

# The OMRS Project: State of the Art

Fausto Giunchiglia<sup>a</sup> Piergiorgio Bertoli<sup>a</sup> Alessandro Coglio<sup>b</sup>

<sup>a</sup> *IRST (Istituto per la Ricerca Scientifica e Tecnologica), Trento, Italy*

<sup>b</sup> *Kestrel Institute, Palo Alto, California (USA)*

---

## Abstract

The state of the art for reasoning systems is unsatisfactory in several respects. In most cases, provers are poorly specified, hardly interconnectible, and they require a deep insight of their custom features in order to fully exploit their capabilities. The OMRS project is aimed at providing a general framework for specifying, structuring, and interoperating provers. This paper surveys the current achievements of the research performed within the OMRS project, under both the theoretical and the experimental side, and provides a perspective of its future evolution.

---

## 1 Introduction

In the last decades, a large variety of provers<sup>1</sup> implementing various kinds of reasoning have been developed and used to accomplish complex tasks. For instance, interactive theorem provers have been employed to mechanically prove deep mathematical theorems ([17,35,37,7]) as well as correctness of non-trivial hardware and software systems ([6,8,36,44]). As another example, the use of model checking tools has become a de-facto standard approach (also in many industrial projects) to validate the design of hardware components or communication protocols ([11,24]). Furthermore, decision procedures of various kinds (e.g., for propositional calculus, linear arithmetic, etc.) are embedded into complex software systems ([25,41,30]). A database of provers can be found in [39] along with links to related surveys and other information.

In spite of the availability of a variety of powerful provers, the state of the art in this field is unsatisfactory in many respects. In fact, provers are designed following ad-hoc approaches, they are often poorly specified, and implement custom interfaces to interact with the user (and with the external environment in general). This raises several issues:

---

<sup>1</sup> We use the word *prover* to denote any piece of software with reasoning capabilities (e.g., mechanized decision procedures, unifiers, rewriters, constraint solvers, automated or interactive theorem provers). When we want to distinguish stand-alone provers from provers embedded inside other software, we refer to the former as *systems*, and to the latter as *modules*.

- it is extremely difficult to interconnect different provers in an effective and semantically sound manner. However, this would be highly desirable (and often even necessary) when tackling complex problems. In fact, interconnecting provers is needed in order to integrate reasoning in different logics as well as to achieve cooperation of different inference abilities;
- complete trust in provers is not achieved. In most cases, provers present themselves as very complex black boxes, whose internal design cannot be understood by a user. This is a problem, especially when using provers in safety-critical applications: considerable effort is spent in detecting possible faults of the reasoning tools;
- users are required a deep know-how of the specific interfaces and heuristics of the provers they make use of. This is an obstacle to reconverting users from established tools to new and more advanced tools. Further, that involves a huge startup cost in verification tasks, which may be a decisive factor when industry choose whether to adopt a formal approach to validate their products, or rather follow standard testing techniques.

The OMRS project is motivated by the above considerations. Our long term goal is the development of a general framework for specifying, structuring, and interoperating provers. Our focus is the formalization of the main architectural and implementational choices which underlie the construction of (state-of-the-art) provers. A prover built according to our framework is called an *OMRS*<sup>2</sup>.

This paper gives a survey of the current status of the OMRS project, in terms of the design of the general architecture, and of its components. It gives a snapshot of our current development of the OMRS architecture, showing the current achievements and including, whenever possible, concrete examples showing the relationships between our concepts and some aspects of real-world provers. Section 2 describes the high-level architectural design of OMRS. Sections 3 and 4 describe two components of the architecture, namely logic and control. Section 5 describes some ongoing theoretical developments for these two components. Finally, Section 6 draws some conclusions.

## 2 The architecture

In general, a prover mechanizes one or more logics, implements some search heuristics over the mechanized logics, and realizes some interfaces to connect with the external environment (e.g., a human user). In current (state-of-the-art) provers, these aspects are usually deeply intertwined (as depicted in the

---

<sup>2</sup> Originally, the OMRS project was essentially motivated by the goal of building *open* provers, i.e. easily interconnectible to each other (or at least more easily than current technology allows). In fact, OMRS is an acronym for *Open Mechanized Reasoning System*. Despite the current broader goals of the project (of which interconnection is one), the name OMRS remains for historical reasons.

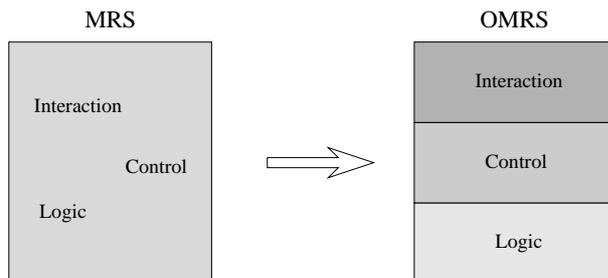


Fig. 1. Current prover vs. OMRS

left part of Figure 1), both within their implementation and their presentation. For instance, data structures may contain a mix of “logic information” (e.g., a representation of the conjecture being proved), “control information” (e.g., some user-supplied hints used by the prover to select among many possible inference steps), and “interaction information” used for the interface (e.g., formatting information for messages to be printed on the screen).

In the OMRS architecture [19], these aspects are clearly separated into three layered components, as shown in the right part of Figure 1. Thus, we can state the following equation:

$$\text{OMRS} = \text{Logic} + \text{Control} + \text{Interaction}$$

The OMRS components are inherently related in a hierarchical way: the definition of the logic is independent of the way heuristics and interfaces are implemented or presented; control is based on the definition of logic, and interaction wraps the other components by “presenting” them to the external world. The hierarchical nature of the architecture allows to decompose the complexity of specifying and implementing the aforementioned aspects into distinct phases. Compositionality of provers is supported by adopting, for every architectural component, a wide-spectrum formalism as a common ground.

### 3 Logic

The logic component of an OMRS defines the assertions manipulated by the prover, and the (elementary) inference steps which the prover can perform upon them. This is formally captured by the OMRS concept of *reasoning theory* (*RTh*). A *RTh* consists of a *sequent system* and some *inference rules*, which respectively model assertions and inference steps. Based on the concept of *RTh*, the concept of *reasoning structure* is defined. A reasoning structure constitutes an extension of the standard notion of proof structure. We now examine each of these concepts in turn.

### 3.1 Sequent systems

The (data structures encoding the) assertions manipulated by actual provers, typically have a richer structure than the assertions of the formal systems that the provers mechanize. For instance, NQTHM ([9]) mechanizes propositional calculus with equality and function symbols, but formulas in such a logic are internally encoded in clausal form (i.e., as lists of lists of literals) in the waterfall, in the form of type assertions in other modules, etc. Furthermore, often some computations of provers consist, from a logical point of view, in performing “schematic” reasoning, i.e. deduction upon assertions containing place-holders which get filled in later as computation progresses. There is a large number of examples of such computations in NQTHM. Performing schematic reasoning often demands for delayed check of applicability conditions of inference rules (e.g., that a variable does not occur free in a formula). This raises the need for an explicit representation of (possibly schematic) constraints, as well as for a mechanism to check constraint satisfaction.

In order to conveniently formally model the above aspects, [19] introduces the following notion of *sequent system*:

$$Ssys = \langle S, C, \models, I, \_ [\_] \rangle$$

$S$  is the set of *sequents*.  $C$  is the set of *constraints*, and  $\models \subseteq (\mathbf{P}_\omega(C) \times C)$  is a satisfaction relation on constraints<sup>3</sup>.  $I$  is the set of *instantiation maps* (or just *instantiations*), and  $\_ [\_]$  is the operation for application of instantiations to sequents and to constraints, that is  $\_ [\_] : S \times I \rightarrow S$  and  $\_ [\_] : C \times I \rightarrow C$ . Sequents represent the assertions or judgments used by provers. They are taken as primitive, with no assumption on the form they may take. Constraints represent applicability conditions for the inference rules (see next subsection), and again no commitment is done on their form.  $\models$  represents the conditions to be met by constraint checking or solving mechanisms. Both sequents and constraints can contain schematic parts. Instantiations abstractly capture the notion of substitution of schematic parts, and  $\_ [\_]$  provides the way to fill in schemata. As sequents and constraints may take any form, instantiations and their application are defined axiomatically in [19]. A concrete example satisfying that axiomatization consists in sequents and constraints being terms, and instantiations being (partial) maps from variables to terms.

We now introduce some example sequent systems, namely a sequent system for the logical calculus ND, and some sequent systems for the ACL2 prover.

**Example (ND sequent system):** ND is a natural deduction system for classical first-order predicate logic [34].  $Ssys_{ND}$  is the sequent system of our representation of ND as a reasoning theory. Its sequents are pairs  $\Gamma \vdash A$  consisting of a set of assumption formulas,  $\Gamma$ , and a conclusion formula,  $A$ . Term and formula meta-variables of traditional presentations are reified as

<sup>3</sup> If  $A$  is a set,  $\mathbf{P}_\omega(A)$  is the sets of all finite subsets of  $A$ .

schematic variables. Terms and formulas are built in the usual way, out of individual constants, individual variables, function and predicate symbols, schematic variables for terms and formulas, propositional connectives, and quantifiers.  $\Gamma \vdash A$  states that  $A$  is a consequence of  $\Gamma$ . Instantiations are maps from schematic variables to syntactic entities of appropriate sort, and instantiation application is the homomorphic lifting of these maps to terms, formulas, and other syntactic entities. Constraints include equations between expressions of the same syntactic sort. Additional constraints are built from predicates which reify meta-predicates of traditional presentations (therefore distinct from those used in formulas), such as the binary *NoFree*, which holds upon a set of formulas  $\Gamma$  and a variable  $a$  iff  $a$  does not occur free in any of the formulas in  $\Gamma$ . ■

**Example (ACL2 sequent systems):** ACL2 ([26]) is a state-of-the-art induction-based theorem prover which extends and enhances the Boyer-Moore prover, Nqthm. ACL2 embeds extremely powerful heuristics and decision procedures; it has been successfully applied to prove mathematical theorems, and to verify properties of systems (see, e.g., [6,8,37]). In ACL2, the proof of a conjecture is driven by a number of heuristics, called *processes*, integrating several inference techniques (such as simplification by a database of rewrite rules, elimination of irrelevants, cross-fertilization, etc.). These processes are called within a top-level search algorithm usually referred to as *waterfall* algorithm. Processes themselves integrate several reasoning modules; for instance, the simplifier process exploits a rewriter module, and a linear arithmetic specialist which operates upon sets of inequalities representing parts of the conjecture being proved.

Corresponding to this structure, the assertions manipulated by ACL2 are formalized, in OMRS, using a number of sequent systems. In [4] an analysis of the sequents involved in the waterfall algorithm is presented, built on top of the work of [19] and [12]. Also, in [19], [14] and [1], the rewriter and linear arithmetic specialist are analyzed in depth. We resume some of the results of these analysis<sup>4</sup>.

Each ACL2 process can be viewed as a derived inference rule backward applied to a clause  $cl$ , producing, in case of success, a set of clauses  $\widetilde{cls}$  whose conjunction implies  $cl$ . Processes act under a context which defines the current logical theory of the prover, recorded as a sequence of theory-modifying events (e.g. function definitions, axiom declarations, etc.). This can be represented by OMRS sequents of the form

$$th \vdash_W cl \rightsquigarrow \widetilde{cls}$$

where  $th$  is an ACL2 theory,  $cl$  is a clause, and  $\widetilde{cls}$  is a set of clauses. Intuitively, an instance of such a sequent asserts that a certain set of clauses has been inferred by backward reasoning starting from a clause, under the

---

<sup>4</sup> For reasons of space, the examples we give of our formalization of ACL2 as an OMRS are slightly simplified. The details can be found in the mentioned references.

given theory. Formal definitions and details are presented in [4].

Reasoning performed by the linear arithmetic specialist consists in manipulating (sets of) linear polynomial disequalities (referred to as polynomials), in order to be able to detect the unsatisfiability of systems of such disequalities. Reasoning is performed within the context of a theory for complex rational numbers; in particular, single polynomials are reasoned about starting from a context of sets of polynomials. This can be represented by sequents of the following form:

$$th \vdash_{LA} db \rightsquigarrow p$$

where  $th$  is the first-order axiomatization of ACL2 theory for numbers (including the properties of the  $<$  and  $\leq$  relations),  $p$  is an ACL2 polynomial, and  $db$  is a set of ACL2 polynomials. Intuitively, this sequent indicates that  $p$  polynomial is satisfiable if the system  $db$  is, under the theory  $th$ . The linear arithmetic specialist composes reasoning onto single disequalities to obtain reasoning over sets of disequalities; this can be represented as a transformation of polynomial sets, using sequents of the following form:

$$th \vdash_{LAP} db \rightsquigarrow db'$$

Similarly to before, this sequent indicates that the  $db'$  system is satisfiable if the system  $db$  is, under the theory  $th$ . ■

### 3.2 Inference rules

Inference steps performed by provers may consist in the application of a variety of forms of rules: “classical” fixed-arity rules (i.e., with a fixed number of premises), schematic rules, (i.e., operating upon schematic sequents), and variable-arity rules (i.e., with a variable number of premises). Such rules can be applied backward as in NuPrl [15] (i.e., generating premises from a conclusion as new subgoals from a goal), forward as in FOL ([42,43]) (i.e., generating a conclusion from premises, as a new theorem from previous theorems), both backward and forward as in HOL ([22]) or GETFOL ([18]), or by linking premises (as theorems) to a conclusion (as goal) as in GETFOL. As already mentioned, the application of these rules can require the satisfaction of constraints.

In order to represent such aspects, [19] introduces the following concepts. Let  $Ssys = \langle S, C, \models, I, \_ \_ \rangle$  be a sequent system, and let  $Id$  be a set of identifiers. Then the set  $\mathbf{Rule}[Ssys]$  of *rules* over  $Ssys$ , and the set  $\mathbf{Rset}[Ssys, Id]$  of *rule sets* over  $Ssys$  and  $Id$  are defined by<sup>5</sup>

$$\begin{aligned} \mathbf{Rule}[Ssys] &= \{R \subseteq (S^* \times S \times \mathbf{P}_\omega(C)) \\ &\quad \mid (\forall \langle \bar{s}, s, \tilde{c} \rangle \in R)(\forall \iota \in I)(\langle \bar{s}, s, \tilde{c} \rangle[\iota] \in R)\} \\ \mathbf{Rset}[Ssys, Id] &= \mathbf{Fmap}[Id, \mathbf{Rule}[Ssys]] \end{aligned}$$

<sup>5</sup> If  $A$  is a set,  $A^*$  is the set of all finite sequences of elements of  $A$ . If  $A$  and  $B$  are sets,  $\mathbf{Fmap}[A, B]$  is the set of all finite maps (i.e., partial functions) from  $A$  to  $B$ .

A rule is a relation whose tuples consist of a finite (possibly empty) sequence of sequents (the premises), a sequent (the conclusion), and a finite (possibly empty) set of constraints (the applicability conditions). We require that rules be closed under instantiation. A rule set is a finite set of rules each associated with a unique identifier. In [19] the notion of *rule generator* is also introduced: a rule generator is a subset  $rg$  of  $S^* \times S \times \mathbf{P}_\omega(C)$ . The rule generated by  $rg$  is the set  $rg[I]$ . These definitions of  $\mathbf{Rule}[Ssys]$  and  $\mathbf{Rset}[Ssys, Id]$  allow great flexibility in the choice of which sequents and constraints a rule consists of, in the way rules are applied (they are defined to be adirectional), and in their arity (which can be fixed or variable).

**Example (ND – propositional rules):** Consider the ND sequent system  $Ssys_{ND}$ , defined in Section 3.1. The set  $Id_{ND}$  includes identifiers for the inference rules for assumption introduction,  $\vee$  introduction and  $\neg$  introduction.

$$Id_{ND} \supset \{\text{ASM}, \vee I_r, \vee I_1, \neg I\}$$

The informal notation used for the corresponding inference rules is

$$\begin{array}{ll} \text{ASM} \quad \frac{}{\{A\} \vdash A} & \neg I \quad \frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A} \\ \vee I_r \quad \frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2} & \vee I_1 \quad \frac{\Gamma \vdash A_1}{\Gamma \vdash A_2 \vee A_1} \end{array}$$

where  $\Gamma$  is a schematic variable standing for a (finite) set of formulas (the assumptions), and  $A, A_1, A_2$  are schematic variables standing for formulas. We will use this informal notation in the rest of the paper. The schemas like those above should be thought of as presenting the rule generators for the rules considered. For instance the schema for ASM presents the (finite) rule generator

$$\{\langle [], \{A\} \vdash A, \emptyset \rangle\}$$

Note that instances of these rules may contain schematic variables. (For example the generating element is itself an instance.) ■

**Example (ND –  $\forall$  introduction):** The propositional rules for ND can all be expressed without explicit constraints, since all the restrictions on their applicability can be expressed schematically in the form of their premises and conclusion. However this is not always the case. The rule for  $\forall$  introduction requires an additional constraint on the occurrences of free variables. The set of rule identifiers also includes  $\forall I$ . The informal notation used to present the rule associated to  $\forall I$  is the following:

$$\forall I \quad \frac{\Gamma \vdash A}{\Gamma \vdash (\forall x)A} \text{NoFree}(\Gamma, x)$$

Recall that  $\text{NoFree}(\Gamma, x)$  holds iff the variable  $x$  does not occur free in any of

the assumptions in  $\Gamma$ . ■

**Example (Some ACL2 rules):**

The top-level waterfall module exploits repeated invocation of processes to combine their deductions. If we are essentially interested in formalizing the waterfall, the processes can be formalized as nullary rules, where their behavioural definitions appear in terms of functional constraints. For instance, let *Simplify* be a function from clauses from clause sets, representing the simplifier module. Then, the following nullary rule is used when representing the inference step performed by invoking the simplifier.

$$\text{Simplify} \frac{}{th \vdash_W cl \rightsquigarrow \widetilde{cls}} \quad \widetilde{cls} = \text{Simplify}(cl)$$

The composition of inferences performed by modules can be formalized by means of the following rule:

$$\text{ComposeResults} \frac{th \vdash_W cl \rightsquigarrow \widetilde{cls}_0 \quad th \vdash_{W'} \widetilde{cls}_0 \rightsquigarrow \widetilde{cls} \quad \widetilde{cls}_0 \neq \{cl\}}{th \vdash_W cl \rightsquigarrow \widetilde{cls}}$$

The linear arithmetic specialist adopts a simple cross-multiplication procedure to detect the unsatisfiability of systems of polynomials. This can be represented by a variary rule like the following<sup>6</sup>:

$$\frac{\begin{array}{c} th \vdash_{LA} p \wedge p_1 \rightsquigarrow p \circ p_1 \\ th \vdash_{LA} p \wedge p_2 \rightsquigarrow p \circ p_2 \\ \vdots \\ th \vdash_{LA} p \wedge p_n \rightsquigarrow p \circ p_n \end{array}}{th \vdash_{LA'} p \wedge p_1 \wedge p_2 \wedge \dots \wedge p_n \rightsquigarrow p \circ p_1 \wedge p \circ p_2 \wedge \dots \wedge p \circ p_n}$$

■

### 3.3 Reasoning theories

A formal system is described in terms of the assertions to be derived and the inference rules used to derive them. Analogously, a RTh is described in terms of a sequent system and a rule set. Formally

$$Rth = \langle Ssys, Id, \tilde{r} \rangle$$

where *Ssys* is a sequent system, *Id* is a set of identifiers, and  $\tilde{r} \in \mathbf{Rset}[Ssys, Id]$  is a rule set over *Ssys* and *Id*. Note the notion of axiom in a formal system is straightforwardly represented in a RTh as a rule with no premises.

<sup>6</sup> The symbol  $\circ$  denotes cross-multiplication.

**Example (ND):** The ND reasoning theory is an example of a (simple) reasoning theory.

$$Rth_{ND} = \langle Ssys_{ND}, Id_{ND}, \tilde{r}_{ND} \rangle$$

where  $Ssys_{ND}$  is the sequent system described in Section 5.  $Id_{ND}$  includes the rule names given in the ND example in Section 6 and additional names for the remaining quantifier rules.  $\tilde{r}_{ND}$  associates to each ND rule name the corresponding rule, as described in the ND example. ■

Most provers, e.g. NQTHM or PVS, have a variety of special-purpose modules. Often these modules employ different logics (e.g., term rewriting vs. linear arithmetic) or deal with the same logic in different ways (e.g., clauses vs. sets of types). Therefore, in these situations it is most natural to employ different RThs to represent (the logic components of) such modules. Since the module cooperate together, from a logical point of view their cooperation can be represented by rules involving sequents from different RThs. Altogether, the various RThs plus these additional rules should form a RTh (which is roughly the union of all the sequents and rules). This is formalized in [19] as the *gluing* of RThs. Let

$$Rth_1 = \langle Ssys_1, Id_1, \tilde{r}_1 \rangle$$

...

$$Rth_n = \langle Ssys_n, Id_n, \tilde{r}_n \rangle$$

be reasoning theories. Let  $Ssys$  be the disjoint union of the sequent systems  $Ssys_i$ <sup>7</sup>,  $Id$  the disjoint union of the sets of identifiers  $Id_i$ ,  $Id_B$  a set of identifiers disjoint from  $Id$ , and  $\tilde{r}_B \in \mathbf{Rset}[Ssys, Id_B]$  a set of rules over the (composite) sequent system. The gluing of the  $Rth_i$  via  $\tilde{r}_B$  is defined by<sup>8</sup>

$$Rth = glueRth([Rth_1, \dots, Rth_n], Id_B, \tilde{r}_B) = \langle Ssys, Id \uplus Id_B, \tilde{r} \uplus \tilde{r}_B \rangle$$

$Rth$  is called a *composite reasoning theory*, with *components*  $Rth_i$ , and *glue*  $Id_B, \tilde{r}_B$ . The rules in  $\tilde{r}_i$  are called *internal rules* of  $Rth_i$ . The rules in  $\tilde{r}_B$  are called *bridge rules*.

### 3.4 Reasoning structures

The overall motivation for the concept of RTh is to represent the computations of provers, from the logical point of view, in a natural and convenient way. So, there are sequents for assertions, rules for inference steps, and constraints for side conditions. The way these ingredients can be manipulated by provers, is formalized by the notion of *reasoning structure*, also introduced in [19]. This

<sup>7</sup> Disjoint union of sequent systems is precisely defined in [19]. The rough idea is taking the disjoint union of sequents and that of constraints.

<sup>8</sup> The symbol  $\uplus$  denotes disjoint union.

notion generalizes the “classical” notion of proof tree in various ways. First, it allows sharing of sub-structures (i.e., a same sequent occurrence used as premise of various rule applications). Then, it allows constraints to occur in the structure, and these constraints can be replaced by others whose satisfaction implies that of the previous ones (according to the relation  $\models$ ). Further, an instantiation can be applied to a reasoning structure (i.e., to all sequents and constraints occurring in it): this can represent the filling in of schemata during a proof attempt by a prover. Last but not least, reasoning structures can be nested into each other, so that they can be hierarchically organized to expose more or less details of an (attempted) proof.

More precisely, a reasoning structure is a directed graph with two kinds of nodes: a node of the first kind is labeled by a sequent; a node of the second kind by a *justification*. Any arc goes from a node to another node of the other kind. A justification either consists of a rule identifier and a set of constraints, or consists of another reasoning structure (*nested* in the first one) plus some additional information which specifies the correspondence between nodes of the outer reasoning structure and nodes of the inner one. A reasoning structure is regarded as a derivation iff it is acyclic, no constraints occur in it, and there is just one sequent node without outgoing arcs. A derivation is regarded as a proof iff there are no open assumptions, i.e. no sequent nodes without incoming arcs. Precise definitions can be found in [19], which also includes examples of reasoning structures and defines operations to build them and to compose them horizontally (by “stitching”) or vertically (by nesting).

## 4 Control

In order to explore the (often huge) search space in a practical and efficient way, provers usually make use of heuristic information which they manipulate along with the logical one, during the tentative construction of a proof. The form of such information and the way it is manipulated, may widely vary among different provers. For instance, ACL2 allows the user to instruct the prover with “hints” of various kinds to drive the proof, and builds internal control information to prune the search, and to avoid infinite recursion wherever possible (see the discussion on “tail biting” in [10]). In ACL2, the heuristics are coded by means of a purely functional language, to be perceived as a subset of Common Lisp. In LCF ([23]), ISABELLE ([32]), NUPRL ([16]) and several other systems, strategies are described by means of a tactical language embedded in the functional language ML ([31]).

The control level of an OMRS must therefore define both the control information used by provers and its possible manipulations, as well as the strategies according to which such manipulations actually take place. OMRS tackles the first of these aspects by extending the notion of RTh to that of *annotated reasoning theory*, which is basically a RTh enriched with control information, and

which naturally gives raise to the concept of *annotated reasoning structure*. The second aspect is instead tackled through the concept of *tactic system*. We now analyze these concepts in turn, which are introduced and defined in detail in [2].

#### 4.1 Annotated reasoning theories and annotated reasoning structures

The idea behind the concept of annotated RTh is the following. Given a RTh  $Rth$  representing the logic component of a prover, we augment its sequents with control information, thus obtaining annotated sequents. Accordingly, we lift rules to also operate upon this control information, besides the logical one. This demands for extending constraints to deal with the additional information, and extending instantiations as well. We thus obtain a new RTh  $Rth'$ . The relationship between  $Rth'$  and  $Rth$  is formalized by an *annotation removal mapping*  $\mu$  which, roughly speaking, maps sequents and constraints of  $Rth'$  to sequents and constraints of  $Rth$ . Indeed,  $\mu$  removes control information leaving logical information intact;  $Rth$  is to be perceived as the logical image of  $Rth'$ . By pairing  $Rth'$  with  $\mu$ , we obtain an annotated RTh  $ARth$  over  $Rth$ . Certain requirements must be satisfied to ensure the soundness of the annotated RTh w.r.t. the underlying (non-annotated) RTh. For instance, the image of the constraint satisfaction relation of  $Rth'$  must be contained into the constraint satisfaction relation of  $Rth$ . In addition, any rule in  $Rth'$  must correspond to the composition of zero or more rules in  $Rth$ . To emphasize the different role of the rules of an annotated RTh from the rules of the associated RTh we call the former *control-inference rules*, leaving the conventional name *inference rules* for the latter. We finally call *control rules* those control-inference rules in which the logical content of the conclusion is identical to the logical content of (at least) one of the premises. Formal definitions are provided in [2].

Annotated reasoning structures are just reasoning structures over  $Rth'$ . Given the requirement that any control-inference rule “maps” to zero or more inference rules, an annotated reasoning structure can be mapped to a non-annotated one.

**Example ( ND annotated reasoning theory):** ND sequents can be annotated in various ways; for instance, a natural number can be added to indicate the maximum number of nested quantifiers appearing in the right part of the sequent (i.e., in its conclusion). This information can be employed by a heuristic prover, e.g. to limit the number of quantifiers in formulas. Thus, annotated sequents for such a theory  $ND_C$  are triples  $\Gamma \vdash A; n$ , where  $n$  is a natural number, according to the previous description. The constraints extend those described in the ND example description, by considering also inequalities between natural numbers. The definition of the annotation removal mapping is straightforward: sequents of  $ND_C$  are mapped to sequents of ND by removing the natural number; constraints of  $ND_C$  either map identically to constraints

of ND or, in case of constraints between control annotations, map to the null constraint. The rule system described for ND in 3.1 can be extended accordingly. For reasons of space, we only show how the  $\forall$  introduction rule can be modified to constrain its application to non-nested formulas:

$$\forall I_C \frac{\Gamma \vdash A; n}{\Gamma \vdash (\forall x)A; n+1} \quad \begin{cases} NoFree(\Gamma, x) \\ n = 0 \end{cases}$$

■

**Example (ACL2 annotated reasoning theory):** ACL2 makes use of different kinds of control information, some provided by the user, others built by internal modules. For instance, at the top level, user-supplied hints are attached to the clauses being manipulated; such hints indicate which processes to use, or not to use, upon a certain clause. Moreover, clauses carry a history with them, i.e. a record of the modules invoked to produce it; histories are used to avoid redundantly invoking certain reasoning modules. Therefore, annotated top-level sequents have the form

$$th \vdash_{W_c} cl; \overline{hst}, \overline{hint} \rightsquigarrow \overline{cls}; \overline{hst}'$$

The annotation removal mapping simply removes histories and hints, and reformulates the sequences of clauses as sets. The simplification rule and the combination rule presented for the underlying reasoning theory are then modified as follows, to take into account the handling of control information.

$$\text{SimplifyC} \frac{}{th \vdash_{W_c} cl; \overline{hst}, \overline{hint} \rightsquigarrow \widetilde{cls}; \overline{hst} \circ \text{Simp}} \quad \begin{cases} \widetilde{cls} = \text{Simplify}(cl) \\ (\text{DoNot.Simplify}) \notin \overline{hint} \end{cases}$$

$$\text{ComposeC} \frac{th \vdash_W cl; \overline{hst}, \overline{hint} \rightsquigarrow \widetilde{cls}_0; \overline{hst}' \quad th \vdash_{WL} \widetilde{cls}_0; \overline{hst}', \overline{hint} \rightsquigarrow \widetilde{cls}; \overline{hst}''}{th \vdash_W cl; \overline{hst}, \overline{hint} \rightsquigarrow \widetilde{cls}; \overline{hst}''} \quad \widetilde{cls}_0 \neq \{cl\}$$

Polynomials used by the linear arithmetic specialist are enriched by a lot of control information; for instance, a list of literals is attached to every polynomial, to indicate that the polynomial has been derived from them (directly or indirectly). These literals are used to avoid redundant cross-multiplications, by considering a set of “literals to be avoided” when performing such operations. Thus annotated linear arithmetic sequents have the form

$$th \vdash_{LA'_c} db \rightsquigarrow (p; \overline{l})$$

$$th \vdash_{LA'_c} db; pt \rightsquigarrow db'$$

where  $db = \{(p_1, \overline{l}_1), \dots, (p_n, \overline{l}_n)\}$  is a set of annotated polynomials. When performing cross-multiplication, the module selects the polynomials to be pro-

cessed by considering a set of literals  $pt$ , and filtering those polynomials whose literals appear in  $pt$ :

$$\begin{array}{c}
 th \vdash_{LAC} (p; \bar{l}) \wedge (p_1; \bar{l}_1) \rightsquigarrow (p \circ p_1; \bar{l} \circ \bar{l}_1) \\
 th \vdash_{LAC} (p; \bar{l}) \wedge (p_2; \bar{l}_2) \rightsquigarrow (p \circ p_2; \bar{l} \circ \bar{l}_2) \\
 \vdots \\
 th \vdash_{LAC} (p; \bar{l}) \wedge (p_n; \bar{l}_n) \rightsquigarrow (p \circ p_n; \bar{l} \circ \bar{l}_n) \\
 \hline
 th \vdash_{LAC'} (p; \bar{l}) \wedge (p_1; \bar{l}_1) \wedge (p_2; \bar{l}_2) \wedge \dots \wedge (p_n; \bar{l}_n); pt \rightsquigarrow \\
 (p \circ p_1; \bar{l} \circ \bar{l}_1) \wedge (p \circ p_2; \bar{l} \circ \bar{l}_2) \wedge \dots \wedge (p \circ p_n; \bar{l} \circ \bar{l}_n)
 \end{array}
 \left\{ \begin{array}{l}
 \bar{l}_1 \cap pt = \emptyset \\
 \vdots \\
 \bar{l}_n \cap pt = \emptyset
 \end{array} \right.$$

■

The gluing of annotated RThs is simply defined by gluing the RThs containing annotated sequents and control-inference rules, taking the disjoint union of the annotation removal mappings, and pairing them.

#### 4.2 Tactic systems

Control-inference rules define how logical and control information are manipulated by the prover. What is then needed, is a way to represent how these rules are actually applied, i.e. in which order, according to which strategies, etc. Besides application of rules, we also need to represent constraint solving (e.g., eliminating constraints which are satisfied, or replacing a constraint with simpler constraints which entail the previous one), as well as instantiation of proof fragments (schematic reasoning).

The formal notion of tactic system can represent the above aspects. Roughly, a *tactic* is a function from  $S$  to  $(S^* \times \mathbf{P}_\omega(C) \times I) \cup F$ , where  $F$  is a set of *failures*. The intuition behind a tactic is the following. If the result of applying it to a sequent  $s$  is a triple  $\langle \bar{s}, \tilde{c}, \iota \rangle$ , then  $s[\iota]$  is derivable from the sequents in  $\bar{s}$  provided the constraints in  $\tilde{c}$  are satisfied. In other words, the goal of proving  $s$  has been decomposed into the subgoals of proving  $\bar{s}$  and checking satisfaction of  $\tilde{c}$ . At the same time, schematic parts of  $s$  are possibly filled in by applying  $\iota$  to it (however, if  $\iota$  is the identity instantiation then the sequent remains the same). If a failure in  $F$  is instead returned, it means that the tactic failed in decomposing the goal into subgoals. Having multiple failures rather than just one (as in LCF) allows tactics to carry some information about the reason why they failed, so that different alternative tactics can be attempted according to the different failures. The set  $F$  varies from OMRS to OMRS.

In order to deal with constraints, a tactic system also includes a *constraint solver*, which is roughly a function from  $\mathbf{P}_\omega(C)$  to  $\mathbf{P}_\omega(C) \times I$ . The idea behind a constraint solver is that if the result of applying it to  $\tilde{c}$  is  $\langle \tilde{c}', \iota \rangle$ , then  $\tilde{c}' \models \tilde{c}[\iota]$ . In other words, the constraints in  $\tilde{c}$  are replaced by the

(supposedly simpler) constraints in  $\tilde{c}'$ , instantiating  $\tilde{c}$  by  $\iota$  (which can be the identity instantiation or not). The constraint solver, in general, may vary from OMRS to OMRS. In fact, different provers may use different kinds of constraints, and therefore their constraint solving mechanism can be optimized on those constraints.

A tactic system always includes *primitive tactics* corresponding to control-inference rules of the annotated RTh (in fact, the concept of tactic system is defined w.r.t. a given annotated RTh). The application of a primitive tactic corresponds to a backward application of a control-inference rules (in the spirit of LCF and other systems), but may involve constraint solving and instantiations. A tactic system then includes *tacticals* (which may vary from OMRS to OMRS), used to build more complex tactics out of the primitive ones. These more complex tactics correspond to multiple applications of (possibly different) control-inference rules, and may involve composition of instantiations, constraint solving, and alternative tactics invoked according to the failure of a tactic.

While the concept of tactic system sketched above does not capture all the possible ways a reasoning structure can be built and manipulated, nevertheless it accounts for a large number of interesting cases. In fact, the use of tactic systems to model substantial parts of NQTHM, with satisfactory results, is described in [12,2,4]. Indeed, tactic systems are a starting point for a more general concept, which is currently the object of ongoing research (see next section).

## 5 Ongoing Theoretical Developments

The formal concepts surveyed in the previous sections are currently being extended and refined to better address the fundamental issues that inspired them. Such extensions and refinements are being carried out in a categorical framework. Category theory [33,3,27] is getting increasingly popular in many areas of Computer Science, because ideas in those areas can be conveniently expressed and reasoned about by means of the plethora of concepts and results of this branch of mathematics. In particular, category theory has proven very successful in the field of formal specification and composition of software systems [38], where OMRS fits naturally. In the remainder of this section we give an overview of the current theoretical developments in the logic and control components of the OMRS architecture, along with their motivations.

### 5.1 Logic

#### 5.1.1 Gluing

The notion of gluing of RThs through bridge rules, presented in 3, is being extended to capture more general forms of composition. We are in the process of defining a category whose objects are RThs and whose arrows are mappings

between RThs. Such mappings can express inclusion, renaming, specialization, etc. Gluing is then applied to a diagram in this category, i.e. a multigraph whose nodes and arcs are respectively labeled by RThs and mappings (which express the relationships among the RThs to be glued): the result of the gluing is the colimit of the diagram. The category should turn out to be finitely cocomplete (i.e., a colimit exists for any diagram).

Preliminary work in such a direction is presented in [13]. In that paper, we restrict our attention to *equational RThs (ERThs)*, where sequents are equivalence classes of terms of an order-sorted equational theory [20]. We define a notion of *faithful inclusion* of an ERTh into another, as well as an operation yielding the *shared part* of two ERThs. Two ERThs are *composable* iff their shared part is an ERTh faithfully included in both, and the result of their composition is an ERTh which is the union of the two (sharing the shared part), in which they are both faithfully included. Composition is therefore a (partial) binary operation over ERThs, which turns out to be commutative, associative, and idempotent. Although in [13] we do not explicitly use category theory, those concepts boil down to the existence of a category whose objects and arrows are respectively ERThs and faithful inclusions, and composition of two ERThs is their pushout (with their shared part). As we consider richer mappings than faithful inclusions (and richer RThs than ERThs as well), an explicit shift to categories becomes necessary.

### 5.1.2 Nesting

As presented in 3, a constraint system is just a set of constraints equipped with a consequence relation. As a RTh substantially defines a consequence relation over sequents (modulo the constraints, so to speak), it makes sense to consider the case where the constraints of a RTh are sequents of another RTh, *nested* into the first. The nested RTh can have constraints, which can be sequents of a further nested RTh. This nesting can go down to an arbitrary finite number of levels (eventually, the deepest RTh must have no constraints). This leads to the definition of *nested RThs (NRThs)* of level  $n$  for any  $n$ .

Again, preliminary work about that is presented in [13], where we define *nested ERThs (NERThs)*, which are ERThs with a nested ERTh for constraints. There are interesting conceptual relations between composition and nesting. The outer and inner ERThs of a NERTh are required to be composable together. This requirement is used to define composition of NERThs, which generalizes that of ERThs and has analogous properties. In a more general setting than NERThs, we are going to define a category of NRThs and mappings, in analogy to the category of RThs sketched above. There will probably be interesting relationships between these two categories.

## 5.2 Control

### 5.2.1 Annotations

Work is also underway to refine the general notion of annotations given in 4. In [13] we define an *annotated ERTh* (and analogously annotated NERThs) as a pair of ERThs plus an *erasing mapping* (which removes annotations). Composition of annotated ERThs is defined based on the composition of the ERThs involved. In a more general setting, we are defining a category of annotated RThs. Roughly, an annotated RTh is a particular mapping from a RTh containing the annotations to one without them, where the mapping defines annotation removal. A mapping between annotated RThs consists in a mapping between the two RThs with annotations and another mapping between the two without. Annotated RThs are glued via colimit in this category, which “projects” to two colimits for the underlying RThs with and without annotations.

### 5.2.2 Tactics and More

We are also refining the concept of tactic system which we presented in 4. The refinement consists in defining a category of tactic systems, and a precise notion of mapping from a tactic system to the RTh whose sequents and rules the tactics “operate upon”. Tactic systems should be glued together via colimit, and there should be an induced mapping from the resulting tactic system to the RTh obtained by gluing the various RThs of the component tactic systems.

While the concept of tactic system is an interesting one, we are aware that to express more general strategies of constructions of reasoning structures we need some more general notion, as mentioned in the previous section. Therefore, tactic systems should be an instance of these more general entities, and these entities should also form a category with gluing being colimit, and there should be mappings from these entities to the RThs they operate upon. An interesting idea is using RThs as these entities, defining suitable mappings which express how a RTh operates upon another one. (So, in particular, tactic systems would be expressed as RThs.) The attractiveness of this approach is the nice uniformity of having RThs as a “central” concept, with various kinds of mappings among them (those for inclusion/renaming/specialization, those for annotation removal, those expressing “operating upon”), and properties relating them. Mappings can be glued together, which involves gluing the underlying RThs.

## 6 Conclusion

This paper has surveyed the current theoretical achievements of the research performed within the OMRS project, and some of its practical applications. A number of useful formal notions has been developed, and tested against non-

trivial case studies, with satisfactory and encouraging results. This testing has taken place in various forms. First of all, we have used OMRS to clearly and formally specify the way substantial and non-trivial parts of a state-of-the-art prover like NQTHM works [19,14,12,2]. Other case studies which we worked on include resolution-based and rippling-based provers. Some work has also been done on the formalization of the integration of Maple and Isabelle [5]. We plan to carry out additional case studies including the Nelson-Oppen cooperating decision procedures and simplifier. Furthermore, our concepts have been employed to re-implement parts of ACL2 as an OMRS [4]. [1] describes a successful attempt of “extracting” the linear arithmetic module from NQTHM, and making it a plug-and-play component.

Besides the ongoing developments described in Section 5, a lot of work is underway or has been planned for. An important part of that is the interaction component of the OMRS architecture, for which no work has essentially been done so far. We are also working on a formal notation to presenting the syntax and semantics of sequent systems and for defining relations between sequent systems. The concept of ERTh [13] is a step in such a direction. To account uniformly for notions of binding we will build on the work on binding structures [40]. As a starting point we expect to use a notation for first-order equational theories such as OBJ [21] or Maude [28]. This gives executable specifications suitable for prototyping. The module composition capability of these systems will also serve as a starting point to glue RThs together.

So far we have considered only the operational and proof theoretic aspects of provers. We need to add a semantic component to the framework and to provide a notion of model for RThs. We need this to be able to reason about how semantics compose when we connect together heterogeneous provers. The work on general logics [29] provides the starting point for this work.

## References

- [1] A. Armando and S. Ranise. From Integrated Reasoning Specialists to “Plug-and-Play” Reasoning Components. In *Fourth International Conference on Artificial Intelligence and Symbolic Computation (AISC98)*, Plattsburgh (NY), 1998.
- [2] Alessandro Armando, Alessandro Coglio, and Fausto Giunchiglia. The control component of open mechanized reasoning systems. Technical report, DIST, 1998.
- [3] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
- [4] P.G. Bertoli. *Using OMRS in practice: a case study with ACL2*. PhD thesis, Computer Science Dept., University Rome 3, Rome, 1998.
- [5] P.G. Bertoli, J. Calmet, and K. Homann F. Giunchiglia. Specification and Integration of Theorem Provers and Computer Algebra Systems. Technical

- Report 9804-03, IRST, Trento, Italy, April 1998. To appear in the proceedings of AISC'98 – Fourth International Conference On Artificial Intelligence and Symbolic Computation.
- [6] W. R. Bevier. *A Verified Operating System Kernel*. PhD thesis, University of Texas at Austin, 1987. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.
  - [7] R. S. Boyer and J S. Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, pages 133–167. American Mathematical Society, Providence, R.I., 1984.
  - [8] R. S. Boyer and Y. Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Proc. of the 11th Conference on Automated Deduction*, pages 416–430. Springer-Verlag, 1992.
  - [9] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.
  - [10] R.S. Boyer and J.S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.
  - [11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking:  $10^{20}$  States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.
  - [12] A. Coglio. Definizione di un formalismo per la specifica delle strategie di inferenza dei sistemi di ragionamento meccanizzato e sua applicazione ad un sistema allo stato dell'arte. Master Thesis DIST - University of Genoa (Italy), 1996.
  - [13] A. Coglio, F. Giunchiglia, J. Meseguer, and C. Talcott. Composing and Controlling Search in Reasoning Theories using Mappings. Technical report, IRST- Technical report, IRST, Trento, Italy, 1998.
  - [14] A. Coglio, F. Giunchiglia, P. Pecchiari, and C. Talcott. A Logic Level Specification of the NQTHM Simplification Process. Technical Report 9700-48, DIST - University of Genoa, Genoa, Italy, 1997. Also IRST-Technical Report 9706-07, IRST, Trento, Italy.
  - [15] R. L. Constable and et. al. *Implementing mathematics with the Nuprl development system*. Prentice-Hall, 1986.
  - [16] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.
  - [17] J. Cowles. Meeting a challenge of Knuth. Technical Report 286, Computational Logic Inc., Austin, Texas, 1993.
  - [18] F. Giunchiglia. The GETFOL Manual - GETFOL version 1. Technical Report 92-0010, DIST - University of Genoa, Genoa, Italy, 1992.

- [19] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning Theories: Towards an Architecture for Open Mechanized Reasoning Systems. Technical Report 9409-15, IRST, Trento, Italy, 1994. Also published as Stanford Computer Science Department Technical note number STAN-CS-TN-94-15, Stanford University. Short version published in Proc. of the First International Workshop on Frontiers of Combining Systems (FroCoS'96), Munich, Germany, March 1996.
- [20] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
- [21] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1993. To appear in J.A. Goguen, editor, *Applications of Algebraic Specification Using OBJ*, Cambridge University Press.
- [22] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer, 1987.
- [23] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A mechanized logic of computation*. Number 78 in Lecture Notes in Computer Science. Springer-Verlag, 1979.
- [24] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [25] D. Kapur and H. Zhang. An Overview of RRL (Rewrite Rule Laboratory). In *Third International Conf. of Rewriting Techniques and Applications*, 1989.
- [26] M. Kaufmann and J. S. Moore. ACL2 Version 1.8 User's Manual. Available on line at <http://www.cs.utexas.edu/users/moore/acl2/index.html>.
- [27] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics Series. Springer-Verlag, Berlin, 1971.
- [28] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.
- [29] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
- [30] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.
- [31] L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–396, 1989.
- [32] L. C. Paulson. *Isabelle - a Generic Theorem Prover*. Springer-Verlag, 1994.

- [33] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, Cambridge, Massachusetts, 1991.
- [34] D. Prawitz. *Natural Deduction: A Proof-theoretical Study*. Almquist and Wiksell, 1965.
- [35] D. M. Russinoff. A mechanical proof of quadratic reciprocity. *Journal of Automated Reasoning*, 8(1):3–21, 1992.
- [36] D. M. Russinoff. Specification and verification of gate-level VHDL models of synchronous and asynchronous circuits. Technical Report 99, Computational Logic Inc., Austin, Texas, 1994.
- [37] N. Shankar. A mechanical proof of the Church-Rosser theorem. *JACM*, 35(3):475–522, 1988.
- [38] Y. Srinivas and R. Jullig. Specware (TM): Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, 1995.
- [39] C. Talcott. Database of automated reasoning systems. Accessible by anonymous ftp or WWW. URL = file://sail.stanford.edu/pub/clt/ARS.
- [40] C. L. Talcott. A theory of binding structures and its applications to rewriting. *Theoretical Computer Science*, 112:99–143, 1993.
- [41] The SRI Staff. The SRI Specification and Verification System, Users Guide. Technical report, SRI International, 1986.
- [42] R. W. Weyhrauch. A Users Manual for FOL. Technical Report STAN-CS-77-432, Stanford University Computer Science Department, 1977.
- [43] R. W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.
- [44] W.D. Young. Verifying the interactive convergence clock synchronization algorithm using the Boyer-Moore theorem prover. Contractor Report 189649, NASA.