Twenty Years of Rewriting Logic

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801, USA

Abstract

Rewriting logic is a simple computational logic that can naturally express both concurrent computation and logical deduction with great generality. This paper provides a gentle, intuitive introduction to its main ideas, as well as a survey of the work that many researchers have carried out over the last twenty years in advancing: (i) its foundations; (ii) its semantic framework and logical framework uses; (iii) its language implementations and its formal tools; and (iv) its many applications to automated deduction, software and hardware specification and verification, security, real-time and cyber-physical systems, probabilistic systems, and bioinformatics.

Keywords: rewriting logic, concurrency, logical frameworks, temporal logics, formal specification and verification, programming language semantics, networks and distributed systems, real-time systems, probabilistic systems, security, bioinformatics.

Contents

1 Introduction 3

2 Rewriting Logic in a Nutshell 4

2.1 Semantic Framework Uses: A Concurrent Objects Example . . . . . . . . . . . . 7
2.2 Logical Framework Uses: A Propositional Satisfiability Example . . . . . . . . . 10

3 Foundations 13

3.1 Rewriting Logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.1 Operational and Denotational Semantics of Rewrite Theories . . . . . . . . 15
3.1.2 Generalized Rewrite Theories . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 Computability and Coherence . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3 Unification, Generalization, Narrowing, and Symbolic Reachability . . . . . . . 20
3.4 Reflection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.5 Strategies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.6 The ρ-Calculus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.7 Sufficient Completeness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.8 Termination . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.9 Real-Time Rewrite Theories . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.10 Probabilistic Rewrite Theories . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.11 Temporal Logic Properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.11.1 Temporal Logics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

Preprint submitted to The Journal of Logic and Algebraic Programming August 10, 2011
3.11.2 Model Checking Verification of Rewrite Theories 37
3.11.3 Deductive Verification of Rewrite Theories 40
3.12 Simulation and Abstraction 42

4 Rewriting Logic as a Logical and Semantic Framework 45
4.1 Representing Logics 45
4.2 Representing Models of Concurrency 48
4.3 Rewriting Logic Semantics of Programming Languages 50
4.4 Representing Distributed Systems, Software Architectures, and Models 52

5 Rewriting Logic Languages 53
5.1 CafeOBJ 54
5.2 ELAN 54
5.3 Maude 55

6 Tools 55
6.1 Formal Tools for Rewriting Logic 55
6.1.1 The Maude Church-Rosser Checker and Coherence Checker (CRChC) 56
6.1.2 The CARIBOO Termination Tool 56
6.1.3 The Maude Termination Tool (MTT) and \( \mu \)-Term 56
6.1.4 The Maude Sufficient Completeness Checker (SCC) 56
6.1.5 The Maude Inductive Theorem Prover (ITP) 57
6.1.6 The Declarative Maude Debugger 57
6.1.8 Real-Time Maude 58
6.1.9 The PMaude Language Design 58
6.1.10 VeStA and PVeStA 58
6.2 Domain-Specific Tools 59
6.2.1 JavaFAN 59
6.2.2 K-Maude 60
6.2.3 The MatchC Tool 60
6.2.4 The Maude-NPA 60
6.2.5 MOMENT2 61

7 Some Applications 62
7.1 Automated Deduction Applications 62
7.2 Software and Hardware Specification and Verification 63
7.2.1 Modeling Languages 63
7.2.2 Programming Languages 63
7.2.3 Hardware Specification and Verification 64
7.2.4 Networks, Distributed Architectures, Middleware and Coordination 66
7.3 Security 67
7.3.1 Cryptographic Protocol Specification and Analysis 67
7.3.2 Network Security 68
7.3.3 Browser Security 70
7.3.4 Access Control 71
7.3.5 Code Security 72
7.4 Real-Time and Cyber-Physical Systems 73
7.4.1 Real-Time Network Protocols 73
7.4.2 Middleware for Distributed Real-Time Systems 74
7.4.3 Real-Time Programming Languages 74
7.4.4 Real-Time Modeling Languages 76
7.4.5 Resource Sharing Protocols 78
7.4.6 Cyber-Physical Systems 79
7.5 Probabilistic Systems 80
7.5.1 Distributed Embedded Systems 80
7.5.2 Distributed Stochastic Hybrid Systems 81
7.6 Bioinformatics, Chemical Systems, and Membranes 82
1. Introduction

The first three papers on rewriting logic were published in 1990 [300, 299, 302]; they were then expanded in [303, 304]. Since that time, many researchers around the world have made important contributions to its foundations, tools, and applications. Since 1996, the WRLA workshop has met biennially, with the 2010 Paphos meeting being its eighth edition, the RTRTS Workshop on applications to real-time systems held its first edition in Spitsbergen in March 2010, and many hundreds of papers have been published on the subject ([289] contains a bibliography up to 2002, and this journal issue contains an up-to-date bibliography). This growth makes it desirable to reflect from time to time upon the advances made, survey such advances, and perhaps get some glimpses and make some guesses about future directions. It is somewhat like taking a snapshot of a person at age twenty. I have taken some similar, total or partial pictures at earlier ages, as a child [306, 310, 309], and as a teenager [289] (with Narciso Martí-Oliet) and [313]. It seems appropriate to attempt taking a coming-of-age picture, and to ask some questions about rewriting logic such as the following:

- How well-developed are its mathematical foundations?
- To what extent have its goals as a semantic framework for concurrency, and as a logical framework, been achieved?
- Which languages and tools supporting rewriting logic programming, specification, and verification have been developed?
- In which application areas has it been shown useful?
- How do its future prospects look like?

This paper is both a survey of the work that has been done, and my own attempt to answer the above questions.

I am grateful to the many gifted researchers who have contributed to the rewriting logic research program. I will explicitly mention some of them and some of their contributions. But I cannot really do justice to either all of them or all their contributions. This is due, in part, to my own limitations in keeping up with a vast and fast-growing literature; and to the impossibility, within the scope of this survey, of discussing, even summarily, the many hundreds of publications on the subject. The compilers of the detailed bibliography contained in this issue have gathered and organized by topic all the contributions that seem to have been made to date. I refer to this bibliography for a more complete picture of the different research directions that here I can only describe in broad outlines.
2. Rewriting Logic in a Nutshell

Since the main goal of this paper is to facilitate access to a large body of research ideas to readers who may not be familiar with rewriting logic, it does not seem out of place to explain and illustrate, in an informal and impressionistic way, what rewriting logic is, and how it can be used.

Rewriting logic is like a coin with two sides: a computational side and a logical side. These two sides are complementary viewpoints on the same reality. Some applications fall more obviously into one of these sides, but when viewed as rewrite theories their other side is always present.

Computationally, rewriting logic is a semantic framework in which many different models of concurrency, distributed algorithms, programming languages, and software and hardware modeling languages can be naturally represented, executed and analyzed as rewrite theories (see Sections 4.2–4.4). Logically, it is a logical framework within which many different logics, and automated deduction procedures can likewise be represented, mechanized, and reasoned about (see Section 4.1).

Whenever anybody is selling you a semantic or logical framework you should be wary. A key reason for waryness is that such a framework may work in principle, but it may create a big gap between what is represented and its representation. I call this the representational distance imposed by the framework. For example, Turing machines provide an, in principle unobjectionable, semantic framework for sequential programming languages; but nobody uses them to define a language’s semantics, except perhaps in the sense that a compiler for a language closely resembles a Turing machine semantics for it. There is just too much distance between a high-level programming language and a Turing machine, and much, including all the language’s features, is lost in translation. In this regard, the evidence accumulated over the last twenty years strongly supports the claim that rewriting logic can rightfully be said to have “ϵ representational distance” as a semantic and logical framework. That is, what is represented and its representation are often isomorphic structures, typically differing only because of the slightly different notations used, but agreeing on all the main features.

Why is this so? Whenever you represent a concurrent system or a logic, there are two key aspects about such a representation, which could be called the static and the dynamic aspects, and rewriting logic happens to be very well-suited for naturally representing both. Representing the static aspect of a concurrent system means representing its distributed states, while representing that of a logic means representing its formulas. Instead, representing the dynamic aspect of a concurrent system means representing its concurrent transitions, while representing that of a logic means representing its inferences.

The reason why rewriting logic’s representational distance is typically $\epsilon$ is that a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ consists of an equational theory $(\Sigma, E)$ and a set of (possibly conditional) rewrite rules $R$, where $(\Sigma, E)$ specifies the statics and $R$ specifies the dynamics. If we are using $(\Sigma, E, R)$ to represent a concurrent system (resp. a logic), then the distributed states (resp. formulas)
of such a system are specified by the equational theory \((\Sigma, E)\), where \(\Sigma\) is a collection of typed operators which includes the state constructors that build up a distributed state out of simpler state components (resp. the logical and non-linear symbols that build up a formula), and where \(E\) specifies the algebraic identities that such distributed states (resp. formulas) enjoy. That is, distributed states (resp. formulas) are specified as elements of an algebraic data type, namely, the initial algebra of the equational theory \((\Sigma, E)\). Concretely, this means that a distributed state (resp. a formula) is mathematically represented as an \(E\)-equivalence class \([t]_E\) of terms (i.e., algebraic expressions) built up with the operators declared in \(\Sigma\), modulo provable equality using the equations \(E\), so that two state (resp. formula) representations \(t\) and \(t'\) describe the same state (resp. formula) if and only if one can prove the equality \(t = t'\) using the equations \(E\). The great generality with which algebraic data types can faithfully represent any data structures such as states or formulas (including binding operators such as quantifiers, \(\lambda\)-abstraction, and so on, which have a natural algebraic specification using a calculus of explicit substitutions such as CINNI [416]) is the reason why the static aspect can typically be represented with an \(\epsilon\) representational distance.

The dynamic aspect of a system or logic represented as a rewrite theory \(R = (\Sigma, E, R)\) is specified by its set \(R\) of rewrite rules. Why are they likewise so flexible? I focus first on concurrent systems specified with unconditional rewrite rules; the case of logics is discussed afterwards. What the rules \(R\) then represent are the system’s local concurrent transitions. Each rewrite rule in \(R\) has the form \(t \rightarrow t'\), where \(t\) and \(t'\) are algebraic expressions in the syntax of \(\Sigma\). The lefthand side \(t\) describes a local firing pattern, and the righthand side \(t'\) describes a corresponding replacement pattern. That is, any fragment of a distributed state which is an instance of the firing pattern \(t\) can perform a local concurrent transition in which it is replaced by the corresponding instance of the replacement pattern \(t'\). Both \(t\) and \(t'\) are typically parametric patterns, describing not single states, but parametric families of states. The parameters appearing in \(t\) and \(t'\) are precisely the mathematical variables that \(t\) and \(t'\) have, which can be instantiated to different concrete expressions by a mapping \(\theta\), called a substitution, sending each variable \(x\) to a term \(\theta(x)\). The instance of \(t\) by \(\theta\) is then denoted \(\theta(t)\).

The most basic logical deduction steps in a rewrite theory \(R = (\Sigma, E, R)\) are precisely atomic concurrent transitions, corresponding to applying a rewrite rule \(t \rightarrow t'\) in \(R\) to a state fragment which is an instance of the firing pattern \(t\) by some substitution \(\theta\). That is, up to \(E\)-equivalence, the state is of the form \(C[\theta(t)]\), where \(C\) is the rest of the state not affected by this atomic transition. Then, the resulting state is precisely \(C[\theta(t')]\), so that the atomic transition has the form \(C[\theta(t)] \rightarrow C[\theta(t')]\). Rewriting is intrinsically concurrent, because many other atomic rewrites can potentially take place in the rest of the state \(C\) (and in the substitution \(\theta\)), at the same time that the local atomic transition \(\theta(t) \rightarrow \theta(t')\) happens. That is, in general one may have complex concurrent transitions of the form \(C[\theta(t)] \rightarrow C'[\theta'(t')]\), where the rest of the state \(C\) has evolved to \(C'\) and the substitution \(\theta\) has evolved to \(\theta'\) by other (possibly many) atomic rewrites simultaneous with the atomic rewrite \(\theta(t) \rightarrow \theta(t')\). The rules of deduction of
rewriting logic [303, 77] (which in general allow rules in $R$ to be conditional) precisely describe all the possible, complex concurrent transitions that a system can perform, so that concurrent computation and logical deduction coincide. Such inference rules are discussed in Section 3.1.

If instead we adopt a logical point of view, so that the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ represents a logic, then the rewrite rules $R$ exactly specify the inference rules of the logic. What the rules rewrite may be formulas, or other formula-based data structures such as sets or lists of formulas, sequents, and so on. In the simplest case of an unconditional rewrite rule $t \rightarrow t'$, we describe an inference step in which we pass from a formula or formula-based structure which is an instance of the pattern $t$ to another such formula or structure which is the corresponding instance of $t'$, perhaps in a context $C$. That is, such atomic inference steps again take the form $C[\theta(t)] \rightarrow C'[\theta'(t')]$, for $\theta$ the substitution instantiating the patterns $t$ and $t'$. Often, however, logical inference steps are conditional, and this may happen in two different ways. First, an inference step $t \rightarrow t'$ may only be allowed if we can previously show that other related steps, say, $u_1 \rightarrow v_1, \ldots, u_n \rightarrow v_n$ can be taken. Second, the inference step may be further constrained by a so-called side condition such as, for example, that a certain variable involved in the step is not a free variable in a given formula. Algebraically, such side conditions can be represented as equational constraints of the form $w_1 = q_1 \land \ldots \land w_m = q_m$. The $\epsilon$ representational distance of rewriting logic as a logical framework is due to the fact that such conditional inference rules can be exactly represented in $R$ as conditional rewrite rules of the form

$$t \rightarrow t' \text{ if } u_1 \rightarrow v_1 \land \ldots \land u_n \rightarrow v_n \land w_1 = q_1 \land \ldots \land w_m = q_m.$$

Of course, what we regard as concurrent computation or as logical deduction may, like beauty, be just in the eyes of the beholder. For example, we may regard any rewrite theory $(\Sigma, E, R)$ where $\Sigma$ has just a binary operator $\otimes$ and some constants, including a unit element $I$, $E$ has associativity and commutativity axioms for $\otimes$ and an axiom for $I$ as identity of $\otimes$, and $R$ is a collection of unconditional ground rewrite rules, as either a Petri net, or as a theory in the linear conjunctive ($\otimes$) fragment of propositional linear logic [287]. But since both structures are mathematically isomorphic, there is no fact of the matter about which viewpoint should be adopted: this is just a pragmatic issue depending on what applications one has in mind.

I illustrate below all the ideas just discussed by means of two simple examples, one of a concurrent object system and another of an automated deduction procedure. For concreteness I give the specifications in Maude [102, 105], a language and system implementation directly based on rewriting logic (rewriting logic languages are discussed in Section 5). This emphasizes that rewriting logic is a computational logical and semantic framework, so that systems and logics can not only be mathematically represented: they can also be efficiently executed if they satisfy some minimum requirements (see Section 3.2).
2.1. Semantic Framework Uses: A Concurrent Objects Example

I present a simple concurrent object-based system specified in Maude. Maude’s syntax is user-definable: operators can be declared with any desired “mixfix” syntax. A concurrent state made up of objects and messages can be thought of as a “soup” in which objects and messages are freely floating and can come into contact with each other in communication events. Mathematically, this means that the concurrent state, called a configuration, is modeled as a multiset or bag built up by a multiset union operator which satisfies the axioms of associativity and commutativity, with the empty multiset as its identity element. We can, for example, denote multiset union with empty syntax, that is, just by juxtaposition by declaring the type (called a sort) Configuration of configurations, which contains the sorts Object and Msg as subsorts, the empty configuration none, and the configuration union operator as follows:

```
sorts Object Msg Configuration .
subsorts Object Msg < Configuration .
op none : -> Configuration [ctor] .
op _ _ : Configuration Configuration -> Configuration
   [ctor config assoc comm id: none] .
```

Each operator is declared with the op keyword, followed by its syntax, the list of its argument sorts, an arrow ->, and its result sort. The configuration union operator has two argument positions, which are made by underbars. Before and/or after such underbars, any desired syntax tokens can be declared. In this case an empty syntax (juxtaposition) has been chosen, so that no syntax tokens at all are declared. Note that constants like none are viewed as operators with no arguments. The keyword config declares that this is a union operator for configurations of objects and messages (the significance of this for fair execution is explained in Section 3.5). The assoc comm id: none declarations declare the associativity axiom \((xy)z = x(yz)\), the commutativity axiom \(xy = yx\), and the identity axiom \(x\ none = x\). Maude then supports rewriting modulo such axioms, so that a rule can be applied to a configuration regardless of parentheses, and regardless of the order of arguments. The ctor keyword declares that both none and _ _ are state-building constructors, as opposed to functions defined on such constructors (see Section 3.7).

Consider an object-based system containing three classes of objects, namely, Buffer, Sender, and Receiver objects, so that a sender object sends to the corresponding receiver a sequence of values (say natural numbers) which it reads from its own buffer, while the receiver stores the values it gets from the sender in its own buffer. In Maude’s Full Maude language extension (see Part II of [105]), such object classes can be declared as subsorts of the Object sort in class declarations, which specify the names and sorts of the attributes of objects in the class. The above three classes can be defined with class declarations:

```
class Buffer | q : NatList, owner : Oid .
class Sender | cell : Nat?, cnt : Nat, receiver : Oid .
class Receiver | cell : Nat?, cnt : Nat .
```
In general, if a class \( \text{Cl} \) has been declared with attributes \( a_1 \) of sort \( A_1 \), ..., \( a_n \) of sort \( A_n \), in a class declaration

\[
\text{class Cl | a_1 : A_1, \ldots, a_n : A_n}.
\]

then an object \( o \) of class \( \text{Cl} \) is a record-like structure of the form:

\[
< o : \text{Cl} | a_1 : v_1, \ldots, a_n : v_n >
\]

where each \( v_i \) is a term of sort \( A_i \). For example, assuming that one uses quoted identifiers as object identifiers (by importing the \textit{QID} module and giving the subsort declaration \textit{Qid < Oid}), and that the supersort \textit{Nat?} of \textit{Nat} containing an empty value \textit{mt}, and the sort \textit{NatList} of lists of natural numbers have been declared as:

\[
\begin{align*}
\text{sorts} & \quad \text{Nat? NatList}. \\
\text{subsorts} & \quad \text{Nat < Nat? NatList}. \\
\text{op} & \quad \text{mt} : \to \text{Nat?} \quad \text{[ctor]}.
\end{align*}
\]

then the following is an initial configuration of a sender and a receiver object, each with its own buffer, and each with its cell currently empty:

\[
\begin{align*}
< 'a : \text{Buffer} | q : 1 . 2 . 3 , \text{owner} : 'b > \\
< 'b : \text{Sender} | \text{cell} : \text{mt} , \text{cnt} : 0 , \text{receiver} : 'd >
\end{align*}
\]

A sender object can send messages to its corresponding receiver object. The specifier has complete freedom to define the format of such messages by declaring operators of sort \( \text{Msg} \), using the \textit{msg} keyword instead of the more general \textit{op} keyword to emphasize that the resulting terms are messages. For example, one can choose the following format:

\[
\begin{align*}
\text{msg} \quad \text{to} \_\_ \text{from} \_\_ : \text{Oid Nat Oid Nat} \to \text{Msg}.
\end{align*}
\]

where a message, say, \( \text{to 'd :: 3 from 'b cnt 1} \), means that \( 'b \) sends to \( 'd \) the data item 3, with counter 1, indicating that this is the first element transmitted. This last information is important, since message passing in a configuration is usually asynchronous, so that messages could be received out-of-order. Therefore, receiver objects need to use the counter information to properly reassemble a list of transmitted data. Of course, out-of-order communication is just one possible situation that can be modeled. If, instead, one wanted to model in-order communication, the distributed state could contain channels, similar for example to the buffer objects, so that axioms of associativity and identity are satisfied when inserting messages into a channel, but not commutativity, which is the axiom allowing out-of-order communication in a configuration of objects and messages. Up to now we have just defined the distributed states of our object-based system as the algebraic data type associated
to the equational theory \((\Sigma, E)\), where \(\Sigma\) is the signature whose sorts have been declared with the `sort` (and `class`) keywords, with subsort relations declared with the `subsort` keyword, and whose operators have been declared with the `op` (or `msg`) keywords; and where the equations \(E\) have been declared\(^1\) as equational axioms of associativity and/or commutativity and/or identity associated to specific operators, declared with the `assoc`, `comm` and `id` keywords.

What about the concurrent transitions for buffers, senders, and receivers? They are specified by rewrite rules \(R\) such as the following (note that, by convention, object attributes not changed by a rule need not be mentioned in it):

\[
\begin{align*}
\text{vars} & \ X \ Y \ Z : \text{Oid} \ . \ \text{vars} \ N \ E : \text{Nat} \ . \ \text{vars} \ L \ L' : \text{NatList} \ . \\
\text{rl [read]} & : < X : \text{Buffer} \ | \ q : L . E , \ \text{owner} : Y > \\
& \quad \quad \quad \quad \rightarrow < X : \text{Buffer} \ | \ q : L , \ \text{owner} : Y > \\
& \quad \quad \quad \quad < Y : \text{Sender} \ | \ \text{cell} : \text{mt} , \ \text{cnt} : N > \\
\text{rl [write]} & : < X : \text{Buffer} \ | \ q : L , \ \text{owner} : Y > \\
& \quad \quad \quad \quad \rightarrow < X : \text{Buffer} \ | \ q : E . L , \ \text{owner} : Y > \\
& \quad \quad \quad \quad < Y : \text{Sender} \ | \ \text{cell} : \text{mt} > . \\
\text{rl [send]} & : < Y : \text{Sender} \ | \ \text{cell} : E , \ \text{cnt} : N , \ \text{receiver} : Z > \\
& \quad \quad \quad \quad \rightarrow < Y : \text{Sender} \ | \ \text{cell} : \text{mt} , \ \text{cnt} : N > (\text{to} \ Z :: E \ \text{from} \ Y \ \text{cnt} \ N) . \\
\text{rl [receive]} & : < Z : \text{Receiver} \ | \ \text{cell} : \text{mt} , \ \text{cnt} : N > \\
& \quad \quad \quad \quad (\text{to} \ Z :: E \ \text{from} \ Y \ \text{cnt} \ N) \\
& \quad \quad \quad \quad \rightarrow < Z : \text{Receiver} \ | \ \text{cell} : E , \ \text{cnt} : N + 1 > .
\end{align*}
\]

That is, senders can read data from the buffer they own and update their count; and receivers can write their received data in their own buffer. Also, each time a sender has a data element in its cell, it can send it to its corresponding receiver with the appropriate count; and a receiver with an empty cell can receive a data item from its sender, provided it has the correct counter. Note that rewriting is intrinsically concurrent; for example, `b` could be sending the next data item to `d` at the same time that `d` is receiving the previous data item or is writing it into its own buffer; furthermore, there could be many different sender-receiver pairs executing concurrently in the same configuration. Note also that the rules `send` and `receive` describe the asynchronous message passing communication between senders and receivers typical of the Actor model \([3]\). Instead, the `read` and `write` rewrite rules describe synchronization events, in which a buffer and its owner object synchronously transfer data between each other. This illustrates the flexibility of rewriting logic as a semantic framework: no assumption of either synchrony or asynchrony is built into the logic. Instead, many different styles of concurrency and of in-order or out-of-order communication can be easily modeled.

\(^1\)In Maude one can also declare explicit equations defining functions on data constructors with the `eq` and `ceq` keywords.
2.2. Logical Framework Uses: A Propositional Satisfiability Example

Procedures for propositional satisfiability (SAT) are very useful in many applications, including SAT solving modulo decidable theories in first-order theorem proving. Sometimes, however, in the quest for performance the algorithmic details of a SAT solver may become so involved that it is unclear whether it is sound. In fact, this is not a theoretical possibility but a real concern in actual SAT solvers. What is needed is a clear separation of concerns between the SAT solver’s inference system and its (typically very sophisticated) heuristics. This separation of concerns has been advocated by Cesare Tinelli, who gave a precise sequent calculus specification of the Davis-Putnam-Logemann-Loveland (DPLL) SAT solving procedure, from which a proof of its correctness is quite direct, in [440]. I discuss in what follows a slightly enhanced version of Tinelli’s inference system in [440], which Tinelli and I then used to develop the rewriting logic specification of the inference system executable in Maude discussed below.

Tinelli’s sequent-based formalization is as follows. To reason about the satisfiability of a propositional formula $\varphi$ we first put it in conjunctive normal form as a conjunction of clauses (i.e., disjunctions of literals) $C_1 \land \ldots \land C_n$, which is logically equivalent to the set of clauses $\Gamma = \{C_1, \ldots, C_n\}$. The DPLL procedure can then be formalized as a sequent-based inference system with sequents of the form $\Delta \vdash \Gamma$, where $\Delta$ is a set of literals, i.e., of atomic propositions $p$ or negations $\neg p$ of such propositions, and where $\Gamma$ is a set of clauses. As usual insequent formulations, a set $\Gamma = \{C_1, \ldots, C_n\}$ is written without the enclosing parentheses as $\Gamma = C_1, \ldots, C_n$. Likewise, a set of literals $\Delta = \{l_1, \ldots, l_m\}$ is written $\Delta = l_1, \ldots, l_m$. The DPLL procedure can then be formalized as the following inference system:

\[
\begin{align*}
\text{(subsume)} & \quad \frac{\Delta \vdash \Gamma, l \lor C}{\Delta \vdash \Gamma} \quad \text{if } l \in \Delta \\
\text{(resolve)} & \quad \frac{\Delta \vdash \Gamma, l \lor C}{\Delta \vdash \Gamma, C} \quad \text{if } \neg l \in \Delta \\
\text{(assert)} & \quad \frac{\Delta \vdash \Gamma, l}{\Delta, l \vdash \Gamma, l} \quad \text{if } l, \neg l \not\in \Delta \\
\text{(close)} & \quad \frac{\Delta \vdash \Gamma, \Box}{\emptyset \vdash \Box} \quad \text{if } \Delta, \Gamma \neq \emptyset \\
\text{(split)} & \quad \frac{\Delta \vdash \Gamma, l \lor C}{\Delta, l \vdash \Gamma, C} \quad \frac{\Delta \vdash \Gamma, \neg l \lor C}{\Delta, \neg l \vdash \Gamma, C} \quad \text{if } l \not\in \Delta, \neg l \not\in \Delta, C \neq \Box
\end{align*}
\]

where $\Box$ denotes the empty clause, $C$ ranges over clauses, and for $l$ any literal, $\neg \neg l = l$. The rewriting logic formalization of this inference system as a rewrite theory $\mathcal{R}_{\text{DPLL}} = (\Sigma_{\text{DPLL}}, E_{\text{DPLL}}, R_{\text{DPLL}})$ must axiomatize sequents as the algebraic data type of the equational theory $(\Sigma_{\text{DPLL}}, E_{\text{DPLL}})$, and then axiomatize the inference rules as rewrite rules in $R_{\text{DPLL}}$. We can, however, do better than that. Because of rewriting logic’s distinction between equations and rules, we can choose to axiomatize as equations those inference rules that are deterministic (in the sense that their combined application will lead to a unique final result) and that should always be applied exhaustively. We only
need to axiomatize as rules the truly nondeterministic rules. This makes the specification both more clever, since it makes explicit the implicit determinism, and much more efficient, because it can drastically reduce the amount of search required, given that search is now only needed for the nondeterministic rules.

For the above DPLL inference system, only the \textbf{split} rule is nondeterministic: all other rules can be axiomatized equationally. The rewriting logic axiomatization $\mathcal{R}_{DPLL} = (\Sigma_{DPLL}, E_{DPLL}, R_{DPLL})$ is in fact executable in Maude as the DPLL module below and can be used as a prototype of the DPLL procedure.

Of course, the real smarts of a SAT solver are in its heuristics; but this is the whole point of Tinelli’s proposal: we should cleanly separate between the inference system and its heuristics and not mix the two together in a confusion of pointers. Nevertheless, the rewrite theory $\mathcal{R}_{DPLL} = (\Sigma_{DPLL}, E_{DPLL}, R_{DPLL})$ captures in a declarative way a simple but important part of those heuristics, namely, it identifies those deterministic rules that should always be applied exhaustively; but it leaves unspecified the heuristics for applying the \textbf{split} rule, which is where the crucial difference resides for obtaining an efficient implementation. Heuristics or, more precisely, \textit{strategies} are a separate and modular dimension of a rewrite theory that I discuss in Section 3.5. The same rewrite theory can be executed with many different strategies, which may be better or worse in various regards; but strategies, being now a particular way of applying intrinsically correct rules, can never affect correctness. For DPLL and DPLL(T) this completely agrees with Tinelli’s approach in [440] and in his later joint work with Nieuwenhuis and Oliveras [344], where the issue of strategies is discussed in depth.

```plaintext
mod DPLL is protecting QID .
sorts Literal Context Clause ClauseSet Sequent .
subsorts Qid < Literal < Context Clause < ClauseSet .
op ~ : Literal -> Literal .
op null : -> Context .
op _,_ : ClauseSet ClauseSet -> ClauseSet [assoc comm id: null] .
op [] : -> Clause .
op _\lor_ : Clause Clause -> Clause [assoc comm id: ([])] .
op _|-_ : Context ClauseSet -> Sequent .

var p : Qid .
var l : Literal .
var CTX : Context .
var C : Clause .
var CS : ClauseSet .

eq ~(~(l)) = l .
eq 1 in 1,CTX = true .

eq [subsume] : 1,CTX |- CS,(l \lor C) = 1,CTX |- CS .
eq [resolve1] : p,CTX |- CS,(~(p) \lor C) = p,CTX |- CS,C .
eq [resolve2] : ~(p),CTX |- CS,(p \lor C) = ~(p),CTX |- CS,C .
```
Let me discuss the rewrite theory $R_{DPLL} = (\Sigma_{DPLL}, E_{DPLL}, R_{DPLL})$ in more detail. The signature $\Sigma_{DPLL}$ describes the sorts, subsorts, constructors, and auxiliary functions needed for sequents. Note that the order-sorted type structure in DPLL precisely captures the types of: (i) propositional symbols, represented here by the sort $\text{Qid}$ of quoted identifiers, (ii) literals, (iii) sets of literals, called contexts, (iv) clauses, and (v) sets of clauses. Sequents are then pairs of a context and a set of clauses. Negation $\neg$ is represented by $\sim$ in typewriter notation, set membership $\in$ by $\in$, and the empty set $\emptyset$ by $\text{null}$. All other operators are typewriter analogues of their mathematical notation.

The equations $E_{DPLL}$ are essentially of two kinds: those axiomatizing the basic properties of sequents, and those expressing the deterministic inference rules $\text{subsume}$, $\text{resolve}$, $\text{assert}$, and $\text{close}$. In any sequent calculus, the first order of business is to define the so-called structural rules enjoyed by sequents $\Delta \vdash \Gamma$. For propositional and first-order logic, sequents $\Delta \vdash \Gamma$ enjoy structural rules making $\Delta$ and $\Gamma$ sets of formulas. This is captured above by the $\text{assoc}$, $\text{comm}$ (corresponding to the so-called exchange structural rule of sequents), and $\text{id}$: attributes of the operator $\cup$ of set union; but there is still one more structural rule, namely, the so-called $\text{contraction}$ rule expressing the idempotency of set union, which is specified above as the $\text{contraction}$ equation. Not all sequent calculi obey all these structural rules: linear logic drops $\text{contraction}$, and Lambek’s logic drops both $\text{contraction}$ and $\text{exchange}$. The general point is that, by choosing the adequate equations, we can capture any desired structural axioms. Furthermore, by declaring some of them as axioms, we can reason modulo such axioms without having to explicitly apply them as structural inference rules: the only exception here is the $\text{contraction}$ rule, which is explicitly applied as a simplification equation modulo the built-in associativity, commutativity, and identity axioms for set union.

Since negations are restricted to literals in the above type structure, we only need the equation stating that the double negation of a literal is the literal itself. Set membership needs only be defined in the positive case by the obvious equation; since we are only defining the positive case, an expression like $'a \in 'b,'c,'d$, where $'a$ is not in the set $'b,'c,'d$, does not have a Boolean value: its value is the expression itself, which belongs to the supersort $[\text{Bool}]$ of $\text{Bool}$ automatically added by Maude. For simplicity and efficiency reasons, except for the $\text{assert}$ rule, all deterministic inference rules that had side conditions in Tinelli’s formulation are now specified as unconditional equations declared with the $\text{eq}$ keyword. Sometimes, as in the case of $\text{resolve}$ and $\text{close}$, two equations are needed to specify one rule. This is done to express the conditions of the corresponding inference rules in the patterns of the unconditional equations.
such as the implicit disjunction of either $\Delta$ or $\Gamma$ being nonempty in the side condition of close, and the side condition of the resolve inference rule. Finally, the two conditional rewrite rules in $R_{DPLL}$, declared with the crl keyword, exactly capture the two inference rules specified by the two different outcomes of the split rule. Again, the representational distance between the textbook formulation of the DPLL sequent calculus and its expression in an executable form in the rewriting logic framework can be fairly described as an $\epsilon$ distance. Furthermore, rewriting logic’s distinction between equations and rules gives a specifier additional expressive power to discriminate between deterministic and nondeterministic inference rules.

3. Foundations

The foundations of rewriting logic begin of course with its proof theory and its model theory, but have various other aspects such as reflection, strategies, and executability properties. Furthermore, rewrite theories themselves can be extended to model real-time systems and probabilistic systems. Finally, the properties enjoyed by a rewrite theory need not be just those expressible in rewriting logic itself: they may also be expressible in other logics, such as temporal logics. Temporal logic properties can then be verified by model checking or deductive methods.

3.1. Rewriting Logic

A rewrite theory\(^2\) is a tuple $\mathcal{R} = (\Sigma, E, R)$, with:

- $(\Sigma, E)$ an equational theory with function symbols $\Sigma$ and equations $E$; and

- $R$ a set of labeled rewrite rules of the general form

\[ r : t \rightarrow t' \]

\(^2\) As already mentioned in Section 2, rewrite rules can be conditional. To simplify the exposition I present here the simplest version of rewrite theories, namely, unconditional rewrite theories over an unsorted equational theory $(\Sigma, E)$. In general, however, the equational theory $(\Sigma, E)$ can be many-sorted, order-sorted, or even a membership equational theory [308]. And the rules can be conditional, where a rule’s condition has a conjunction of rewrites, equalities, and even memberships, that is, rules have the general form

\[ r : t \rightarrow t' \text{ if } (\bigwedge_i u_i = u'_i) \land (\bigwedge_j v_j : s_j) \land (\bigwedge_i w_i \rightarrow w'_i) \]

Furthermore, the theory may also specify an additional mapping $\phi : \Sigma \rightarrow \mathcal{P}(N)$, assigning to each function symbol $f \in \Sigma$ (with, say, $n$ arguments) a set $\phi(f) = \{i_1, \ldots, i_k\}$, $1 \leq i_1 < \ldots < i_k \leq n$ of frozen argument positions under which it is forbidden to perform any rewrites. Rewrite theories in this more general sense are studied in detail in [77]; they are clearly more expressive than the simpler unconditional and unsorted version presented here. This more general notion is the one supported by the Maude language [103]. I discuss further these generalized rewrite theories in Section 3.1.2.
with \( r \) a label and \( t, t' \) \( \Sigma \)-terms which may contain variables in a countable set \( X \) of variables which we assume fixed in what follows; that is, \( t \) and \( t' \) are elements of the term algebra \( T_{\Sigma}(X) \). In particular, their corresponding sets of variables, \( \text{vars}(t), \text{vars}(t') \) are both contained in \( X \).

Given \( R = (\Sigma, E, R) \), the sentences that \( R \) proves are rewrites of the form, \( t \rightarrow t' \), with \( t, t' \in T_{\Sigma}(X) \), which are obtained by finite application of the following rules of deduction:

- **Reflexivity.** For each \( t \in T_{\Sigma}(X) \), \( t \rightarrow t \)
- **Equality.** \( u \rightarrow v \quad E \vdash u = u' \quad E \vdash v = v' \quad u' \rightarrow v' \)
- **Congruence.** For each \( f : k_1 \ldots k_n \rightarrow k \) in \( \Sigma \), and \( t_i, t_i' \in T_{\Sigma}(X) \), \( 1 \leq i \leq n \),
  
  \[
  t_1 \rightarrow t_1' \quad \ldots \quad t_n \rightarrow t_n' \quad f(t_1, \ldots, t_n) \rightarrow f(t_1', \ldots, t_n')
  \]
- **Replacement.** For each rule \( r : t \rightarrow t' \) in \( R \), with, say, \( \text{vars}(t) \cup \text{vars}(t') = \{x_1, \ldots, x_n\} \), and for each substitution \( \theta : \{x_1, \ldots, x_n\} \rightarrow T_{\Sigma}(X) \), with \( \theta(x_i) = p_i, 1 \leq l \leq n \), then
  
  \[
  p_1 \rightarrow p_1' \quad \ldots \quad p_n \rightarrow p_n' \quad \theta(t) \rightarrow \theta'(t')
  \]
  
  where for \( 1 \leq i \leq n \), \( \theta'(x_i) = p_i' \).
- **Transitivity**
  
  \[
  t_1 \rightarrow t_2 \quad t_2 \rightarrow t_3 \quad t_1 \rightarrow t_3
  \]

We can visualize the above inference rules as follows:

**Reflexivity**

\[
\begin{array}{c}
\Delta \\
\rightarrow \\
\Delta
\end{array}
\]

**Equality**

\[
\begin{array}{c}
\Delta \\
\parallel \\
\Delta
\end{array}
\]

**Congruence**

\[
\begin{array}{c}
\Delta \\
\parallel \\
\Delta
\end{array}
\]

**Replacement**

\[
\begin{array}{c}
\Delta \\
\parallel \\
\Delta
\end{array}
\]

**Transitivity**

\[
\begin{array}{c}
\Delta \\
\parallel \\
\Delta
\end{array}
\]
The notation $\mathcal{R} \vdash t \rightarrow t'$ states that the sequent $t \rightarrow t'$ is provable in the theory $\mathcal{R}$ using the above inference rules. Intuitively, we should think of the inference rules as different ways of constructing all the (finitary) concurrent computations of the concurrent system specified by $\mathcal{R}$. The Reflexivity rule says that for any state $t$ there is an idle transition in which nothing changes. The Equality rule specifies that the states are in fact equivalence classes modulo the equations $E$. The Congruence rule is a very general form of "sideways parallelism," so that each operator $f$ can be seen as a parallel state constructor, allowing its arguments to evolve in parallel. The Replacement rule supports a different form of parallelism, which I call "parallelism under one’s feet," since besides rewriting an instance of a rule’s lefthand side to the corresponding righthand side instance, the state fragments in the substitution of the rule’s variables can also be rewritten. Finally, the Transitivity rule allows us to build longer concurrent computations by composing them sequentially.

3.1.1. Operational and Denotational Semantics of Rewrite Theories

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has both a deduction-based operational semantics, and an initial model denotational semantics. Both semantics are defined naturally out of the proof theory just described. The deduction-based operational semantics of $\mathcal{R}$ is defined as the collection of proof terms [303] of
the form $\alpha : t \rightarrow t'$. A proof term $\alpha$ is an algebraic description of a proof tree proving $R \vdash t \rightarrow t'$ by means of the inference rules of rewriting logic. As already mentioned, such proof trees describe the different finitary concurrent computations of the concurrent system axiomatized by $R$. When we specify $R$ as a Maude module and rewrite a term $t$ with the rewrite or rewrite commands, obtaining a term $t'$ as a result, we can use Maude's trace mode to obtain a sequentialized version of a proof term $\alpha : t \rightarrow t'$ of the particular rewrite proof built by the Maude interpreter.

A rewrite theory $R = (\Sigma, E, R)$ has also a model-theoretic semantics, so that the inference rules of rewriting logic are sound and complete with respect to satisfaction in the class of models of $R$ [303]. Such models are categories with a $(\Sigma, E)$-algebra structure [303]. These are “true concurrency” denotational models of the concurrent system axiomatized by $R$. That is, this model theory gives a precise mathematical answer to the question: when do two descriptions of two concurrent computations denote the same concurrent computation? The class of models of a rewrite theory $R = (\Sigma, E, R)$ has an initial model $T_R$ [303]. The initial model semantics is obtained as a quotient of the just-mentioned deduction-based operational semantics, precisely by axiomatizing algebraically when two proof terms $\alpha : t \rightarrow t'$ and $\beta : u \rightarrow u'$ denote the same concurrent computation. Of course, $\alpha$ and $\beta$ should have identical beginning states and identical ending states. By the Equality rule this forces $E \vdash t = u$, and $E \vdash t' = u'$. That is, the objects of the category $T_R$ are $E$-equivalence classes $[t]$ of ground $\Sigma$-terms, which denote the states of our system. The arrows or morphisms in $T_R$ are equivalence classes of proof terms, so that $[\alpha] = [\beta]$ iff both proof terms denote the same concurrent computation according to the “true concurrency” axioms. Such axioms are very natural. They express that the Transitivity rule behaves as an arrow composition and is therefore associative. Similarly, the Reflexivity rules provides an identity arrow for each object, satisfying the usual identity laws. Furthermore, they state that each $f$ in the Congruence rule acts not only on states but also on arrows as a functor, i.e., preserving arrow compositions and identities; this axiomatizes the true concurrency semantics of “sideways parallelism.” Finally, the “parallelism under one’s feet” semantics of the Replacement inference rule is axiomatized by giving equational axioms making each rewrite rule $r : t \rightarrow t'$ a natural transformation $r : t \Rightarrow t'$ between the functors $t$ and $t'$.

Categorical models for rewrite theories go back to [300, 302, 303]. As pointed out in those papers and mentioned above, the models of a rewrite theory are (small) categories with an algebraic structure. They generalize ordinary algebras, which are sets with an algebraic structure. This means that the underlying universe in which these models and their morphisms should be considered is the 2-category $\textbf{Cat}$ of small categories [302, 303, 332], as opposed to the underlying universe of algebras, which is the category $\textbf{Set}$ of sets. There is also a generalization of Lawvere’s functorial semantics [268] for ordinary algebras: the models of a rewrite theory $R$ have a functorial semantics as 2-product-preserving 2-functors into $\textbf{Cat}$ from its associated Lawvere 2-theory $L_R$ [301, 311]. Such Lawvere 2-theories have been replaced by weaker sesqui-categories in [422, 121];
and in the context of tile logic (which I discuss further in Section 4.2) by Lawvere double theories in [317, 75, 78].

3.1.2. Generalized Rewrite Theories

Since rewriting logic is parameterized by its underlying equational logic, the more expressive its underlying equational part, the more expressive also the resulting rewriting logic. Increased expressiveness is not a theoretical luxury, but an eminently practical goal, since formal specification languages should describe as simply and naturally as possible the widest possible class of systems. As explained in [307], membership equational logic is indeed a very expressive equational logic generalizing order-sorted equational logic (which generalizes many-sorted equational logic, which, in turn, generalizes unsorted equational logic). It supports sorts, subsorts, partiality, and sorts defined by equational conditions through membership axioms. Its atomic formulas are either equalities \( t = t' \), or memberships \( t : s \) stating that \( t \) has sort \( s \). Its sentences are universally quantified Horn clauses on such atoms. Therefore, as already pointed out in Footnote 2, a rewrite theory \( R = (\Sigma, E, R) \), whose underlying equational theory \( (\Sigma, E) \) is a membership equational theory, may have conditional rules in \( R \) whose conditions can be conjunctions of: (i) equations, (ii) memberships, and (iii) rewrites.

In the quest for more expressive versions of rewriting logic, another feature, namely, frozenness, has proved to be very useful in many applications. The idea of frozenness is that some argument positions in a state constructor should be “frozen,” in the sense that no rewrites are allowed below that position. For example, if \( \cdot \) is an action concatenation operator in a process calculus, then an expression like \( a.P \), with \( a \) an action and \( P \) a process expression, should typically not be rewritten on the \( P \) part, that is, on its second argument. This can be simply captured by saying that \( \cdot \) is frozen on its second argument. More generally, given a signature \( \Sigma \), its frozenness information is defined as a function \( \phi : \Sigma \rightarrow P_{\text{fin}}(\mathbb{N}) \), where \( \phi(f) \) is the set of frozen argument positions. For example, \( \phi(\cdot) = \{2\} \). In summary, a generalized rewrite theory is a 4-tuple \( R = (\Sigma, E, R, \phi) \) where: (i) \( (\Sigma, E) \) is a membership equational theory; (ii) the rules in \( R \) may be conditional, where conditions are conjunctions of equations, memberships and rewrites, and (iii) \( \phi \) is the frozenness map. As shown in detail in [77], all the good properties of the proof theory and the model theory of rewriting logic, including the existence of initial and free models, extend naturally to the case of generalized rewrite theories.

A theme already developed in [303], which is extended to generalized rewrite theories in [77], is that of reachability models. For some purposes (for example, model checking or reachability analysis), we may not need the initial model of a rewrite theory \( R \) in its full glory as a category of truly concurrent computations: a much more abstract model, namely, its reachability relation may be sufficient for such purposes. It is well-know that any small category can be collapsed to a binary relation on its objects which is a preorder. In exactly this way, the initial model of \( R = (\Sigma, E, R, \phi) \) is collapsed to a preorder, namely, its reachability initial model, whose elements are \( E \)-equivalence classes \([t]\) of ground terms \( t\);
and where the reachability relation \([t] \xrightarrow{\mathcal{R}} [t']\) is defined by the equivalence:
\[
[t] \xrightarrow{\mathcal{R}} [t'] \iff \mathcal{R} \vdash t \rightarrow t'.
\]

It is also possible to distinguish in the initial reachability model between one-step transitions \([t] \rightarrow^1 \mathcal{R} [t']\), corresponding to the application of a single rewrite rule, and general transitions \([t] \rightarrow \mathcal{R} [t']\), corresponding to zero, one, or more rewrite steps. This distinction is useful for various purposes, for example for giving semantics in the initial reachability model of \(\mathcal{R}\) to the next operator \(\bigcirc\) in temporal logic, a topic further discussed in Section 3.11.

### 3.2. Computability and Coherence

For execution purposes a rewrite theory \(\mathcal{R} = (\Sigma, E, R, \phi)\) should satisfy some additional requirements. As already illustrated by the DPLL example, the equations \(E\) may decompose as a union \(E = E_0 \cup B\), where \(B\) is a (possibly empty) set of structural axioms, and \(E_0\) is a set of equations used as simplification rules modulo \(B\). We should require that matching modulo \(B\) is decidable, and that the equations \(E_0\) are sort-decreasing, ground confluent and terminating modulo \(B\) and \(B\)-coherent.\(^3\) This makes the initial algebra \(T_{\Sigma/E_0 \cup B}\), that is, the set of states of the system axiomatized by \(\mathcal{R}\), computable; in fact, equality becomes obviously decidable, and the elements of the initial algebra \(T_{\Sigma/E_0 \cup B}\) have a very simple description as the (irreducible) canonical forms \(\text{can}_{E_0 \cup B}(t)\) of ground terms \(t\) by the equations \(E_0\) modulo the axioms \(B\).

What about the computability of the one-step rewrite relation \(\rightarrow^1\mathcal{R}\) in \(\mathcal{R} = (\Sigma, E, R, \phi)\)? If we want the number of states reachable in one step from a given state to be finite, for unconditional rules \(R\) we should first of all require that for any rule \(r : t \rightarrow t'\) in \(R\) we have \(\text{vars}(t') \subseteq \text{vars}(t)\). But because of rewriting logic’s Equality inference rule, computability is not at all obvious just by requiring \(\text{vars}(t') \subseteq \text{vars}(t)\), or even by further requiring that \(E = E_0 \cup B\) with the equations \(E_0\) ground Church-Rosser and terminating modulo \(B\). The problem is that the term \(t\) we rewrite need not be in canonical form, and there may easily be an infinite number of terms having the same canonical form. Otherwise put, model-theoretically the transitions in the initial model \(T_{\mathcal{R}}\), or in its collapse as an initial reachability model, are between states \([t]\) which are \(E_0 \cup B\)-equivalence classes of terms, and therefore possibly infinite sets. Finding a rewritable term in such a set is the proverbial search for a needle in a haystack and may be undecidable.

Of course, all would be easy if the existence of a one-step rewrite proof \(\mathcal{R} \vdash t \rightarrow t'\) guarantees the existence of another such one-step rewrite proof of the form \(\mathcal{R} \vdash \text{can}_{E_0 \cup B}(t) \rightarrow t''\) such that \([t'] = [t'']\), since then, assuming \(R\) is finite, the one-step rewrite relation becomes easily computable: to rewrite \([t]\) what we can do is: (i) compute the canonical form \(\text{can}_{E_0 \cup B}(t)\) of \(t\), and

---

\(^3\)For \(B\) any combination of associativity and/or commutativity and/or identity axioms, \(B\)-coherence can be automatically guaranteed by a simple theory transformation, as done automatically in Maude (see [103], Section 4.8).
(ii) try to rewrite $\text{can}_{E_0/B}(t)$ with the rules $R$ modulo $B$ in all possible ways. By the assumptions on $B$ and the finiteness of $R$ there is only a finite set of such one-step rewrites that can be effectively computed, say, $\text{can}_{E_0/B}(t) \rightarrow t_1, \ldots, \text{can}_{E_0/B}(t) \rightarrow t_k$. Then the next states reachable from $[t]$ in one step are exactly $[t_1], \ldots, [t_k]$. Furthermore, we can conveniently represent such states by their unique canonical forms $\text{can}_{E_0/B}(t_1), \ldots, \text{can}_{E_0/B}(t_k)$. This is exactly how Maude computes with a rewrite theory: it reduces $t$ to canonical form with $E_0$ modulo $B$, and then applies a rule in $R$ modulo $B$, and keeps doing this until termination or until a user-given maximum number of rewrites with $R$, that is, of one-step transitions. Similarly, in reachability analysis or model checking, Maude stores the states in the state space as their canonical forms $\text{can}_{E_0/B}(t)$.

But is this complete? Couldn’t we be missing rewrite proofs, and therefore transitions, by adopting this strategy? Completeness is guaranteed if we have the implication:

$$\mathcal{R} \vdash t \rightarrow^1 t' \Rightarrow (\exists t'') \mathcal{R} \vdash \text{can}_{E_0/B}(t) \rightarrow^1 t'' \land [t'] = [t'']$$

where $\mathcal{R} \vdash t \rightarrow^1 t'$ denotes a one-step rewrite proof. This property is called the ground coherence of $R$ with $E_0$ modulo $B$. If we do not require $t$ to be a ground term, we talk instead of the coherence of $R$ with $E_0$ modulo $B$. This coherence property was first axiomatized by Viry [449, 450]. A similar but weaker property, what Viry calls “weak coherence,” was independently identified in [304]. For the case of rewrite theories $\mathcal{R} = (\Sigma, E_0 \cup B, R)$ where $(\Sigma, E_0 \cup B)$ is an untyped equational theory, $E_0$ is confluent and terminating modulo $B$, and the axioms $B$ consist of the associativity or the associativity-commutativity of some binary function symbols in $\Sigma$, a detailed study of critical pair criteria for checking coherence of $R$ with $E_0$ modulo $A$ was given by Viry in [453]. Since coherence is such a fundamental property to ensure the computability and efficient executability of rewrite theories, coherence needed to be generalized to support more expressive rewrite theories $\mathcal{R} = (\Sigma, E_0 \cup B, R, \phi)$ with: (i) an order-sorted signature $\Sigma$ with sorts and subsorts; (ii) possibly conditional equations $E_0$; (iii) more general axioms $B$ such as any axioms whose equations are unconditional, linear and regular and have a finitary unification algorithm; (iv) conditional rules $\bar{R}$ which can have a conjunction of equations in their condition; and (v) a frozenness map $\phi$. Furthermore, proof methods and tools not only for coherence (the case studied by Viry) but also for ground coherence had to be developed. This has been done recently in [156], where the Maude Coherence Checker tool is also described (I further discuss this tool in Section 6.1.1). But of course, to check coherence or ground coherence under such general conditions is only possible if we can first check the confluence and termination of the underlying order-sorted conditional specification $(\Sigma, E_0 \cup B)$. Proof methods for checking confluence of equational theories under such general conditions and a tool (the Maude Church-Rosser Checker (CRC)) are presented in [156] (I discuss the CRC tool in Section 6.1.1). I postpone discussion of the termination methods until Section 3.8, and of termination tools until Section 6.1.

To summarize, equality of states, operations on states, and the one-step rewrite relation are all effectively computable in a finitary rewrite theory $\mathcal{R} =$
such that: (i) the (possibly conditional) equations $E$ are sort-decreasing, ground confluent and terminating modulo $B$ and $B$-coherent, and there is a $B$-matching algorithm; and (ii) the rules in $R$ are coherent with the equations $E$ modulo $B$ and have only equalities and memberships in their conditions, and if they have extra variables in their righthand side or condition which do not appear in the lefthand side, then they are admissible rules in the sense of [103, Section 6.3].

An interesting question to ask is: how expressive is rewriting logic to specify computable transition systems and computable Kripke structures (for more on Kripke structures see Section 3.11)? For equational logic the same question was asked and answered by Bergstra and Tucker in [51]: any computable algebra, i.e., any computable data type, can be specified by a finitary equational theory $(\Sigma, E)$, where the equations $E$ are confluent and terminating. For rewriting logic the same question has been asked and answered in [321]: any computable transition system, resp., computable Kripke structure, is isomorphic to one specified by a finitary rewrite theory $R = (\Sigma, E \cup B, R, \phi)$ satisfying conditions (i)–(ii) and with a chosen kind $\text{State}$ of states, so that the transition system’s set of states is the algebraic data type $T_{\Sigma/E \cup B(\text{state})}$, and its transition relation is $\rightarrow_R$.

3.3. Unification, Generalization, Narrowing, and Symbolic Reachability

The rewrite rules of a rewrite theory $R$, and the rewrite sequents we can deduce from it using the inference rules discussed in Section 3.1, are all (implicitly) universally quantified. But what about existential formulas of the form

$\exists \overline{\tau} : t(\overline{\tau}) \rightarrow t'(\overline{\tau})$

with $\overline{\tau}$ some variables; what do such formulas mean? and how can we reason formally about them? An existential formula $\exists \overline{\tau} . t(\overline{\tau}) \rightarrow t'(\overline{\tau})$ is of course a reachability property. It says that there is some instance of the state pattern $t$ from which we can reach, by some possibly complex computation, another state which is an instance of the state pattern $t'$. A negated existential formula $\neg \exists \overline{\tau} . t(\overline{\tau}) \rightarrow t'(\overline{\tau})$, which is of course equivalent to the universal formula $\forall \overline{\tau} . \neg (t(\overline{\tau}) \rightarrow t'(\overline{\tau}))$, is then an unreachability property. Reachability and unreachability properties are among the most useful properties of rewrite theories. Typically, an unreachability property expresses a safety property such as an invariant (invariants are further discussed in Section 3.11.3). An invariant says that for the states reachable from a specified set of initial states something bad can never happen. By describing our, possibly infinite, set of initial states as the ground instances of the state pattern $t$, and likewise describing the bad states as the ground instances of the state pattern $t'$, the unreachability property $\forall \overline{\tau} . \neg (t(\overline{\tau}) \rightarrow t'(\overline{\tau}))$ says that bad states in $t'$ are never reachable from the initial states in $t$ or, equivalently, that the complement of the set of bad states which are ground instances of $t'$ is an invariant, relative to the initial states in $t$. Understood this way, proving the formula $\exists \overline{\tau} . t(\overline{\tau}) \rightarrow t'(\overline{\tau})$ means proving that such a supposed invariant can be violated.

So the question now is: how can we prove existential formulas of the form $\exists \overline{\tau} . t(\overline{\tau}) \rightarrow t'(\overline{\tau})$ for a rewrite theory $R = (\Sigma, E \cup B, R, \phi)$ (where we assume
the good executability properties already discussed in Section 3.2, i.e., that $E$ is confluent and terminating modulo $B$, and $R$ is coherent with $E$ modulo $B$? Prasanna Thati and I studied this question in [328, 436] and gave several conditions on $R$ and several forms of narrowing modulo $E \cup B$ providing complete proof methods for formulas of the form $\exists \vec{x}. t(\vec{x}) \rightarrow t'(\vec{x})$. Let me summarize the simplest condition that can be given on $R$, namely, the frequently occurring case of topmost rewrite theories. These are theories having a kind $k$ (a topmost sort in some connected component in the poset of sorts) such that: (i) no operator has $k$ as sort for any of its arguments; and (ii) the terms in all rewrite rules in $R$ are of kind $k$. For example, the DPLL module satisfies these two conditions with $k = \text{Sequent}$. Our object-based example in Section 2.1 does not quite satisfy requirements (i) and (ii) (the sort Configuration has itself as an argument), but can be easily transformed into a semantically equivalent rewrite theory which does: we can just add a new sort, say, State, and declare an operator embracing a whole configuration to make a global distributed state:

$$\text{op} \{ \_ \} : \text{Configuration} \rightarrow \text{State}.$$ 

Then, to satisfy condition (ii) we can just place all the rules in our object-based example in the bigger context of a state by adding an extra variable $C$ of sort Configuration to represent “the rest of the state” (which could be empty). For example, rule send now becomes:

$$\text{rl [send]} : \{ < Y : \text{Sender} | \text{cell} : E, \text{cnt} : N, \text{receiver} : Z > C \} \rightarrow \{ < Y : \text{Sender} | \text{cell} : \text{mt}, \text{cnt} : N > (\text{to Z :: E from Y cnt N}) C \}.$$ 

As shown in [328], under conditions (i)–(ii), narrowing with $R$ modulo $E \cup B$ is a complete method for proving formulas of the form $\exists \vec{x}. t(\vec{x}) \rightarrow t'(\vec{x})$, that is, for symbolic reachability analysis. Specifically, under such conditions $\exists \vec{x}. t(\vec{x}) \rightarrow^* t'(\vec{x})$ holds for $R$ iff there is a narrowing sequence $t \sim^*_{R,E \cup B} u$ such that $u$ and $t'$ have a $E \cup B$-unifier. Narrowing is just like rewriting, but replacing matching modulo an equational theory by (semantic) unification modulo such a theory. That is, the one-step ($R,B \cup E$)-narrowing relation is defined as $t \sim_{R,E \cup B} t'$ iff there is a non-variable position $p$ of $t$, a (possibly renamed) rule $l \rightarrow r$ in $R$, and a unifier $\sigma \in \text{Unif}_{E \cup B}(t|_p,l)$ such that $t' = \sigma(t|_p)$, where $\text{Unif}_{E \cup B}(t|_p,l)$ denotes a complete set of unifiers of the equation $t|_p = l$, that is, of substitutions $\theta$ solving such an equation in the equational theory $E \cup B$, in the sense that $\theta(t|_p) =_{E \cup B} \theta(l)$. This has many applications to automated deduction, verification of safety properties, model checking, and security. Some of these applications were discussed in [328, 182]. I discuss some of the applications to model checking in Section 3.11.2, and to the analysis of cryptographic protocols in Section 7.3.

There is, however, a nontrivial problem, namely, how to obtain practical unification algorithms to compute $\text{Unif}_{E \cup B}(t|_p,l)$. If $E = \emptyset$, and $B$ is a set of axioms for which a unification algorithm exists, then things are easier. For example, for the object-based system of sender and receiver objects with buffers in Section 2.1, $E = \emptyset$ and $B$ are just the axioms of associativity, commutativity
and identity for the operators \( \_ \_ \) and \( \_ \_ \) for which there is a finitary unification algorithm generating a finite set of solutions. There is, however, the remaining problem that the signature of the above example is order-sorted (indeed, the operators \( \_ \_ \) and \( \_ \_ \) have different sorts), whereas the standard unification algorithms modulo associativity, commutativity and identity are unsorted. The paper [228] gives an algorithm, under very general conditions on \( B \), by which one can use an unsorted \( B \)-unification algorithm to obtain a complete set of order-sorted \( B \)-unifiers. Currently, Maude supports order-sorted unification for any combination of: (i) free function symbols; (ii) commutativity axioms; (iii) associativity-commutativity axioms; and (iv) associativity, commutativity and identity axioms [147].

When \( E \) is nonempty, the matter of finding a \( E \cup B \)-unification algorithm is more complex. In principle, one can assume good properties about \( E \) such as confluence, termination, and coherence modulo \( B \) and use the results in [240] to compute \( E \cup B \)-unifiers by \((E, B)\)-narrowing.\(^4\) But there are two main problems: (i) in general the number of \( E \cup B \)-unifiers is not finite; and (ii) for \( B \neq \emptyset \) unrestricted narrowing can be horribly inefficient in the sense of leading to huge search spaces, and known strategies making narrowing efficient such as basic narrowing can be incomplete. For example, basic narrowing is incomplete when \( B \) is the theory of associativity-commutativity (AC) [119]. To make things even worse, it is very easy to give examples of narrowing modulo, e.g., AC such that there is a finite set of most general narrowing solutions to a unification problem, but the narrowing algorithm modulo AC will loop forever looking for more solutions.

In fact, narrowing with (oriented) equations \( E \) modulo axioms \( B \) when \( B \neq \emptyset \) has been for a long time a terra incognita, where little was known about any practical methods to deal with these problems. Using the idea of variants of a term proposed by Comon and Delaune in [119], Santiago Escobar, Ralf Sasse and I have defined a complete narrowing strategy with equations \( E \) modulo \( B \) called folding variant narrowing [183] (see also the longer paper by Escobar, Sasse and Meseguer in this issue), that is optimally terminating, that is, if any complete narrowing strategy terminates on an input term, then folding variant narrowing will terminate on that term. Furthermore, if \( E \cup B \) has the so-called finite variant property [119], folding variant narrowing will terminate on all input terms. For \( E \cup B \)-unification purposes this means that, if \( E \cup B \) has the finite variant property, folding variant narrowing then provides a finitary \( E \cup B \)-unification algorithm. Escobar, Sasse and I have also given methods to check the finite variant property of a theory in [181]. It turns out that many cryptographic theories of interest have the finite variant property [119]. I explain in Section 7.3

\(^4\)This reduces the problem of computing \( E \cup B \)-unifiers to a symbolic reachability problem. Specifically, we add a new binary operator \( \approx \) and a fresh constant \( \text{true} \) to our syntax, and add a new rule \( x \approx x \rightarrow \text{true} \) to our equations \( E \) oriented as rewrite rules. Then the \( E \cup B \)-unification problem \( \exists \xi. t(\xi) = t'(\xi) \) is transformed into the symbolic reachability problem \( \exists \xi : t(\xi) \approx t'(\xi) \rightarrow \text{true} \) for the rewrite theory with equations \( B \) and rules \( E \cup \{ x \approx x \rightarrow \text{true} \} \), which is solved by narrowing with rules \( E \cup \{ x \approx x \rightarrow \text{true} \} \) modulo \( B \).
how — using folding variant narrowing to compute $E \cup B$-unifiers and narrowing with protocol rules $R$ modulo $E \cup B$ to perform symbolic reachability analysis — this has been exploited in the Maude-NPA protocol analyzer [177] to provide complete formal analysis for security protocols modulo a variety of cryptographic theories. More generally, Maude 2.6 supports folding variant narrowing, and symbolic reachability analysis of topmost rewrite theories, modulo a large class of equational theories $E \cup B$ having the finite variant property [147].

Generalization is the dual of unification. Given two terms $t$ and $t'$ a set of most general $B$-unifiers for the equation $t = t'$ is, as already mentioned, a set $\text{Unif}_B(t, t')$ giving us a set of most general instances $\{\theta(t) | \theta \in \text{Unif}_B(t, t')\}$, which are common instances of $t$ and $t'$ up to $B$-equivalence, i.e., $\theta(t) =_B \theta(t')$. But we can ask the dual question: given terms $t$ and $t'$, can we compute a set $\text{Gral}_B(t, t')$ of least general patterns of which $t$ and $t'$ are instances modulo $B$, i.e., least general terms $u$ such that there are substitutions $\theta, \rho$ with $\theta(u) =_B t$ and $\rho(u) =_B t'$? For example, for $B = \emptyset$ and $\Sigma$ untyped, the terms $f(f(a, a), b)$ and $f(f(b, b), c)$ have a least general generalization in the pattern $f(f(x, x), y)$. Generalization has many useful applications, for example, to automated deduction, machine learning, testing, and partial evaluation. María Alpuente, Santiago Escobar, Pedro Ojeda and I have developed generalization algorithms for two cases that are important for rewriting logic, namely, order-sorted generalization [17], and generalization modulo $B$, for $B$ any combination of associativity and/or commutativity and/or identity axioms [16].

3.4. Reflection

Reflection is a very important property of rewriting logic [111, 99, 113, 114]. Intuitively, a logic is reflective if it can represent its metalevel at the object level in a sound and coherent way. Specifically, rewriting logic can represent its own theories and their deductions by having a finitely presented rewrite theory $U$ that is universal, in the sense that for any finitely presented rewrite theory $R$ (including $U$ itself) we have the following equivalence

$$R \vdash t \rightarrow t' \iff U \vdash \langle \mathcal{R}, t \rangle \rightarrow \langle \mathcal{R}, t' \rangle,$$

where $\mathcal{R}$ and $t$ are terms representing $R$ and $t$ as data elements of $U$. Since $U$ is representable in itself, we can achieve a “reflective tower” with an arbitrary number of levels of reflection [111, 99, 113], since we have

$$R \vdash t \rightarrow t' \iff U \vdash \langle \mathcal{R}, t \rangle \rightarrow \langle \mathcal{R}, t' \rangle \iff U \vdash \langle U, \langle \mathcal{R}, t \rangle \rangle \rightarrow \langle U, \langle \mathcal{R}, t' \rangle \rangle \cdots$$

Reflection is a very powerful property: (i) it allows defining rewriting strategies by means of metalevel theories that extend $U$ and guide the application of the rules in a given object-level theory $R$ (this is further discussed in Section 3.5); (ii) it is efficiently supported in the Maude implementation by means of descent functions [101] in the META-LEVEL module; (iii) it can be used to build a variety of theorem proving and theory transformation tools (this is further discussed in Sections 4.1 and 6.1); (iv) it can endow a rewriting logic language
like Maude with powerful theory composition operations [154, 145, 146, 155]; (v) it can be used to prove metalogical properties about families of theories in rewriting logic, and about other logics represented in the rewriting logic metalogical framework [47, 107] (this is further discussed in Section 4.1); and (vi) has important connections with distributed object-based reflection and adaptation [326].

3.5. Strategies

Recall the DPLL rewrite theory in Section 2.2. The most sophisticated aspect of a SAT solver is precisely its heuristics or strategy. In the case of the rewrite theory specified in DPLL this means that performance will crucially depend on the strategies used to apply the split1 and split2 rewrite rules. Of course, this is a general issue that applies not just to SAT solving but to any rewrite theory; and that involves not only performance but also any goal-oriented use of a rewrite theory. The key issue is the potential nondeterminism of rules, as opposed to the determinism of confluent and terminating equations.

Strategies are still relevant for equations for performance and termination reasons, even when the equations are confluent and terminating, or to ensure their termination as in the case of context-sensitive rewriting for equations (see, e.g., [278] and references there). Context-sensitive rewriting of equational specifications is supported by OBJ, CafeOBJ, and Maude. Note that the addition of a frozenness map $\phi$ to a generalized rewrite theory, as explained in Section 3.1.2, provides a similar form of context-sensitive rewriting at the rule level, as opposed to the equation level. But for nondeterministic rules, strategies become a much more essential issue, because such rules, depending on when and where they are applied, can yield totally different outcomes. Frozenness provides a very simple form of strategic rewriting with rules, but more than frozenness is needed.

The role of strategies is to tame the potentially wild nondeterminism of rules for various purposes, which may include: (i) realistic modeling of the behavior of a truly nondeterministic system, whose nondeterminism we cannot or do not intend to control, but where some behaviors may be utterly unrealistic; and (ii) goal-oriented (and perhaps performance-oriented) control of the nondeterminism in a system’s execution. It is of course possible to mix purposes (i) and (ii): for example, we may have an asynchronous object system where the asynchronous behavior is only restricted by a few fairness assumptions, but where the objects are intelligent and use sophisticated game-theoretic strategies when interacting with each other. In all cases, what strategies do is to restrict the set of all possible dynamic behaviors of the system axiomatized by the given rewrite theory. That is, roughly speaking a strategy determines a subset of the set of all the possible computations of a system specified by a rewrite theory $R$.

---

5Maude supports both forms of context-sensitive rewriting: with equations using the strat attribute, and with rules using the frozen attribute.
where those computations need not be just the finite ones but may also include
infinite computations.

If we are modeling a concurrent, asynchronous system whose nondetermi-
nism is an intrinsic fact of life which cannot really be controlled, and we want
to simulate such a system, strategies may still be relevant, not so much to con-
trol the outcome of system executions as to observe the behavior of the system
under realistic assumptions about its execution. Recall the example of sender,
receiver, and buffer objects in Section 2.1. It is easy to extend such a system to
one where there are also sensor objects that are periodically writing numerical
data observations into the sender’s buffer. In this way the system immediately
becomes a nonterminating reactive system. Such a system can have executions
that are totally unrealistic. For example, a sensor can be regularly writing new
data into the sender’s buffer, the sender object can be sending this potentially
infinite stream of data to the receiver, but the receiver never receives anything!
Intuitively, such a behavior is unfair. Therefore, fair strategies, which restrict
the set of behaviors to those were starvations such as this are ruled out, are
very important to model a system’s behavior realistically, and to reason for-
ma
d
formally about system properties such as termination or satisfaction of temporal
logic formulas (I further discuss fair termination in Section 3.8, and model check-
ing of temporal logic formulas under fairness assumptions in Section 3.11). As
explained in [312], fair rewriting is not just a matter of rule fairness, that is,
of making sure that all rewrite rules are given a chance to be executed. For
example, in the above concurrent object system with sensor, buffer, sender and
receiver objects, if we have two different sensors hooked up to two different
senders through their respective buffers and two corresponding receiver objects
with their own buffers, we can be rule fair by making sure that the receive and
write rules are executed infinitely often; but we can still starve one of the
receivers, just by only executing receive and write rules for the other. That is,
we here need not only rule fairness but also object fairness: each object should
be treated fairly. The general notion is that of localized fairness in rule appli-
cations [312]. This is of course important to obtain realistic simulations. For
example, Maude provides rule fair executions through its rewrite command;
and rule and position fair executions through its frewrite, which becomes also
object fair for object-based concurrent systems specified with a multiset union
operator using the config keyword, as illustrated in the example of Section
2.1. But what can be done if we want to obtain fair behaviors besides the ones
provided by a language implementation? Fairness is just a particular kind of
temporal logic property. More generally, we can view a temporal logic formula
as a strategy expression which defines a corresponding class of behaviors. In
Section 3.11.2, I explain how an expressive temporal logic such as TLR can be
used as a strategy language, which is then implemented by a model checker.

If instead our purpose is to control the nondeterministic behavior of a rewrite
theory $R$ for goal-oriented and perhaps performance-oriented purposes, an ap-
propriate way to achieve that end is to provide a strategy language that can be
used to guide and control the way in which the rules of $R$ are applied. To give a
logical example, $R$ can be the inference system of a theorem prover or of a SAT
solver, and then the strategies correspond to proof tactics or to solving heuristics. In concurrent system applications the relevant strategies may have other purposes, such as, for example, having a winning strategy in a game-theoretic interaction between agents. Given all these useful purposes, different rule-based languages such as, for example, ELAN [68, 67], Maude [111, 112, 291], and Stratego [454], provide strategy languages to guide and control rule executions. The ELAN researchers deserve much credit as pioneers in this area for having made key contributions to rewriting strategy ideas from the beginning of the ELAN language.

For modularity and reasoning purposes it is very useful to keep a clear separation between the rewrite theory \( \mathcal{R} \) and the strategies used to control it. As discussed in Section 2.2, this was one of the key motivations of Tinelli in seeking formal specifications of SAT solvers by inference systems, so that the proof of correctness of a SAT solver is completely decoupled from its, possibly quite complex, heuristics. Following this point of view, a strategy language \( SL \) is understood in [291] as a theory transformation of the form:

\[
(\mathcal{R}, SM) \mapsto SL(\mathcal{R}, SM)
\]

where \( SM \) is a strategy module completely separated from the rewrite theory \( \mathcal{R} \), and \( SL(\mathcal{R}, SM) \) is a transformed rewrite theory which executes the rules in \( \mathcal{R} \) using the strategy expressions of \( SM \). Modularity and separation of concerns are thus achieved, because we can have different strategy modules, say, \( SM_1, \ldots, SM_n \), to control the executions of the same rewrite theory \( \mathcal{R} \) in different ways for different purposes. The fact that \( SL(\mathcal{R}, SM) \) is another rewrite theory means that the operational semantics of the strategy language \( SL \) is also defined by rewriting, as done, for example, in [68, 67, 112, 291]. But what is now rewritten is not just a term \( t \) in \( \mathcal{R} \), but a pair \( s @ t \), consisting of a strategy expression \( s \) in \( SM \) which is applied to a term \( t \) in \( \mathcal{R} \). What the term \( s @ t \) rewrites to are solutions (plus possibly pending strategy tasks); that is, terms \( t' \) in \( \mathcal{R} \) that are reachable from \( t \) when the rules in \( \mathcal{R} \) are applied according to the strategy \( s \). Therefore, one can also give to \( SL \) a more abstract set-theoretic semantics that assigns to \( s @ t \) the set of all its solutions, as done, for example, in [68, 67, 291].

Of course, the theory \( SL(\mathcal{R}, SM) \) manipulates or controls the theory \( \mathcal{R} \). It needs to know and handle notions such as term, subterm, rule, position, matching substitution, and so on. This makes an explicit use of reflection in the definition of \( SL(\mathcal{R}, SM) \) very natural, in the sense that \( SL(\mathcal{R}, SM) \) can be viewed as a rewrite theory that extends the universal theory \( \mathcal{U} \) with special combinators aimed at controlling the execution of \( \mathcal{R} \) at the metalevel. This has been the approach taken in Maude since its first strategy languages until its current one [111, 112, 291]. In this way, strategies are made internal to rewriting logic itself. There are of course various requirements that one would like a strategy language to satisfy, the most basic one being its soundness, i.e., only terms reachable from \( t \) in \( \mathcal{R} \) should be among the solutions of \( s @ t \). The paper [291] discusses several such requirements, emphasizing the fact that
the determinism of \( SL(R, SM) \) is a highly desirable feature: since we want to control the nondeterminism of \( R \), once we fix a strategy \( s \), the solutions of \( s @ t \) should not depend on how \( s @ t \) is executed in \( SL(R, SM) \), in the sense that any possible solution not yet seen should always be obtainable by further rewriting.

3.6. The \( \rho \)-Calculus

One of the attractive aspects of the \( \lambda \)-calculus is that it is very simple, both in its syntax and its rules, yet all of higher-order functional programming can be encoded in it, or in some variant of it such as a typed version. Couldn’t there be a similar calculus for rewriting? And could such a calculus be general enough as to naturally embed the \( \lambda \)-calculus as a sub-calculus? Horatiu Cirstea and Claude Kirchner both posed these intriguing questions and gave an elegant positive answer to them in their \( \rho \)-calculus [92, 93, 94]. The key idea is to replace the \( \lambda \)-abstraction operator \( \lambda x. u \) by a \( \rho \)-abstraction \( t \leftarrow u \), where the role of the bound variable \( x \) in \( \lambda x. u \) is now played by the bound term \( t \) in \( t \leftarrow u \). As in the \( \lambda \)-calculus, there is also an application operator \([\cdot]_\cdot\). The intended meaning of an application \([t \leftarrow u](v)\) is to rewrite the term \( v \) at the top with the rewrite rule \( t \rightarrow u \). The \( \lambda \)-calculus is then naturally encoded in the \( \rho \)-calculus as a special case. For example, the \( \lambda \)-term \( \lambda x. (y x) \) is encoded as the \( \rho \)-term \( x \leftarrow [y](x) \). The entire \( \rho \)-calculus is then described by a small set of evaluation rules; furthermore, such evaluation rules, particularly the Fire rule, can be made parametric to the matching algorithm employed, i.e., the \( \rho \)-calculus can express not only syntactic rewriting, but also rewriting modulo axioms such as associativity-commutativity. In similarity to the \( \lambda \)-calculus, there are also typed versions of the \( \rho \)-calculus [96, 276], and even a “\( \rho \)-cube” [95].

From the point of view of reflection, the \( \rho \)-calculus can be understood as a convenient simple calculus specifying a universal theory (modulo using an explicit substitution calculus such as, e.g., CINNI [416] to turn the \( \rho \)-calculus itself into a first-order rewrite theory). Indeed, it is shown in [93, 94] that the \( \rho \)-calculus can faithfully simulate at the metalevel the rewriting behavior of any other rewrite theory. Since, as pointed out in Section 3.5, from a reflective point of view a strategy language \( SL \) can be understood as the addition of appropriate strategy combinators to a universal theory \( U \), it is entirely natural to see that one of the important uses of the \( \rho \)-calculus has been to give a rewriting semantics at the metalevel to strategy languages such as ELAN, and that the \( \rho \)-calculus itself has been extended with such strategy combinators to become in effect a powerful strategy language [97].

3.7. Sufficient Completeness

Given a rewrite theory \( R = (\Sigma, E \cup B, R, \phi) \), with good executability conditions such as \( E \) being ground confluent and terminating modulo \( B \), and \( R \) being coherent with \( E \) modulo \( B \), we can represent its states uniquely up to \( B \)-equality as canonical forms \( \text{can}_{E/B}(t) \) with \( t \) a ground term. The equations \( E \) may define various auxiliary functions (for example, numerical functions), which operate on some parts of the state, that is, that manipulate elements of
the initial algebra $T_{Σ/E∪B}$. Therefore in $\text{can}_{E/B}(t)$ all such auxiliary functions should have already disappeared and only state constructors should remain. This is the (equational) sufficient completeness problem: given a subsignature $Ω ⊆ Σ$ of operators called constructors, is it the case that for any ground $Σ$-term $t$, the term $\text{can}_{E/B}(t)$ is an $Ω$-term? If this holds, $(Σ, E∪B)$ is called sufficiently complete with respect to the constructor subsignature $Ω$: if it fails to hold, this is clear indication that we have not given enough equations to define some auxiliary function $f ∈ Σ−Ω$, so that there is something wrong with the specification. For a rewrite theory $R = (Σ, E∪B, R)$ this means that there are extra states that we had not intended to have in our system and which are not built by the state constructors $Ω$ alone.

It is therefore important to check that an equational theory $(Σ, E∪B)$, or the equational part of a rewrite theory $R = (Σ, E∪B, R, φ)$, is sufficiently complete. When $B = ∅$, $Σ$ is unsorted, and the equations $E$ are unconditional, several algorithms to check sufficient completeness are known (see, e.g., [118] and references there). An attractive possibility is to further assume that the equations $E$ are left-linear (i.e., if $(t = t′) ∈ E$, then each variable $x$ in $t$ occurs at a single position $p$ of $t$), because then the problem can be reduced to an emptiness problem for tree automata (see [118]). In general, however, one would like to have sufficient completeness proof methods that can apply more broadly to: (i) order-sorted or even membership-equational signatures; (ii) modulo axioms $B$; and (iii) with $E$ containing conditional equations and even conditional memberships. In such a broad generality the problem becomes undecidable, but proof obligations can be generated. For example, the tool described in [225] addresses (i) and (iii) by providing a decision procedure to check the sufficient completeness of unconditional order-sorted equational theories without requiring left linearity, and generates proof obligations which are sent to the Maude Inductive Theorem Prover (ITP) (see Section 6.1.5), to prove sufficient completeness of order-sorted and membership-equational conditional specifications. Instead, the Maude Sufficient Completeness Checker tool (SCC) [229, 227] addresses (i) and (ii) by providing a decision procedure which can check sufficient completeness of order-sorted equational specifications modulo combinations of associativity and/or commutativity and/or identity axioms when the equations $E$ are unconditional and left-linear. The SCC tool reduces the problem to an emptiness problem for propositional tree automata [231], and uses the CETA library that efficiently implements tree automata operations for propositional tree automata [224]. As already mentioned, sufficient completeness for membership equational logic (MEL) is in general undecidable, but proof obligations can be generated. The MEL sufficient completeness problem has been studied in [69, 230, 224].

For a rewrite theory $R = (Σ, E∪B, R, φ)$ there are actually two different sufficient completeness problems. The first, of course, is the equational sufficient completeness of its equational part $(Σ, E∪B)$ relative to a constructor subsignature $Ω$ described above. The second problem is the sufficient completeness of the rules $R$. But what does that mean? If $(Σ, E∪B)$ is sufficiently complete in the equational sense, are not all states of $R$ already representable
as Ω-constructor terms of the form \( can_{E/B}(t) \)? Yes indeed, but what about the set of final states, that is, states for which it is not possible to perform any further transitions with \( R \)? They are in general a subset of all ground Ω-terms, so that they may be describable by an even smaller constructor subsignature \( \Lambda \subseteq \Omega \subseteq \Sigma \). By specifying \( \Lambda \), a user makes clear a set of state constructors that is enough to generate all such final states. What is then a failure of sufficient completeness for the rules \( R \)? What does it mean? It means exactly a violation of deadlock freedom. A deadlock is an unintended and unwanted final state. Lack of sufficient completeness for \( R \) means that there is a final state of \( \mathcal{R} \) which is not a \( \Lambda \)-term, that is, \( \mathcal{R} \) has a deadlock. Therefore, checking sufficient completeness of \( R \) means checking deadlock-freedom. This has been proposed by Camilo Rocha and I in [384], where we show that the same propositional tree automata techniques used to verify sufficient completeness for order-sorted equational specifications modulo axioms can be extended to check sufficient completeness of the rules \( R \) in \( \mathcal{R} \) under the assumption that they are unconditional, left-linear, and weakly terminating; we also extend the Maude SCC tool to also support such checking. For the case of rewrite theories of the form \( \mathcal{R} = (\Sigma, R) \), with \( \Sigma \) unsorted and \( R \) unconditional, a different method to check the sufficient completeness of \( R \) using narrowing techniques has been proposed by I. Gnaedig and H. Kirchner in [207].

3.8. Termination

Termination of a rewrite theory \( \mathcal{R} = (\Sigma, E \cup B, R, \phi) \) is a very important problem, and there is a rich body of termination techniques for term rewriting systems that can be used. However, the standard termination proof methods address the much simpler case of untyped rewrite theories of either the form \( \mathcal{R} = (\Sigma, \emptyset, R) \), or the from \( \mathcal{R} = (\Sigma, B, R) \) for some restricted set \( B \) of axioms, perhaps with some context-sensitive information \( \phi \). These standard methods are clearly insufficient for rewrite theories and need to be substantially generalized in several dimensions such as: (i) support for sorts, subsorts, and memberships; (ii) support for conditional rules with extra variables in their conditions in both \( E \) and \( R \); (iii) the existence, when \( E \) and \( R \) are conditional, of two separate rewrite relations \( \to_E \) and \( \to_R \) that cannot be easily combined into a single one; (iv) the need to support a wide range of equational axioms \( B \) containing at the very least any combination of associativity and/or commutativity and/or identity axioms. Furthermore, standard termination methods were developed in the context of equational logic and automated deduction and do not address important kinds of termination relevant for rewriting logic applications such as: (a) termination under fairness assumptions; (b) termination under strategies; and (c) probabilistic termination.

To address problems (i)–(iv) in the context of generalized rewrite theories \( \mathcal{R} = (\Sigma, E \cup B, R, \phi) \) whose equational part is a (possibly conditional) membership equational theory \( (\Sigma, E) \), the first thing to observe is that the “vanilla flavored” description of the computations by a single rewrite relation \( \to_R \), or even by two relations \( \to_E \) and \( \to_R \), is utterly inadequate, because the computation of the membership relations \( t : s \) is just as important and is entwined
with that of rewrites using $\rightarrow_E$ and $\rightarrow_R$. What one needs to make explicit is an inference system involving both rewrites (with $R$ and $E$) and memberships. This, in turn, poses the problem of conditional termination not in terms of a rewrite relation $\rightarrow_R$, but in terms of different logics with different inference systems. This has led to proposing the notion of operational termination in [150], not only for membership rewriting, but for logical inference systems in general. Although very general, this notion is also very practical, because it captures the idea of an interpreter carrying out the inference steps, so that operational termination means that such an interpreter will never loop. Even for the vanilla-flavored case of untyped conditional rewrite theories $\mathcal{R} = (\Sigma, \emptyset, R)$ this notion provides useful insights: as shown in [279], operational termination coincides there with the notion of quasi-decreasing conditional term rewriting systems, making it clear that other conditional rewrite systems, which are so dissant terminating, such as those enjoying “effective termination,” are not effective at all, since interpreters can loop on such systems. The relations of operational termination with other notions of conditional termination for untyped conditional term rewriting systems have been further investigated in [400].

Although the approach to the operational termination of membership rewrite theories in [150] already dealt with rewriting modulo axioms $B$, and was extended in [152] to deal simultaneously with the relations $\rightarrow_E$ and $\rightarrow_R$ plus the memberships $t : s$, there is great practical interest in being able to use existing state-of-the-art termination tools for term rewriting systems to prove the termination of generalized rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ beyond their scope. To bridge this gap, several important problems need to be solved. First, the rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, or even the membership equational theories $(\Sigma, E)$ need to be transformed into untyped vanilla-flavored term rewriting systems, eliminating features such as sorts, subsorts, memberships, and even conditions. This is accomplished in [150, 152] by appropriate non-termination preserving theory transformations. The second problems is that the sets of axioms $B$ for which proofs of termination modulo $B$ are supported in existing tools are quite restricted. To solve this problem, semantics-preserving theory transformations based on the notion of variant (see Section 3.3) that transform a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ into a semantically equivalent one $\tilde{\mathcal{R}} = (\Sigma, \tilde{E} \cup \tilde{D} \cup B_0, \tilde{R}, \tilde{\phi})$ with simpler axioms $B_0$, where $B = B_0 \cup D$, is presented in [153]. However, transformational methods come at a nontrivial cost, since the transformed theories are usually more complex. Therefore, more intrinsic proof methods to handle the above two problems are also of great interest. For example, in [282] the transformations in [150] are replaced by transformations into order-sorted rewrite theories, which still keep a lot of sort information, and in [280] dependency-pair-based methods are generalized from the unsorted to the order-sorted level. Similarly, in [10] intrinsic methods to prove termination modulo useful combinations of equational axioms by dependency pair techniques are proposed. Many of the above-mentioned techniques for proving termination of rewrite theories are already supported by the Maude Termination Tool (MTT), which I discuss in Section 6.1.3.
My current view is that the class of order-sorted rewrite theories of the form $\mathcal{R} = (\Sigma, B_0, R, \phi)$, where: (i) $B_0$ is the widest possible class of axioms for which dependency pair proof methods are available; and (ii) the rules $R$ are unconditional, is a good target class for which intrinsic methods should be further developed, since the transformations of general rewrite theories into that class become much simpler than the transformations into untyped rewrite theories, and therefore the proof methods will become considerably more effective in practice.

Another, orthogonal set of techniques that need to be further developed in order for termination proofs to scale up to large rewrite theories are modularity techniques that work at the richer level of at least order-sorted rewrite theories modulo axioms $B_0$. At the vanilla-flavored level of untyped rewrite theories of the form $\mathcal{R} = (\Sigma, \emptyset, R)$, there is already a substantial body of such techniques available (see, e.g., [346, 443]), and even some very useful work for untyped rewrite theories of the form $\mathcal{R} = (\Sigma, AC, R)$, with $AC$ associative-commutative axioms [284]. Felix Schernhammer and I have initiated the study of modularity techniques for the termination of unconditional order-sorted specifications modulo combinations of associativity and/or commutativity and/or identity axioms in [401].

All the termination techniques described above provide an important necessary core. However, this core is not sufficient to cover important applications. Suppose that our rewrite theory $\mathcal{R}$ specifies a communication protocol whose termination we want to prove. Very often $\mathcal{R}$ will not terminate in the standard sense, but will terminate under appropriate fairness assumptions. That is, infinite rewrite sequences do exist, but all such sequences are unfair and therefore unrealistic. Proof techniques for termination of rewrite theories under fairness assumptions have been studied in [281], substantially extending prior work in [373, 374]. Another way in which termination techniques need to be extended is to reason about termination of $\mathcal{R}$ when executed under a given strategy (see Section 3.5). This extension has been carried out in [193, 208] and is supported by the CARIBOO tool, which I discuss in Section 6.1.2. Yet another topic requiring a substantial extension of standard termination techniques is the termination of probabilistic rewriting, a topic investigated in [206] (for a discussion of probabilistic rewriting and the different notions that have been proposed see Section 3.10).

3.9. Real-Time Rewrite Theories

In many reactive and distributed systems, including, for example, schedulers, networks, and so-called cyber-physical systems, real-time properties are essential to their design and correctness. Therefore, the question of how systems with real-time features can be best specified, analyzed, and proved correct in the semantic framework of rewriting logic is an important one. This question has been investigated by several authors from two related perspectives. On the one hand, an extension of rewriting logic called timed rewriting logic has been investigated, and has been applied to several examples and specification languages [262, 355, 263, 415]. On the other hand, Peter Ölveczky and I found a simple
way to express real-time and hybrid system specifications directly in rewriting logic [356, 347, 357, 360]. Such specifications are called real-time rewrite theories and have rules of the form

\[
\{ t \} \xrightarrow{r} \{ t' \} \text{ if } C
\]

with \( r \) a term denoting the duration of the transition (where the time can be chosen to be either discrete or continuous), \( \{ t \} \) representing the whole state of a system, and \( C \) an equational condition. Peter ¨Olveczky and I showed that, by making the clock an explicit part of the state, these theories can be desugared into semantically equivalent ordinary rewrite theories [356, 347, 357]. That is, in the desugared version we can model the state of a real-time or hybrid system as a pair \( (\{ t \}, r_0) \), with \( \{ t \} \) the current state, and \( r_0 \) the current global clock time. Then the above rule becomes desugared as

\[
(\{ t \}, r_0) \longrightarrow (\{ t' \}, r_0 + r) \text{ if } C
\]

Rewrite rules can then be either instantaneous rules, that take no time and only change some part of the state \( t \), or tick rules, that advance the global time of the system according to some time expression \( r \) and may also change the state \( t \). By characterizing equationally the enabledness of each rule and using conditional rules and frozen operators [76], it is always possible to define tick rules so that instantaneous rules are always given higher priority; that is, so that a tick rule can never fire when an instantaneous rule is enabled [358]. When time is continuous, tick rules may be nondeterministic, in the sense that the time \( r \) advanced by the rule is not uniquely determined, but is instead a parametric expression (however, this time parameter is typically subjected to some equational condition \( C \)). In such cases, tick rules need a time sampling strategy to choose suitable values for time advance. Besides being able to show that a wide range of known real-time models (including, for example, timed automata, hybrid automata, timed Petri nets, and timed object-oriented systems) and of discrete or dense time values, can be naturally expressed in a direct way in rewriting logic (see [357]), an important advantage of the above approach is that one can use an existing implementation of rewriting logic to execute and formally analyze real-time specifications. Because of some technical subtleties, this seems difficult for the alternative of timed rewriting logic, although a mapping into the above framework does exist [357].

Of course, one would like to simulate and formally analyze real-time systems specified as real-time rewrite theories. The Real-Time Maude tool [347, 360] has been developed for this purpose (I further discuss Real-Time Maude in Section 6.1.8). In this way, a wide range of applications, including schedulers, networks, cyber-physical systems, and real-time programming and modeling languages, have been specified (I discuss such applications in Section 7.4), and have been formally analyzed by model checking their temporal logic properties (I discuss the model checking of temporal logic properties, including the model checking of such properties for real-time systems in Section 3.11.2).
3.10. Probabilistic Rewrite Theories

Many systems are probabilistic in nature. This can be due either to the uncertainty of the environment in which they must operate, such as message losses and other failures in an unreliable environment, or to the probabilistic nature of some of their algorithms, or to both. In general, particularly for distributed systems, both probabilistic and nondeterministic aspects may coexist, in the sense that different transitions may take place nondeterministically, but the outcomes of some of those transitions may be probabilistic in nature. To specify systems of this kind, rewrite theories have been generalized to probabilistic rewrite theories in [265, 266, 6]. Rules in such theories are probabilistic rewrite rules of the form

\[ l : t(\vec{x}) \rightarrow t'(\vec{x}, \vec{y}) \text{ if } \text{cond}(\vec{x}) \text{ with probability } \vec{y} := \pi_{r}(\vec{x}) \]

where the first thing to observe is that the term \( t' \) has new variables \( \vec{y} \) disjoint from the variables \( \vec{x} \) appearing in \( t \). Therefore, such a rule is nondeterministic; that is, the fact that we have a matching substitution \( \theta \) such that \( \theta(\text{cond}) \) holds does not uniquely determine the next state fragment: there can be many different choices for the next state, depending on how we instantiate the extra variables \( \vec{y} \) in \( t' \). In fact, we can denote the different such next states by expressions of the form \( t'(\theta(\vec{x}), \rho(\vec{y})) \), where \( \theta \) is fixed as the given matching substitution, but \( \rho \) ranges along all the possible substitutions for the new variables \( \vec{y} \). The probabilistic nature of the rule is expressed by the notation: with probability \( \vec{y} := \pi_{r}(\vec{x}) \), where \( \pi_{r}(\vec{x}) \) is a probability distribution which may depend on the matching substitution \( \theta \). We then choose the values for \( \vec{y} \), that is, the substitution \( \rho \), probabilistically according to the distribution \( \pi_{r}(\theta(\vec{x})) \).

The fact that the probability distribution may depend on the substitution \( \theta \) can be illustrated by means of a simple example. Consider a battery-operated clock. We may represent the state of the clock as a term \( \text{clock}(T, C) \), with \( T \) a natural number denoting the time, and \( C \) a positive real denoting the amount of battery charge. Each time the clock ticks, the time is increased by one unit, and the battery charge slightly decreases; however, the lower the battery charge, the greater the chance that the clock will stop, going into a state of the form \( \text{broken}(T, C') \). We can model this system in PMaude notation (see Section 6.1.9) by means of the probabilistic rewrite rule

\[
\text{rl [tick]: } \text{clock}(T, C) \Rightarrow \begin{cases} 
\text{if } B \text{ then } \text{clock}(s(T), C - (C / 1000)) \\
\text{else } \text{broken}(T, C - (C / 1000)) 
\end{cases} \text{ with probability } B := \text{BERNOULLI}(C / 1000) .
\]

that is, the probability of the clock breaking down instead of ticking normally depends on the battery charge, which is here represented by the battery-dependent bias of the coin in a Bernoulli trial. Note that here the new variable on the rule's righthand side is the Boolean variable \( B \), corresponding to the result of tossing the biased coin. As shown in [265], probabilistic rewrite theories can
express a wide range of models of probabilistic systems, including continuous-time Markov chains [423], probabilistic non-deterministic systems [375, 403], and generalized semi-Markov processes [205]; they can also naturally express probabilistic object-based distributed systems [266, 6], including real-time ones. Yet another class of probabilistic models that can be simulated by probabilistic rewrite theories is the class of object-based stochastic hybrid systems discussed in [324].

A completely different notion of probabilistic rewriting has been proposed in [73, 71]. The key idea in both of these papers is that the rewrite rules themselves, \( r : t \to t' \), are still deterministic (the lefthand side \( t' \) has no extra variables); what is probabilistic is the choice of which rule to apply and where. In [73] it is shown how such choices can be defined in quite sophisticated ways by probabilistic ELAN strategies to model, for example, probabilistic algorithms; and in [71] ordinary deterministic rewrite rules are endowed with weights to achieve a notion of probabilistic rewrite system. A good way to understand how the ideas in [73, 71] are different from those in [265, 266, 6] is to observe that in a rewrite theory \( \mathcal{R} \) there are two completely different potential sources of nondeterminism: (i) the choice of which rule to apply at any given moment and where to apply it; and (ii) once a choice of rule, term position and matching substitution has been made, if the rule \( r : t(\vec{x}) \to t'(\vec{x}, \vec{y}) \) has extra variables \( \vec{y} \) on its righthand side, the choice of a ground substitution \( \rho \) to instantiate the variables \( \vec{y} \). The semantics in [73, 71] makes the choice (i) probabilistic while keeping the rules themselves deterministic; while the semantics in [265, 266, 6] keeps the choice (i) non-deterministic while making the instantiation of non-deterministic rewrite rules governed by probability distributions that are parametric on the lefthand side’s matching substitution. A final observation to make is that the existence of nondeterminism in the choice (i) of which transition to fire and where, with the transitions themselves being probabilistic in their outcome, is well-known in the modeling of probabilistic systems, e.g., in probabilistic non-deterministic systems [375, 403]; and in the probabilistic model checking of such systems, which introduces the notion of a scheduler to eliminate the nondeterminism in the choice of transitions, and then model checks the system considering all such possible schedulers.

It is highly desirable to be able to specify, simulate and analyze probabilistic systems specified as probabilistic rewrite theories. The PMaude language design [6] has exactly this purpose; I further discuss PMaude in Section 6.1.9. The kinds of formal analyses possible go beyond simulations and include statistical model checking with respect to properties expressed in either a probabilistic temporal logic or even a quantitative probabilistic temporal logic where the result of evaluating a formula on a path is a real number corresponding to some quantity associated to a system behavior; I discuss probabilistic temporal logics and model checking of probabilistic properties in Section 3.11. Many applications to probabilistic systems are thus made possible; I discuss some of them in Section 7.5.
3.11. Temporal Logic Properties

As already observed at the end of Section 3.1.2, the reachability initial model of a rewrite theory \( R = (\Sigma, E, R, \phi) \) has an associated one-step rewrite relation \( [t] \rightarrow^1_R [t'] \) relating the states, i.e., the \( E \)-equivalence classes \([t]\) of ground \( \Sigma \)-terms \( t \). Since \( R \) can have different sorts and kinds, we should furthermore specify which is the preferred kind of states, so that terms of other kinds describe state fragments, or data components of the state, but not an entire state of our system. Let us call \([\text{State}]\) to such a kind. Then we can associate to \( R \) a transition system, namely, the pair \((T_{\Sigma/E[\text{State}]}, \rightarrow^1_R)\) where \( T_{\Sigma/E[\text{State}]} \) denotes the set of \( E \)-equivalence classes \([t]\) of ground \( \Sigma \)-terms \( t \) of kind \([\text{State}]\).

Without loss of generality we may also assume that the equations \( E \) already define a desired collection of state predicates (if they do not, we can just add new function symbols and equations defining such state predicates as Boolean-valued functions). That is to say, we can associate to \( R \) not just a transition system \((T_{\Sigma/E[\text{State}]}, \rightarrow^1_R)\), but in fact a Kripke structure\(^6\) \((T_{\Sigma/E[\text{State}]}, \rightarrow^1_R, L_R)\), where \( L_R \) is a labeling function, associating to each state predicate \( p \) the set of all states where \( p \) holds.

All this means that, since rewrite theories model concurrent systems and we can naturally associate to them Kripke structures, their temporal logic properties can then be defined semantically in terms of such Kripke structures (or for real-time or probabilistic rewrite theories the analogue real-time or probabilistic transition systems). For expressing such properties, suitable temporal logics can be used. Then, both model checking, or theorem proving, or a combination of both approaches, can be used to verify that a rewrite theory (more precisely, its reachability initial model) satisfies some desired temporal logic properties.

3.11.1. Temporal Logics

Which temporal logic is best suited for specifying which properties of a rewrite theory is itself a very good question. Here are several choices with specific advantages.

**State-Based Logics.** There are many choices. The most common is CTL* [98], or one of its subsets such as CTL or LTL. These logics are well suited for properties based on state predicates; but not well suited for properties based on events, which need to be encoded unnaturally in the state itself to be expressible.

**TLR and Parameterized Fairness.** To avoid the limitations of state-based logics in expressing events, while keeping all their good state-based features; and to take advantage of the expressive power of rewrite theories in expressing parameterized events by rewrite rules, and spatial information by term patterns,
the temporal logic of rewriting TLR [314] can be used. TLR is a simple extension of CTL* where just one more construct is added to the syntax of formulas, namely, spatial action patterns. The simplest such patterns are just labels of rewrite rules, stating that a transition event with a rule having that label has taken place. For example, for the object-based system of Section 2.1, we can state the liveness property that each message send is always eventually followed by a receive event by the (implicitly universally path quantified) TLR formula □(send → ○receive). However, more complex patterns are possible taking advantage of both the parametric nature of rewrite rules (whose parameters are the mathematical variables of each rule) and the context where the rewrite takes place. For example, we can localize the above property both to sender object ‘b and its associated receiver object ‘d by the formula □(send(‘b) → ○receive(‘d)). It is also very easy to express localized (that is, parameterized) fairness conditions as universally quantified TLR properties. For example, the (weak) object fairness of the receive and write actions needed for a realistic modeling of the object-based system of Section 2.1 when sensor objects are added, as explained in Section 3.5, can be succinctly captured by the TLR formulas (∀x : Oid) □receive.enabled(x) → □ ○ receive(x), and (∀x : Oid) □write.enabled(x) → □ ○ write(x), where receive.enabled(x) and write.enabled(x) are the obvious state predicates stating that the object x can perform the receive, resp., write action. Of course, the reachability initial model of a rewrite theory R and its associated Kripke structure (TΣ/E[State], →R, L_R) throw away all information about actions and therefore cannot be used to give semantics to TLR. We need to use the initial model TR of R and its associated labeled Kripke structure, where labeled transitions are of the form $[t][\alpha] \rightarrow R [t']$, with $\alpha$ a one-step proof term [314].

**Metric Temporal Logic and TCTL.** For real-time systems, standard temporal logics, although able to express many useful properties (particularly when the state predicates refer to timers or even to the global clock), are not expressive enough: one often wants to express the requirement that a certain property must hold within certain time bounds. Various temporal logics for real-time systems can be used. A simple possibility is to use the metric temporal logic MTL [264], which extends LTL to timed paths by qualifying LTL’s until operator $\mathcal{U}$ with a time interval $[t, r]$. The meaning of a formula $\varphi \mathcal{U}_{[t, r]} \psi$ is then that $\varphi \mathcal{U} \psi$ holds in the standard LTL sense and, furthermore, $\psi$ must hold at a time $t' \in [t, r]$, and $\varphi$ must continuously hold until time $t'$. Instead, **Timed CTL (TCTL)** [25] extends CTL by qualifying the until operator $\mathcal{U}$ with a time bound $t$ plus an indication of whether the second formula must hold before, after, or exactly at time $t$, that is, we have formulas of the form $\varphi \mathcal{U}_{[t, b]} \psi$, where $b \in \{\geq, >, \leq, <, =\}$, with the expected meaning. For example, $\varphi \mathcal{U}_{\geq t} \psi$ is equivalent to $\varphi \mathcal{U}_{[t, +\infty)} \psi$ in an interval formulation.

**PCTL, CSL, and QuaTEX.** For probabilistic systems, temporal logics that extend standard ones are also needed. One well-known such logic is *Probabilistic*
CTL (PCTL) [220]. The basic idea is that sets of computation paths in a probabilistic system have probability measures associated to them, and we can qualify temporal logic formulas by requiring that the set of paths satisfying a certain formula has a probability greater (resp., smaller) or equal to a certain $p \in [0,1]$. For example, the PCTL formula $P > 0.7(\varphi U \psi)$ states that the set of paths where $\varphi U \psi$ holds has a probability measure greater or equal to 0.7.

Since many probabilistic systems are also real-time systems, for such systems there is also a need to have temporal logics which combine both probabilistic and time-bounded features. Continuous Stochastic Logic (CSL) [1, 41] is one such logic extending PCTL by qualifying temporal logic operators by a time bound. For example, the formula $P \geq 0.7 (\varphi U \leq 3.2 \psi)$ states that the set of paths where $\varphi U \psi$ holds and, furthermore, $\psi$ holds at a time $t \in [0,3.2]$, and $\varphi$ holds continuously until time $t$, has a probability measure greater or equal to 0.7.

In the analysis of probabilistic systems we are often interested not just in the probabilities associated to the satisfaction of certain temporal logic formulas, but in quantitative properties such as, for example, the expected latency of a communication protocol when hardened against DoS attacks under specific assumptions about the attacker and the network. Such a latency is not a probability but a real number. To be able to express such quantitative properties, PCTL and CSL have been generalized to a logic of Quantitative Temporal Expressions (QuaTEx) in [6]. The key idea is to generalize state formulas and path formulas to real-valued state expressions and path expressions, where the appropriate real-valued functions can be defined by the user, just as the appropriate state predicates are defined by the user in standard temporal logics. Boolean-valued and probability-valued formulas are now regarded as special cases of real-valued QuaTEx formulas by using the subset containments $\{0,1\} \subset [0,1] \subset \mathbb{R}$. For example, Boolean-valued CSL formulas such as $P \geq 0.7 (\varphi U \leq 3.2 \psi)$ are also expressible in QuaTEx, but QuaTEx can express properties beyond CSL [6].

3.11.2. Model Checking Verification of Rewrite Theories

Model Checking of State-Based Temporal Properties. The simplest, yet very useful, form of model checking analysis of rewrite theories is the verification of invariants. As usual in model checking, what we search for is the violation of a property, in this case the invariant. An invariant $I$ is a Boolean-valued state predicate, so we can express a search for its violation as a search for a proof of the existential formula

$$(\exists x : [\text{State}]) \ (\text{init} \rightarrow x \land I(x) = \text{false})$$

where $\text{init}$ is the initial state, and $[\text{State}]$ is our chosen kind of states. If the number of states reachable from $\text{init}$ is finite, breadth first search is a complete model checking procedure to verify the invariant. If the number of states reachable from $\text{init}$ is infinite, breadth first search still gives us a semidecision procedure to check the failure of the invariant: if $I$ fails, we are guaranteed to find a counterexample in finite time.

More generally, we can model check properties in state-based temporal logics such as CTL, LTL, or CTL$^*$ using the model checking algorithms described in
by using the Kripke structure \((T_{\Sigma/E_{\text{state}}} \to^1_R, L_R)\) associated to the given rewrite theory \(R\), provided the number of states reachable from the given initial state \(\text{init}\) is finite.

Model Checking of TLR Properties. To verify TLR properties on a rewrite theory \(R\), assuming again that the number of states reachable from the given initial state \(\text{init}\) is finite, we have two different possibilities: (i) to transform \(R\) and the property \(\varphi\) into a new rewrite theory \(\tilde{R}\) and a CTL* formula \(\tilde{\varphi}\) and then model check \(\tilde{R}, \text{init} \models \tilde{\varphi}\) as described in [314] and implemented in Maude in [35] for the linear time temporal logic fragment LTLR; or (ii) to use a more efficient algorithm that can directly verify LTLR formulas on a rewrite theory \(R\) on the fly, as the one developed and implemented in the Maude system in [36]. One of the good features of TLR is that it is very easy to express fairness assumptions in it [314], so a first approach to the verification of a TLR property \(\psi\) under fairness assumptions \(\varphi\) is to verify the implication \(\varphi \rightarrow \psi\). However, this suffers from two major drawbacks: (i) in a logic like LTL the Büchi automaton associated to \(\varphi \rightarrow \psi\) grows exponentially with the size of the formula; and since \(\varphi\) typically contains several fairness formulas and can be relatively complex, we can easily hit severe performance barriers; and (ii) to make things worse, the approach of model checking \(\varphi \rightarrow \psi\) has no reasonable way of dealing with localized fairness formulas which are parametric, i.e., what we have is not a propositional formula \(\varphi\), but a universally quantified first-order formula \((\forall x) \varphi(x)\). For example, \((\forall x) \varphi(x)\) may express an object fairness assumption in a system with dynamic object creation. Even if we could predict the set \(O\) of all such objects, which may not be possible unless we explore the entire state space, the only way to encode this directly at the propositional level would be as a conjunction \(\bigwedge_{o \in O} \varphi(o)\), something quite unfeasible to model check in practice because of the typically huge size of the corresponding Büchi automaton. For these reasons, Kyungmin Bae and I have developed a completely new model checking algorithm for LTLR which can model check LTLR formulas under parametric fairness assumptions of the form \((\forall x) \varphi(x)\). The algorithm and its Maude implementation are described in [37].

An interesting, additional aspect of LTLR model checking is its use as a strategy language. Since TLR formulas contain action patterns corresponding to how rules are applied, with which substitutions, and where in the state, and describe complex behaviors involving such elementary actions and tests expressed by state predicates, a TLR path formula \(\varphi\) can be naturally understood as a strategy expression, which defines a corresponding set of computations in the given rewrite theory \(R\). Assuming that \(\varphi\) does not contain any path quantifiers, we can use a LTLR model checker to generate a behavior for the strategy expression \(\varphi\) by giving to the model checker the LTLR state formula \(\forall \neg \varphi\). If the strategy expression \(\varphi\) can be realized by a concrete behavior, the LTLR model checker will provide such a behavior as a counterexample for \(\forall \neg \varphi\), that is, as a constructive proof of the existentially path quantified TLR state formula \(\exists \varphi\).
Narrowing-Based Symbolic Model Checking of Rewrite Theories. One important limitation of standard model checking algorithms such as those described in [98] is that they work under the assumption that the set of states reachable from the initial state is finite. There are several ways to avoid this limitation: (i) to use deductive methods as those I discuss in Section 3.11.3; (ii) to use some kind of abstraction or simulation that transforms the system into a finite-state one (I discuss this in Section 3.12); and (iii) to use a model checking approach that does not require the system to be finite-state. Regarding approaches of type (iii), Section 3.3 has explained how narrowing can be used as a complete symbolic reachability analysis method to model check the failure of an invariant for a possibly infinite-state rewrite theory $R$. This is of course a very different notion of “symbolic model checking” than the usual one based on BDDs, which uses the representation of a finite set of states as a propositional formula assuming a finite state space. But Section 3.3 dealt only with reachability and invariants. What about other temporal logic properties? In [180] Santiago Escobar and I show how the same narrowing approach can be extended to model check $ACTL^*$ properties of a possibly infinite system specified as a toposmost rewrite theory $R$.

Model Checking of Real-Time Rewrite Theories. The simplest models of real-time systems are timed automata [26], whose $TCTL$ properties are decidable by model checking [25]. The paper [49] shows how timed automata model checking can be expressed as a symbolic procedure using appropriate strategies in the ELAN rewriting logic language. Timed automata can be seen as very simple real-time rewrite theories [357], but their simplicity also involves a severe limitation: they are finite-state systems. Even a relatively simple system such as a scheduler whose state includes unbounded queues cannot be modeled by a timed automaton [353]. What real-time rewrite theories offer is a more expressive high-level way of specifying many real-time systems of interest, such as network protocols and distributed object systems, whose states are in principle unbounded and often contain complex data structures. The challenge is to identify temporal logic properties and conditions on the real-time rewrite theory that make the verification of such properties decidable by model checking. A very broad class of real-time rewrite theories (whose time may be continuous) has been identified in [359], where it is shown that the following temporal logic properties are decidable for such systems: (i) time-bounded $LTL/\Box$ formulas of the form $\varphi$ in time $r$, where $\varphi$ is an $LTL$ formula and $r$ is a time bound (for a detailed explanation of the semantics of such formulas see [360]); and (ii) $LTL/\Box$ formulas whose state predicates do not refer to the global clock, provided the set of discrete states reachable from the initial state is finite. Recall that a state of a system specified by a real-time rewrite theory is a pair $\{t\},r$, with $\{t\}$ a ground term describing the global state and $r$ a (possibly continuous) clock value. By the “discrete state” I mean the global state $\{t\}$. Formulas of types (i) or (ii) can already express many properties of practical interest, but formalisms such as $MTL$ and $TCTL$ are obviously more expressive. More recent work has developed two new model checking algorithms for real-time rewrite theories. In [272], a model checking algorithm to verify properties in a subset of $MTL$ for
object-oriented real-time rewrite theories whose state is a multiset of objects and messages is presented; and [271] presents an algorithm to model check real-time rewrite theories for the satisfaction of TCTL formulas, except for formulas of the form $\varphi U t \psi$. In Section 6.1.8 I discuss the Real-Time Maude tool, which supports all the model checking procedures mentioned above; and in Section 7.4 I discuss many real-time system applications that have been specified and analyzed in Real-Time Maude.

**Statistical Model Checking of Probabilistic Rewrite Theories.** Temporal logic properties of a probabilistic system can be model checked either by exact model checking algorithms, or in an approximate, but more scalable and more widely applicable way, by statistical model checking (see, e.g., [404, 461, 6]). The idea of statistical model checking is to verify the satisfaction of a temporal logic property by statistical methods up to a user-specified level of statistical confidence. For this, a large enough number of Monte-Carlo simulations of the system are performed, and the formula is evaluated on each of the simulations.

Recall the discussion in Section 3.10 about how a probabilistic rewrite theory in general has a nondeterministic aspect corresponding to the choice of which probabilistic transition to fire. One important requirement of statistical model checking algorithms is that they assume that the system is purely probabilistic: there is no nondeterminism in the choice of transitions. This seems like a strong requirement. However, using the methodology presented in [6], a wide class of object-oriented probabilistic real-time rewrite theories specifying many concurrent, actor-based systems of interest can be expressed so that no nondeterminism is involved in the application of rewrite rules. The key idea is to take advantage of three facts: (i) time is continuous; (ii) the probability distributions governing message arrival latencies are also continuous; and (iii) the probability that two messages will arrive to any two objects at the same time is then zero. Since the rewrite rules specify how an actor changes state when it receives a message, and at each instant in time at most one message has arrived to at most one object, there is at most one rewrite rule that can be applied at each continuous instant and all nondeterminism disappears.

Properties expressed in either CSL or QuaTEx can then be statistically model checked for such probabilistic real-time rewrite theories, using the algorithms presented in, respectively, [404] and [6]. Furthermore, as shown in [23], the above algorithms are naturally parallelizable and can scale up very well using such parallelization. A related algorithm for statistical model checking of quantitative properties is presented in [250]. In Section 6.1.10 I discuss how the VeStA and PVeStA tools support the statistical model checking of CSL and QuaTEx properties for the above-mentioned class of probabilistic rewrite theories; and in Section 7.5 I discuss various applications that have been specified and analyzed this way.

**3.11.3. Deductive Verification of Rewrite Theories**

Model checking, while extremely useful, is not sufficient for all verification purposes. This is clear from the fact that satisfaction of properties is in general
undecidable, from the infinite-state nature of many systems, and, even when a system is finite-state for each initial state, from the fact that in general there may be an infinite number of initial states. Furthermore, even if we succeed in reducing the verification problem to a finite-state model checking problem by the use of an abstraction as discussed in Section 3.12, deduction still plays a fundamental role in verifying the correctness of such an abstraction. The late Amir Pnueli expressed the situation succinctly in his motto “Deduction is Forever” [372].

Given a rewrite theory \( \mathcal{R} \) (resp. a parameterized\(^7\) rewrite theory \( \mathcal{R}[P] \) with \( P \) its parameter theory), there are different kinds of properties that one may want to verify deductively about its initial model \( \mathcal{I}_\mathcal{R} \), or the Kripke structure associated to its initial reachability model (resp. the free models of \( \mathcal{R}[P] \) or their associated Kripke structures). Properties we may want to verify include: (i) temporal logic properties; (ii) inductive properties about the rewrite relation itself; and (iii) inductive equational properties about the states of \( \mathcal{R} \). The termination methods for rewrite theories discussed in Section 3.8 can be naturally regarded as proof methods for a particular kind of type (i) property.

Regarding deductive verification of temporal logic (type (i)) properties, the general idea is to use a sound and relatively complete proof system for a temporal logic to get rid of the temporal logic operators as much as possible and try to reduce the proof task to the verification of proof obligations of type (iii). The term “relatively complete” expresses the fact that the original temporal logic property holds for the given model iff the proof obligations of type (iii) generated by the inference system do; but since these are inductive proof obligations, a complete proof system for properties of type (iii) does not exist in general. A good example of a sound and relatively complete deductive proof system for \( CTL^* \) is the one proposed by Gabbay and Pnueli in [198]. An important remaining problem in using a deductive system of this kind is how to deal with the resulting proof obligations of type (iii). In this regard, rewrite theories are particularly attractive, because there is a rich body of inductive proof methods for equational logic which can then be used to discharge such proof obligations. For example, for Maude specifications one can use various formal tools described in Section 6.1 for this purpose.

For rewrite theories, this approach to the verification of type (i) properties has so far focused mostly on safety properties, including invariants. For the deductive proof of invariants there is a rich body of work, including several substantial case studies, using proof scores in CafeOBJ to verify invariants of observational transition systems (OTSs) (see, e.g., [345, 196]). The CafeOBJ

---

\(^7\)A parameterized rewrite theory \( \mathcal{R}[P] \) can be understood as a theory inclusion \( P \hookrightarrow \mathcal{R} \) of the parameter theory \( P \) into the “body” \( \mathcal{R} \) and specifies a parametric family of concurrent systems. \( \mathcal{R}[P] \) can then be instantiated by views, i.e., theory interpretations \( V : P \rightarrow Q \), by the usual “pushout construction.” Semantically, what is used is the fact that rewriting logic is a “liberal institution,” i.e., that it has not only initial models, but also free models along theory interpretations. For the treatment of parameterized rewrite theories in Maude see [103, Section 8.3].
researchers have also shown how deductive verification of invariants for an OTS can be combined with model checking verification of the rewrite theory associated to the OTS, or an abstraction of it [462, 196]. Another approach to invariant and temporal logic verification which can be viewed as both deductive and algorithmic is the narrowing-based reachability analysis method already discussed in Sections 3.3 and 3.11.2. Rusu and Clavel [397], and Rusu [396], present a different approach to invariant verification that reduces the problem to a type (iii) proof task by associating to a rewrite theory \( R \) a corresponding membership equational theory \( \mathcal{M}(\mathcal{R}) \) with a sort \( \text{Reachable} \) of reachable states characterized by appropriate membership predicates. In a sense, this can be seen as using an enrichment of the characterization of the initial reachability model of \( \mathcal{R} \) as the initial model of a membership equational theory given in [77] and discussed in Section 3.1.2. Camilo Rocha and I have presented a different approach to the verification of safety properties in [385]. The basic idea is to use narrowing-based proof methods to reduce the proof of: (a) invariants, (b) stability properties of the form \( P \Rightarrow \Box P \), and (c) strengthenings of invariants, to proof obligations of type (iii); and to then discharge many such proof obligations automatically, so that a considerably smaller set of proof obligations is left for an inductive theorem prover.

Finally, Camilo Rocha and I have initiated a study of constructor-based proof methods for inductive properties about the rewrite relation of the initial reachability model of a rewrite theory \( \mathcal{R} \) (type (ii) properties) in [384]. That is, we want to prove that the initial reachability model of \( \mathcal{R} \) satisfies some property of the form \( (\forall \bar{x}) t \rightarrow t' \), which is equivalent to proving \( \mathcal{R} \vdash \theta(t) \rightarrow \theta(t') \) for all ground substitutions \( \theta \). A related task is to prove that the initial reachability model of \( \mathcal{R} \) satisfies inductive joinability properties of the form \( (\forall \bar{x}) t \downarrow t' \), stating that all ground instances of \( t \) and \( t' \) can be rewritten to a common term. The key idea is that, the same way that equational constructors are crucial for proving inductive equalities \( t = t' \), both equational constructors for \( (\Sigma, E \cup B) \), and constructors for \( R \) (as discussed in Section 3.7) are crucial for proving inductive properties of the form \( (\forall \bar{x}) t \rightarrow t' \) for a rewrite theory \( \mathcal{R} = (\Sigma, E \cup B, R, \phi) \).

### 3.12. Simulation and Abstraction

As already mentioned, the application of standard model checking methods to the verification of a temporal logic property \( \varphi \) by (the initial model of) a rewrite theory \( \mathcal{R} \) may be hindered by \( \mathcal{R} \) being infinite-state. Even if \( \mathcal{R} \) is finite-state, the huge size of its state space may still make it unfeasible to model check such a property. Under such circumstances a very useful approach is to find a different rewrite theory \( \mathcal{R} \) which has a much smaller (and finite) state space than \( \mathcal{R} \), to verify \( \varphi \) for \( \mathcal{R} \), and to show that we have an implication

\[
\mathcal{R}, \text{init} \models \varphi \Rightarrow \mathcal{R}, \text{init} \models \varphi.
\]

As shown in, e.g., [98, 283, 321], this can be done if we can relate the sets of states of \( \mathcal{R} \) and \( \mathcal{R} \) and the initial states \( \text{init} \) and \( \text{init} \) by a binary relation
Given a rewrite theory $R = (\Sigma, E \cup B, R, \phi)$, a very simple, yet powerful, approach to obtaining such a theory $\hat{R}$ is to realize that rewriting logic comes with a built-in “abstraction dial” which allows us to turn some rewrite rules in $R$ into equations that can be removed from $R$ and added to $E$. That is, we can decompose $R$ into a disjoint union $R = G \cup R_0$ and define $\hat{R} = (\Sigma, E \cup G \cup B, R_0, \phi)$, where $G$ denoted the set of equations associated to the rules $G$. A good example of the use of such an abstraction dial is the DPLL module in Section 2.2, where $G$ consisted of the subsume, resolve, assert, and close rules. Of course, for the use of this abstraction dial to be natural, the rules $G$ should be deterministic in nature, so that the equations $E \cup G$ are still ground confluent and terminating modulo $B$. But in order for the Kripke structure associated to $\hat{R}$ to be computable (an essential requirement for model checking it) we also need $R_0$ to be coherent with $E \cup G$ modulo $B$. If these two conditions are satisfied, and, furthermore, the rules $G$ preserve all the state predicates in $\phi$, Azadeh Farzan and I proved in [186] that the quotient $\Sigma$-homomorphism $q : T_{\Sigma/E \cup B} \rightarrow T_{\Sigma/E \cup G \cup B}$ defines a stuttering bisimulation, so that for any $\varphi \in ACTL^*/\Box$ we have the equivalence $\hat{\varphi}, init \models \varphi \iff \hat{\varphi}, init \models \varphi$, where $init = q(init)$.

If the theory $\hat{R}$ thus obtained by turning the abstraction dial as much as possible is still too big to be model checked, a second, also very useful approach is to further collapse the set of states by an equational abstraction. Given a rewrite theory $R = (\Sigma, E \cup B, R, \phi)$ and a set $G \cup B'$ of $\Sigma$-equations, we can collapse $R$ into the rewrite theory $R/G \cup B' = (\Sigma, E \cup G \cup B \cup B', R, \phi)$ which has a much smaller state space than $R$. Again, we need the equations $G \cup B'$ to preserve the state predicates appearing in the formula $\varphi$ we want to model check; and we need $R/G \cup B'$ itself to yield a computable Kripke structure, i.e., $E \cup G$ should be ground confluent and terminating modulo $B \cup B'$, and $R$ should be coherent with $E \cup G$ modulo $B \cup B'$. Under these conditions, Miguel Palomino, Narciso Martí-Oliet and I proved in [320] that the quotient $\Sigma$-homomorphism $q : T_{\Sigma/E \cup B} \rightarrow T_{\Sigma/E \cup G \cup B}$ defines a simulation, so that for any $\varphi \in ACTL^*$ we have the implication $R/G \cup B', q(init) \models \varphi \implies \hat{R}, init \models \varphi$.

In the two methods just discussed for collapsing the state space of a rewrite theory $R = (\Sigma, E \cup B, R, \phi)$, the signature $\Sigma$ did not change at all: we just added some more equations to its equational part. But this is not a necessary requirement: our more abstract rewrite theory $\hat{R}$ may be based on a different signature $\Sigma'$, so that it is of the form $\hat{R} = (\Sigma', E' \cup B', R', \phi')$. All we need is to find an appropriate simulation relation $\hat{H}$ between $R$ and $\hat{R}$. Several methods for finding such simulation, or stuttering simulation, relations are presented in [290, 321] under the general banner of “algebraic simulations.” The general idea is to use algebraic and/or rewriting logic methods to define such an $\hat{H}$ as either
a function or a relation. Another idea explored in depth in [365, 321] is that simulations and stuttering simulations are arrows in appropriate categories, so that they can be composed, i.e., the entire approach is compositional, so that we can combine several of the above-mentioned abstraction methods to arrive at the desired abstraction. A general emphasis common to all the abstraction methods presented in [186, 320, 290, 321] is on the inductive proof obligations that need to be discharged in order to prove that the proposed simulation $H$ is correct. That is, although $H$ is used to verify a property by model checking, the correctness of the verification requires the interplay between model checking and inductive theorem proving: deduction is forever!

Another stuttering-simulation-based method frequently used to reduce the state space is partial order reduction (POR). The general idea is that a concurrent system can have a huge number of states due to the many different interleavings involved; however, many concurrent transitions are independent, in the sense that they can be interleaved with each other in arbitrary order without affecting the resulting state. This leads to the idea of cutting down the number of interleavings by only considering a subset of the computations involving independent transitions (see [98] for a detailed discussion). To support POR at the level of rewrite theories, Azadeh Farzan and I proposed in [187] a general theory transformation mapping rewrite theories of a certain type into their corresponding POR versions. In particular we showed how this transformation can be applied as a generic method to model check programs much more efficiently in a wide range of concurrent programming languages whose semantics has been defined in rewriting logic by the methods outlined in Section 4.3.

In Section 3.11.2 I explained how for topmost rewrite theories $ACTL^*$ properties can be model checked symbolically by narrowing by the methods presented in [180]. The reason why the $CTL^*$ property must be in the universal fragment $ACTL^*$ is precisely that what is used is a simulation relating the ground term instances of a term to such a term. That is, the original system we want to verify is the Kripke structure associated to the given rewrite theory $R$, whose states are $E \cup B$-equivalence classes of ground terms; but we simulate it symbolically by another Kripke structure where the states are terms with variables. The abstraction relation $H$ is precisely the “being an instance of modulo $E \cup B$” relation, denoted $\preceq_{E \cup B}$, where $E \cup B$ are the equations in $R$. Given a ground term $t$ and a term $t'$, $t \preceq_{E \cup B} t'$ holds iff there is a substitution $\sigma$ such that $t = E \cup B \sigma(t')$. Since the transition system defined on terms with variables by the narrowing relation $\leadsto_{R, E \cup B}$ in general has still an infinite number of states reachable from a symbolic initial state, a further abstraction can be obtained by adding a folding relation between terms with variables. This gives rise to an even more abstract simulation relation, where now the symbolic transition system can in many cases become finite-state. In order for the number of $E \cup B$-unifiers to be finite, which is needed to get the finite-state property for this more abstract symbolic system, the finite variant property is required of $E \cup B$.

I have already mentioned in Section 3.11.2 that, under very general conditions, time-bounded $LTL$/□ properties and standard $LTL$/♢ properties of a real-time rewrite theory can be effectively verified by model checking, even when
time is continuous. The reason for this is also related to simulations and abstractions. Specifically, we show in [359] that there is a stuttering bisimulation between the fair timed computations of a “time-robust” real-time rewrite theory and the much smaller set of computations obtained by always advancing the clock as much as possible until the next zero-time transition becomes enabled. For continuous time rewrite theories and time-bound $LTL\cup$ properties, this provides an abstraction from an infinite-state system to a finite-state one; but even for discrete time rewrite theories this provides a huge abstraction, marking in practice the difference between feasible and unfeasible model checking. One can further prove that when the state predicates in $\varphi \in LTL\cup$ do not depend on the value $r$ of the global clock, but only on the global state $\{t\}$, the projection map $({\{t\}, r} \mapsto \{t\})$ provides a further abstraction allowing the model checking of time unbounded properties in $LTL\cup$ when the set of discrete states (of the form $\{t\}$) reachable from the initial state is finite.

4. Rewriting Logic as a Logical and Semantic Framework

I further discuss here the logical and semantic framework uses already illustrated by means of simple examples in Section 2.

4.1. Representing Logics

Using rewriting logic as a logical framework can be best understood within a metatheory of logics such as the theory of general logics [298], 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 [210] 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 $\Phi$ from a logic $\mathcal{L}$ to a logic $\mathcal{L}'$ as a morphism

\[(\dagger) \quad \Phi : \mathcal{L} \longrightarrow \mathcal{L}'\]

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

\[(\ddagger) \quad \Psi : \mathcal{L} \longrightarrow \mathcal{F}\]

called representation maps, that have particularly good properties such as conservativity.\(^8\)

\(^8\)A map of logics is conservative [298] if the translation of a sentence is a theorem if and only if the sentence was a theorem in the original logic. Conservative maps are sometimes said to be adequate and faithful by some authors.
A number of logics, particularly higher-order logics based on typed lambda calculi, have been proposed as logical frameworks, including the Edinburgh logical framework LF [223, 32, 200], generic theorem provers such as Isabelle [369], λProlog [336, 190], and Elf [371], and the work of Basin and Constable [48] on metalogical frameworks. Other approaches, such as Feferman’s logical framework \( FS_0 \) [189]—that has been used in the work of Matthews, Smaill, and Basin [293]—earlier work by Smullyan [411], and the 2OBJ generic theorem prover of Goguen, Stevens, Hobley, and Hilberdink [213] are instead first-order. The role of rewriting logic as a logical framework 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 [316]).

As I have already pointed out in Section 2, one key property by which the practicality of a logical framework should be judged is by how short its representational distance is, and of course by how general it is in representing other logics. Regarding generality, since various typed lambda calculi have been extensively used as logical frameworks, a logical framework that can represent them with 0 representational distance can a fortiori represent anything they can represent, and possibly better. As Mark-Oliver Stehr and I have shown in [420], rewriting logic can represent with 0 representational distance not just some particular typed lambda calculus, but the parametric family of typed lambda calculi called pure type systems [50], which generalize the \( \lambda \)-cube and therefore contain virtually all typed lambda calculi of interest. The reverse is not at all the case: there is no representation of rewriting logic, or even of equational logic, into such calculi which could be said to have \( \epsilon \) representational distance. The obvious reason for this is the well-known difficulty of lambda calculi in dealing with equational reasoning, since the only equational reasoning native to such calculi is that between lambda expressions by \( \beta \)-reduction. Furthermore, in LF there is no adequate representation for linear logic in a precise technical sense of “adequate” [200, Corollary5.1.8]. Instead, linear logic can be faithfully represented in rewriting logic with 0 representational distance [288].

All these representations of logics are easily mechanizable using a rewriting logic language like Maude, leading to useful prototypes supporting formal reasoning for the logic in question. The nontrivial matter of quantifiers and substitutions is elegantly supported by Stehr’s CINNI calculus of explicit substitutions [416]. In particular, pure type systems can not only be represented: they can also be efficiently executed in a rewriting logic language like Maude. This trivial representation in one direction, and the serious difficulties for lambda calculi to deal with equality in the converse direction, were seen by Stehr as an opportunity to generalize the Coquand-Huet Calculus of Constructions (CC) [120] into his own Open Calculus of Constructions (OCC) [417, 418, 419] within rewriting logic (implemented in Maude as a theorem prover) to naturally support both what CC does and equational reasoning in a seamless way.

The above remarks make it already quite obvious that rewriting logic has very good properties as a logical framework. Several other examples of well-known logics which can be represented in rewriting logic with \( \epsilon \) representational
distance are given in [286, 288], and a more detailed discussion of logical framework application is given in Section 7.1. An additional good feature of rewriting logic as a logical framework is its ability to deal naturally with state changes, and therefore to solve in a straightforward way the thorny issue of the "frame problem," which has plagued for decades AI researchers using first-order logic as a knowledge representation formalism; this is explained in detail in [287].

Yet another very useful representational feature is rewriting logic’s "abstraction dial" (see Section 3.12). This was already obvious in the DPLL example of Section 2.2 and is systematically exploited for model checking purposes as explained in Section 3.12. For logical framework uses the general point is that: (a) there is a very useful distinction to be made between (i) computation, which is deterministic and can be blindly and exhaustively applied with high efficiency, and (ii) deduction, which is nondeterministic, requires search, and can be very inefficient; and (b) this computation vs. deduction distinction is naturally supported by a rewrite theory \( R = (\Sigma, E \cup B, R, \phi) \) as the distinction between its deterministic equations \( E \cup B \) and its nondeterministic rules \( R \). The practical meaning of all this is that one can make the implementation of a logic much more efficient, and the level at which a user interacts with a tool much higher, if millions of trivial computations are automatically performed, so that the strategic thinking about proofs can be focused at a much higher level. This was emphasized since the early papers on logical framework uses of rewriting logic [316, 286, 288], has been later dubbed "deduction modulo" by some researchers [143], and has been illustrated with interesting examples of rewrite theories representing logics such as those in [452, 383].

All the above remarks are fine and well, but even with all those good features a mapping \( \Psi : L \rightarrow F \) of a logic \( L \) into a logical framework \( F \) is still a complex metalevel entity: how can \( \Psi \) itself be represented? It is neither in \( L \) nor in the framework \( F \) but hovers above both. This is not a theoretical question but an eminently practical one: how is \( \Psi \) going to be implemented? and how can we reason about such a \( \Psi \)? Here is where rewriting logic’s reflective features play a key role, so that it is not just a good logical framework, but a reflective metalogical framework in the precise, technical sense given to the term in [47].

The key advantage of having a reflective logical framework 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 \( \mathcal{U} \) (see Section 3.4) of our reflective framework logic \( F \) (namely, rewriting logic), which has a sort \( \text{Theory} \) representing rewrite theories \( R \) as terms \( \overline{R} \) of sort \( \text{Theory} \), with equational abstract data type definitions for the data type of theories \( \text{Theory}_L \) for each logic \( L \) of interest. Then, a map of the form (†) can be reified as an equationally-defined function

\[
\Phi : \text{Theory}_L \rightarrow \text{Theory}_{L'}.
\]

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

\[
\Psi : \text{Theory}_L \rightarrow \text{Theory}.
\]
If the maps $\Phi$ and $\Psi$ are computable, then, by a metatheorem of Bergstra and Tucker [51] it is possible to define the functions $\bar{\Phi}$ and $\bar{\Psi}$ 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.

The point worth emphasizing again is that all this is not a theoretical divertimento but an enormously practical feature. For example, Pavel Naumov, Mark-Oliver Stehr and I used exactly the above approach to represent the logics of the HOL and NuPrl theorem provers within rewriting logic, define a conservative map of logics between them, prove its correctness, make such a formal definition executable in Maude, and automatically translate several megabytes of HOL theories into correct-by-construction NuPrl theories in [341] (a mechanical proof of correctness of such a map of logics was later given in [402]). Many more examples of how reflection is enormously useful to define and implement within $\mathcal{F}$ itself maps of logics, particularly maps of the form $\Psi : \mathcal{F} \rightarrow \mathcal{F}$ mapping the reflective framework to itself and corresponding to theory transformations are discussed in Sections 3.4, 6.1 and 7.1.

The last point is that rewriting logic is not just a logical framework but a metalogical one. As explained in [48], what a metalogical framework adds to a logical framework is the capacity to reason formally within itself about the metalogical properties of the logics represented in it. Typically such reasoning requires induction. As explained in [47], the reflective features of membership equational logic and of rewriting logic, combined with the fact that both logics have initial models supporting inductive reasoning principles, and with the fact that, in particular, their universal theories do come with their own induction principles, is what makes them into reflective metalogical frameworks. For several practical applications of rewriting logic’s metalogical reasoning capabilities see [47, 110, 107].

4.2. Representing Models of Concurrency

Since rewriting logic is a coin with two sides, a logical side and a computational one, the exact same reasons making it a very flexible logical framework with 0- or $\epsilon$-distance representations make it also a very flexible semantic framework. Since this is one of the main uses of rewriting logic since the beginning [303], so much work has been done that it is hardly possible to survey it all. But perhaps what is most important is for me to explain the philosophical distinction between a model and a logic, and why that distinction is crucial for representing concurrency models within rewriting logic.

The way concurrency models have been traditionally compared is by building encodings from one model into another. For example, some researchers encoded the CCS process calculus into Petri nets; and others encoded the lambda calculus and some variants of the actor model into the $\pi$-calculus. These are Turing-machine-like representations, where in principle one can show that some model can be simulated by another model by some kind of compilation process, but in general there is a substantial representational distance and much is lost in translation. If rewriting logic were to be one more such model into which other
models are similarly compiled, there would be little point in such a futile repre-
sentational exercise. The key observation is that rewriting logic is not a model
at all. It is instead a logic within which widely different models can be specified
as rewrite theories without any encoding. One can think of it as an “ecumenical
movement” with no sectarian ax to grind: it makes no commitments to specific
concurrency mechanisms. Is it better to be synchronous or asynchronous? Is
message-passing the best communication mechanism? Should channels be con-
ceived as names, or as communication links containing messages? Should the
order of messages be preserved or not? Should processes have unique names? All
these are questions for each specific model, that is, each specific rewrite theory,
to address or ignore. Rewriting logic remains politely silent about the choices
made in each model, but tries to be as flexible as possible in representing different
choices. My own opinion is that concurrency is such a motely phenomenon
(much more so than, say, functional computation) that the question “what is
the best model of concurrency?” is both meaningless and unwise. Chivalrous
quests for the Holy Grail of Concurrency, while commendable and probably
quite useful in their side effects, are likely to remain inconclusive. The point is
that any model must make some commitments about what concurrency mech-
anisms to favor; and this will automatically create a representational distance
between it and other models making other equally valid commitments, perhaps
for different purposes and reasons.

Just to give some feeling for the vast amount of work which has been done
in defining different models of concurrency as rewrite theories, typically with
0 or ϵ representational distance, I mention first some well-known models not
involving real time or probabilities, and then discuss real time and probabilistic
models. Next to each model I mention some references for illustration purposes,
without any attempt to cover them all (see the bibliography in this issue for a
hopefully complete list of references).

1. Actors and Concurrent Objects [304, 428].
2. CCS [124, 446, 74].
3. Dataflow [306].
4. Gamma and the CHAM [303].
5. Graph Rewriting [306, 407].
6. Neural Networks [306, 398].
7. Parallel λ-Calculus [267].
8. Petri Nets [303, 421].
9. π-Calculus [451, 416, 437].
10. Tile Logic [317, 80, 75, 79].
11. The UNITY Model [303].
An important point not made explicit by the above list is that the initial model semantics of rewriting logic (see Section 3.1.1) plays also a crucial role, because it unifies within a single semantics very different denotational models that have been independently proposed for various models of concurrency. For example, rewriting logic’s initial model semantics specializes to: (i) for Actors to the event diagram partial order of events model of [42, 117], as shown in [325]; (ii) for Petri nets to the Best-Devillers commutative process model [54], as shown in [125, 421]; (iii) for the parallel lambda calculus to its traditional model, shown to be a simple quotient of the initial model of the corresponding rewrite theory in [267]; and (iv) for CCS to the proved transition causal model of Degano and Priami [126], shown to be a simple quotient of the initial model of the corresponding rewrite theory in [81].

For real-time models, real-time rewrite theories also provide a very general and flexible semantic framework. For example, the following models of real time can all be naturally specified as real-time rewrite theories:

1. Hybrid Automata [357].
2. Timed Petri Nets [357, 421].
3. Timed Automata [357].
4. Timed Transition Systems [357].
5. Object-Oriented Real-Time Systems [357].
6. The Orc Model of Concurrent Real-Time Computation [20, 21].
7. Phase Transition Systems [357].

Probabilistic rewrite theories can also be used as a semantic framework for a wide range of probabilistic systems, including:

1. Continuous Time Markov Chains [265].
3. Object-Oriented Probabilistic Systems [266, 6].
4. Object-Oriented Stochastic Hybrid Systems [324].
5. Probabilistic Nondeterministic Systems [265].

4.3. Rewriting Logic Semantics of Programming Languages

The flexibility of rewriting logic to naturally express many different models of concurrency can be exploited not just at the theoretical level, for expressing such models both deductively, and denotationally in the model theory of rewriting logic [303, 306]: it can also be applied to give formal definitions of concurrent programming languages by specifying the concurrent model of a language \( \mathcal{L} \) as a rewrite theory \((\Sigma_\mathcal{L}, E_\mathcal{L}, R_\mathcal{L})\), where: (i) the signature \( \Sigma_\mathcal{L} \) specifies both the
syntax of $\mathcal{L}$ and the types and operators needed to specify semantic entities such as the store, the environment, input-output, and so on; (ii) the equations $E_\mathcal{L}$ can be used to give semantic definitions for the deterministic features of $\mathcal{L}$ (a sequential language typically has only deterministic features and can be specified just equationally as $(\Sigma_\mathcal{L}, E_\mathcal{L})$); and (iii) the rewrite rules $R_\mathcal{L}$ are used to give semantic definitions for the concurrent features of $\mathcal{L}$ such as, for example, the semantics of threads. By specifying the rewrite theory $(\Sigma_\mathcal{L}, E_\mathcal{L}, R_\mathcal{L})$ in a rewriting logic language like Maude, it becomes not just a mathematical definition but an executable one, that is, an interpreter for $\mathcal{L}$. Furthermore, one can leverage Maude’s generic search and LTL model checking features to automatically endow $\mathcal{L}$ with powerful program analysis capabilities. For example, the search command can be used in the module $(\Sigma_\mathcal{L}, E_\mathcal{L}, R_\mathcal{L})$ to detect any violations of invariants, e.g., a deadlock or some other undesired state, of a program in $\mathcal{L}$. Likewise, for terminating concurrent programs in $\mathcal{L}$ one can model check any desired LTL property. All this can be effectively done not just for toy languages, but for real ones such as Java and the JVM, Scheme, and C (see Section 7.2 for a discussion of such “real language” applications), and with performance that compares favorably with state-of-the-art model checking tools for real languages.

There are essentially three reasons for this surprisingly good performance. First, rewriting logic’s distinction between equations $E_\mathcal{L}$, used to give semantics to deterministic features of $\mathcal{L}$, and rules $R_\mathcal{L}$, used to specify the semantics of concurrent features, provides in practice an enormous state space reduction. Note that a state of $(\Sigma_\mathcal{L}, E_\mathcal{L}, R_\mathcal{L})$ is, by definition, an $E_\mathcal{L}$-equivalence class $[t]_{E_\mathcal{L}}$, which in practice is represented as the state of the program’s execution after all deterministic execution steps possible at a given stage have been taken. That is, the equations $E_\mathcal{L}$ have the effect of “fast forwarding” such an execution by skipping all intermediate deterministic steps until the next truly concurrent interaction is reached. For example, for $\mathcal{L} = \text{Java}$, $E_{\text{Java}}$ has hundreds of equations, but $R_{\text{Java}}$ has just 5 rules. The second reason is of course the high performance of rewriting logic languages such as Maude, which can reach millions of rewrite steps per second. The third reason is that the intrinsic flexibility of rewriting logic means that it does not prescribe a fixed style for giving semantic definitions. Instead, many different styles such as, for example, small-step or big-step semantics, reduction semantics, CHAM-style semantics, modular structural operational semantics, or continuation semantics, can all be naturally supported [409]. But not all styles are equally efficient; for example, small-step semantics makes heavy use of conditional rewrite rules, insists on modeling every single computation step as a rule in $R_\mathcal{L}$, and is in practice horribly inefficient. Instead, the continuation semantics style described in [409] and used in, e.g., [185] is very efficient.

As for models of concurrency, the general idea for SOS definitions is that rewriting logic provides a general framework for such definitions, but has no ax to grind regarding specification style choices. From its early stages rewriting logic has been recognized as ideally suited for SOS definitions [315, 288], and has been used to give SOS definitions of programming languages in quite different
styles, e.g., [445, 74, 446, 447, 185, 188]. What the paper [409] makes explicit is both the wide range of SOS styles supported, and the possibility of defining new styles that may have specific advantages over traditional ones. Where the “abstraction dial” is placed in such choices is of course crucial for the efficiency of model checking analyses: traditional styles will tend to force the least abstract choices that specify all computation steps with rules; but many more choices are available when the underlying logic supports a distinction between equations and rules.

The good theoretical and practical advantages of using rewriting logic to give semantic definitions to programming languages have stimulated an international research effort called the rewriting logic semantics project (see [322, 323, 409] for some overview papers). Not only have semantic definitions allowing effective program analyses been given for many real languages such as Java, the JVM, Scheme, and C, and for hardware description languages such as ABEL and Verilog: it has also been possible to build a host of sophisticated program analysis tools for real languages based on different kinds of abstract semantics. The point is that instead of a “concrete semantics” \((\Sigma_L, E_L, R_L)\), describing the actual execution of programs in a language \(L\), one can just as easily define an “abstract semantics” \((\Sigma^A_L, E^A_L, R^A_L)\) describing any desired abstraction \(A\) of \(L\).

A good example is type checking, where the values manipulated by the abstract semantics are the types. All this means that many different forms of program analysis, much more scalable than the kind of search and model checking based on a language’s concrete semantics, become available essentially for free by using Maude to execute and analyze one’s desired abstract semantics \((\Sigma^A_L, E^A_L, R^A_L)\).

I further discuss different applications of both concrete and abstract rewriting semantics of programming languages in Section 7.

Two further developments of the rewriting logic semantics project, both pioneered by Grigore Roșu with several collaborators, are worth mentioning. One is the K semantic framework for programming language definitions [388], which provides a very concise and highly modular notation for such definitions. The K-Maude tool then automatically translates language definitions in K into their corresponding rewrite theories in Maude for execution and program analysis purposes (I further discuss K and the K-Maude tool in Section 6.2.2). Another is matching logic [394, 387], a program verification logic, with substantial advantages over both Hoare logic and separation logic, which uses a language’s rewriting logic semantics, including the possibility of using patterns to symbolically characterize sets of states, to mechanize the formal verification of programs, including programs that manipulate complex data structures using pointers (I further discuss matching logic and the MatchC tool in Section 6.2.3).

### 4.4. Representing Distributed Systems, Software Architectures, and Models

It is well known that the most expensive errors in system development are design errors. They are not coding errors having to do with some mistake in the details of a program: they happened much earlier, when the system was designed and no programs yet existed. Because design errors affect the overall structure of a system and are often discovered quite late in the development
cycle, they can be enormously expensive to fix. All this is uncontroversial: there is widely-held agreement that, to develop systems, designs themselves should be made machine-representable, and that tools are needed to keep such designs consistent and to uncover design errors as early as possible. This has led to the development of many software modeling languages and of architectural notations to describe software designs.

There are however two main limitations at present. The first is that some of these notations lack a formal semantics: they can and do mean different things to different people. The second is that this lack of semantics manifests itself at the practical level as a lack of analytic power, that is, as an incapacity to uncover expensive design errors which could have been caught by better analysis. It is of course virtually impossible to solve the second problem without solving the first: without a precise mathematical semantics any analytic claims about satisfaction of formal requirements are meaningless.

The practical upshot of all this is that a semantic framework such as rewriting logic can play an important role in: (i) giving a precise semantics to modeling languages and architectural notations; and in (ii) endowing such languages and notations with powerful formal analysis capabilities. Essentially the approach is the same as for programming languages. If, say, \( \mathcal{M} \) is a modeling language, then its formal semantics will be a rewrite theory of the form \( (\Sigma_\mathcal{M}, E_\mathcal{M}, R_\mathcal{M}) \). If the modeling language \( \mathcal{M} \) provides enough information about the dynamic behavior of models, the equations \( E_\mathcal{M} \) and the rules \( R_\mathcal{M} \) will make \( \mathcal{M} \) executable, that is, it will be possible to simulate models in \( \mathcal{M} \) before they are realized by concrete programs, and of course such models thus become amenable to various forms of formal analysis. There is a large body of research in rewriting logic that has done just this. For example, giving formal semantics to various object-oriented design notations, architectural notations, and software modeling languages, e.g., [191, 257, 258, 460, 31, 108, 149, 58, 334, 57, 40, 380, 381, 163, 333, 59, 62, 335, 63, 38, 351], and to various middleware and distributed coordination mechanisms, e.g., [338, 13, 14, 161, 162, 391, 148]. I discuss all this work in more detail in Section 7, and the MOMENT-2 tool in Section 6.2.

Since many of the software architectures needed in practice are distributed architectures, the flexibility of rewriting logic to naturally represent a wide range of distributed communication and interaction mechanisms has proved very useful in all the applications mentioned above. But the medium of a modeling language or an architectural description language is not a necessary requirement. It is also possible to specify and analyze a wide range of distributed system designs and algorithms directly in rewriting logic. In practice this has been often the case for many network algorithms, e.g., [127, 129, 448, 215, 362, 364, 247, 378], and middleware designs and distributed reflective architectures, e.g., [130, 326, 429, 158] I further discuss all this work in Section 7.

5. Rewriting Logic Languages

In this section I discuss CafeOBJ, ELAN, and Maude, three languages that implement rewriting logic and whose researchers, through their language de-
sign and implementation work and through a host of important new techniques and applications, have made fundamental contributions to the rewriting logic research program. These are not the only rule-based languages that I could discuss. For example, OBJ [212], ASF+SDF [444], Tom [44], and Stratego [454] are other important rule-based languages; but they are somewhat more specialized in nature: OBJ and ASF+SDF deal with equational specifications; Tom enriches Java with rewriting capabilities; and Stratego is a rewriting strategy language aimed particularly at program transformation applications.

5.1. CafeOBJ

CafeOBJ [137] is a language containing in essence OBJ [212] as a functional sublanguage but extending substantially order-sorted equational logic in two orthogonal and complementary directions: (i) it supports behavioral specifications and their execution by behavioral rewriting in behavioral equational logic; and (ii) it also supports rewriting logic specifications. Furthermore, these orthogonal logical features are combined in the “CafeOBJ Cube” [137]. As OBJ, CafeOBJ has powerful module composition features through module hierarchies, parameterization, and module expressions. Two additional important features are CafeOBJ’s support for object-oriented modules, and its support for Observational Transition Systems (OTS), a special type of behavioral specifications ideally suited to specify transitions systems such as network protocols and other distributed systems. CafeOBJ specifications can be formally analyzed in various ways. An important theme is the use of proof scores [195, 196] which reduce the proof of inductive properties about a CafeOBJ specification to rewriting on the underlying CafeOBJ engine. Of particular interest from the rewriting logic point of view is CafeOBJ’s search feature, which supports breadth-first search modulo a user-specified equality predicate [196], a very useful form of abstraction-based model checking. Also interesting in this direction is the synergistic way, already mentioned in Section 3.11.3, in which CafeOBJ and Maude can be used together to analyze OTS specification by model checking [462]. I discuss some CafeOBJ applications in Section 7.

5.2. ELAN

I have already mentioned in Section 3.5 the importance of strategies for controlling the rewriting process when the rules can be highly nondeterministic, and the key contributions that the ELAN researchers have made in this area. ELAN [67, 66] supports the specification of sophisticated strategies that can guide the rewriting process to achieve complex tasks. This has applications in many areas that have been developed by the ELAN researchers; I discuss some of them in Section 7. In particular, from the beginning of the language the ELAN researchers have developed many applications of rewriting logic as a logical framework which greatly benefit from the use of strategies. The key idea is that the logical inference system used in a theorem prover or in some other logical procedure is typically nondeterministic. Therefore search, as opposed
to deterministic computation, is essential. ELAN supports a corresponding distinction at the language level between computation rules, which are applied exhaustively without using strategies, and strategy-guided rules. At the language implementation level, besides the contributions to efficiently support strategies, an important additional contribution has been the development of novel compilation techniques for efficient rewriting modulo associativity-commutativity.

5.3. Maude

Maude [102, 103] supports both membership equational logic (its functional sublanguage of functional modules), and rewriting logic (system modules) in the fullest possible generality: equations and rules can be conditional and can have extra variables in their right-hand sides and conditions, and rewriting modulo any combination of associativity and/or commutativity and/or identity axioms is supported [103]. All this is achieved without sacrificing high performance thanks to Maude’s use of advanced semi-compilation techniques and novel matching modulo algorithms [109, 164, 102, 165]. Maude has also powerful module composition operations and support for parameterized modules, theories and views. A key feature is its efficient support for reflection (see Section 3.4) through its META-LEVEL module. Besides providing powerful higher-order meta-programming features (functions can take not just other functions as arguments, but entire modules as arguments), this makes the Maude module composition operations extensible [155], which is exploited in the Full Maude language extension [103] to support, for example, very convenient syntax for object-oriented specifications. Reflection is also exploited in an essential way in Maude’s strategy language [168, 291]. A unique feature of Maude is its efficient built-in support for model checking. Reachability analysis and invariant verification are supported by its breadth-first search command; and LTL model checking by its MODEL-CHECKER module. Another important feature is its support for order-sorted unification modulo axioms, and for variant computations and symbolic reachability analysis modulo equational theories with the finite variant property [100, 147]. I discuss Maude’s formal environment in Section 6.1, and some Maude applications in Section 7.

6. Tools

In Section 6.1 I discuss some tools supporting various kinds of formal reasoning about rewriting logic specifications. In Section 6.2 I discuss several more specialized tools that use rewriting logic and its reasoning methods to support formal analysis in various application domains.

6.1. Formal Tools for Rewriting Logic

In Section 3 I discussed in detail various formal properties that one often wants to verify about a rewrite theory. Tools supporting verification of such properties are very important. I discuss some of them here with the exception
of the search and model checking capabilities already native to rewriting logic languages: CafeOBJ, ELAN, and Maude support various forms of search analysis, and Maude also supports LTL model checking. Some of these formal tools, particularly the Maude-based ones, systematically use reflection (see Section 3.4) in their design: since formal analysis tools manipulate and transform theories, a reflective approach making such theories data structures manipulable within rewriting logic is very useful in practice. Indeed, several of the Maude formal tools use the Full Maude reflective extension of Maude [105, Part II] as their basis, and then use the general methodology outlined in [157] to add tool-specific reflective features. The tools mentioned below are an incomplete set of tools; see the rewriting logic bibliography in this issue for references to other tools.

6.1.1. The Maude Church-Rosser Checker and Coherence Checker (CRChC)

These are two closely-related tools combined into one [156]. The CRC tool checks the confluence and sort-decreasingness of conditional order-sorted specifications modulo axioms, assuming they are operationally terminating (see Section 3.8). Instead, the ChC tool checks the coherence, or ground coherence, of a rewrite theory’s rules $R$ with respect to their equations $E$ modulo axioms $B$, assuming that the equations themselves are (ground) confluent, sort-decreasing and operationally terminating.

6.1.2. The CARIBOO Termination Tool

Cariboo [192, 208] is a termination tool written in ELAN which can prove ground termination of rewrite theories written in ELAN with respect to a given strategy (see Section 3.5). It uses induction, an abstraction mechanism to represent sets of terms symbolically with logical variables, and narrowing controlled by abstraction constraints. Orderings need not be chosen in advance but can be partially and incrementally determined by means of constraints.

6.1.3. The Maude Termination Tool (MTT) and $\mu$-Term

The Maude termination tool (MTT) [151, 152] supports termination proofs for generalized rewrite theories and for membership equational theories, which can both be conditional and have axioms such as associativity, commutativity and identity. As already explained in Section 3.8, the main technique used by MTT is that of non-termination-preserving theory transformations that transform such theories to either order-sorted or unsorted context-sensitive unconditional specifications modulo axioms. Termination tools such as $\mu$-Term [9] or AProVE [204] can then be invoked by the user to try to prove the transformed theory terminating. $\mu$-Term is in some ways closer to MTT because of its unrivaled support for context-sensitive termination and its support for order-sorted termination.

6.1.4. The Maude Sufficient Completeness Checker (SCC)

The Maude Sufficient Completeness Checker (SCC) [229] can check the sufficient completeness (see Section 3.7) of context-sensitive unconditional left-linear
order-sorted equational theories modulo axioms [227], and in its most recent version also the sufficient completeness of both equations and rules in unconditional order-sorted left-linear theories modulo axioms [384]. SCC uses the CETA library of propositional tree automata operations developed by Joe Hendix as part of his Ph.D. dissertation [224] to reduce all the above sufficient completeness problems to tree automata emptiness problems.

6.1.5. The Maude Inductive Theorem Prover (ITP)

The Maude Inductive Theorem Prover (ITP) was originally developed by Manuel Clavel and has been substantially extended by Joe Hendix [104, 115, 224]. It supports inductive reasoning about membership equational theories in Maude and has been applied to a wide range of problems and also to build more specialized theorem provers for imperative programming languages and for modeling languages [116, 399, 108]. Its original support for structural induction has been more recently extended to also support coverset induction [224, 226]. An important feature of the ITP is its natural support for partiality, which is nicely demonstrated by the extended powerlist case study developed by Joe Hendrix as part of his Ph.D. thesis [224, 226].

6.1.6. The Maude Formal Environment (MFE)

Often a verification task requires interoperating different tools. For example, the proof of an inductive theorem using the ITP may be based on a structural induction scheme using constructors whose proof is provided by the SCC tool, but the sufficient completeness proof relies on a weak termination assumption for which the MTT tool may be invoked. Similarly, a proof of ground coherence using the ChC tool may generate inductive proof obligations for the ITP, and requires a proof of confluence of the equations using the CRC, which itself relies on a proof of operational termination of those equations using the MTT. To support the seamless interoperability of formal tools for rewriting logic within a single formal environment, the Maude Formal Environment (MFE) [159] has been developed as an extensible framework to which different Maude-based tools can be added. Besides allowing the user to ship proof tasks from one tool to another, MFE keeps track of the overall proof effort, and stores a record of the tool interactions and subproof invocations involved in such an overall proof, so that proof scripts can be stored and reused.

6.1.7. The Declarative Maude Debugger

In addition to the debugging capabilities already provided by Maude itself [103], the Declarative Maude Debugger [379] can interact with a user to find the causes of wrong answers in a Maude program execution and also of missing answers, which are particularly important for nondeterministic programs such as rewrite theories (system modules), but are also meaningful for deterministic ones (functional modules) because of sort information. The debugger traverses an abbreviated proof tree, which stores an abbreviated declarative summary of the computation, and interacts with the user asking questions until the cause of the bug is found.
6.1.8. Real-Time Maude

Real-Time Maude [360] is a specification language and a formal tool built as an extension of Full Maude by reflection. It provides special syntax to specify real-time systems, including distributed object-oriented ones, where the time can be either discrete or continuous. It offers a range of formal analysis capabilities, including simulation, reachability analysis, and model checking. Real-Time Maude systematically exploits the underlying Maude efficient rewriting, search, and LTL model checking capabilities to both execute and formally analyze real-time specifications, which are internally desugared into ordinary Maude specifications and Maude search and model checking queries using reflection [360]. It furthermore supports model checking in a subset of MTL [272], and in TCTL [271] (see Section 3.11). Real-Time Maude has been applied in a wide range of industrial applications, including networks, embedded car software, and scheduling algorithms. It has also been used to give formal semantics to, and provide formal analysis for, several real-time programming languages and software modeling languages. I further discuss these applications in Section 7.4.

6.1.9. The PMaude Language Design

The PMaude language [266, 4] is an experimental specification language whose modules are probabilistic rewrite theories. It is still language design, since it has not yet passed the prototyping level. However, since its methodology has already been successfully applied to a wide range of applications such as sensor networks, defenses against Denial of Service (DoS) attacks, and stochastic hybrid systems (I further discuss these applications in Section 7.5), it seems appropriate to discuss it here. Recall from Section 3.10 that, due to their nondeterminism, probabilistic rewrite rules are not directly executable. However, probabilistic systems specified in PMaude can be simulated in Maude. This is accomplished by transforming a PMaude specification into a corresponding Maude specification in which actual values for the new variables appearing in the righthand side of a probabilistic rewrite rule are obtained by sampling the corresponding probability distribution functions (see Section 3.3 in [313] for a detailed explanation). Using the transformed Maude module one can perform Monte-Carlo simulations of the given PMaude module. Using the methodology presented in [4] and discussed in Section 3.11.2, one can then use the VeStA and PVeStA tools discussed below to perform statistical model checking verification of temporal logic properties of a real-time PMaude module expressed in either CSL or QuaTEx (see Section 3.11.1).

6.1.10. VeStA and PVeStA

The VeStA statistical model checking tool [405, 4] supports statistical model checking (see Section 3.11.2) of probabilistic real-time systems specified as either: (i) discrete or continuous Markov Chains; or (ii) probabilistic rewrite theories in Maude. Furthermore, the properties that it can model check can be expressed in either: (i) CSL/PCTL, or (ii) the QuaTEx quantitative temporal logic (see Section 3.11.1). One important practical issue for any model checking analysis is scalability. Since statistical model checking is parametric on a

58
user-specified level of statistical confidence, if such a level is high, the number of Monte-Carlo simulations that have to be performed before VeStA can return an answer to a model checking query can be very large. Fortunately, Monte-Carlo simulations can be run in parallel on different processors. This has led to the design and implementation of the PVeStA tool [23], which parallelizes the statistical model checking analysis of probabilistic rewrite theories, making it highly efficient and scalable. For example, a realistic model checking problem can be sped up by a factor of 46 on a 60 node parallel machine using PVeStA, compared with the time required for VeStA to perform the same task on a single node [23].

6.2. Domain-Specific Tools

This section is much more of a random sample than Section 6.1: there are many more domain-specific tools based on a rewriting logic semantics than the ones mentioned below, and discussing them all is out of the question. For example, any rewriting logic semantics of a programming language or of a modeling language expressed in Maude or Real-Time Maude automatically provides a tool supporting simulation, reachability analysis, and LTL model checking for such a language. A number of other tools are discussed much more briefly in Section 7, and the bibliography in this issue gives a more comprehensive picture. My main goal here is to give the reader a feeling through concrete examples for some of the advanced applications that can be supported by tools of this kind.

6.2.1. JavaFAN

The Java Formal Analyzer (JavaFAN) [185, 188] is a tool supporting the execution and analysis of the source code and the JVM code of Java programs. It is based on rewriting logic semantic definitions in Maude at both the Java and the JVM level. The entire language, except for the libraries, is supported. Such definitions provide interpreters for Java and for the JVM. Also, multithreaded Java and JVM programs can be formally analyzed to detect violations of invariants using Maude’s breadth search command; and terminating multithreaded programs can likewise be model checked with respect to LTL properties using Maude’s LTL model checker. To facilitate the use of the tool and make knowledge of the underlying semantics unnecessary for users, Java and JVM code can be directly entered into JavaFAN and is then automatically translated into Maude. Similarly, JavaFAN provides an intuitive Java-like syntax for defining atomic predicates which makes it easy for users to define search commands and LTL queries only in terms of their programs. The performance of JavaFAN compares favorably with other state-of-the-art tools such as Java PathFinder on various benchmarks [185, 188], which is encouraging since JavaFAN is just a formal semantic definition of Java. One of the reasons for this is rewriting logic’s distinction between equations and rules (the “abstraction dial” mentioned in Section 3.12), which, while still faithfully capturing the concrete semantics, allows a huge equational abstraction of the state space by expressing all deterministic features equationally and reserving rules for the nondeterministic, concurrent features.
6.2.2. K-Maude

As mentioned in Section 4.3, one of the important recent contributions to the rewriting logic semantics project is the K framework [388], which provides a concise and highly modular notation for programming language definitions. K is a new definitional style offering specific advantages over SOS-based styles such as those discussed in [406]. Furthermore, the relation between a K definition and its corresponding rewriting logic semantics is essentially one of desugaring, where what is conveniently implicit in the more compact K notation is made fully explicit in its rewriting logic counterpart. The K-Maude tool [408, 407] allows a user to define the semantics of a programming language in K and provides two main features. The first one is the automatic generation of a Latex rendering of the given K definition for ease of readability in two different styles, one more textual and another more graphical and intuitive. The second and main feature is that the rewriting logic semantics of K is supported by the tool, so that the rewrite theory corresponding to a language definition in K is automatically generated as a Maude module. In this way, K definitions can be executed as interpreters, and programs can be formally analyzed by reachability analysis and LTL model checking. K-Maude has already been applied to give K definitions for entire languages such as, for example, Scheme and C.

6.2.3. The MatchC Tool

As mentioned in Section 4.3, matching logic [394, 387] is another key contribution to the rewriting logic semantics project. It is a logic of programs with clear advantages over Hoare logic and separation logic. The key idea is to leverage a programming language’s rewriting logic definition as the mathematical basis for the matching logic inference system. What matching logic essentially does is to extend such a definition into a full-fledged first-order reasoning system which manipulates symbolic descriptions (with existential and universal variables) of programs and their properties, and uses the term matching (modulo axioms) native to rewriting logic to express both properties about program configurations, and the application of semantic rules to such configurations. This accomplishes at a simpler, structural level all the separation properties achieved by separation logic at the logical level. In this way, programs involving pointers and complex data structures on the heap can be easily reasoned about. A very appealing feature of matching logic is that there is essentially no gap between the level of a language’s semantic definition and that of its logic, whereas proving soundness and relative completeness of a Hoare logic with respect to an operational semantics is a highly nontrivial task. Although the matching logic ideas are very general, the current MatchC tool [387] realizes them for the C language with a remarkable level of automation and with very high efficiency. An impressive web-accessible collection of benchmarks has already been assembled [387].

6.2.4. The Maude-NPA

The Maude NPA [177] is a tool to verify security properties of cryptographic protocols modulo the algebraic properties of their cryptographic functions. The
point is that one can “verify” that a protocol is correct with respect to the traditional Dolev-Yao model which treats the cryptography as a “black box,” but an attacker can sometimes break such a protocol by making use of algebraic properties. For example, if the protocol uses an exclusive or operation ⊕, and the attacker has already seen a message m, then it can get message m′ from the message $m \oplus m'$ just by performing the operation $m \oplus m' \oplus m$, since ⊕ is associative and commutative, and satisfies the equation $x \oplus x = 0$ and $x \oplus 0 = x$. All this means that reasoning modulo such axioms is an essential feature of security proofs, since attacks can be mounted using them. The Maude-NPA does exactly this by: (i) axiomatizing a protocol $P$ as a (topmost) rewrite theory $(\Sigma_P, E_P \cup B, R_P)$, where $P$’s equational properties are axiomatized by the equations $E_P \cup B$, and $P$’s transitions are axiomatized by the rules $R_P$; (ii) characterizing attack patterns as terms with variables describing a possibly infinite set of concrete attack states; and (iii) using the rules $R_P$ in reverse\(^9\) to search for an initial state from the given attack pattern $p$. This is accomplished by narrowing $p$ with the reversed rules $R_P^{-1}$ modulo $E_P \cup B$; which, as explained in Section 3.3 and in [328], is a complete reachability analysis method for topmost rewrite theories. Of course this still leaves the problem of computing $E_P \cup B$-unifiers. Fortunately, many equational theories $E_P \cup B$ of interest satisfy the finite variant property (see Section 3.3), so that the Maude-NPA uses narrowing at two levels: with $R_P^{-1}$ modulo $E_P \cup B$ for reachability analysis; and with $E_P$ modulo $B$ to compute $E_P \cup B$-unifiers. Since the narrowing tree generated by a search from an attack pattern $p$ is typically infinite, an important additional feature of the Maude-NPA is the use of very powerful state space reduction techniques [176] that often make such a symbolic search space finite, so that not finding an attack is in fact a proof that the protocol is safe from the given attack modulo the algebraic properties $E_P \cup B$. I further discuss applications of the Maude-NPA in Section 7.3.

### 6.2.5. MOMENT2

MOMENT2 is an algebraic model management framework and tool written in Maude and developed by Artur Boronat [57]. It permits manipulating software models in the Eclipse Modeling Framework (EMF). It uses OMG standards, such as Meta-Object Facility (MOF), Object Constraint Language (OCL) and Query/View/Transformation (QVT), as a clean interface between rewriting-logic-based formal methods and model-based industrial tools. Specifically, it supports formal analyses based on rewriting logic and graph transformations to endow model-driven software engineering with strong analytic capabilities. MOMENT2 supports not just one fixed modeling language, but any modeling language whose meta-model is specified in MOF. In more detail, a modeling language is specified as a pair $(\mathcal{M}, \mathcal{C})$, where $\mathcal{M}$ is its MOF-based metamodel, and $\mathcal{C}$ are the OCL constraints that $\mathcal{M}$ should satisfy. Using rewriting-logic-based reflection and its efficient support in Maude, MOMENT2 provides an executable

\(^9\)That is, a rule $t \rightarrow t'$ is now viewed in reverse as a rule $t' \rightarrow t$. 

61
algebraic semantics for such metamodel specifications \((M, C)\) in the form of a theory in membership equational logic (MEL) \(A(M, C)\), so that a model \(M\) conformant with the metamodel \((M, C)\) is exactly a term of sort Model in \(A(M, C)\), and so that satisfaction of OCL constraints is also decidable using the algebraic semantics \([61, 63]\).

Due to the executability of MEL specifications in Maude, the realization of MOF metamodels as MEL theories enhances the formalization and prototyping of model-driven development processes, such as: (i) model transformations; (ii) model-driven roundtrip engineering; (iii) model traceability; and (iv) model management. These processes permit, for example, merging models, generating mappings between models, and computing differences between models; they can be used to solve complex scenarios such as the roundtrip problem. In MOMENT2 the formal semantics of model transformations is given by rewrite theories specified in a user-friendly QVT-based syntax \([59]\). Such model transformations can describe the dynamic evolution of systems at the level of their models. Using the search and LTL model checking features of Maude, properties about the dynamic evolution of a model \(M\) conformant with a metamodel specification \((M, C)\) can then be formally analyzed by model checking \([59]\). Real-time modeling languages can likewise be supported and analyzed \([64]\); this is further discussed in Section 7.4.4.

7. Some Applications

I discuss applications in areas such as automated deduction, software and hardware specification and verification, security, real-time and cyber-physical systems, probabilistic systems, and bioinformatics. Neither the choice of areas nor the work discussed in each of them aim at any completeness: this is just a sample.

7.1. Automated Deduction Applications

Perhaps the most important automated deduction applications are formal tools for different logics and automated deduction procedures that use rewriting logic as a logical framework. As explained in Section 4.1, the systematic idea common to all such tools is the faithful representation of their underlying inference systems as rewrite theories. Furthermore, using reflection very sophisticated tools can be built this way for many logics and for rewriting logic itself \([106]\). All the rewriting-logic-based tools discussed in Section 6.1 exemplify this general approach. But many other tools or prototypes for different automated deduction procedures have likewise been developed this way using either ELAN or Maude, including, for example,

- Constraint solving \([65, 255, 256, 458, 235]\).
- Higher-order logics, procedures, and provers, explicit substitution calculi, and translations between such logics \([43, 53, 141, 416, 142, 341, 420, 418, 419]\).
• Proof certification [342, 393].
• Rule Completion [253].
• Timed automata verification [49].
• Other theorem proving systems and procedures [91, 136, 452, 143, 382, 383].

7.2. Software and Hardware Specification and Verification

Systems need to be specified and verified at various levels of abstraction. Rewriting logic has very good properties as a semantic framework to support such specification and verification at different levels: at the level of models in the early stages of software design; at the level of code written in different programming languages; and at the hardware level. Furthermore, specification and verification of different network systems, and of distributed architectures, middleware, and coordination and reflection mechanisms can likewise be supported. All this has been described in broad outlines in Sections 4.3 and 4.4. Here I discuss in more detail some of the concrete applications that have been developed at all these levels.

7.2.1. Modeling Languages

As explained in Section 4.4, software design notations and modeling languages are quite useful, but they can be made even more useful by substantially increasing their analytic power through formal analysis, since this can make it possible to catch expensive design errors very early. Formal analysis is impossible or fraudulent without a formal semantics. Early work in developing rewriting-logic-based formal semantics focused on object-oriented design notations and languages [459, 340, 339], and stimulated subsequent work on UML and UML-like notations, e.g., [191, 257, 258, 460, 31, 108, 334, 333, 163, 335].

A more ambitious question is: can we give semantics not just to a single modeling language, but to an entire modeling framework where different modeling languages can be defined? This question has been answered positively in [58, 57, 59, 62, 63, 381], and has led to the MOMENT2 and the e-Motions tools (see Sections 6.2.5 and 7.4.4).

I further discuss the semantics of real-time modeling languages [40, 380, 381, 38, 351, 64, 39] in Section 7.4. Some recent work has also considered the semantics of multi-modeling languages [60], that is, languages that can combine different models describing various perspectives about the same system.

7.2.2. Programming Languages

I have already given an overview of the rewriting logic semantics project in Section 4.3. Here I discuss concrete applications within this project. Early work focused on SOS definitions of process calculi and of small programming languages [315, 288, 445, 74, 446, 447]. The first application to a “real” programming language showing that this approach could scale up to large languages
and could be used to analyze programs with competitive performance was the semantics of Java and the JVM [185, 188] described in Section 6.2.1. Since then, many other languages have been partially or totally defined in rewriting logic, sometimes using the K notation. For example, Beta [232] and KOOL [234] have been so defined; all of Scheme has been defined in [295, 296], and the formal semantics of C in [169] is arguably the most complete ever and will soon cover the entire C language. Another real language whose rewriting semantics has been fully defined in Maude is PLEXIL, a synchronous language developed by NASA to support autonomous spacecraft operations. The Maude-based formal executable semantics of PLEXIL [144] has become the de facto PLEXIL standard at NASA, against which the correctness of PLEXIL implementations is judged, and is the basis of other PLEXIL tools [386].

As mentioned in Section 4.3, the rewriting semantics of a language can be extended and/or abstracted to provide other kinds of static and dynamic analysis, for example, for units of measurement [88, 233], type checking [170], and runtime verification [395, 407]. Two extensions of a programming language’s rewriting logic semantics to model fault detection (resp. hang detection) have been developed by Pattabiraman et al. [367] (resp. Wang et al. [455]). In [367], the authors use rewriting logic to model both the semantics of an assembly language and the hardware on which it runs, as well as various hardware errors. The overall goal is to provide a formal semantic framework (called SymPLFIED) to analyze the effectiveness of error detection mechanisms. Maude’s search command is used for complete reachability analysis. In [455], a Linux-like operating system, as well as the underlying hardware, are formally specified in Maude in order to verify the detection effectiveness of an operating system’s hang detector. In order to exhaustively explore all the possible hanging behaviors, Maude’s search command is used (up to a specified depth) to explore all behaviors. It is also possible to use a language’s rewriting logic semantics as the basis for program refactoring, as shown for C in [202] and for Java in [201].

Regarding tools supporting rewriting-logic-based language definitions, besides the direct use of rewriting logic languages for this purpose and the K-Maude tool discussed in Section 6.2.2, the Maude MSOS tool [86] supports definition, execution and analysis of language definitions on the MSOS style. Also, tools to simulate and analyze CCS processes and LOTOS specifications based on their rewriting semantics are discussed in [103], Section 21.2.3. Deductive tools based on rewriting logic semantic definitions include the MatchC tool discussed in Section 6.2.3, and two Hoare logic provers built on top of the Maude ITP [116, 399]. Furthermore, the rewriting logic semantics of Java was used in [8] to automatically validate the semantics of a Java verification tool.

7.2.3. Hardware Specification and Verification

Prior to the use of rewriting logic, its equational logic subset (plus inductive principles) has been used for hardware specification and verification by various researchers, e.g., [209, 414, 241]. The earliest work I know on hardware specification and verification using Maude is by Neil Harman [221, 222]. Subsequent work has focused mostly on extending the rewriting logic semantics project from
the level of programming languages to that of hardware description languages (HDLs). In this way, hardware designs written in an HDL can be both simulated and analyzed using the executable rewriting semantics of the HDL and tools like ELAN, CafeOBJ, or Maude. The first HDL to be given a rewriting logic semantics in Maude was ABEL [243]; this semantics was used not only for hardware designs, but also for hardware/software co-designs. An important new development has been the use of the rewriting logic semantics of an HDL for generating sophisticated test inputs for hardware designs. The point is that random testing can catch a good number of design errors, but uncovering deeper errors after random testing is hard and costly and requires a good understanding of the design to exercise complex computation sequences. The key insight, due to Michael Katelman, is that the rewriting semantics can be used symbolically to generate desired test inputs, not on a device’s concrete states, but on states that are partly symbolic (contain logical variables) and partly concrete. Broadly speaking, this is an instance of the symbolic reachability analysis of rewrite theories I have discussed in Section 3.3; but for hardware verification the approach, first outlined in [246] and more fully developed in [245], has a number of unique features including: (i) the use of SAT solvers to symbolically solve Boolean constraints; (ii) support for user-guided random generation of partial instantiations; and (iii) a flexible strategy language, in which a hardware designer can specify in a declarative, high-level way the kind of test that needs to be generated. The effectiveness of this approach for generating sophisticated tests on real hardware designs has already been demonstrated for medium-sized Verilog designs [245]. The vlogsl tool is currently undergoing further enhancements to efficiently handle large designs.

But the value of the rewriting semantics of an HDL is not restricted to testing. For example, the recent Maude-based rewriting logic semantics of Verilog in [297] is arguably the most complete formal semantics to date, both in the sense of covering the largest subset of the language and in its faithful modeling of nondeterministic features. Besides being executable and supporting formal analysis, this semantics has uncovered several nontrivial bugs in various mature Verilog tools, and can serve as a practical and rigorous standard to ascertain what the correct behavior of such tools should be in complex cases.

A more exotic application of rewriting logic semantics, for which it is ideally suited due to its intrinsically concurrent nature, is that of asynchronous hardware designs. These are digital designs which do not have a global clock, so that different gates in a device can fire at different times. Such devices can behave correctly in much harsher environments (e.g., a satellite in outer space) and with much wider ranges of physical operating conditions, than clocked devices. Asynchronous designs can be specified with the notation of production rules, which roughly speaking describe how each gate behaves when inputs to its wires are available. In [242] a rewriting logic semantics of asynchronous digital devices specified as sets of production rules is given and is realized in Maude. This is the first executable formal semantics of such devices I am aware of. It can be used both for simulation purposes and for model checking verification of small-sized devices (about 100 gates). An interesting challenge is how to
scale up model checking for larger devices; this is nontrivial due to the large combinatorial explosion caused by their asynchronous behavior.

7.2.4. Networks, Distributed Architectures, Middleware and Coordination

Networks and network protocols are among the most basic distributed systems, on top of which other systems communicate. There is a long history of work on formal specification and verification of network protocols. Early work using rewriting logic in this area includes [127, 129, 292, 448]. What rewriting logic seems to be particularly good at is its support for distributed objects, which naturally describes network nodes, and its flexibility in handling many different network and communication models: in-order or out-of-order, link-based communication, broadcast, multicast and unicast, active networks, wireless communication, and so on; and to also handle naturally real time and probabilistic features. For example, to faithfully model wireless communication in a sensor network the geometry of the network, the varying power at each node, the time required for transmission, and the radius that a wireless message broadcast can travel without being lost depending on the power with which it is transmitted, all need to be modeled as done in [364]; likewise, probabilistic algorithms for sensor networks, modeling of packet contention, clock synchronization, and formal analysis by statistical model checking are all naturally handled in [247]. Network specifications and analyses have tackled not just single protocols, but composable collections of them in actual active network systems, where important design problems not revealed by standard testing have been uncovered [362].

In some cases, e.g., [362], the network protocols specified and analyzed in rewriting logic had already been implemented before the formal analysis was done; but the most useful application of these methods is before a protocol is implemented. The reason is obvious, although not always perceived by the unenlightened: it is much easier to debug a design expressed as a formal executable specification which can be very quickly specified and can then be subjected to exhaustive formal analysis, than it is to adopt the standard alternative of testing successive prototypes written in, say, C. Also, using formal executable specifications one can much more easily explore different design alternatives and get a better understanding of the design choices. Everybody knows that debugging distributed code is notoriously hard to do, but the brute force approach still remains a widespread, wasteful and unreliable way to develop protocols. One of the key contributions of [215] was to make exactly this point in a very thorough way by taking to heart the idea of using formal specification and model checking analysis in Maude to design a completely new protocol (L3A) and using this as a method to make the right choices between design alternatives and to fully debug the design. The beauty of it was that the subsequent implementation of L3A (reported in [216]) was essentially a transcribing of the executable Maude specification into imperative code, which was accomplished much faster and in a much more reliable way than if the formal analysis had not been done. In the words of one of the authors [214],
Maude modeling and analysis gave us a complete story of a model with proofs and an implementation that was really done from the Maude model. In essence, the debugging was done in Maude and we could focus on implementation and performance issues and not the correctness of the protocol.

For a similar detailed case study of using Maude to fully explore a protocol design (in this case one that was not implemented, precisely because of the complexities uncovered by the formal analysis) see [217]. Some of the above protocols, e.g., [127, 129, 215, 217], are security protocols. I discuss them from a security perspective, as well as other security protocols, in Section 7.3.

Besides networks themselves, different distributed architectures and middleware systems, and various distributed coordination and reflection mechanisms, have also been modeled and formally analyzed in rewriting logic. For example, there is work on formalizing different aspects of ODP [338, 161, 162, 149, 160, 391], SOAP [13], CORBA [14], and the SMEPP P2P middleware [148]. Similarly, work on formal models of coordination includes [82, 83, 429, 431]. Closely related to coordination models is work on formal models of distributed object reflection and adaptation [130, 326, 429, 250, 85]. For work on formal analysis of web applications and services using rewriting logic specifications see [15, 158].

7.3. Security

Security is a concern of great practical importance for many systems, making it worthwhile to subject system designs and implementations to rigorous formal analysis. Security, however, is many-faceted: on the one hand we are concerned with properties such as secrecy and authenticity: malicious attackers should not be able to get secret information or to falsely impersonate honest agents; on the other, we are also concerned with properties such as availability, which may be destroyed by a (DoS) attack: a highly reliable communication protocol ensuring secrecy may be rendered useless because it spends all its time checking spurious signatures generated by a DoS attacker. Furthermore, security concerns span many different levels and subsystems, such as network protocols, programming languages, browsers, web applications, operating systems, and hardware.

Rewriting logic has been successfully applied to analyze various security properties for a wide range of systems and at different levels of abstraction. Research in this general area includes: (i) work on cryptographic protocols; (ii) work on network security; (iii) work on browser security; (iv) work on access control, and (v) work on code security.

7.3.1. Cryptographic Protocol Specification and Analysis

The earliest work on the formal specification and analysis of cryptographic protocols in rewriting logic is by Denker, Meseguer, and Talcott [128, 129]. This stimulated further work by Rodriguez [389, 390], and inspired Millen and Denker to use Maude to give a formal semantics to their cryptographic protocol specification language CAPSL, and to endow CAPSL with an execution and formal analysis environment [131, 132, 133, 134]. In a similar vein, Cervesato, Stehr,
and Reich gave a rewriting logic semantics to the MSR security specification formalism, leading to the first executable environment for MSR [84, 377].

An important breakthrough was the realization that, by specifying a crypto protocol as a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, where $E \cup B$ describes the algebraic properties of the protocol’s cryptographic functions, and $R$ are the protocol rules, one could use narrowing with $R$ modulo the equations $E \cup B$ as a complete reachability analysis method (see Section 3.3). This was first pointed out in [327, 328]. This advance was crucial for two main reasons: (i) protocols could be analyzed modulo their algebraic properties $E \cup B$; it is well known (as already pointed in Section 6.2.4), that the traditional Dolev-Yao analysis treating cryptography as a “back box” is too weak, since protocols proved secure under the black box assumption can sometimes be broken by an attacker using the properties $E \cup B$; and (ii) by adopting a narrowing-based symbolic model checking approach, the fact that the number of protocol states, and even the number of protocol sessions, is unbounded does not preclude performing a complete analysis. Based on these ideas and on the rich experience about symbolic reachability methods in the NRL Protocol Analyzer [294], Santiago Escobar, Catherine Meadows and I have developed the Maude-NPA protocol analysis tool and its foundations [174, 177]. To the best of my knowledge the Maude-NPA is the most advanced analysis tool to date for analyzing cryptographic protocols modulo algebraic properties with an active intruder and an unbounded number of sessions in a complete way and without using any abstractions or approximations. For many protocols, Maude-NPA can exploit the fact that $E \cup B$ happens to enjoy the finite variant property to obtain a finitary $E \cup B$-unification algorithm by variant narrowing (see [184] and Section 3.3). But finitary algorithms for theories $E \cup B$ not having the finite variant property, e.g., homomorphic encryption, are also supported by Maude-NPA. In this way, we have formally analyzed protocols of the form $\mathcal{R} = (\Sigma, E \cup B, R)$, where $E \cup B$ can be a cryptographic theory involving a combination of functionalities such as: (i) encryption-decryption; (ii) bounded associativity; (iii) Diffie-Hellman exponentiation; (iv) exclusive-or; and (v) homomorphic encryption [171, 177, 175, 172, 173]. In general, of course, protocol analysis with an unbounded number of sessions is undecidable. However, thanks to Maude-NPA’s use of grammars [174] and of other state space reduction techniques [176, 178], a protocol’s symbolic state space can often become finite while remaining complete. This means that one can not only be sure to find attacks if they exist, but that one can sometimes also prove that the specified attacks are not possible modulo the algebraic properties $E \cup B$. Protocols are often compositions of smaller protocols, so that, even when the subprotocols are secure, unforeseen insecure interactions may take place in a composition. To support compositional reasoning in Maude-NPA, new composition constructs and associated analysis methods have been developed in [179].

7.3.2. Network Security

I have already discussed in Section 7.2.4 the usefulness for protocol design of the formal specifications in [215] and [217]. Since both specify network security
protocols, I briefly discuss them here from a security perspective. The work in [215] describes in detail the design steps, using Maude and its model checking formal analysis, to arrive at the design of L3A, an accounting protocol built on top of IPsec (using IPsec tunnels) to support billing which was subsequently implemented in [216]. One of the unique features of L3A is that it is resilient under cramming attacks, where a malicious attacker can direct traffic to a client for the purpose of having the client billed for the spurious traffic. The work in [217] uses Maude and its model checking features to explore and analyze a new protocol design called Sectrace. The problem addressed by Sectrace is the setting up of associations and policies assumed, but not provided, by the IPSec protocol in order to provide encryption and authentication services. Due to the presence of nested channels and concatenated channels involving several security gateways, setting up such security associations and policies is highly nontrivial. Indeed, the formal analysis uncovered quite complex issues, such as the fact that certain possibilities to set up correct security associations could be missed; and that concurrent runs of the protocol could cause undesirable interference effects. The design of Sectrace was not further advanced to resolve these issues, but the lessons learned were very valuable and could not have been learned without such kind of formal specification and analysis.

The work by Gutierrez-Nolasco et al. in [219] uses formal specification and verification in Maude to address a very important and real problem: how can the security requirements of a protocol be balanced with other equally important requirements such as timeliness or other QoS requirements? And how can a design be made adaptive, so that such a balancing can take place at runtime? This problem was addressed in the context of the Secure Spread group communication protocol [413], for which a formal specification in Maude had been previously developed. One problem with Secure Spread was its assumption of virtual synchrony (VS), which is more restrictive and expensive than the extended virtual synchrony (EVS) semantics. What the work in [219] accomplished was to extend the formal Maude specification of Secure Spread to a considerably more flexible and dynamically adaptive secure group communication protocol with two simultaneous dimensions of adaptation: (i) synchrony, which could be chosen to have the VS or EVS semantics; and (ii) group key security, where various levels of laziness in the key establishment protocol could be specified.

Regarding availability properties, a big problem in network security is Denial of Service (DoS) attacks, which are often distributed (DDoS) and employ many “bots,” i.e., large numbers of compromised machines from which a simultaneous attack is mounted. Two key questions are how to make network protocols resilient to DoS attacks, and how to formally analyze such resilience. A probabilistic approach to the formal specification and verification of DoS-resilient protocols is very natural for two reasons: (i) both the attacker models and the defense algorithms may be probabilistic; and (ii) the answers from a formal analysis will typically not be “true” or “false” answers; they will instead be numerical quality of service (QoS) answers, such as the expected latency for a client to get a response from a server during an attack of given intensity.
This means that probabilistic rewrite theories (see Section 3.10) and statistical model checking of qualitative properties in QuaTEx (see Section 3.11.1) are ideally suited for specification and analysis of DoS-resilient protocols. This is exactly the approach taken by Agha et al. in [5] to analyze the DoS-resilience of a hardening of the TCP/IP protocol by means of the Selective Verification (SV) probabilistic DoS-defence mechanism. This work has been later extended by AlTurki et al. [24] to the formal specification and verification of a more sophisticated DoS-defense protocol, namely, Adaptive Selective Verification (ASV), where both clients and servers ramp up or slow down their response to a DoS attack based on its perceived intensity. In his recent Ph.D. thesis [18], Musab AlTurki has modularized ASV as a meta-object wrapper that can be added to the objects of a distributed application without changing the application code; he has also extended the study of DoS defense mechanisms from simple client-server architectures to complex orchestrations of web services in Orc: he has shown how combinations of web services can be secured against DoS attacks by wrapping its distributed objects with ASV wrappers (for Orc and its rewriting logic semantics see Section 7.4.3).

However, neither the analysis nor the DoS-defense mechanisms need to be probabilistic. For example, in [410], Shankesi et al. give a formal specification in Maude of the VoIP Session Initiation Protocol, and of defense mechanisms against DoS Amplification attacks, and use LTL model checking in Maude with parametric predicates, which can actually measure performance metrics, to formally analyze the effectiveness of the specified defenses. Another DoS defense mechanism not involving probabilities is that of cookies. In [85], Chadha et al. propose a formal specification of the cookie-based DoS defense mechanisms as a modular wrapper, which can be composed with an underlying communication protocol without any modifications to the protocol’s code; and they prove that this modular approach preserves all the safety properties, for example secrecy properties, enjoyed by the underlying protocol. That is, the addition of this DoS defense can be made modular both at the code level and at the level of verifying safety properties, which need not be re-verified when the cookie wrappers are added.

### 7.3.3. Browser Security

To achieve end-to-end security, traditional machine-to-machine security measures are insufficient if the human-computer interface is compromised. This is particularly the case for browsers, where visual spoofing attacks that exploit GUI logic flaws can lure even security-conscious users to perform unintended actions. In [89], Shuo Chen, Ralf Sasse, Helen Wang, Yi-Min Wang and I called the preventing of such visual spoofing attacks “securing the last 20 inches.” That is, all the machine-to-machine protocols, code and hardware may be secure, but these visual attacks take place in the last 20 inches separating a user’s eyes from the screen where he/she is interacting with a browser using the browser’s GUI. Before we performed a rewriting-logic based formal analysis of Microsoft’s Internet Explorer (IE), it seems fair to say that the approach to IE security was basically reactive, i.e., each new attack was patched up, but there was no...
systematic way to predict and prevent future attacks. Based on an in-depth study of IE’s code, we developed a formal specification of IE (including a model of the user) in Maude as a rewrite theory. We then characterized status bar and address bar spoofing attacks as violations of visual invariants, where the web site that the user assumes he/she is interacting with is different from the real web site: what you see is not what you get. Our model-checking-based formal analysis uncovered nine status bar types of spoofing attacks and four address bar spoofing attack types that had not been previously mounted against IE. For each attack type, a malicious web page producing an actual attack could be built. The IE team then confirmed all these attack scenarios and proceeded to make IE secure for these new types of attacks.

This work stimulated new research by C. Grier, S. Tang and S. King at the University of Illinois at Urbana-Champaign. They asked the question: can we use rewriting logic not to uncover browser security flaws a posteriori, but to design a browser that is secure by construction? This question was answered in their paper [218], where they presented the design and implementation of the OP secure browser, whose design was specified in Maude and was subjected to model checking analysis to uncover design flaws. A more advanced browser variant of OP, OP2, as well as IBOS, a design and system implementation which integrates into a single architecture a secure browser and a secure operating system, are described in detail in Shuo Tang’s thesis [435], where Maude is again used to specify and verify by model checking the browser designs.

The security of web browsers is part of a bigger problem, namely, the security of web applications. In [15] Alpuente et al. give a rewriting semantics of web applications which formalizes the interactions between multiple browsers and a web server through a request/response protocol that supports the main features of HTTP and models browsers actions such as refresh, forward/backward navigation, and window/tab openings. Their formal model also supports a scripting language which abstracts the main common features (e.g. session data manipulation, data base interactions) of the most popular Web scripting languages and formalizes adaptive navigation, where page transitions may depend on users data or previous computation states of the Web application. They also show how the temporal logic of rewriting LTLR and its Maude-based model checker (see Section 3.11.1) is ideally suited to express and verify various safety and security properties of web applications specified this way.

7.3.4. Access Control

Access control policies specify the conditions under which access to information is permitted or should be denied in a system. They are a key security feature of many systems and apply way beyond the original setting of operating systems: enterprise systems, web-based systems, and even cloud computing applications all need and use access control policies. Such a policy is typically specified as a collection of access control rules. An important insight due to D. Dougherty et al. [140] is that access control rules can be formalized as rewrite rules. To further increase the expressive power of access control rules, the corresponding rewrite rules may be conditional, and they may be controlled by some
given strategy. This leads to the notion of rewrite-based access control policies, and to a corresponding notion of policy composition [140]. One important advantage of this rewriting-based formalization is that sophisticated forms of formal analysis about an access control policy become possible. C. Kirchner, H. Kirchner, and A. Santana de Oliveira show in [252] how narrowing-based analysis (see Section 3.3) with the rewrite rules formalizing an access control policy and following given strategies can provide an in-depth understanding of policies and their dynamic behavior to policy designers. Furthermore, since the rewrite rule formalization is directly executable and, using a language like Tom, can be automatically translated into Java code, the paper [123] shows how rewrite-based access control policies can be used to generate Java monitoring code for such policies. The monitoring code can then be automatically “weaved” with the application code it monitors using aspect-oriented methods.

7.3.5. Code Security

Many security attacks such as format string, heap corruption and buffer overflow involve malicious code performing pointer manipulations. The insight of Shuo Chen and his collaborators in [90] is that all these problems have a common cause that they call pointer taintedness, where a pointer is tainted whenever a user input can directly or indirectly be used as a pointer value. The formal approach taken in [90] is a good example of the general way of giving a rewriting logic semantics to a programming language already described in Sections 4.3 and 7.2.2. Indeed, what it is done in [90] is to give a rewriting semantics to a sequential programming language (since the language used is deterministic, only equations are needed) which includes a memory model. This formal model is then used to reason formally about pointer taintedness. This reasoning is applied to several library functions to extract security preconditions which guarantee the absence of pointer taintedness. In this way, various commonly occurring security vulnerabilities, such as format string, heap corruption and buffer overflow vulnerabilities can be both detected and prevented.

The topic of application level insider attacks, where a malicious insider tries to overwrite one or more data items in an application, has been systematically studied by Pattabiraman et al. in [368]. The application code is modeled at the assembly level by defining the rewriting logic semantics of assembly code. An insider attack is then represented as a corruption of data values at specific points in the program’s execution (called attack points). The behavior of an application code subjected to security attacks in the specified attack points is then formally modeled by replacing concrete values by appropriate symbolic values when attack points are reached; and by systematically modeling with rewrite rules the behaviors that such symbolic values can generate. Given the application code and its inputs, a set of attack points, and a goal state that the attacker intends to achieve, Maude is then used to generate a comprehensive set of insider attacks that lead to the goal state.

A very elegant application of a programming language’s abstract rewriting logic semantics (see Section 4.3) to Java code security is presented by Alba-Castro et al. in [11, 12] as part of their rewriting-logic-semantics-based ap-
proach to proof carrying code. The key idea is to use an abstract rewriting logic semantics of Java that correctly approximates security properties such as noninterference (that is, the specification of what objects should not have any effects on other objects according to a stated security policy [211]) and erasure (a security policy that mandates that secret data should be removed after its intended use). Since the abstract rewriting logic semantics is finite-state, it supports the automatic creation of certificates for noninterference and erasure properties of Java programs that are independently checkable and small enough to be used in practice.

Yet another code security application is M. LeMay and C. Gunter's verification of the security and fault-tolerance requirements of their Cumulative Attestation Kernel (CAK). This kernel runs on a flash MicroController Unit (MCU) as part of an advanced metering infrastructure for utilities in the Power Grid. For example, a meter for electricity consumption in a household or business will use such an MCU, connected to a communications network, to automatically gather and send power consumption data. Security threats include the installation of malware in the MCU to send false data. The CAK code protects the MCU against such attacks and also provides fault tolerance. The CAK's behavior has been fully specified as a rewrite theory in Maude, and Maude's LTL model checker has been used to verify that key security and fault tolerance requirements of the CAK are satisfied [270].

7.4. Real-Time and Cyber-Physical Systems

I have already mentioned in Section 3.11.2 that ELAN has been used to model check timed automata in [49]. Here I focus on the more general issue of specification and formal analysis of real-time and cyber physical systems which, by having arbitrary data structures in their discrete states, may not be specifiable at all as timed automata but have a natural specification as real-time rewrite theories (see Section 3.9). The best tool currently available to specify and analyze systems as real-time rewrite theories is Real-Time Maude (see Section 6.1.8). A wide range of applications have been specified and analyzed in Real-Time Maude including: (i) network protocols; (ii) middleware for distributed real-time systems; (iii) real-time programming languages; (iv) real-time modeling languages; (v) scheduling algorithms; and (vi) cyber-physical systems. Furthermore, in some cases the Real-Time Maude specifications have been used to easily derive actual system prototypes operating in physical time.

7.4.1. Real-Time Network Protocols

Because of their frequent use of timers, timeouts, roundtrip times, and so on; many network protocols (discussed already in Section 7.2.4) are in fact real-time systems. This means that their rewriting logic specification naturally takes the form of a real-time rewrite theory, and that their model checking analysis can best be performed by the kinds of real-time model checking supported by Real-Time Maude. Important network protocols that have been specified and have been thoroughly analyzed in Real-Time Maude include: (i) the AER/NCA suite
of active network protocols [347, 354, 362] already mentioned in Section 7.2.4; (ii) the NORM multicast protocol standard [274, 275]; and (iii) the OGDC wireless sensor network algorithm [438, 439]. This last work is quite unique, because it seems to be the first time that a sensor network was fully formally modeled in all its main aspects, such as geometry, power, transmission times, effective broadcast radius for each node, and so on; and because the formal analysis turned out to be more accurate and to uncover flaws in prior simulation-based analyses of OGCD. It is also noteworthy in terms of scalability, since a network of up to 600 nodes was modeled and analyzed. In fact, a sensor network is more than a network: it is a cyber-physical system, which in this work was fully modeled as such.

7.4.2. Middleware for Distributed Real-Time Systems

Many Distributed Real-Time Systems (DRTS), such as integrated modular avionics systems and distributed control systems in motor vehicles, are made up of a collection of components that communicate asynchronously and that must change their state and respond to environment inputs within hard real-time bounds. Such systems are often safety-critical and need to be certified; but their certification is currently very hard due to their distributed nature. The Physically Asynchronous Logically Synchronous (PALS) architectural pattern [329] can greatly reduce the design and verification complexities of achieving virtual synchrony in a DRTS. A key property that the PALS pattern should satisfy is to be provably correct-by-construction. This of course requires that the pattern itself should be formally specified as a parameterized construction. In [319, 318] Peter Ölveczky and I have used Real-Time Maude to specify PALS as a formal model transformation that maps a synchronous design, together with performance bounds of the underlying infrastructure, to a formal DRTS specification that is semantically equivalent to the synchronous design. This semantic equivalence is proved, showing that the formal verification of temporal logic properties of the DRTS can be reduced to their verification on the much simpler synchronous design. Furthermore, the PALS period is shown to be the shortest possible. The issue of how to mechanize PALS at the Maude metalevel, and an application of PALS to a wireless network protocol are discussed in [244].

7.4.3. Real-Time Programming Languages

How should the formal semantics of a real-time programming language be defined? And how can programs in such a language be formally analyzed? For an ordinary programming language, the rewriting logic semantics project answers the first question by saying: “with a rewrite theory,” and the second by saying: “by model checking and/or deductive reasoning based on such a theory.” The obvious answers for real-time programming languages are: (i) “with a real-time rewrite theory,” and (ii) “by real-time model checking and/or deductive reasoning based on such a theory.” Of course, the effectiveness of such answers has to be shown in actual languages. Three real-time programming languages have been given semantics in exactly this way, and their semantics has been used to verify their programs.
In [19], AlTurki et al. present a language for real-time concurrent programming for industrial use in DOCOMO Labs called L. The goal of L is to serve as a programming model for higher-level software specifications in SDL or UML. A related goal is to support formal analysis of L programs by both real-time model checking and static analysis, so that software design errors can be caught at design time. The way all this is accomplished is by giving a formal semantics to L in Real-Time Maude, which automatically provides an interpreter and a real-time model checker for L. Static analysis capabilities are added to L by using Maude to define an abstract semantics for L in rewriting logic, which is then used as the static analyzer.

As already mentioned in Section 4.2, the Orc model of real-time concurrent computation [330, 331, 456] has been given semantics in rewriting logic using real-time rewrite theories [20, 21]. Although Orc is a very simple and elegant language, its real-time semantics is quite subtle for two reasons. First, in the evaluation of any Orc expression, internal computation always has higher priority than the handling of external events; this means that, even without modeling time, a vanilla-flavored SOS semantics is not expressive enough to capture these different priorities: two SOS relations are needed [331]. Second, Orc is by design a real-time language, where time is a crucial feature. Using real-time rewrite theories, this double subtlety of the Orc semantics was faithfully captured by Musab AlTurki and I in [20]; furthermore, this semantics yielded of course an Orc interpreter and a real-time model checker. But Orc is not just a model of computation: it is also a concurrent programming language. This suggested the following challenge question: can a correct-by-construction distributed Orc implementation be derived from its rewriting logic semantics? This question was answered in two stages. Since, as discussed in Section 4.3, a small-step SOS semantics is typically horribly inefficient and it was certainly so in the case of Orc, a much more efficient reduction semantics was first defined in [21], and was proved to be bisimilar to the small-step SOS semantics. This semantics provided a much more efficient interpreter and model checker. Furthermore, to explicitly model different Orc clients and various web sites, and their message passing communication, the Orc semantics was seamlessly extended in [21] to a distributed object-based Orc semantics, which modeled what a distributed implementation should look like. The only remaining step was to pass from this model of a distributed implementation to an actual Maude-based distributed real-time implementation. This was accomplished in [22] using three main ideas: (i) the use of sockets in Maude to actually deploy a distributed implementation; (ii) the systematic replacement of logical time by physical time, supported by Ticker objects external to Maude, while retaining the rewriting semantics throughout; and (iii) the experimental estimation of the physical time required for “zero-time” Maude subcomputations, to ensure that the granularity of time ticks is such that all “instantaneous transitions” have already happened before the next tick. Ideas (i)–(iii) are of course much more widely applicable: they have subsequently been used to derive prototypes of real-time systems from their rewriting logic specifications for other applications such as medical devices, as explained in Section 7.4.6.
Creol is an object-oriented language supporting concurrent objects which communicate through asynchronous method calls. Its rewriting-logic-based operational semantics was defined in [237] without real-time features. However, to support applications such as sensor systems with wireless communication, where messages expire and may collide with each other, Creol’s design and operational semantics have been extended in [55] to Timed Creol using rewriting logic. The notion of time used by Timed Creol is described as a “lightweight” one in [55]. Time is discrete and is represented by a time object. This approach does not require a full use of the features in Real-Time Maude (Maude itself is sufficient to define the real-time semantics). The effectiveness of Timed Creol in the modeling and analysis of applications such as sensor networks is illustrated in [55] through a case study.

7.4.4. Real-Time Modeling Languages

The usefulness and importance of giving a formal a rewriting logic semantics to software modeling languages has already been discussed in Sections 4.4 and 7.2.1. In particular, there is strong interest in modeling languages for real-time and embedded systems. The rewriting logic semantics for such modeling languages can be naturally based on real-time rewrite theories. Using a tool like Real-Time Maude, what this means in practice is that such models can then be simulated; and that their formal properties, in particular their safety requirements, can be model checked. Furthermore, the simulations and formal analysis capabilities added to the given modeling language can be offered as “plugins” to already existing modeling tools, so that much of the formal analysis happens “under the hood,” and somebody already familiar with the given modeling notation can make use of such formal analysis without needing to have an in-depth understanding of the underlying formalism.

The Ptolemy II modeling language (http://ptolemy.eecs.berkeley.edu) supports design and simulation of concurrent, real-time, embedded systems expressed in several models of computation (MoCs), such as state machines, data flow, and discrete-event models, that govern the interaction between concurrent components. A user can visually design and simulate hierarchical models, which may combine different MoCs. Furthermore, Ptolemy II has code generation capabilities to translate models into other modeling or programming languages such as C or Java. Discrete-Event (DE) Models are among the most central in Ptolemy II. Their semantics is defined by the tagged signal model [269]. The work by Bae et al. in [40] endows DE models in Ptolemy II with formal analysis capabilities by: (i) defining a semantics for them as real-time rewrite theories; (ii) automating such a formal semantics as a model transformation using Ptolemy II’s code generation features; (iii) providing a Real-Time Maude plugin, so that Ptolemy II users can use an extended GUI to define temporal logic properties of their models in an intuitive syntax and can invoke Real-Time Maude from the GUI to model check their models. This work has been further advanced in [38] to support not just flat DE models, but hierarchical ones. That is, above tasks (i)–(iii) have been extended to hierarchical DE models; this extension is nontrivial, because it requires combining synchronous
fixpoint computations with hierarchical structure.

AADL (http://www.aadl.info/) is a standard for modeling embedded systems that is widely used in avionics and other safety-critical applications. However, AADL lacks a formal semantics, which severely limits both unambiguous communication among model developers and the formal analysis of AADL models. In [351] Ölveczky et al. define a formal object-based real-time concurrent semantics for a behavioral subset of AADL in rewriting logic, which includes the essential aspects of AADL’s behavior annex. Such a semantics is directly executable in Real-Time Maude and provides an AADL simulator and LTL model checking tool called AADL2Maude. AADL2Maude is integrated with OSATE, so that OSATE’s code generation facility is used to automatically transform AADL models into their corresponding Real-Time Maude specifications. Such transformed models can then be executed and model checked by Real-Time Maude. One difficulty with AADL models is that, by being made up of various hierarchical components that communicate asynchronously with each other, their model checking formal analysis can easily experience a combinatorial explosion. However, many such models express designs of distributed embedded systems which, while being asynchronous, should behave in a virtually synchronous way. This suggest the possibility of using the PALS pattern (see Section 7.4.2) to pass from simple synchronous systems, which have much smaller state spaces and are much easier to model check, to semantically equivalent asynchronous systems, which often cannot be directly model checked but can be verified indirectly through their synchronous counterparts. This has led to the design of the Synchronous AADL sublanguage in [39], where the user can specify synchronous AADL models by using a sublanguage of AADL with some special keywords. A synchronous rewriting semantics for such models has also been defined in [39]. Using OSATE’s code generation facility, synchronous AADL models can be transformed into their corresponding Real-Time Maude specifications in the SynchAADL2Maude tool, which is provided as a plugin to OSATE. Likewise, the user can define temporal logic properties of synchronous AADL models based on their features, without requiring knowledge of the underlying formalism, and can model check such models in Real-Time Maude.

A more ambitious goal is to provide a framework, where a wide range of real-time Domain-Specific Visual Languages (DSVLs), as well as their dynamic real-time behavior, can be specified with a rigorous semantics. This is precisely the goal of two frameworks and associated tools: (i) the e-Motions framework [381]; and (ii) MOMENT2’s support for real-time DSVLs [64].

- In e-Motions, DSVLs are specified by their corresponding metamodels, and dynamic behavior is specified by rules that define in-place model transformations. But the goals of e-Motions do not remain at the syntax/visual level; they also include giving a precise rewriting logic semantics in Real-Time Maude to the different real-time DSVLs that can be defined in e-Motions, and to automatically support simulation and formal analysis of models by using the underlying Real-Time Maude engine. The formal semantics translates the metamodel of a DSVL as an object class,
the corresponding models as object configurations of that class, and the e-Motions rules as rewrite rules. Since all these translations are automatic and define a DSVL’s formal semantics, a modeling language designer using e-Motions does not have to explicitly define the DSVL’s formal semantics: it comes for free, together with the simulation and model checking features, once the DSVL’s metamodel and the dynamic behavior rules are specified.

- In [64], the MOMENT2 framework (see Section 6.2.5) has been extended to support the formal specification and analysis of real-time model-based systems. This is achieved by means of a collection of built-in timed constructs for defining the timed behavior of such systems. Timed behavior is specified using in-place model transformations. Furthermore, the formal semantics of a timed behavioral specification in MOMENT2 is given by a corresponding real-time rewrite theory. In this way, models can be simulated and model checked using MOMENT2’s Maude-based analysis tools. In addition, by using in-place multi-domain model transformations in MOMENT2, an existing model-based system can be extended with timed features in a non-intrusive way, in the sense that no modification is needed for the class diagram.

7.4.5. Resource Sharing Protocols

Real-time resource sharing protocols are protocols governing the way in which multiple tasks can share common resources such as a data structure, a memory area, a file, a set of registers in a peripheral device, and so on. The dynamic behavior of such protocols divides naturally into a scheduling part, and a resource access part. Although this is a very well-established area, the emergence of multicore machines has brought about new protocols and more sophisticated approaches, for which correctness is not obvious, so that formal modeling and analysis can be a valuable design methodology. The first work applying rewriting logic in this area was by P. Őlveczky and M. Caccamo, who modeled and analyzed in Real-Time Maude the CASH capacity sharing scheduling algorithm [352], corresponding to the scheduling part of a resource sharing protocol. Search analysis of CASH’s Real-Time Maude specification uncovered a previously unknown behavior that led to missed deadlines. This was a subtle error that it would have been virtually impossible to detect by testing. Indeed, extensive Monte-Carlo simulation was utterly incapable of detecting the flaw. The CASH protocol furthermore illustrated a broad class of applications beyond the pale of (timed) automata-based analysis techniques. The point is that model checking algorithms for such techniques work only for finite-state real-time systems, but the Real-Time Maude formal analysis showed that the queues in the state of the CASH protocol could grow in an unbounded manner.

A broader framework for formally modeling and analyzing real-time resource sharing protocols, in both their scheduling and resource access parts, is presented by P. Őlveczky, P. Prabhakar and X. Liu in [363]. In particular, [363] shows how crucial properties such as: (i) unbounded priority inversion; (ii) dead-
locks; and (iii) schedulability, can be analyzed for such protocols when they are specified as real-time rewrite theories. The effectiveness of this framework is illustrated by means of the analysis of the priority inheritance protocol (PIP).

7.4.6. Cyber-Physical Systems

Cyber-physical systems are real-time systems, often distributed, which interact with the physical world by sensing and possibly by means of actuators. A number of such systems have been specified and modeled in Real-Time Maude. One example is the OGCD wireless sensor network algorithm in [438, 439] already described in Section 7.4.1. Another example is the family of traffic system designs specified and analyzed in [361], where one of the experiences gained was the ease with which the use of distributed objects and class inheritance provided a very high degree of genericity and extensibility of the different designs (including European and American light regimes, a special regime for emergency vehicles, and so on), and allowed for a distributed control without any need for a centralized controller. A third example is the modeling in Real-Time Maude of object-oriented real-time systems that follow the Actor model, and the application of this modeling style to the specification and analysis of the simplex architecture, a software architecture for for fault-tolerant real-time control systems. Yet a fourth example is the use of Real-Time Maude to analyze embedded code in a Japanese car design; the analysis uncovered flaws in the embedded code but has not been published for proprietary reasons.

The safe interoperation of medical devices has been the topic of several research papers, which have formally modeled and analyzed various device configurations in Real-Time Maude. For example, in [348] P. Olveczky describes the application of Real-Time Maude to the formal modeling and analysis of a network integrating an X-ray machine, a ventilator, and a controller. This configuration automates a similar manual interoperation between an X-ray machine and a ventilator for which an accidental death in an operating room was reported in the literature. As part of the formal specification and analysis, [348] introduces novel techniques for: (i) modeling nondeterministic transmission delays while maintaining completeness and reasonable performance of the analysis; (ii) modeling clock drifts; and (iii) analyzing bounded response properties. Subsequent work by M. Sun, J. Meseguer, and L. Sha in [425] has focused on the development of patterns for interoperation of medical devices (among themselves and with a patient) that are safe by construction, and generic, so that they can be instantiated for many different devices. Specifically, one such pattern, called the Command-Shaper pattern, is formally specified as a parameterized Real-Time Maude module and proved correct in [425]. The key idea of the Command-Shaper is to intercept the commands from external devices (possibly including the patient), so that the patient is never placed in a medically dangerous state, including states where the patient’s medical constants may be stressed for a dangerously long time. Instances of the Command-Shaper pattern include a mechanism for enforcing that a sophisticated pacemaker, which can adapt to changes in the patient’s activity, will never place the patient’s heart in stressful situations, and a patient-operated infusion pump for mor-
phine. As already pointed out for the Orc orchestration language in Section 7.4.3, Real-Time Maude specifications of distributed real-time systems can be easily transformed into distributed real-time implementations using Maude’s socket mechanism. For the Command-Shaper pattern this has been done by M. Sun and J. Meseguer in [424]. One attractive feature of this transformation is that formal specifications can be interoperated with actual physical devices in a system that emulates a final implementation.

In connection with the PALS pattern discussed in Section 7.4.2, Peter Olveczky and I have specified in Real-Time Maude synchronous and asynchronous versions of an active standby avionics system [319, 318], and, using the synchronous version plus its bisimulation equivalence with the asynchronous one, have verified by model checking that it satisfies (appropriately enhanced versions of) all the informal requirements listed by the designers. This example underscores the power and usefulness of the PALS pattern, since the synchronous version had just a few hundred states and each property was model checked in less than 0.8 seconds, whereas the simplest possible asynchronous version (with no message delays) had over 3 million states.

7.5. Probabilistic Systems

Probabilistic rewrite theories (see Section 3.10) can model a wide variety of probabilistic systems, including many cyber-physical systems. As already mentioned, both the environments in which such systems operate and the very algorithms they use are often probabilistic. Furthermore, the verification of their quantitative properties may be just as important as that of Boolean-valued properties such as safety requirements. For this purpose, one can use statistical model checking methods (see Section 3.11.2) of quantitative properties expressed in a formalism such as QuaTEx (see Section 3.11.1). As the PVeStA tool demonstrates, such statistical model checking analyses can be quite scalable (see Section 6.1.10).

Up to now, the probabilistic system applications that have been specified and analyzed using the just-mentioned methods fall into three areas: (i) DoS-resistant protocols; (ii) distributed embedded systems; and (iii) distributed stochastic hybrid systems. There are of course many other possibilities, including applications for the quite different notion of probabilistic rewriting proposed in [73, 71] and discussed in Section 3.10. Since DoS-resistant protocols have already been discussed in Section 7.3.2, I focus here on areas (ii) and (iii).

7.5.1. Distributed Embedded Systems

For many distributed embedded systems, particularly those including energy-constrained components such as hand-held devices, Quality of Service (QoS) properties are essential. For achieving such properties in an end-to-end manner, adaptive resource management policies across different layers of the system, such as the application, middleware, and OS layers, are needed. M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian have used probabilistic rewrite theories specified in Maude, and statistical model checking analysis
of quantitative properties of such theories (using the algorithm described in [250]), to model and formally analyze various sophisticated adaptive designs of distributed embedded systems that can provide desired QoS guarantees. Their general methodology is presented in [250], where it is applied to a multi-mode multimedia case study. Furthermore, in [248] they show how these methods can be combined with direct observation of system executions to refine the probabilistic models of the system, and how this can be used to achieve system adaptation under timing constraints by iteratively tuning system parameters. This line of research is continued in [249], where they present a compositional method for cross-layer system optimization based on a constraint refinement technique which can be used to fine tune system parameters in a compositional manner, allowing coordinated interaction among sub-layer optimizers to achieve cross-layer optimization. Experiments on a realistic multimedia application demonstrate that constraint refinement can generate robust and near optimal parameter settings.

An important class of energy-constrained distributed embedded systems is that of wireless sensor networks, since the power of the sensors must be used very carefully to ensure an acceptable network lifetime. In [247], Michael Katelman, the late Jennifer Hou and I used probabilistic rewrite theories and qualitative analysis in VeStA to study in depth and under realistic conditions the design of the local minimum spanning tree (LMST) topology control protocol, which tries to maintain connectivity in an ad-hoc wireless sensor network while minimizing power consumption and maximizing data bandwidth. Our starting point was an idealized LMST design with perfect clocks and perfect communication, which did in fact maintain connectivity at an abstract level. However, our formal analysis revealed that, as soon as more realistic implementation details such as clock synchronization and network contention were introduced, the idealized LMST design failed rather badly to maintain network connectivity. The problem we then addressed was how to use probabilistic modeling and statistical model checking to redesign LMST at a realistic level, so that it would meet its intended goals. For this purpose we developed a system redesign methodology supporting three mutually-reinforcing tasks: (i) to uncover flaws in a given design; (ii) to conjecture the causes of the various malfunctions and to confirm such conjectures by means of statistical correlations between further analyses; and (iii) to then use the confirmed conjectures of the hypothesized causes of flaws to redesign a system and verify by statistical model checking that the final design satisfies the desired requirements. Our application of this methodology to LMST resulted in a new, implementable design that satisfied all the desired requirements under realistic operating conditions.

7.5.2. Distributed Stochastic Hybrid Systems

Stochastic hybrid systems generalize ordinary hybrid systems by allowing continuous evolution to be governed by stochastic differential equations (SDEs) and/or by allowing instantaneous changes in system modes to be probabilistic. This fits well the intrinsic uncertainty of the environments in which many hybrid systems must operate, and is also very useful when some of the systems algorithms are probabilistic. Indeed, there is a wide range of application
areas, including communication networks, air traffic control, economics, fault tolerant control, and bioinformatics. However, in practice many stochastic hybrid systems are not autonomous: they are distributed as collections of objects that communicate with other objects by exchanging messages through an asynchronous medium such as a network. In [324], Raman Sharikin and I used probabilistic rewrite theories to investigate several open issues such as: (i) how to compositionally specify distributed object-based stochastic hybrid systems; (ii) how to formally model them, and (iii) how to verify their properties. Specifically, in [324] we addressed these issues by: (i) defining a mathematical model for such systems; (ii) proposing a formal specification language in which system transitions are specified in a modular way by probabilistic rewrite rules; and (iii) showing how these systems can be subjected to statistical model checking analysis to verify their probabilistic temporal logic properties. Maude and VeStA were used to illustrate the approach with specific examples such as: (i) an international auction system in which bidders reside in different countries and their different currencies fluctuate according to an SDE; and (ii) a system consisting of \( N \) rooms, each equipped with a thermostat, plus a central server unit controlling them, where each thermostat can be in either heating, cooling, or idle mode, and the temperature in each room changes randomly according to an SDE.

In Section 7.6.1 I discuss another very useful application of probabilistic rewriting to the modeling of biological systems as stochastic hybrid systems [2].

7.6. Bioinformatics, Chemical Systems, and Membranes

I discuss here several related research strands where rewriting logic has been applied to bioinformatics, to modeling the dynamics of chemical systems, and to chemically and biologically inspired membrane systems.

7.6.1. Bioinformatics

Biology lacks at present adequate mathematical models that can provide something analogous to the analytic and predictive power that mathematical models provide for, say, Physics. Of course, the mathematical models of Chemistry describing, say, molecular structures are still applicable to biochemistry. The problem is that they do not scale up to something like a cell, because they are too low-level. One can of course model biological phenomena at different levels of abstraction. Higher, more abstract levels seem both the most crucial and the least supported. The most abstract the level, the better the chances to scale up.

All this is analogous to the use of different levels of abstraction to model digital systems. There are great scaling up advantages in treating digital systems and computer designs at a discrete level of abstraction, above the continuous level provided by differential equations, or, even lower, the quantum electrodynamics (QED) level. The discrete models, when they can be had, can also be more robust and predictable: there is greater difficulty in predicting the behavior of a system that can only be modeled at lower levels. Indeed, the level
at which biologists like to reason about cell behavior is typically the discrete level; however, at present descriptions at this level consist of semi-formal notations for the elementary reactions, together with informal and potentially ambiguous notations for things like pathways, cycles, feedback, etc. Furthermore, such notations are static and therefore offer little predictive power. What are needed are new computable mathematical models of cell biology that are at a high enough level of abstraction so that they fit biologist’s intuitions, make those intuitions mathematically precise, and provide biologists with the predictive power of mathematical models, so that the consequences of their hypotheses and theories can be analyzed, and can then suggest laboratory experiments to prove them or disprove them.

As first pointed out in [166], and vigorously developed in the subsequent Pathway Logic research which I discuss later, rewriting logic seems ideally suited for this task. The basic idea is that we can model a cell as a concurrent system whose concurrent transitions are precisely its biochemical reactions. In fact, the chemical notation for a reaction like $AB \rightarrow CD$ is exactly a rewriting notation. In this way we can develop symbolic bioinformatic models which we can then analyze in their dynamic behavior just as we would analyze any other rewrite theory.

Implicit in the view of modeling a cell as a rewrite theory $(\Sigma, E, R)$ is the idea of modeling the cell states as elements of an algebraic data type specified by $(\Sigma, E)$. This can of course be done at different levels of abstraction. We can for example introduce basic sorts such as AminoAcid, Protein, and DNA and declare the most basic building blocks as constants of the appropriate sort. For example,

\begin{verbatim}
ops T U Y S K P : -> AminoAcid .
ops 14-3-3 cdc37 GTP Hsp90 Raf1 Ras : -> Protein .
\end{verbatim}

But sometimes a protein is modified, for example by one of its component amino acids being phosphorylated at a particular site in its structure. Consider for example the c-Raf protein, denoted above by Raf1. Two of its $S$ amino acid components can be phosphorylated at sites, say, 259 and 261. We then obtain a modified protein that we denote by the symbolic expression,

$[Raf1 \ \text{phos}(S\ 259) \ \text{phos}(S\ 621)]$

A fragment, relevant for this example, of the signature $\Sigma$ needed to symbolically express and analyze such modified proteins is given by the following sorts, subsorts, and operators:

\begin{verbatim}
sorts Site Modification ModSet .
subsort Modification < ModSet .

op phos : Site -> Modification .
op none : -> ModSet .
op __ : ModSet ModSet -> ModSet [assoc comm id: none] .
op __ : AminoAcid MachineInt -> Site .
op [_\ldots_] : Protein ModSet -> Protein [right id: none] .
\end{verbatim}
Proteins can stick together to form *complexes*. This can be modeled by the following subsort and operator declarations:

```plaintext
sort Complex .
subsort Protein < Complex .
```

In the cell, proteins and other molecules exist in “soups,” such as the cytosol, or the soups of proteins inside the cell and nucleus membranes, or the soup inside the nucleus. All these soups, as well as the “structured soups” making up the different structures of the cell, can be modeled by the following fragment of sort, subsort, and operator declarations,

```plaintext
sort Soup .
subsort Complex < Soup .
op __ : Soup Soup -> Soup [assoc comm] .
op cell{_{_}} : Soup Soup -> Soup .
op nucl{_{_}} : Soup Soup -> Soup .
```

that is, soups are made up out of complexes, including individual proteins, by means of the above binary “soup union” operator (with juxtaposition syntax) that combines two soups into a bigger soup. This union operator models the fluid nature of soups by obeying *associative* and *commutative* laws. A *cell* is then a *structured soup*, composed by the above *cell* operator out of two subsoups, namely the soup in the membrane, and that inside the membrane; but this second soup is itself also structured by the cytoplasm and the nucleus. Finally, the nucleus itself is made up of two soups, namely that in the nucleus membrane, and that inside the nucleus, which are composed using the above *nucl* operator. Then, the following expression gives a partial description of a cell:

```plaintext
cell{cm (Ras : GTP) {cyto
  (([Raf1 \ phos(S 259)phos(S 621)] : (cdc37 : Hsp90)) : 14-3-3)
  nucl{n(m(n))}}}
```

where *cm* denotes the rest of the soup in the cell membrane, *cyto* denotes the rest of the soup in the cytoplasm, and *nm* and *n* likewise denote the remaining soups in the nucleus membrane and inside the nucleus.

Once we have cell states defined as elements of an algebraic data type specified by \((\Sigma, E)\), the only missing information has to do with cell *dynamics*, that is, with its biochemical reactions. They can be modeled by suitable rewrite rules \(R\), giving us a full model \((\Sigma, E, R)\). Consider, for example, the following reaction described in a survey by Kolch [260]:

> “Raf-1 resides in the cytosol, tied into an inactive state by the binding of a 14-3-3 dimer to phosphoesterines-259 and -621. When activation ensues, Ras-GTP binding ... brings Raf-1 to the membrane.”

We can model this reaction by the following rewrite rule:
where CM and CY are variables of sort Soup, representing, respectively, the rest of the soup in the cell membrane, and the rest of the soup inside the cell (including the nucleus). Note that in the new state of the cell represented by the righthand side of the rule, the complex has indeed migrated to the membrane.

Given a type of cell specified as a rewrite theory \((\Sigma, E, R)\), rewriting logic then allows us to reason about the complex changes that are possible in the system, given the basic changes specified by \(R\). That is, we can then use \((\Sigma, E, R)\) together with Maude and its supporting formal tools to simulate, study, and analyze cell dynamics. In particular, we can study in this way biological pathways, that is, complex processes involving chains of biological reactions and leading to important cell changes. In particular we can:

- observe progress in time of the cell state by symbolic simulation, obtaining a corresponding trace;
- answer questions of reachability from a given cell state to another state satisfying some property; this can be done both forwards and backwards;
- answer more complex questions by model checking LTL properties; and
- do meta-analysis of proposed models of the cell to weed out spurious conjectures and to identify consequences of a given model that could be settled by experimentation.

Since the first paper in this direction [166], on which the above summary is based, this line of research has been vigorously advanced by the Pathway Logic (PL) team of computer scientists and molecular biologists at SRI led by Carolyn Talcott [167, 427, 433, 434, 426, 430, 2, 442, 432, 441] (for a good overview, see Talcott’s tutorial [432]). The PL researchers have used rewriting logic to develop sophisticated analyses of cell behavior in biological pathways, and have built useful notations and visualization tools, such as the Pathway Logic Assistant [433], that can represent the Maude-based analyses in forms more familiar to biologists. The papers [427, 434] contain good discussions of related work in this area, using other formalisms, such as Petri nets or process calculi, that can also be understood as particular rewrite theories; and show how cell behavior can be modeled with rewrite rules and can be analyzed at different levels of abstraction, and even across such levels. A very exciting more recent development is the use of several probabilistic rewriting methods to model cell behaviors as stochastic hybrid systems [2]. Yet another very exciting development is the use of rewriting logic in neuroinformatics, at a much higher level of abstraction than that of reactions in molecular biology. What are now modeled are neural systems, with neurons as objects, in the O-O sense,
plus what might be called “wiring information” about neuron interconnections. Changes in neuron states due to firings are then described by rewrite rules. A Maude model of the neural system responsible for the feeding behavior of the marine mollusk *Aplysia* has been used to model quite accurately Aplysia’s neural behavior in a way consistent with other studies [2]; furthermore, using symbolic model checking, more ambitious properties of Aplysia’s neural behavior have been verified in [441]. In general, one of the important contributions of the PL project is the combination of various modeling and analysis techniques to model biological systems; in addition to all the already-mentioned techniques, SAT-solving is yet one more weapon in PL’s arsenal [442].

The PL research has stimulated the use of rewriting logic and Maude by other bioinformatics researchers. For example, M.G. Sriram has used Maude to model protein functional domains in signal transduction, and to obtain testable hypotheses at various levels of abstraction [412], and, in work not yet published, my UIUC colleague Thomas Anastasio has used Maude to analyze and obtain useful hypotheses about biological pathways whose malfunction is related to Alzheimer’s disease.

Although the research by O. Andrei and H. Kirchner in [29] makes also valuable contributions to the bioinformatics applications of rewriting logic, I discuss it in the next section because of its similarities with other work on chemical systems.

### 7.6.2. Chemical Systems

The already-mentioned fact that the chemical notation for a reaction like $AB \rightarrow CD$ is a rewriting notation suggests that rewrite theories can be used to symbolically model not just cell biology but *any* chemical systems, with the reactions modeled as rewrite rules. This is exactly the research approach taken by O. Bournez et al. in [70], and further developed by O. Bournez, L. Ibanescu and H. Kirchner in [72], and by O. Andrei, L. Ibanescu and H. Kirchner in [28]. This research makes a number of novel contributions. First of all, it emphasizes the fact that chemical compounds are *graphs*, so that chemical reactions can be more properly modeled as graph rewrite rules. Second, it identifies an appropriate *term representation* for chemical graphs so that: (i) equivalent representations can be effectively identified; (ii) “soups” of different chemical compounds can be represented as multisets by an AC operator; and (iii) the graph rewriting modeling of chemical reactions can be faithfully represented as term rewriting modulo AC. In particular, the paper [72] provides a detailed study of this dual graph/term representation and proves the faithfulness of the associated term rewriting in capturing the desired graph rewriting. A third contribution is the use of *strategies* to characterize chemical processes, which do no correspond to arbitrary sequences of rewrites, but obey certain dynamic constraints. A fourth contribution is the implementation of all these ideas in the *GasEl* system, first implemented in ELAN in [70], but subsequently implemented in TOM for enhanced efficiency, as reported in [28].

The already-mentioned work by O. Andrei and H. Kirchner in [29], although belonging to the more specific area of biochemistry and bioinformatics applica-
tions —indeed, to the modeling of biochemical networks— has a some similarities with the just-mentioned work on chemical modeling, but makes different contributions. It models the molecular complexes appearing in cell biology as *labeled multigraphs with ports*, with molecules represented as nodes, sites as ports, and bonds as edges. Biochemical reactions are then modeled as graph transformation rules; and biochemical networks are finally modeled as strategies which express the appropriate control between the different reactions and the dynamic evolution of molecular complexes. In analogy with [72], careful attention is paid to finding a faithful *term representation*, that is, a faithful representation as an (order-sorted) rewrite theory of the corresponding graphs and graph transformation rules associated to a given biochemical network.

### 7.6.3. Membrane Systems

Transfer of ideas can sometimes go in both directions. Not only can rewriting logic provide formal models for cell biology and bioinformatics, but *chemical and biological metaphors* may suggest models of computation. Indeed, chemical metaphors understood as mutiset rewriting —so that a mutiset of entities is visualized as a chemical “soup,” and atomic computation steps as chemical reactions— go back to the *Gamma* model of computation of J.-P. Banâtre and D. Le Métayer [45], which inspired the Chemical Abstract Machine (CHAM) of G. Berry and G. Boudol [52]. A further development of this line of research has been the study of *membrane systems* in the sense of O. Andrei, G. Ciobanu, and D. Lucanu [27], who base their ideas on the cell-inspired proposal of membrane computing by Gh. Paun [370]. The basic idea is that membrane systems are hierarchical systems consisting of nested cells, each surrounded by a membrane enclosing a mutiset of elements, which may include other cells. This bears some similarities to the Meseguer-Talcott “Russian Dolls” model of distributed object-oriented reflection [326] already mentioned in Section 7.2.4. Another important idea is that rules describing local changes in a membrane system have *priorities*, and that *maximal parallelism* is the desired model of computation. A careful study of all these issues within the rewriting logic framework has been presented in [27]. The issue of maximal parallelism using the idea of “promoters and inhibitors” is further studied by O. Agrigoraie and G. Ciobanu in [7]. Of course, since rules in membrane systems have priorities and should fire with maximal parallelism, not all rewriting computations are desirable; this leads to the issue of characterizing membrane computations by appropriate rewriting strategies, a topic studied by O. Andrei and D. Lucanu in [30], and by D. Lucanu in [277].

### 8. Some Future Research Directions

Of course, all the research areas already discussed are promising future directions. The question is rather, which new or recent areas seem most in need of development and look particularly promising? Answers to such questions are necessarily subjective, and can only be *guesses*. In fact, the emergence of other
areas which one has not anticipated should be a cause for rejoicement. With that said, here are some directions I think need development and are promising:

1. *Rewriting Logic as a New Paradigm for Declarative Concurrent Programming*, as well as new multicore and distributed rewriting logic language implementations.

2. *Advancing the Rewriting Logic Semantics Project*, including future advances in K, Matching Logic, and Compiler Generation from language definitions.


5. *New Verification Methods and Tools for Probabilistic Rewrite Theories*, including languages, verification methods, and tools.

9. Conclusions

In the introduction I raised the following questions about rewriting logic:

- How well-developed are its mathematical foundations?
- To what extent have its goals as a semantic framework for concurrency, and as a logical framework, been achieved?
- Which languages and tools supporting rewriting logic programming, specification, and verification have been developed?
- In which application areas has it been shown useful?
- How do its future prospects look like?

I believe that I have given quite extensive answers to all these questions, except perhaps for a briefer treatment of the last one on future prospects. The foundations are in my mind rock-solid. At this point the wide range of models of concurrency and of logics that have been *naturally* expressed within the rewriting logic framework provides overwhelming evidence that it is a very suitable framework. The languages supporting rewriting logic are mature, provide many features, and are furthermore still growing. The spectrum of formal tools is quite adequate, although more advances are and will be happening. And the range of applications is quite wide and exciting. I think some of us will be busy pushing the envelope for years to come; and I hope this survey will encourage other researchers to use rewriting logic in their own work and to make new contributions.
Acknowledgements. I thank the organizers of WRLA 2010 for giving me the opportunity and the stimulus to do some reflecting, surveying, and guessing about rewriting logic at this point, when twenty years have passed since the first papers were published, for their further encouragement to turn my WRLA lecture into a survey paper, and for their patience in allowing me the necessary time to finish it. As already mentioned, I feel a debt of gratitude to the many gifted researchers who have made important contributions to the rewriting logic research program. This work has been supported in part by NSF Grants CNS 07-16638, CNS 08-34709, CNS 08-31064, CNS 09-04749, and CCF 09-05584.


[133] G. Denker and J. Millen. CAPSL integrated protocol environment. In Koob et al. [261], pages 207–222.


[252] C. Kirchner, H. Kirchner, and A. S. de Oliveira. Analysis of rewrite-based access control policies. In Dougherty and Escobar [139], pages 55–75.


112


P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In Bonsangue and Johnsen [56], pages 122–140.


P. Viry. Input/output for ELAN. In Meseguer [305], pages 51–64.

P. Viry. Adventures in sequent calculus modulo equations. In Kirchner and Kirchner [251], pages 21–32.


