of specifying control has also been applied to the situation calculus [42].

In a different, but related, branch of the literature—called strategic proof planning—strategies that guide the proof search are explicitly represented as plans and dynamically refined during the theorem proving process [15]. This approach adds a form of explicit global control to the low-level, local, tactical control decisions of a theorem prover. Work in this area spans from the earliest research by Bundy [14] to contemporary theorem provers and integrated mathematical assistants (e.g., [7, 17, 40, 47, 48]).
The Automated Deduction community has produced a large body of systems and literature on tactical and strategic control of deduction (see, for example, [35]). Denzinger, Fuchs and Fuchs [23] describe a system that learns to re-enact previously successful proof attempts in the domain of purely equational theorem proving. The system finds solved problems that are analogous to the current problem and adapts the corresponding known proof. The search is distributed across a multi-agent architecture whose selection strategies and heuristics are expressed declaratively, using domain-specific terms.

There are important differences between the control of theorem provers in the field of automated deduction and the work presented in this paper. In automated deduction, proving theorems is the primary aim of the system. Our system, on the other hand, is designed to facilitate the expression of control knowledge in the context of a particular application domain. Theorem proving is just the vehicle, not the goal. In the application domain example in the previous section, for example, the formulation of meta-level control knowledge is crucial to the effective orchestration of a heterogeneous reasoning process: automated system identification. The underlying mathematical theory of system identification—ODE theory—is expressed as a logical theory; hence, the control of PRET’s reasoning is an instance of control of automated deduction. However, PRET employs only one theorem prover. Unlike automated deduction systems that orchestrate a suite of theorem provers (e.g., [22]), PRET orchestrates a suite of non-logic-based reasoning modules by deciding when to invoke which one and by making the results available to other modules in form of logical formulae.

In PRET’s logical paradigm [60], the invocation of modules is triggered by the evaluation of resolution goals. Therefore, some of the control information that is expressed through the dynamic ordering of goals and clauses corresponds to what in automated deduction is called strategic information. This uniform framework of expressing all control as control of a resolution prover has proven useful and intuitive in our application domain. The combination of static abstraction levels and dynamic meta level control rules allows for an effective orchestration of the automated system identification task.

Meta languages have a long history in logic and logic programming [62]. Meta lan-
guage constructs whose semantics are similar to the constructs in our system were suggested by Gallaire and Lasserre [31, 32]; however, their specification of the semantics was vague. Declaration of relevancy has a similar effect as Gallaire and Lasserre’s *finish* predicate. The idea of establishing a relationship between clauses and their names also stems from [32]. Our *notready/1* predicate is also similar to Nu-PROLOG’s *wait* predicate [53] and Gödel’s *delay* predicate [38]. Amalgamated meta-level inference for SLDNF resolution was presented by [8, 69].

A terminology for meta-level systems was suggested by van Harmelen [66]. According to that classification, our system is a bilingual object-level inference system with a ground representation of object-level goals and clauses on the meta-level. Unlike some other systems in that category [4], our system provides no guards to express directionality. If guards were available, meta predicates like *var/1* and *ground/1* could be used to choose appropriate clauses without interfering with the declarative semantics of the clauses. In our system, however, such meta predicates appear in the body of the clause and must therefore be used with care. In the bodies of meta rules we allow the full GHCIL language instead of restricting the meta language to Horn clauses. To evaluate meta-level control clauses, the GHCIL inference engine simply calls itself pseudo-recursively instead of switching to an Earley theorem prover [3, 26]. We also added the notion of abstraction levels, as described in previous sections.

Another deviation from [4]—as discussed in Section 4.4—is that we do not make use of a full ATMS. Instead, we store proven goals with relevant predicates in a hash table. Another approach to caching partial results is known as tabling, which is implemented in XSB\(^{20}\). From a computational complexity perspective, our approach is currently inferior to an ATMS or a tabling approach because adding cached solutions at the front of the regular solutions may in the worst case increase the program complexity exponentially. However, in our specific application, extensive backtracking and/or search for multiple answers is rare, and we were thus able to avoid this problem. Also, we are currently developing a version of the caching in which every cached formula will maintain a pointer to the rules from which it was derived, along with some other book-keeping information. From a functionality perspective, our approach provides additional functionality over current implementations of tabling: for hypothetical

\(^{20}\text{http://xsb.sourceforge.net}\)
goals, which are derived in the context of a nested (non-Horn) clause, the goal is stored together with the context in which it is valid.

In the Foreword to [38], Robinson calls the difference between pure logic programming and applied logic programming “a gap that has plagued the relational logic programming community since the birth of Prolog in the early 1970s.” In a perfect world, “programs are first-order theories, and computations are deductions from them.” Recently, several papers (for example, [41]) have assigned declarative semantics to procedural constructs like the cut or negation as failure by stratifying programs or restricting program models. Our solution to this problem is to disallow procedural constructs and to restrict negation syntactically to negation as inconsistency with intuitionistic semantics.

The problem of ordering query subgoals—and optimizing queries in general—has been studied in both the database and the logic programming communities (e.g., [55, 68]). The continuing attempts of the logic programming community to make applied logic programs more declarative and thus more readable and comprehensible has a strikingly similar counterpart in the database community. Relational query languages [64] allow the desired data to be specified declaratively. Query optimizers, however, are typically programmed in procedural terms. One might argue that query optimizers in databases correspond to control components in logic programs. Cherniack has developed a system that expresses the information as to how queries are optimized declaratively as well, namely as declarative rewrite rules [19]. In a sense, the concept of declarativeness is moving down the food chain. Naturally, this has to stop somewhere: Cherniack’s system specifies the information “which rewrite rule should be applied when” in procedural terms. Similarly, our system executes the bodies of control rules from left to right, i.e., procedurally.

8 Conclusion

We have presented an implemented logic system whose language is that of generalized Horn clause intuitionistic logic with negation as inconsistency. The system achieves three important goals: equivalence of declarative and operational semantics, explicit and declarative representation of control information, and smooth interaction among
8 CONCLUSION

various heterogeneous reasoning modes. We believe that these ideas can play an important role in solutions to the general problem of how to specify control and reasoning strategies in AI systems.

These goals have been accomplished by integrating and implementing several carefully chosen techniques. Static abstraction levels and dynamic meta control rules explicitly specify the deduction strategy of the inference engine, thereby allowing the reasoner to intelligently navigate in the search tree. An explicit representation of the theorem prover’s state allows information to be passed between various logical and non-logical reasoning modules. The abstraction levels and meta control rules specified by the programmer orchestrate the calls to these reasoning modules. Furthermore, an intelligent caching mechanism stores relevant formulae and makes them available for reuse in later proof attempts. Typical examples of such relevant formulae are intermediate results of expensive calls to various reasoning modules.

As an example, we have incorporated our system into PRET, an automated modeling tool that reasons about ordinary differential equations (ODEs). The logic system described in this paper is an effective and efficient reasoning core for this process. Its design allows a domain expert to express knowledge about dynamic systems and ODEs in a natural, declarative manner. The control information, which specifies how the domain knowledge is to be processed, is also formulated declaratively, but separately from the domain knowledge. This approach facilitates correctness and clarity of the domain knowledge because the expert need not be concerned with control strategies when formulating knowledge about mathematical truths about dynamic systems and ODEs. As demonstrated in the PRET system, an appropriate set of control rules leads to the desirable behavior of the reasoner, which—in this case—amounts to an efficient search for an ODE: one that prioritizes cheap, abstract-level reasoning over expensive low-level reasoning whenever possible. Finally, the system identification task draws on a variety of heterogeneous reasoning modules. The logic system described in this paper allows PRET to smoothly integrate these modules with each other, orchestrating them through careful control of its first-order theorem prover.

The system described in this paper implements an approach that can be viewed as a hybrid between what is known as tactical and strategic control in automated theorem
proving. Rather than using different control mechanisms for tactical and strategic control, a PRET knowledge engineer expresses all control as control of a resolution prover. Combined with the abstraction hierarchy of more and more refined domain theories, this approach provides—in our experience—just the right tools for an effective orchestration of the automated system identification task. More generally, we expect that our approach will prove useful in other complex AI tasks that require the integration of heterogeneous reasoning modules that employ different reasoning paradigms. The conceptually clear separation between object-level domain knowledge and dynamic meta-level control knowledge, along with the domain theory's abstraction levels, allows for a formulation of control information that corresponds directly to the user’s intuitions about more-abstract and less-abstract concepts and about more-expensive and less-expensive reasoning techniques in the application domain. The main goal of the design presented in this paper is not to give the logic system the means to learn or improve its reasoning strategies. Rather, it is to give the user (or knowledge engineer) the means to formalize domain-dependent information about various degrees of abstraction and about various degrees of reasoning cost in a way that is conceptually clear and that corresponds to a body of knowledge and expertise in the application domain. For this reason, an numerical comparison of PRET’s performance with and without meta-control (along the lines of [50], for example) would be besides the point. PRET’s control information does not merely make its reasoning more efficient; it makes it feasible. Nevertheless, it would be interesting to better understand the computational cost of PRET’s meta-control module, and we are currently working on an empirical evaluation.

Acknowledgements

Matt Easley and Tom Wrensch contributed ideas and code to this project. Much of the work described in this paper is built on concepts developed by Clemens Beckstein and Gerhard Tobermann. The first author, in particular, wishes to thank them for many helpful discussions and continuing support.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


