The Rewriting Logic Semantics Project and its Maude Implementation

José Meseguer

University of Illinois at Urbana-Champaign

Summer School on Language Frameworks, Sinaia, July 2012

The Rewriting Logic Semantics (RLS) Project

Goal: use rewriting logic as a framework for formally defining and analyzing programming languages.

The RLS project was started in 2003 by Meseguer and Roșu, but there are many other people involved. Here is a short list:

Wolfgang Ahrendt, Musab Al-Turki, Marcelo d'Amorim, Eyvind W. Axelsen, Christiano Braga, Illiano Cervesato, Fabricio Chalub, Feng Chen, Manuel Clavel, Chucky Ellison, Azadeh Farzan, Alejandra Garrido, Mark Hills, Einar Broch Johnsen, Ralph Johnson, Michael Katelman, Narciso Marti-Oliet, Patrick Meredith, Olaf Owe, Stefan Reich, Andreas Roth, Juan Santa-Cruz, Ralf Sasse, Koushik Sen, Andrei Ştefănescu, Mark-Oliver Stehr, Carolyn Talcott, Prasanna Thati, Ram Prasad Venkatesan, Alberto Verdejo ...

Motivation

Given language $\mathcal{L}$, there is a substantial gap between:

1. Formal semantics for $\mathcal{L}$
2. Implementation of $\mathcal{L}$
3. Analysis tools for $\mathcal{L}$

Even if a formal semantics exists for $\mathcal{L}$, there may not be any formal semantics available at the higher level of software designs and models, or at the lower level of hardware.

The Rewriting Logic Semantics Approach

Rewriting logic semantics is a wide-spectrum framework, where:

1. The formal semantics of $\mathcal{L}$ is given as an executable rewrite theory and is used as the basis on which both language implementations and language analysis tools are built.
2. The same semantics-based approach is used not just for programming languages, but also for software and hardware modeling languages.

The RLS approach has proved to be expressive, scalable and, using Maude's implementation of rewriting logic, quite efficient.

Rewriting Logic in a Nutshell

Rewriting logic is a flexible logical framework to specify concurrent systems.

- A concurrent system specified as rewrite theory $\mathcal{R} = (\Sigma, E, R)$
  - Σ signature defining the syntax of the system and of its state
  - E equations defining system's states as an algebraic data type
  - R set of rewrite rules of the form $t \longrightarrow t'$, specifying system's local concurrent transitions.
- Rewriting logic deduction consists of applying rewriting rules R concurrently, modulo the equations E.

Maude is a rewrite engine capable of efficiently executing rewriting logic theories. Maude additionally provides a series of formal analysis analysis tools.

Maude in a Nutshell

Maude is a high-performance language and system whose modules are either:

- Equational theories, called functional modules, of the form `fmod` $(\Sigma, E \cup A)$ `endfm`, where E are (possibly conditional) confluent equations modulo axioms A of associativity and/or commutativity and/or identity, or
- Rewrite theories, called system modules, of the form `fmod` $(\Sigma, E \cup A, R)$ `endfm`, where the equations E are confluent modulo A, and the (possibly conditional) rules R are coherent with E modulo A.

Maude specifications can be: (i) executed, (ii) model checked with the `search` command and with Maude's LTL model checker; and (iii) formally analyzed by Maude formal tools such as the CRC, ChC, MTT, SCC, ITP, and InvA tools.

Defining Programming Languages in Rewriting Logic

Define a concurrent language $\mathcal{L}$ as a rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$

- $\Sigma_{\mathcal{L}}$ 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;
- $E_{\mathcal{L}}$ give semantics to the deterministic features of $\mathcal{L}$;
- $R_{\mathcal{L}}$ give semantics to the concurrent features of $\mathcal{L}$.

Then one can use Maude and its formal analysis tools to obtain executable models and formal analysis tools for the defined language.

This scales up to realistic languages, such as Java, the JVM, Scheme, C, Verilog, etc.

Operational vs. Denotational Semantics

Rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ yields both

- An operational semantics by means of rewriting logic deduction as explained above; and
- A denotational semantics, by means of its initial model.

More precisely, the derivation proof trees associated to rewriting logic deduction formally capture the operational semantics of the defined language.

In the tradition of initial model semantics, there is a unique morphism from the initial model of $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ into any model.

The initial model serves as the canonical denotational model of the defined language.

The Abstraction Dial

Given language $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, we can vary its abstractness by moving sentences between the sets $E_{\mathcal{L}}$ and $R_{\mathcal{L}}$:

- Turning equations in $E_{\mathcal{L}}$ into rules in $R_{\mathcal{L}}$ will make the semantics more concrete, making $E_{\mathcal{L}}$-steps visible in the corresponding transition system of the program.
- Turning rules in $R_{\mathcal{L}}$ into equations in $E_{\mathcal{L}}$ will make the semantics more abstract, making $R_{\mathcal{L}}$-steps invisible in the transition system.

Turning equations into rules increases the state space, while turning rules into equations reduces the state space.

Abstraction is desirable to reduce large state spaces to analyzable sizes. However, not all rules can be turned into equations: one should keep $E_{\mathcal{L}}$ confluent.

Rewriting Logic Semantics of `PARALLEL`

To illustrate how rewriting logic and Maude can be used to give semantics to concurrent programming languages, I will use a simple language called `PARALLEL`.

I will also illustrate how programs in `PARALLEL` can then be verified by model checking using Maude's LTL model checker.

Rewriting Logic as a Unified Semantic Framework

RLS is not a competitor to other semantic styles! [Şerbănuţă *et al.* 2009] showed how virtually all operational semantic styles, namely:

- Small-Step Structural Operational Semantics
- Big-Step Structural Operational Semantics
- Modular Structural Operational Semantics (MSOS)
- Reduction Semantics with Evaluation Contexts
- The Chemical Abstract Machine
- Continuation-Based Semantics

can be faithfully (i.e., step-by-step) represented in rewriting logic.

Various styles can co-exist in rewriting logic, which also allows combinations of styles. Traian will lecture on this later this afternoon.

Modular Definitions and the K Framework

The modularity of a semantic framework is critical for its reusability and scalability. Modular SOS [Mosses 1999] brings modularity to SOS. Its natural representation in rewriting logic has already been demonstrated by Braga and Meseguer and will also be discussed in Traian's lecture.

An alternative approach to modular language definitions is provided Roșu's K framework [Rosu 2003] and will be explained later in this Summer School.

K is one of the key developments in the RLS Program. It provides a very compact and modular notation to define a language's semantics by rewrite rules. A K definition can then be translated into a corresponding rewrite theory in Maude thanks to the K-Maude tool for: (i) execution, (ii) model checking, and (iii) (using Matching Logic) theorem proving.

Defining Real-Time Languages

Rewriting logic can also specify real-time languages, as real-time rewrite theories, which are special rewrite theories (Σ, E, R) s.t.:

- Σ contains special sort *Time* and E contains an algebraic axiomatization of the *Time* data type, which can be either discrete or continuous
- Σ also contains a sort *GlobalState*, whose terms are pairs (t, r), with t an "untimed state" (a term) and r a "global clock" (a term of sort *Time*)
- R contains two types of rules:
  - instantaneous rules, which do not change the time; and
  - tick rules, which advance the time, of the form:

$$(t, r) \rightarrow (t', r') \;\; \text{if } C$$

Real-Time Maude [Ölvecki *et al.*] supports execution, search and model-checking of real-time rewrite theories.

Defining Modeling Languages

The most expensive errors are design errors. To make designs machine-representable software modeling languages are used.

There are two main limitations: (i) modeling notations tend to lack a formal semantics; and (ii) this lack of semantics manifests itself as a lack of analytic power.

The practical advantage of giving an executable formal semantics to a modeling language is that it can then be executed, reasoned about, and analyzed to uncover costly design errors.

Modeling languages $\mathcal{M}$ can be defined also rigorously as rewrite theories $(\Sigma_{\mathcal{M}}, E_{\mathcal{M}}, R_{\mathcal{M}})$, and then we can use the same machinery as for programming languages to formally analyze models at the design stage, before they are implemented.

Defining Hardware Description Languages

Similarly, hardware description languages (HDLs) can be defined as rewrite logic theories and can then be formally analyzed using the same generic mechanisms as for programming and modeling languages.

Several hardware description languages (ABEL, Verilog, BlueSpec, Production Rules) have been defined using this approach.

This talk reports on recent advances on using rewriting logic semantics in defining and formally analyzing programming, modeling, and hardware description languages.

General Philosophy

To use the rewriting logic semantics approach, we typically follow the steps below:

1. Define the desired programming, modeling or hardware description language as a rewrite theory (Σ, E, R)
2. Using Maude or similar tools execute the definition on many examples, until the designer is happy with the formal definition
3. Using an abstract semantics based on the original semantics, one can statically analyze programs
4. Using generic formal analysis tools of Maude, one can search for behaviors of interest in a non-deterministic or concurrent system, or even model check LTL properties
5. Using matching logic, a deductive verification approach on top of rewriting logic, one can prove programs correct by deductive verification.

Real-Time Language Semantics

Three real-time programming languages have been given formal semantics as real-time rewrite theories in Real-Time Maude:

1. AlTurki et al. have given concrete and abstract rewriting logic semantics to DOCOMO Labs' *L* real-time language.
2. AlTurki and Meseguer have given a rewriting logic semantics to J. Misra's Orc model of real-time concurrent computation. Both an SOS semantics and a much more efficient, yet equivalent, reduction semantics have been defined, as well as a tool to simulate and model check Orc programs.
3. Timed Creol extends the Creol object-oriented language defined in Oslo by researchers in O. Owe's group. Timed Creol and has been given a real-time rewriting logic semantics.

Programs in the above languages can be both simulated and model checked. Using Maude's socket mechanism a correct by construction distributed implementation of Orc has been obtained.

Modeling Language Semantics

Beginning with work of Knapp and Wirsing, OO Modeling languages, including UML, have been given semantics in rewriting logic by various authors.

But can we give semantics to modeling frameworks where different modeling languages are defined? This has been answered by Boronat et al. in MOMENT2, a model management framework and tool written in Maude and based on an executable rewriting semantics of the Meta-Object Facility (MOF), Object Constraint Language (OCL), and Query/View/Transformation (QVT).

MOMENT2 automatically checks conformance of a model M to a metamodel $\mathcal{M}$ with OCL constraints $\mathcal{C}$. Also, model transformations can be defined by rewrite theories specified in a user-friendly QVT-based syntax, and dynamic properties of systems can be model checked at the level of their models using Maude.

Real-Time Modeling Language Semantics

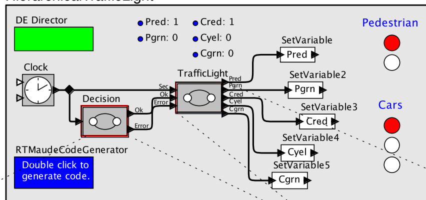
One can give formal executable semantics to real-time modeling languages using real-time rewrite theories, and then simulate and analyze their models in Real-Time Maude. Specifically:

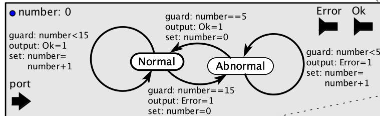
1. Ptolemy II DE models have been given semantics this way by [*Bae et al.*] with Real-Time Maude invoked as a Ptolemy plugin.
2. AADL models and particularly synchronous AADL models have likewise been given semantics and formal analysis capabilities in the AADL2Maude [*Ölveczky et al.*] and SynchAADL2Maude [*Bae et al.*] tools, which invoke Real-Time Maude in an OSATE plugin.
3. Entire real-time modeling frameworks such as: (i) e-Motions [*Rivera et al.*]; and (ii) MOMENT2's real-time extension [*Boronat&Ölveczky*] have been given formal semantics and analysis capabilities using Real-Time Maude.

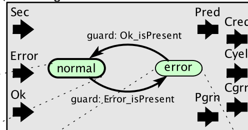
Rewriting Logic Semantics of Ptolemy II

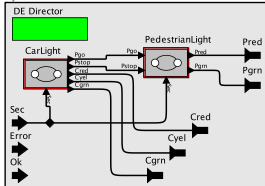
Ptolemy II is a widely used graphical modeling and simulation tool at UC Berkeley for real-time and embedded systems. Such systems are modeled as discrete-event (DE) models, which consist of a set of components called actors, having input ports and output ports, and linked by communication channels that pass events from one port to another.

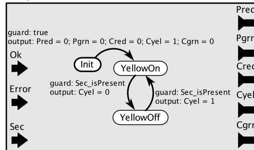
Next slide shows a Ptolemy-II hierarchical model of a fault-tolerant traffic light, together with the Real-Time Maude plugin for model checking Ptolemy II models.


HierarchicalTrafficLight
DE Director
Pred: 1
Pgrn: 0
Cred: 1
Cyel: 0
Cgrn: 0
Pedestrian
Cars
Clock
Decision
TrafficLight
SetVariable
SetVariable2
SetVariable3
SetVariable4
SetVariable5
RTMaudeCodeGenerator
Double click to generate code.


Decision
number: 0
guard: number<15
output: Ok=1
set: number= number+1
guard: number==5
output: Ok=1
set: number=0
Normal
Abnormal
guard: number<5
output: Error=1
set: number= number+1
guard: number==15
output: Error=1
set: number=0
port
Error
Ok


TrafficLight
Sec
Error
Ok
Pred
Cred
Cyel
Cgrn
Pgrn
guard: Ok_isPresent
normal
error
guard: Error_isPresent


Normal
DE Director
CarLight
PedestrianLight
Sec
Error
Ok
Pred
Pgrn
Cred
Cyel
Cgrn


Error
guard: true
output: Pred = 0; Pgrn = 0; Cred = 0; Cyel = 1; Cgrn = 0
Ok
Error
Sec
Init
YellowOn
YellowOff
guard: Sec_isPresent
output: Cyel = 0
guard: Sec_isPresent
output: Cyel = 1
Pred
Pgrn
Cred
Cyel
Cgrn

Rewriting Logic Semantics of Ptolemy II (slide 3)

In Ptolemy II's rewriting logic semantics the different components are modeled as distributed objects.

The semantics has three rewrite rules and several equations. The first rule is a 'tick' rule that advances time until the first events in the event queue are scheduled.

```
vars SYSTEM : ObjectConfiguration . var EVTS : Events . var QUEUE : EventQueue . var NZT : NzTime .
var N : Nat .
rl [tick] : {SYSTEM < global : EventQueue | queue : (EVTS ; NZT ; N) :: QUEUE >} => {delta(SYSTEM, NZT)
< global : EventQueue | queue : (EVTS ; 0 ; N) :: delta(QUEUE, NZT) >} in time NZT .
```

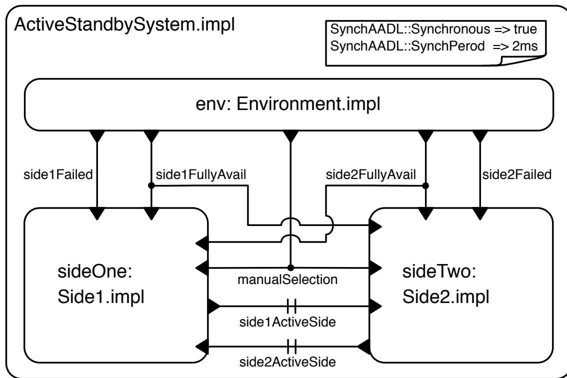
Rewriting Logic Semantics of AADL

AADL 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.

[Ölveczky et al. 2010] have given a formal executable semantics to an AADL fragment in Real-Time Maude. And [Bae et al. 2011] have identified a Synchronous AADL sublanguage and developed an OSATE plugin called SynchAADL2Maude that automatically tanslates synchronous AADL models into Real-Time Maude, according to the formal semantics, for execution and model checking.

The following is a Synchronous AADL model of an Active Standby System in Modular Avionics whose safety properties have been analyzed in Real-Time Maude.

Rewriting Logic Semantics of AADL (slide 2)



Rewriting Logic Semantics of AADL (slide 3)

Given a *Synchronous AADL* system, a synchronous transition step of the system is then formalized by the following' tick' rewrite rule:

```
var SYSTEM : Object . var VAL : Valuation . var VALS : ValuationSet .
crl [syncStepWithTime] : {SYSTEM} => {applyTransitions(transferData(applyEnvTransitions(VAL, SYSTEM)))}
in time period(SYSTEM) if containsEnvironment(SYSTEM) / VAL ;; VALS := allEnvAssignments(SYSTEM).
```

Hardware Description Language Semantics

The rewriting logic semantics project has been naturally extended from the level of programming languages to that of hardware description languages (HDLs):

1. Formal analysis of hardware/software codesigns was achived for the ABEL HDL by [*Katelman&Meseguer*].
2. The most complete formal executable semantics to-date for Verilog has been given by [*Meredith et al.*], finding significant erros on some tools; and Katemaln has used such a semantics as a basis for a novel test generation tool `vlogsl`.
3. The semantics of BlueSpec has also been defined by Katelman.
4. A formal rewriting semantics for the production rules asynchronous hardware formalism has been given by [*Katelman et al.*], who have then model checked several asynchronous hardware designs.

Abstract vs. Concrete Semantics and Static Analysis

A language rewriting logic semantics can be used unchanged for many analysis purposes. However, for static analysis we may change the concrete language semantics into a semantics within an abstract domain. Three static analyses have been investigated:

1. Hills [Hills 2008] has developed the C pluggable policy framework, which allows one to design static analysis on top of C, using a common infrastructure
2. Alba-Castro *et al.* [Castro 2010] have developed an abstract semantics for Java security properties, in particular for non-interference
3. Ellison *et al.* [Ellison 2009] have developed techniques to define type systems also in K. For example:

Deductive Verification using Matching Logic

Matching logic [Rosu 2010] is a combination of rewriting logic and FOL, which allows us to achieve Hoare-like program verification without redefining the language semantics axiomatically. Its formulae are called patterns and their satisfaction is based on pattern matching. Matching logic verification consists of:

- User annotates program with pattern assertions (pre-conditions, post-conditions, invariants)
- Use the language rewriting logic semantics to execute program symbolically
- On each path, show that all claimed assertions hold, by proving them using matching logic

As will be explained by Roșu, the Matching logic tool uses a rewriting logic semantics of a language in Maude (currently the K semantics of a C fragment) plus an SMT solver to verify programs.

Conclusions and Future Work

Rewriting logic semantics is closing the gap between theory and practice by supporting executable semantic definitions that scale up to real modeling, programming and hardware languages.

Rewriting logic definitions can be directly used to obtain interpreters and sophisticated program analysis tools, including static analyzers, model checkers, and program proving tools.

A future challenge is the automatic generation of high-performance language implementations from language definitions.

Another research area is meta-reasoning methods, to prove formal properties not about programs, but about language definitions.

Another research direction is the interplay between abstract semantics and model checking, and the application of state space reduction techniques in the model checking of programs from their rewriting logic language definitions