Maude as a Formal Meta-Tool*

M. Clavel[1], F. Durán[2], S. Eker[2], and J. Meseguer[2]

[1] Department of Philosophy, University of Navarre, Spain
[2] SRI International, Menlo Park, CA 94025, USA

Abstract. Given the different perspectives from which a complex software system has to be analyzed, the multiplicity of formalisms is in some sense unavoidable. This poses two important technical challenges: how to rigorously meet the need to inter-relate formalisms, and how to reduce the duplication of effort in tool and specification building across formalisms. These challenges could be answered by adequate *formal meta-tools* that, when given the specification of a formal inference system, generate an efficient inference engine, and when given a specification of two formalism and a translation, generate an actual translator between them. Similarly, module composition operations that are logic-independent, but that at present require costly implementation efforts for each formalism could be provided for logics in general by module algebra generator meta-tools. The foundations of meta-tools of this kind can be based on a meta-theory of general logics. Their actual design and implementation can be based on appropriate logical frameworks having efficient implementations. This paper explains how the reflective logical framework of rewriting logic can be used, in conjunction with an efficient reflective implementation such as the Maude language, to design formal meta-tools such as those described above. The feasibility of these ideas and techniques has been demonstrated by a number of substantial experiments in which new formal tools and new translations between formalisms, efficient enough to be used in practice, have been generated.

1 Introduction

At present, formal methods for software specification and verification tend to be monolithic, in the sense that in each approach only one formal system or specification language is used to formalize the desired system properties. For this reason, formal systems, and the tools based on them, can be as it were *autistic*, because they lack the meta-tools and methods for being related to other formalisms and to their supporting tools.

As a consequence, it is at present very difficult to integrate in a rigorous way different formal descriptions, and to reason across such descriptions. This situation is very unsatisfactory, and presents one of the biggest obstacles to the

* Supported by DARPA through NASA Contract NAS2-98073, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grant CCR-9633363.

use of formal methods in software engineering because, given the complexity of large software systems, it is a fact of life that no single perspective, no single formalization or level of abstraction suffices to represent a system and reason about its behavior. We use the expression *formal interoperability* to denote this capacity to move in a mathematically rigorous way across the different formalizations of a system, and to use in a rigorously integrated manner the different tools supporting such formalizations [44, 41].

By transforming problems in a formalism lacking tools into equivalent problems in a formalism that has them, formal interoperability can save much time and effort in tool development. Also, libraries of theories and specifications can in this way be amortized across many formalisms, avoiding much duplication of effort. One would similarly like to have rigorous meta-methods and tools making it easy to solve different parts of a complex problem using different formal tools, and to then integrate the subproblem solutions into an overall solution.

These considerations suggest that it would be very fruitful to investigate and develop new *formal meta-tools*. That is, tools in which we can easily and rigorously develop many formal tools at a very high level of abstraction; and also tools through which we can rigorously interoperate existing and future tools. Specifically, it would be very useful to have:

- *Formal Tool Generators*, that given a formal description of an inference system generate an *inference engine* for it that is sufficiently efficient to be used in practice as a tool.
- *Formal Translation Generators*, that given formal descriptions of two formalisms and of a translation between them generate an actual *translator* that can be used to translate specifications and to interoperate tools across the given formalisms.
- *Module Algebra Generators*, that given a formalism with appropriate metalogical properties extend its language of basic specifications into a much richer algebra of specification-combining operations, including specification hierarchies, parameterized specifications, and many other specification transformations.

But where will the meta-theory supporting such meta-tools come from? To make such tools mathematically rigorous, the first thing obviously needed is to have a mathematical meta-theory of logics and of translations between logics. We have been investigating the theory of general logics [39, 36, 44, 9, 14] for this purpose. This theory axiomatizes the proof-theoretic and model-theoretic facets of logics and their translations, includes the theory of institutions as its model-theoretic component [25], and is related to other similar meta-theories (see the survey [44]).

But meta-tools need more than a meta-theory. They have to "run," and therefore they need an *executable* meta-theory. This can be provided by an adequate *logical framework*, that is, by a logic with good properties as a meta-logic in which other logics can be naturally represented. Then, an implementation of such a framework logic could serve as a basis for developing the meta-tools.

This paper reports on our results and experiments in using the Maude language [13, 11] as a formal meta-tool in the senses described above. Maude is a reflective language based on rewriting logic [40] that essentially contains the OBJ3 language as an equational sublanguage. Rewriting logic extends equational logic and has very good properties as a logical framework, in which many other logics and many semantic formalisms can be naturally represented [35, 43]. A very important property of the rewriting logic framework is its being *reflective* [15, 10]. Reflection is efficiently supported by the Maude implementation and, together with the high-performance of Maude, is the key feature making possible the use of Maude as a meta-tool.

The rest of the paper is organized as follows. Section 2 explains in more detail in which sense rewriting logic is a reflective logical framework, and some basic principles and methods underlying the use of a rewriting logic implementation as a formal meta-tool. Section 3 describes the key features of Maude allowing it to be used as a meta-tool. Our experience in building formal tools in Maude is described in Section 4, where we report on several formal tool generator and formal translation generator uses, and on the beginnings of a module algebra generator capability. We finish the paper with some concluding remarks and future research directions.

2 A Reflective Logical Framework

A *formal* meta-tool must both rely on, and support, a precise axiomatization of the metalevels of different logics. That is what makes it formal, and what distinguishes it from tool implementations in conventional languages, say Java, in which the implementation itself is not a suitable formal axiomatization of the tool being implemented.

This leads us to the need for a meta-theory of logics, as a necessary foundation for the design of formal meta-tools. In our work we have used the theory of *general logics* proposed in [39], which provides an axiomatic framework to formalize the proof theory and model theory of a logic, and which also provides adequate notions of *mapping* between logics, that is, of logic translations. This theory contains Goguen and Burstall's theory of institutions [25] as its model-theoretic component.

The theory of general logics allows us to define the space of logics as a *category*, in which the objects are the different logics, and the morphisms are the different mappings translating one logic into another. We can therefore axiomatize a translation Φ from a logic $\mathcal{L}$ to a logic $\mathcal{L}'$ as a morphism

$$(\dagger)\ \ \Phi : \mathcal{L} \longrightarrow \mathcal{L}'$$

in the category of logics. A *logical framework* is then a logic $\mathcal{F}$ such that a very wide class of logics can be mapped to it by maps of logics

$$(\ddagger)\ \ \Psi : \mathcal{L} \longrightarrow \mathcal{F}$$

called *representation maps*, that have particularly good properties such as conservativity[1].

A number of logics, particularly higher-order logics originating in constructive type theory, have been proposed as logical frameworks, including the Edinburgh logical framework LF [30, 2, 23], meta-theorem-provers such as Isabelle [47], λProlog [46, 21], and Elf [48], and the work of Basin and Constable [3] on metalogical frameworks. Other approaches, such as Feferman's logical framework FS_0 [20]—that has been used in the work of Matthews, Smaill, and Basin [38]—earlier work by Smullyan [50], and the the 2OBJ meta-theorem-prover of Goguen, Stevens, Hobley, and Hilberdink [28] are instead first-order. Our work should of course be placed within the context of the above related work, and of experiments carried out in different frameworks to prototype formal systems (for more discussion see the survey [44]).

2.1 Rewriting Logic and Reflection

We and other researchers [37, 36, 44, 33, 52, 34, 5, 49, 8, 10, 12] have investigated the suitability of rewriting logic [40] as a logical framework and have found it to have very good properties for this purpose. One important practical advantage is that, what might be called the *representational distance* between a theory T in the original logic and its rewriting logic representation $\Psi(T)$ is often practically zero. That is, both T's original syntax and its rules of inference are faithfully mirrored by the rewrite theory $\Psi(T)$.

A rewrite theory (Ω, E, R) is an equational theory (Ω, E) with signature of operations Ω and equations E together with a collection R of labeled rewrite rules of the form

$$r : t \longrightarrow t'.$$

Logically, such rules mean that we can derive the formula t' from the formula t. That is, the logical reading of a rewrite rule is that of an *inference rule*.

Since the syntax Ω and the equational axioms E of a rewrite theory are entirely *user-definable*, rewriting logic can represent in a direct and natural way the formulas of any finitary logic as elements of an algebraic data type defined by a suitable equational theory (Ω, E). Furthermore, the *structural axioms* satisfied by such formulas—for example, associativity and commutativity of a conjunction operator, or of a set of formulas in a sequent—can also be naturally axiomatized as equations in such an equational theory. Each inference rule in the logic is then naturally axiomatized as a rewrite rule, that is applied *modulo* the equations E. If there are *side conditions* in the inference rule, then the corresponding rewrite rule is *conditional* [40]. Rewriting logic has then very simple (meta-) rules of deduction [40], allowing it to mirror deduction in any finitary logic as rewriting inference. In earlier work we have shown how this general method for representing logics in the rewriting logic framework allows very natural and direct representations

[1] A map of logics is *conservative* if the translation of a sentence is a theorem if and only if the sentence was a theorem in the original logic.

for many logics, including also a general method for representing quantifiers [35, 37].

Besides these good properties, there is an additional key property making rewriting logic remarkably useful as a metalogic, namely *reflection*. Rewriting logic is reflective [15, 10] in the precise sense that there is a finitely presented rewrite theory U such that for any finitely presented rewrite theory T (including U itself) we have the following equivalence

$$T \vdash t \longrightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle,$$

where $\overline{T}$ and $\overline{t}$ are terms representing T and t as data elements of U, of respective types *Theory* and *Term*. Since U is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$T \vdash t \longrightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle \iff U \vdash \langle \overline{U}, \overline{\langle \overline{T}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{U}, \overline{\langle \overline{T}, \overline{t'} \rangle} \rangle \ldots$$

The key advantage of having a reflective logical framework logic such as rewriting logic is that we can represent—or as it is said *reify*—within the logic in a computable way maps of the form (†) and (‡). We can do so by extending the universal theory U with equational abstract data type definitions for the data type of theories $\mathit{Theory}_{\mathcal{L}}$ for each logic $\mathcal{L}$ of interest. Then, a map of the form (†) can be reified as an equationally-defined function

$$\overline{\Phi} : \mathit{Theory}_{\mathcal{L}} \longrightarrow \mathit{Theory}_{\mathcal{L}'}.$$

And, similarly, a representation map of the form (‡), with $\mathcal{F}$ rewriting logic, can be reified by a function

$$\overline{\Psi} : \mathit{Theory}_{\mathcal{L}} \longrightarrow \mathit{Theory}.$$

If the maps Φ and Ψ are computable, then, by a metatheorem of Bergstra and Tucker [4] it is possible to define the functions $\overline{\Phi}$ and $\overline{\Psi}$ by means of corresponding finite sets of Church-Rosser and terminating equations. That is, such functions can be effectively defined and executed within rewriting logic.

2.2 Forma Meta-Tool Techniques

How can we systematically exploit all these properties to use a reflective implementation of rewriting logic as a meta-tool? *Formal tool generator* uses can be well supported by defining representation maps Ψ that are conservative. In conjunction with a reflective implementation of rewriting logic, we can reify such representation maps as functions of the form $\overline{\Psi}$ that give us a systematic way of executing a logic $\mathcal{L}$ by representing each theory T in $\mathcal{L}$—which becomes a data element $\overline{T}$ of $\mathit{Theory}_{\mathcal{L}}$—by the rewrite theory that $\overline{\Psi}(\overline{T})$ meta-represents. By executing such a rewrite theory we are in fact executing the (representation of) T. In our experience, the maps Ψ are essentially identical maps, preserving the original structure of the formulas, and mirroring each inference rule by a corresponding rewrite rule. Therefore, a user can easily follow and understand the rewriting logic execution of the theory T thus represented.

But how well can we *execute* the representation of such a theory T? In general, the inference process of T may be highly nondeterministic, and may have to be guided by so-called *strategies*. Will the status of such strategies be logical, or extra-logical? And will strategies be representable at all in the framework logic? Rewriting logic reflection saves the day, because strategies have a *logical* status: they are computed by rewrite theories at the metalevel. That is, in the reflective tower they are always one level above the rewrite theory whose execution they control. Furthermore, there is great freedom for creating different *internal strategy languages* that extend rewriting logic's universal theory U to allow a flexible logical specification of strategies [15, 10, 11].

Formal translator generator uses are of course supported by formally specifying the algebraic data types $\mathit{Theory}_{\mathcal{L}}$ and $\mathit{Theory}_{\mathcal{L}'}$ of the logics in question and the translation function $\overline{\Phi}$. *Module algebra generator* uses can be supported by defining a *parameterized* algebraic data type, say $\mathit{ModAlg}[X]$, that, given a logic $\mathcal{L}$ having good meta-logical properties, extends the data type $\mathit{Theory}_{\mathcal{L}}$ of theories to an algebra of theory-composition operations $\mathit{ModAlg}[\mathit{Theory}_{\mathcal{L}}]$.

Section 3 explains the reflective metalanguage features of Maude that make meta-tool uses of this kind possible, and Section 4 summarizes our practical meta-tool experience with Maude.

3 Maude's Metalanguage Features

Maude [13, 11] is a reflective language whose modules are theories in rewriting logic. The most general Maude modules are called *system modules*. Given a rewrite theory $T = (\Omega, E, R)$, a system module has essentially the form `mod` T `endm`, that is, it is expressed with a syntax quite close to the corresponding mathematical notation for its corresponding rewrite theory.[2] The equations E in the equational theory (Ω, E) underlying the rewrite theory $T = (\Omega, E, R)$ are presented as a union $E = A \cup E'$, with A a set of *equational axioms* introduced as *attributes* of certain operators in the signature Ω—for example, a conjunction operator $\wedge$ can be declared associative and commutative by keywords `assoc` and `comm`—and where E' is a set of equations that are assumed to be Church-Rosser and terminating *modulo* the axioms A. Maude supports rewriting modulo different combinations of such equational attributes: operators can be declared associative, commutative, with identity, and idempotent [11]. Maude contains a sublanguage of *functional modules* of the form `fmod` (Ω, E) `endfm`, with the equational theory (Ω, E) satisfying the conditions already mentioned. A system module `mod` T `endm` specifies the initial model [40] of the rewrite theory T. Similarly, a functional module `fmod` (Ω, E) `endfm` specifies the initial algebra of the equational theory (Ω, E).

[2] See [11] for a detailed description of Maude's syntax, which is quite similar to that of OBJ3 [27].

3.1 The Module META-LEVEL

A naive implementation of reflection can be very expensive both in time and in memory use. Therefore, a good implementation must provide efficient ways of performing reflective computations. In Maude this is achieved through its predefined META-LEVEL module, in which key functionality of the universal theory U of rewriting logic has been efficiently implemented. In particular, META-LEVEL has sorts Term and Module, so that the representations $\overline{t}$ and $\overline{T}$ of a term t and a module (that is, a rewrite theory) T have sorts Term and Module, respectively. As the universal theory U that it implements in a built-in fashion, META-LEVEL can also support a reflective tower with an arbitrary number of levels of reflection. We summarize below the key functionality provided by META-LEVEL:

- Maude terms are reified as elements of a data type Term of terms;
- Maude modules are reified as terms in a data type Module of modules;
- the process of reducing a term to normal form is reified by a function meta-reduce;
- the process of applying a rule of a system module to a subject term is reified by a function meta-apply;
- the process of rewriting a term in a system module using Maude's default interpreter is reified by a function meta-rewrite; and
- parsing and pretty printing of a term in a module are also reified by corresponding metalevel functions meta-parse and meta-pretty-print.

Representing Terms. Terms are reified as elements of the data type Term of terms, with the following signature

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid Qid -> Term .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
```

The first declaration, making the sort Qid of quoted identifiers a subsort of Term, is used to represent variables in a term by the corresponding quoted identifiers. Thus, the variable N is represented by 'N. The operator {_}_ is used for representing constants essentially as pairs, with the first argument the constant, in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant 0 in the module NAT discussed below is represented as {'0}'Nat. The operator _[_] corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form, and the second argument the list of its subterms, where list concatenation is denoted _,_. For example, the term s s 0 + s 0 of sort Nat in the module NAT is meta-represented as

```
'_+_['s_['s_[{'0}'Nat]],'s_[{'0}'Nat]].
```

Representing Modules. Functional and system modules are meta-represented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are now meta-represented as explained above; and (2) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operator `_;_`.

To motivate the general syntax for representing modules, we illustrate it with a simple example—namely, a module `NAT` for natural numbers with zero and successor and with a commutative addition operator.

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s N + M = s (N + M) .
endfm
```

The syntax for the top-level operator representing functional modules is as follows.

```
sorts FModule Module .
subsort FModule < Module .

op fmod_is_______endfm : Qid ImportList SortDecl
      SubsortDeclSet OpDeclSet
      VarDeclSet MembAxSet EquationSet -> FModule .
```

The representation $\overline{\texttt{NAT}}$ of `NAT` in `META-LEVEL` is the term

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
  op 's_ : 'Nat -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  var 'N : 'Nat .
  var 'M : 'Nat .
  none
  eq '_+_[{'0}'Nat, 'N] = 'N .
  eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
endfm
```

Since `NAT` has no list of imported submodules and no membership axioms (see [42, 11] for more on membership axioms) those fields are filled by the `nil` import list, and the none set of membership axioms. Similarly, since the zero and successor operators have no attributes, they have the none set of attributes.

Note that—just as in the case of terms—terms of sort Module can be meta-represented again, yielding then a term of sort Term, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels. A good example is the inductive theorem prover described in Section 4.1, where modules are meta-represented as terms in the inference rules for induction, but they have to be meta-meta-represented as terms of sort Term when used in strategies that control the application of the inductive inference rules.

There are many advanced applications that the META-LEVEL module makes possible. Firstly, strategies or tactics to guide the application of the rewrite rules of a theory can be defined by rewrite rules in *strategy languages* [15, 10, 11], which are Maude modules extending META-LEVEL in which the more basic forms of rewriting supported by functions like meta-apply and meta-reduce can be extended to arbitrarily complex rewrite strategies defined in a declarative way within the logic. Secondly, as further explained in Section 4.4, an extensible *module algebra* of module composition and transformation operations can be constructed by defining new functions on the data type Module and on other data types extending it. Thirdly, as explained in Section 4, many uses of Maude as a *metalanguage* in which we can implement other languages, including formal specification languages and formal tools, are naturally and easily supported.

3.2 Additional Metalanguage Features

Consider, for example, formal tool generator uses of Maude. That is, how can we use Maude as a meta-tool to build other formal tools? Suppose that we want to build a theorem prover for a logic, or an executable formal specification language. We can do so by representing the logic $\mathcal{L}$ of the theorem prover or specification language in question in rewriting logic by means of a representation map

$$\Psi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

Such representations are typically very simple and natural. Using reflection we can, as already explained in Section 2, *internalize* such a map as an executable equationally defined function $\overline{\Psi}$. In Maude this is accomplished using the module META-LEVEL and its sort Module. Specifically, we can reify the above representation map Ψ by defining an abstract data type $\mathtt{Module}_{\mathcal{L}}$ representing theories in the logic $\mathcal{L}$. We can then internalize the representation map Φ as an equationally-defined function

$$\overline{\Phi} : \mathtt{Module}_{\mathcal{L}} \longrightarrow \mathtt{Module}$$

in a module extending META-LEVEL. We can then use the functions meta-reduce, meta-apply, and meta-rewrite, or more complex strategies that use such functions, to execute in Maude the metarepresentation $\overline{\Psi}(\overline{T})$ of a theory T in $\mathcal{L}$. In other words, we can in this way *execute* $\mathcal{L}$ in Maude.

But we need more. To build a usable formal tool we need to build an *environment* for it, including not only the execution aspect just described, but parsing, pretty printing, and input/output. If we had instead considered formal translator generator uses of Maude, we would have observed entirely similar needs, since we need to get the specifications in different logics—originating from, or going to, different tools—in and out of Maude by appropriate parsing, pretty printing, and input-output functions. In Maude, these additional metalanguage features are supported as follows:

- The *syntax definition* for $\mathcal{L}$ is accomplished by defining the data type $\texttt{Module}_{\mathcal{L}}$. In Maude this can be done with very flexible user-definable *mixfix* syntax, that can mirror the concrete syntax of an existing tool supporting $\mathcal{L}$.
- Particularities at the *lexical* level of $\mathcal{L}$ can be accommodated by user-definable *bubble sorts*, that tailor the adequate notions of token and identifier to the language in question (see [11]). This is very useful when the theories in $\mathcal{L}$ have user-definable syntax, since then we can use such bubble sorts to represent strings of symbols corresponding to expressions that can only be analyzed after the user-defined syntax has been analyzed.
- Parsing and pretty printing for $\mathcal{L}$ is accomplished by the `meta-parse` and `meta-pretty-print` functions in `META-LEVEL`, in conjunction with the bubble sorts defined for $\mathcal{L}$.
- Input/output of theory definitions, and of commands for execution in $\mathcal{L}$ is accomplished by the predefined module `LOOP-MODE`, that provides a generic read-eval-print loop (see [11]).

In Section 4 we describe our experience in using the `META-LEVEL` and the above metalanguage features of Maude as a meta-tool to build formal tools.

4 Using Maude as a Formal Meta-Tool

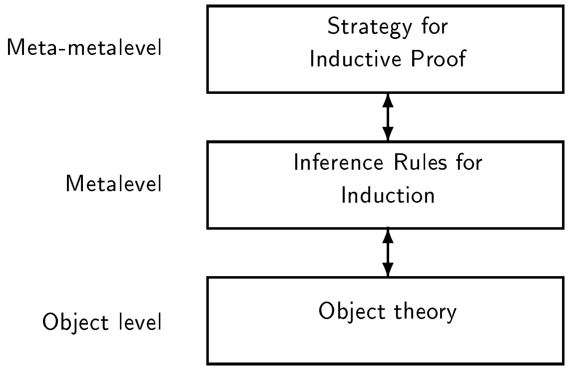
This section summarizes our experience using Maude as a formal meta-tool. Specifically, we report on two formal tool generator uses—to build an inductive theorem prover and a Church-Rosser Checker tool—three formal translator generator uses, and on the beginnings of a module algebra generator use.

4.1 An Inductive Theorem Prover

Using the reflective features of Maude's `META-LEVEL` module, we have built an inductive theorem prover for equational logic specifications [12] that can be used to prove inductive properties of both CafeOBJ specifications [22] and of functional modules in Maude.

The specifications we are dealing with are equational theories T having an initial algebra semantics. The theory T about which we want to prove inductive properties is at the object level. The rules of inference for induction can be naturally expressed as a rewrite theory $\mathcal{I}$. But since this rewrite theory uses T

as a data structure—that is, it actually uses its representation $\overline{T}$—the theory $\mathcal{I}$ should be defined at the metalevel. Proving an inductive theorem for T corresponds to applying the rules in $\mathcal{I}$ with some *strategy*. But since the strategies for any rewrite theory belong to the metalevel of such a theory, and $\mathcal{I}$ is already at the metalevel, we need *three levels of reflection* to clearly distinguish levels and make our design entirely *modular*, so that, for example, we can change the strategy without any change whatsoever to the inference rules in $\mathcal{I}$. This is illustrated by the following picture, describing the modular architecture of our theorem prover.



This tool uses several levels of reflection and associative-commutative rewriting, and expresses the inference rules at a very high level. However, thanks to the efficient implementation of Maude—that can reach more than 800,000 rewrites per second for some applications—the resulting implementation is a tool of competitive performance that can be used in practice in interactive mode with typically fast response times. Furthermore, our tool-building experience has been very positive, both in terms of how quickly we were able to develop the tool, and how easily we can extend it and maintain it.

We are currently extending this theorem prover by extending both its logic, from equational to rewriting logic, and its inference rules, to support more powerful reasoning methods, including meta-logical reasoning. In this case the logic in question is that of Maude; but a similar modular architecture could be used to build in Maude a theorem prover, inductive or not, for any logic whatsoever. All we need to do is to represent the prover's inference system as rewrite rules, and to define tactics as a separate module at the metalevel.

4.2 A Church-Rosser Checker

We have also built a Church-Rosser checker tool [12] that analyzes equational specifications to check whether they satisfy the Church-Rosser property. This tool can be used to analyze order-sorted [26] equational specifications in CafeOBJ and in Maude. The tool outputs a collection of proof obligations that can be used to either modify the specification or to prove them.

Even for specifications that are already Church-Rosser such proof obligations are in general unavoidable for two reasons. Firstly, the equations may be *conditional*, and therefore confluence may require reasoning about the conditions. Secondly, a specification with an initial algebra semantics may be *ground Church-Rosser* without being Church-Rosser, that is, it may only satisfy the property for ground terms without variables.

The Church-Rosser Checker has a reflective design similar to that of the inductive theorem prover, but somewhat simpler. Again, the module T, that we want to check is Church-Rosser, is at the object level. An inference system $\mathcal{C}$ for checking the Church-Rosser property uses $\overline{T}$ as a data structure, and therefore is a rewrite theory at the metalevel. However, since the checking process can be described in a purely functional way, there is no need in this case for an additional strategy layer at the meta-metalevel: two levels suffice.

Maude does not yet have built-in support for unification, but only for matching. Therefore, we implemented the order-sorted unification algorithm using rewrite rules which—with unification being the real workhorse of the tool—is of course inefficient. However, in spite of this inefficiency, of using several levels of reflection, and of making heavy use of associative-commutative rewriting—which is NP-complete—our tool has competitive performance. For example, it generates a long list of proof obligations for a substantial example, namely the number hierarchy from the naturals to the rationals, after 2,091,898 rewrites in 12 seconds running on a 450 MHz Pentium II.

We are currently extending this tool in several ways. Firstly, unification will be performed by Maude in a built-in way. This will greatly improve performance, and will enhance the general capabilities of Maude as a formal meta-tool. Secondly, besides Church-Rosser checking we will support Knuth-Bendix completion of membership equational logic specifications [6] and coherence completion of rewrite theories [51].

4.3 Formal Interoperability Experiments

Using the general methods explained in Section 2.2, Maude can be used as a "logical bus" to interoperate in a systematic and rigorous way different formalisms and their associated tools.

The goal is twofold. Firstly, the mappings relating different formalisms should themselves be formalized in a metalogic, so that they are rigorously defined and it becomes possible to subject them to formal metalogical analysis to verify their correctness. Secondly, the formal definition of a mapping between two logics should be *executable*, so that it can be used to carry out the translation and to interoperate in practice different formal tools. This is precisely what defining such mappings in Maude makes possible.

Maps of logics can relate any two logics of interest. In particular, when the target logic is rewriting logic, we can execute in Maude the translated theories. However, in other cases the goal may be to relate two different formalisms which may have tools of their own. We describe below some formal interoperability experiments—carried out in cooperation with several colleagues—that illustrate the different uses just discussed and some combined uses.

HOL → Nuprl. The HOL theorem proving system [29] has a rich library of theories that can save a lot of effort by not having to specify from scratch many commonly encountered theories. Potentially, this is a very useful resource not only for HOL, but for other theorem proving systems based on other logics. Howe [32] defined a map of logics mapping the HOL logic into the logic of Nuprl [17], and implemented such a mapping to make possible the translation from HOL theories to Nuprl theories. In this way, the practical goal of relating both systems and making the HOL libraries available to Nuprl was achieved. However, the translation itself was carried out by conventional means, and therefore was not in a form suitable for metalogical analysis.

After studying this mapping with the kind help of D. Howe and R. Constable, Stehr and Meseguer have recently formally specified it in Maude. The result is now an executable formal specification of the mapping that translates HOL theories into Nuprl theories. Large HOL libraries have already been translated into Nuprl this way. Current work is focusing on optimizing the specification and on expressing the map of logics within the theory of general logics.

LinLogic → RWLogic. As an illustration of the naturalness and flexibility with which rewriting logic can be used as a logical framework to represent other logics, Martí-Oliet and Meseguer defined two simple mappings from linear logic [24] to rewriting logic: one for its propositional fragment, and another for first-order linear logic [35]. In addition, they explained how—using the fact that rewriting logic is reflective and the methods discussed in Section 2.2—these mappings could be specified and executed in Maude, thus endowing linear logic with an executable environment. Based on these ideas, Clavel has specified in Maude the mapping from propositional linear logic to rewriting logic [10].

Wright → CSP → RWLogic. Architectural description languages (ADLs) can be useful in the early phases of software design and software maintenance and evolution. Furthermore, if architectural descriptions can be subjected to formal analysis, design flaws and inconsistencies can be detected quite early in the design process. The Wright language [1] is an ADL with the attractive feature of having a formal semantics based on CSP [31]. However, Wright's current tool support does not include an executable environment.

Meseguer, Nodelman, and Talcott have recently developed in Maude a prototype executable environment for Wright using two mappings. The first mapping gives an executable formal specification of the CSP semantics of Wright, that is, it associates to each Wright architectural description a CSP process. The second mapping gives an executable rewriting logic semantics to CSP itself. The composition of both mappings provides a prototype executable environment for Wright, which can be used—in conjunction with appropriate rewrite strategies—to both animate Wright architectural descriptions, and to submit such descriptions to different forms of formal analysis.

4.4 Implementing Formal Specification Languages

The efforts required for building adequate tools for formal specification languages are considerable. Such efforts can be particularly intense when such languages are *executable*, since a good execution engine must also be developed. The methods described in this paper can be used in practice to develop tools and environments for formal specification languages, including executable ones, and to endow such languages with a powerful module algebra of specification-combining operations.

We have applied these methods to the design and implementation of Maude itself. The most basic parts of the language—supporting module hierarchies of functional and system modules and some predefined modules—are implemented in C++, giving rise to a sublanguage called Core Maude. This is extended by special syntax for object-oriented specifications, and by a rich *module algebra* of parameterized modules and module composition in the Clear/OBJ style [7, 27] giving rise to the Full Maude language.

All of Full Maude has been formally specified in Core Maude [18, 19]. This formal specification—about 7,000 lines—is in fact its implementation, which is available in the Maude web page (`http://maude.csl.sri.com`). Our experience in this regard is very encouraging in several respects. Firstly, because of how quickly we were able to develop Full Maude. Secondly, because of how easy it will be to maintain it, modify it, and extend it with new features and new module operations. Thirdly, because of the competitive performance with which we can carry out very complex module composition and module transformation operations, that makes the interaction with Full Maude quite reasonable.

But the reflective methods described in this paper, that underly our development of Full Maude, are much more general. They can equally be used to develop high-performance executable environments for other formal specification languages with much less effort and much greater flexibility, maintainability, and extensibility than what would be required in conventional implementations. For example, an executable environment for Action Semantics [45] is currently being considered, and it would be quite interesting to explore Maude implementations for other specification languages such as a next-generation CafeOBJ [22] and CASL [16].

Furthermore, we plan to generalize the module algebra that we have developed for Maude into a module algebra *generator*, that could endow many other specification languages with powerful and extensible algebras for combining and transforming specifications. As explained in Section 2.2, this can be done by defining such a module algebra as a *parameterized* algebraic data type. The module algebra of Maude provided by the Full Maude specification should then be regarded as the particular instance of such a generic construction, namely, for the case in which the underlying logic $\mathcal{L}$ is rewriting logic.

5 Conclusions

We have argued that, given the different perspectives from which a complex software system has to be analyzed, the multiplicity of formalisms is in some sense unavoidable. We have also argued that the technical challenges posed by the need to inter-relate formalisms require advances in formal interoperability and in meta-tool design that can be based on a meta-theory of general logics and on appropriate logical frameworks having efficient implementations. We have explained how the reflective logical framework of rewriting logic can be used, in conjunction with an efficient reflective implementation such as Maude, to design formal meta-tools and to rigorously support formal interoperability. The feasibility of these ideas and techniques has been demonstrated by a number of substantial experiments in which new formal tools and new translations between formalisms, efficient enough to be used in practice, have been generated.

Much work remains ahead to further advance these ideas. Maude 1.0 was made publicly available on the web in January 1999. It is well documented [11] and already supports all the formal meta-tool uses described in this paper. We are currently working towards version 2.0. In that new version we plan to enhance the formal meta-tool features of Maude. Specifically, we plan to increase Maude's flexibility in tailoring the lexical level of any language, to enhance its input/output capabilities by means of built-in objects, to provide efficient built-in support for unification modulo different equational theories, to support efficient search in the space of rewrite paths, and to further extend the expressiveness of Maude and of its `META-LEVEL` module.

We also plan to develop a module algebra generator by generalizing the current module algebra of Full Maude to a parameterized algebraic data type. The further development of Maude's theorem proving tools will also be very important, because it will allow carrying out proofs of *meta-logical* properties about the formalisms and translations represented in Maude.

Finally, more experience on using Maude as a formal meta-tool is needed. We hope that the recent release of Maude, and the positive experience already gained will help us and others gain a broader experience in the future.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Soft. Eng. and Meth.*, July 1997.
2. A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, December 1992.
3. D. A. Basin and R. L. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.
4. J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

5. P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *Proc. SOFTSEM'95*, pages 363–368. Springer LNCS 1012, 1995.
6. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. To appear in *Theoretical Computer Science*, `http://maude.csl.sri.com`.
7. R. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In D. Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.
8. C. Castro. An approach to solving binary CSP using computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.
9. M. Cerioli and J. Meseguer. May I borrow your logic? (Transporting logical structure along maps). *Theoretical Computer Science*, 173:311–347, 1997.
10. M. Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, `http://maude.csl.sri.com`.
12. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. `http://maude.csl.sri.com`.
13. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.
14. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288, 1996. `http://jerry.cs.uiuc.edu/reflection/`.
15. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.
16. CoFI Task Group on Semantics. CASL—The CoFI algebraic specification language, version 0.97, Semantics. `http://www.brics.dk/Projects/CoFI`, July 1997.
17. R. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1987.
18. F. Durán and J. Meseguer. An extensible module algebra for Maude. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1998.
19. F. Durán and J. Meseguer. The Maude specification of Full Maude. Technical report, Computer Science Laboratory, SRI International, February 1999.
20. S. Feferman. Finitary inductively presented logics. In R. Ferro et al., editors, *Logic Colloquium'88*, pages 191–220. North-Holland, 1989.
21. A. Felty and D. Miller. Encoding a dependent-type λ-calculus in a logic programming language. In M. Stickel, editor, *Proc. 10th. Int. Conf. on Automated Deduction, Kaiserslautern, Germany, July 1990*, volume 449 of *LNCS*, pages 221–235. Springer-Verlag, 1990.
22. K. Futatsugi and R. Diaconescu. CafeOBJ report. AMAST Series, World Scientific, 1998.

23. P. Gardner. *Representing Logics in Type Theory.* PhD thesis, Technical Report CST-93-92, Department of Computer Science, University of Edinburgh, 1992.
24. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.
25. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.
26. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
27. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen and G.R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1999.
28. J. A. Goguen, A. Stevens, K. Hobley, and H. Hilberdink. 2OBJ: A meta-logical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992.
29. M. Gordon. *Introduction to HOL: A Theorem Proving Environment.* Cambridge University Press, 1993.
30. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association Computing Machinery*, 40(1):143–184, 1993.
31. C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
32. D. Howe. Semantic foundations for embedding HOL in Nuprl. Manuscript; Bell Labs, 1996.
33. C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.
34. H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, 1995.
35. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.
36. N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.
37. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.
38. S. Matthews, A. Smaill, and D. Basin. Experience with FS_0 as a framework theory. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.
39. J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
40. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
41. J. Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence, Fort Laurerdale, Florida, January 1998*, 1998. `http://rutcor.rutgers.edu/~amai/Proceedings.html`.

42. J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., *Proc. WADT'97*, 18–61, Springer LNCS 1376, 1998.
43. J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*. Springer-Verlag, 1999.
44. J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, Santa Margherita, Italy, May/June 1994*, pages 48–80. Springer LNCS 906, 1995.
45. P. Mosses. *Action Semantics.* Cambridge University Press, 1992.
46. G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth Int. Joint Conf. and Symp. on Logic Programming*, pages 810–827. The MIT Press, 1988.
47. L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–39, 1989.
48. F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proc. Fourth Annual IEEE Symp. on Logic in Computer Science*, pages 313–322, Asilomar, California, June 1989.
49. C. Ringeissen. Prototyping combination of unification algorithms with the ELAN rule-based programming language. In H. Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications.* Springer LNCS 1232, 1997.
50. R. M. Smullyan. *Theory of Formal Systems*, volume 47 of *Annals of Mathematics Studies.* Princeton University Press, 1961.
51. P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.
52. M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* PhD thesis, Université Henry Poincaré — Nancy I, 1994.