Putting Declarative Meta Control to Work

Apollo Hogan and Reinhard Stolle and Elizabeth Bradley*
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309–0430
{hogan,stolle,lizb}@cs.colorado.edu

December 20, 1998

University of Colorado Technical Report CU-CS-856-98
In review for publication in the
"IEEE Transactions on Systems, Man, and Cybernetics"

Abstract

We present a logic programming system that accomplishes three important goals: equivalence of declarative and operational semantics, declarative specification of control information, and smoothness of interaction with non-logic-based programs. The language of the system is that of Generalized Horn Clause Intuitionistic Logic with negation as inconsistency. 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, logic programming, declarative programming, artificial intelligence.

*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.

1 Introduction

Many languages that are designed for the declarative representation of domain knowledge—one of the core concepts of Artificial Intelligence—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 (**R**eason Maintenance Based **I**nference **S**ystem for Generalized Horn **C**lause Logic) at the University of Erlangen [BST96, BT92]. The logic system presented in this paper is the core of PRET, an automated modeling tool that finds ordinary differential equations (ODEs) that model black-box dynamical systems [BS96, SB98]. The achievement of the three goals listed above is crucial to the success of this modeling task.

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, 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 [Llo87]. 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.

The language of the logic system presented in this paper is that of Generalized Horn Clause Intuitionistic Logic (GHCIL) [McC88a, McC88b]. 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 framework. We conclude the paper with some pointers to related work.

2 The Language

GHCIL clauses are (implicitly) universally quantified implications of the following form.

1. Every definite Horn clause is a GHCIL clause.[1]
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,

$$\mathit{dedicated}(P) \leftarrow (\mathit{working}(P) \leftarrow \mathit{assigned}(W, P), \mathit{unfinished}(W))$$

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 [McC88a, McC88b].)

In our system, there are several distinguished predicates that may occur in GHCIL clauses. One of them is *falsum*: GHCIL clauses having *falsum* 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* [GS86] and interprets $\mathit{not}(p)$ as an abbreviation for $\mathit{falsum} \leftarrow p$. For example, consider the following rulebase:

$$\begin{aligned} 1: &\ \mathit{falsum} \leftarrow \mathit{male}(X), \mathit{female}(X). \\ 2: &\ \mathit{male}(\mathit{john}). \\ 3: &\ \mathit{female}(\mathit{betty}). \\ 4: &\ \mathit{male}(\mathit{pat}). \end{aligned}$$

[1] 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.

The query *?not*(*male*(*X*)) succeeds with X bound to *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.

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 [SS86]. 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.

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:

$$\begin{aligned} 1: &\ \mathit{ord}(\mathit{succ}(X)) \leftarrow \mathit{ord}(X). \\ 2: &\ \mathit{ord}(0). \\ 3: &\ \mathit{ord}(\omega). \end{aligned}$$

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 *?ord*(*succ*(*Z*)). When adding new rules (e.g., $\mathit{ord}(\omega_1)$), 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, ω, 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 over non-ground clauses for the predicate *ord/1*.[2]

3.1 Static Control: Abstraction Levels

One method for specifying meta control in a declarative fashion is *abstraction levels*: static numeric annotations that describe the order in which clauses are searched, enforcing the preference of abstract proofs over less-abstract ones. These impose static *global* constraints on the search. For example, suppose that there are only two abstraction levels, *low* and *high*. Then any proof that uses only clauses with *high* abstraction levels will be preferred to any proof that uses a clause with a *low* abstraction level, even if the latter proof is much shorter.[3]

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 where dynamic meta control is needed.

3.2 Dynamic Control: Meta Rules

In PROLOG, for example, control information interferes with logical statements in order to achieve an efficient evaluation of huge sets of unit clauses [SS86]. Consider the following example (from [BST96]).

$$\begin{aligned} &\mathit{grandparent}(X,Y) \leftarrow \mathit{var}(Y), !, \mathit{parent}(X,Z), \mathit{parent}(Z,Y). \\ &\mathit{grandparent}(X,Y) \leftarrow \mathit{parent}(Z,Y), \mathit{parent}(X,Z). \end{aligned}$$

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 *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

$$\mathit{grandparent}(X,Y) \quad \leftarrow \quad \mathit{parent}(X,Z), \mathit{parent}(Z,Y).$$

[2] A predicate p of arity n is denoted p/n.

[3] However, the programmer will typically assign abstraction levels to the rules in such a way that short proofs are also abstract proofs.

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:

$$\begin{aligned} \mathit{before}(L_1, L_2) \;\leftarrow\; & \mathit{goal}(L_1, \mathit{parent}(X, Y)), \\ & \mathit{goal}(L_2, \mathit{parent}(Y, Z)), \\ & \mathit{ground}(X), \mathit{var}(Z). \\ \mathit{before}(L_2, L_1) \;\leftarrow\; & \mathit{goal}(L_1, \mathit{parent}(X, Y)), \\ & \mathit{goal}(L_2, \mathit{parent}(Y, Z)), \\ & \mathit{ground}(Z), \mathit{var}(X). \end{aligned}$$

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 *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 *notready/1* and *hot/1* for the selection of subgoals to be resolved and *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 *before/2*. Subgoals that are proved to be *notready/1* may not be chosen; within these constraints, *hot/1* subgoals receive priority. The rule

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

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 *clause/2* and *goal/2* establish names for clauses and currently active subgoals.[4] If the meta rules do not completely specify the control decisions, the default control is from left to right.

The semantics of all meta predicates follows that of [BST96]; we omit the details of the declarative and procedural specification of the meta predicates.

[4]The meta predicates *var/1* and *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.

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, even if it is not part of the current proof tree (the current path of the search tree). 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 [FdK93]. 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.[5] 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 what motivated 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 any place in the search tree where a decision must be made. 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 $p(X)$.

$$\begin{aligned} 1: &\quad p(Y) \leftarrow q(y), r(Y). \\ 2: &\quad p(a) \leftarrow q(a). \end{aligned}$$

Then the system will create a choice point for the goal $p(X)$ that contains the matching clauses 1 and 2. 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

[5] A graphical user interface (GUI) also takes advantage of the explicit internal state of the inference engine: The engine is easily interrupted and restarted and the GUI allows for examination of the current state of the search.

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.[6]

The 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.

4.2 Sample Inference

A flow chart of the inference process is shown in Fig. 1. Circles ①, ②, ③, ④, and ⑤ are the important internal states of the inference engine.

To illustrate how the inference engine operates, we step through the inference process for a simple query. Let the database of clauses be given by

$$
\begin{aligned}
1: &\ A \leftarrow (B \leftarrow C), D.\\
2: &\ A \leftarrow X.\\
3: &\ B \leftarrow F.\\
4: &\ B \leftarrow X.\\
5: &\ D \leftarrow E.\\
6: &\ E.\\
7: &\ F \leftarrow C.
\end{aligned}
$$

Suppose the query is $?A$. The initial conditions of the inference engine are shown in Fig. 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[7] for the engine selects A (the only choice in this case) as the next goal to prove. A choice point (written c.p. in the figure) is created from all clauses in the database and all

[6]There are other methods for handling the need for recursive calls. Another embedding of PROLOG into SCHEME [Hay87] 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.

[7]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.

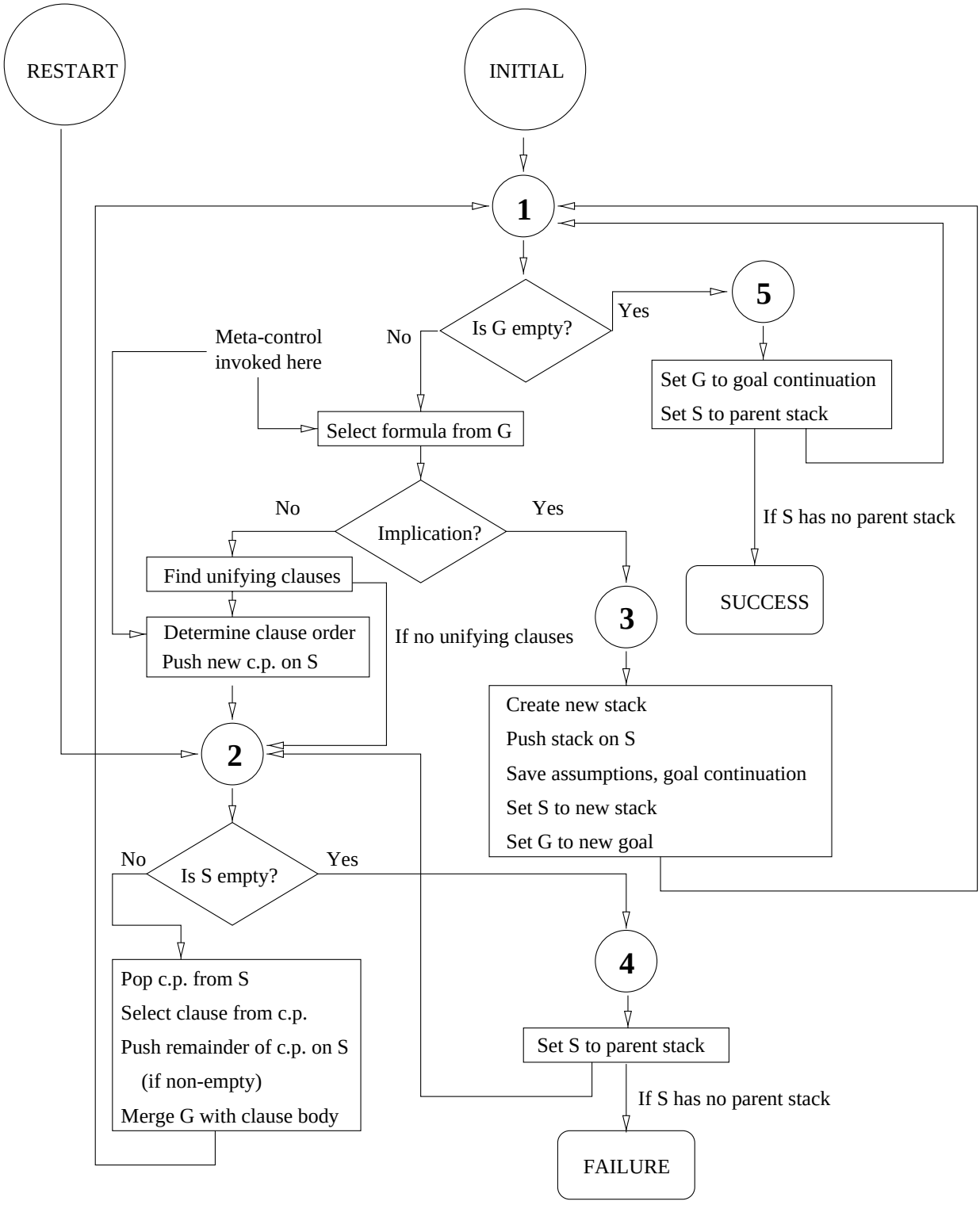

Figure 1: Flow-chart of the inference engine.

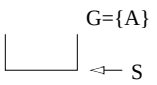

Figure 2: Initial conditions of the inference engine.

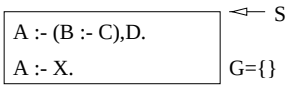

Figure 3: Inference engine, state=②

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 Fig. 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). 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 (Fig. 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 ① until G becomes empty (Fig. 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 Fig. 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 the original, outer-most one (Fig. 11). 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 for two reasons. First, it allows previously derived

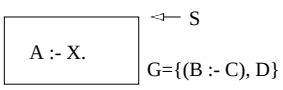

Figure 4: Inference engine, state=①

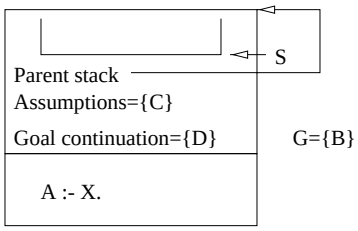

Figure 5: Inference engine, state=①

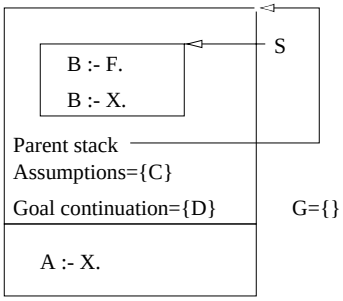

Figure 6: Inference engine, state=②

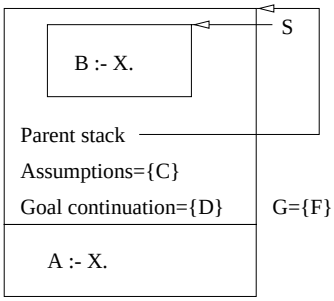

Figure 7: Inference engine, state=①

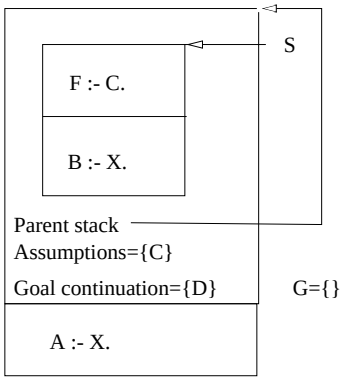

Figure 8: Inference engine, state=②

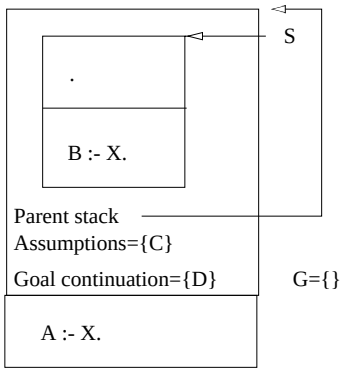

Figure 9: Inference engine, state=①

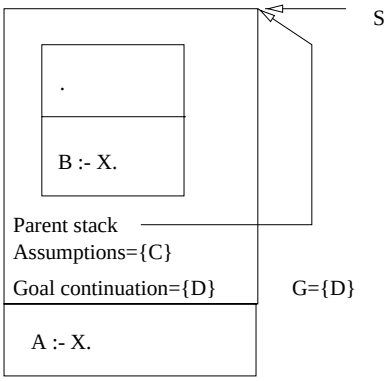

Figure 10: Inference engine, state=①

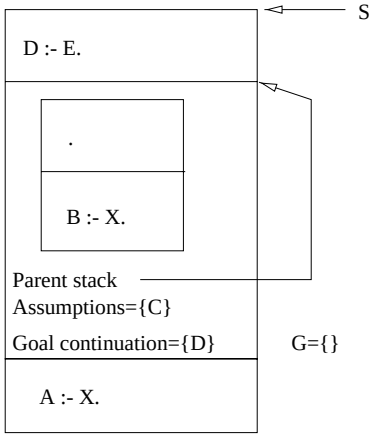

Figure 11: Inference engine, state=②

knowledge to be reused and passed around to other (possibly non-logical) reasoning modules. Second, the graphical user interface has access to the complete search tree. These two features are of crucial importance for a multi-modal reasoning system, an example of which we describe in Section 6 of this paper.[8] 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.

4.3 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 *relevant* predicates: those predicates in which other modules are interested and those that are expensive to evaluate. Relevant predicates are declared as such using the meta predicate *relevant/1* [BT92].

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

[8] A third advantage of an explicit representation is that the meta control can choose between all subgoals in the call stack as opposed to just the subgoals of the inner-most stack. In this case, nested implications are not necessarily evaluated before the goal in which they are embedded. However, our implementation does not take advantage of this possibility. Currently, the reasoner always finishes embedded subgoals before returning to the embedding goal.

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 θ, then we store only A. This amounts to θ-subsumption [vdL95] in the case of atomic formulae. In the case of embedded goals (implications) this is only a crude form of caching; handling full θ-subsumption in this general case is NP-complete [GJ79]. 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 is also the reason why we decided against the use of a full truth maintenance system [FdK93, dK86]. 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; our system maintains just enough information to be able to pass all relevant current knowledge to other modules and to avoid duplicating work in evaluating time-intensive predicates.

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.

$$
\begin{aligned}
\mathit{falsum} &\leftarrow \mathit{time\_series}(T), \mathit{chaotic}(T), \mathit{periodic}(T).\\
\mathit{falsum} &\leftarrow \mathit{time\_series}(T), \mathit{chaotic}(T), \mathit{linear}(T).\\
\mathit{chaotic}(T) &\leftarrow \mathit{time\_series}(T), \mathit{expensive\_test}(T, \mathit{chaotic}).\\
\mathit{periodic}(T) &\leftarrow \mathit{time\_series}(T), \mathit{expensive\_test}(T, \mathit{periodic}).\\
\mathit{linear}(T) &\leftarrow \mathit{time\_series}(T), \mathit{expensive\_test}(T, \mathit{linear}).\\
\mathit{time\_series}(\mathit{ts}).
\end{aligned}
$$

Suppose that *ts* is an experimental time series that happens to be chaotic (hence non-periodic). Consider the query $\mathit{falsum} \leftarrow \mathit{linear}(\mathit{ts})$ whose interpretation is: "is the time-series non-linear?"[9] If we assume a depth-first-left-right strategy, the system evaluates the formulae in the following order:

[9]The formula *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 *linear-system*(*current-model*)) cannot be chaotic.

not(*linear*(*ts*))
falsum ← *linear*(*ts*)
falsum
chaotic(*ts*)
expensive_test(*ts*, *chaotic*)
periodic(*ts*)
expensive_test(*ts*, *periodic*) **fails**
falsum
chaotic(*ts*)
expensive_test(*ts*, *chaotic*)
linear(*ts*) **succeeds**

The system does not do the numeric test for linearity because we are assuming *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 the program in which we have tested this inference system. The PRET program, which constructs ODE models of physical systems, 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*.

4.4 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 embed-

ded implications are. Conceptually, a new incarnation of an inference process tries to finish a proof of the relevant subgoal before other subgoals receive attention.[10] 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. 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 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[11] 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 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. (Consult [BST96] for a formal description of the semantics of the control predicates.)

[10] Gallaire and Lasserre [GL82] achieve a similar effect using the predicate *finish*.

[11] 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.

5 Correctness and Completeness

Generalized Horn Clause Logic is intuitionistically equivalent to a certain subset of McCarty's *Clausal Intuitionistic Logic* [McC88a, McC88b]. According to Tobermann [Tob94], 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 [Tob94] has shown that the selection function for a RISC-type prover may perform arbitrary computations. The only condition 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. It does, however, affect combinatorial completeness; a different order may make the prover follow an infinite path before it finds some logical consequence of the program. One of the intended usages of the meta predicate *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. However, it does remain the responsibility of the programmer.

The calculus that results from adding the abstraction level mechanism 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. Now consider completeness. Relying on the completeness of the underlying inference engine, 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 since no rules are removed from the logic program; rather, the cache is *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. The underlying idea of that improved version is that every cached formula can keep a pointer to the rules from which it was derived and 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 [BS96, SB98] automates system identification [Lju87]: 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.

The model checker makes use of several non-logic-based modules, e.g., the commercial symbolic algebra package Maple [CGG$^+$91], a simple qualitative envisioning module, a nonlinear numerical parameter estimator [BOR98], and a geometric reasoner for intelligent data analysis [BE98]. Calls to these modules require knowledge to be passed to them explicitly. This knowledge is made available by the inference engine by declaring the necessary predicates as relevant.

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 PRET's parameter estimation module NPER [BOR98], for example, ranges between a couple of seconds and several minutes. What lets PRET manage the complexity of its task (finding an ODE model

for a given dynamic system) is its ability to dynamically orchestrate application of its ODE rules and the various reasoning modes that are triggered by the resulting evaluations. 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, PRET tries to discard models by purely qualitative means before resorting to numerical techniques. In other qualitative reasoning systems that reason at various abstraction levels (e.g., [YZ96, MB97]), the levels are implicitly defined by the system's architecture and data structures. In PRET, every rule is explicitly assigned an abstraction level number. Similarly, the logic engine's dynamic control is used in PRET to guide the search toward a cheap and quick proof of *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.

$$\begin{aligned} \mathit{stable} &\leftarrow \mathit{linear}, \mathit{all\_roots\_in\_left\_half\_plane}. \\ \mathit{stable} &\leftarrow \mathit{non\_linear}, \mathit{stable\_in\_all\_basins}. \\ \mathit{hot}(L) &\leftarrow \mathit{linear}, \mathit{goal}(L, \mathit{stable}). \end{aligned}$$

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. *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. PRET's meta theory allows the inference system to take advantage of such dynamic dependencies. 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.

Even though the computational complexity of PRET's model checker has not yet been formally analyzed, experiments show that it performs well on engineering 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. The only use of embedded implication so far was the interpretation of *not/1* as negation as inconsistency.

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 [Men64] that encodes the structure of the search space [GG91]. While this elegant result is true for logic programming in general, the PRET project (and others projects as well, e.g., [Min96]) 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.

PRET's graphical user interface (GUI) also takes advantage of the explicit state representation of the inference engine. For example, the GUI allows the user to interrupt the inference engine at any time. Having interrupted the computation, the user is able to examine the inference engine's state, restart the computation, or even save the computation in progress and start a new computation. The GUI displays the knowledge that has been derived so far, proof trees for this derived knowledge, and the proof tree in progress. Whereas in traditional logic systems proof trees are usually built by meta interpreters, in PRET this task is trivially implemented using the explicit state representation.

7 Related Work

Throughout the body of this paper we have discussed a variety of related papers; those citations shall not be repeated here. We mention here only the most closely related publications from the large body of literature on meta level systems and control.

Some of the earliest work on meta control includes [Dav80, DFN86, DL84]. Meta language constructs whose semantics are similar to the constructs in our system were suggested by Gallaire and Lasserre [GL82, GL79]; 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 [GL82]. Our *notready/1* predicate is also similar to NU-PROLOG's *wait* [Nai85] and GÖDEL's *delay* [HL94]. Amalgamated meta level inference for SLDNF resolution was presented by [BK82, Yal91].

Unlike [BST96], 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. Also, as discussed in this paper, we do not make use of a full ATMS. In the bodies of meta rules we allow the full GH-

CIL 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 [Ear70, BK91]. We also added the notion of abstraction levels, as described in previous sections.

A terminology for meta level systems was suggested by van Harmelen [vH91]. Similar to [BST96], our system is—according to that classification—a bilingual object-level inference system with a ground representation of object-level goals and clauses on the meta level.

In the Foreword to [HL94], 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, [Lin97]) 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 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 [Ull88] 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 [CZ98]. 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 various heterogeneous reasoning modes.

These goals have been accomplished by integrating and implementing several 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 to non-logical reasoning modules that

are orchestrated through the specified control information. 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.

We have shown an application example of our system in the domain of the theory of ordinary differential equations. Our program PRET is an implemented automated system identification tool that uses the logic system described in this paper as its reasoning core.

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 second author, in particular, wishes to thank them for many helpful discussions and continuing support.

References

[BE98] E. Bradley and M. Easley. Reasoning about sensor data for automated system identification. *Intelligent Data Analysis*, 2(2), 1998. Also in *Second International Symposium on Intelligent Data Analysis (IDA-97)*.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, London, 1982.

[BK91] C. Beckstein and M. Kim. Generalized Earley deduction and its correctness. In Thomas Christaller, editor, *GWAI-91: 15th German Workshop on Artificial Intelligence*, volume 285 of *Informatik-Fachberichte*, pages 1–10. Springer, Berlin, Heidelberg, 1991.

[BOR98] E. Bradley, A. O'Gallagher, and J. Rogers. Global solutions for nonlinear systems using qualitative reasoning. *Annals of Mathematics and Artificial Intelligence*, 1998. To appear. Also in *Eleventh International Workshop on Qualitative Reasoning (QR-97)*.

[BS96] E. Bradley and R. Stolle. Automatic construction of accurate models of physical systems. *Annals of Mathematics and Artificial Intelligence*, 17:1–28, 1996.

[BST96] C. Beckstein, R. Stolle, and G. Tobermann. Meta-programming for generalized horn clause logic. In *Fifth International Workshop on Metaprogramming and Metareasoning in Logic (META-96)*, pages 27–42, 1996. Bonn, Germany.

[BT92] C. Beckstein and G. Tobermann. Evolutionary logic programming with RISC. In *Fourth International Workshop on Logic Programming Environments*, pages 16–21, November 1992. Washington, D.C. Technical Report TR 92-143, Center for Automation and Intelligent Systems Research at Case Western Reserve University.

[CGG+91] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual*. Springer, New York, 1991.

[CZ98] M. Cherniack and S. Zdonik. Changing the rules: Transformations for rule-based optimizers. In *ACM SIGMOD International Conference on Management of Data*, June 1998. Seattle, WA.

[Dav80] R. Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15(3), 1980.

[DFN86] P. Devanbu, M. Freeland, and S. A. Naqvi. A procedural approach to search control in PROLOG. In *European Conference on Artificial Intelligence (ECAI-86)*, pages 53–57, 1986.

[dK86] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[DL84] M. Dincbas and J. P. Le Pape. Metacontrol of logic programs in METALOG. In *International Conference on Fifth Generation Computing Systems*, pages 361–370, 1984.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[FdK93] K. D. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, Cambridge, MA, 1993.

[GG91] M. Ginsberg and D. Geddis. Is there any need for domain-dependent control information? In *Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 452–457, 1991.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[GL79] H. Gallaire and C. Lasserre. Controlling knowledge deduction in a declarative approach. In *International Joint Conference on Artificial Intelligence (IJCAI-79)*, pages S1–S6, 1979.

[GL82] H. Gallaire and C. Lasserre. Metalevel control for logic programs. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*. Academic Press, London, 1982.

[GS86] D. M. Gabbay and M. J. Sergot. Negation as inconsistency I. *The Journal of Logic Programming*, 3(1):1–36, April 1986.

[Hay87] C. T. Haynes. Logic continuations. *The Journal of Logic Programming*, 4:157–176, 1987.

[HL94] P. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, Cambridge, MA, 1994.

[Lin97] F. Lin. Applications of the situation calculus to formalizing control and strategic information: The Prolog cut operator. In *Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1412–1418, August 1997. Nagoya, Japan.

[Lju87] L. Ljung, editor. *System Identification; Theory for the User*. Prentice-Hall, Englewood Cliffs, N.J., 1987.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 2nd extended edition, 1987.

[MB97] P. J. Mosterman and G. Biswas. Formal specifications for hybrid dynamical systems. In *Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 568–573, August 1997. Nagoya, Japan.

[McC88a] L. T. McCarty. Clausal intuitionistic logic I. Fixed-point semantics. *The Journal of Logic Programming*, 5:1–31, 1988.

[McC88b] L. T. McCarty. Clausal intuitionistic logic II. Tableau proof procedures. *The Journal of Logic Programming*, 5:93–132, 1988.

[Men64] E. Mendelson. *Introduction to mathematical logic*. Van Nostrand, Princeton, NJ, 1964.

[Min96] S. Minton. Is there any need for domain-dependent control information? A reply. In *Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 855–862, 1996.

[Nai85] L. Naish. *Negation and Control in PROLOG*, volume 238 of *Lecture Notes in Computer Sciences*. Springer, Berlin, 1985.

[SB98] R. Stolle and E. Bradley. Multimodal reasoning for automatic model construction. In *Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998. Madison, Wisconsin.

[SS86] L. Sterling and E. Shapiro. *The Art of PROLOG*. MIT Press, Cambridge, MA, 1986.

[Tob94] G. Tobermann. *Verallgemeinerte Hornklausellogik: vom logischen Kalkül zum Logik-Programmiersystem*. PhD thesis, Universität Erlangen-Nürnberg, 1994.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, Rockville, Maryland, 1988.

[vdL95] P. van der Laag. *An Analysis of Refinement Operators in Inductive Logic Programming*, volume 102 of *Tinbergen Institute Research Series*. Thesis Publishers, Amsterdam, 1995.

[vH91] F. van Harmelen. *Meta-level Inference Systems*. Morgan Kaufmann, 1991.

[Yal91] L. U. Yalcinalp. *Meta-Programming for Knowledge Based Systems in PROLOG*. PhD thesis, Case Western Reserve University, 1991. Tech. Rep. TR 91-141.

[YZ96] K. Yip and F. Zhao. Spatial aggregation: Theory and applications. *Journal of Artificial Intelligence Research*, 5:1–26, 1996.