Symbolic Analysis of Maude Theories with Narval

Symbolic Analysis of Maude Theories with Narval∗ (system description)

MARÍA ALPUENTE, SANTIAGO ESCOBAR, JULIA SAPIÑA
VRAIN (Valencian Research Institute for Artificial Intelligence), Universitat Politècnica de València
(*e-mail:* {alpuente,sescobar,jsapina}@dsic.upv.es)

DEMIS BALLIS
DMIF, University of Udine
(*e-mail:* demis.ballis@uniud.it)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Concurrent functional languages that are endowed with symbolic reasoning capabilities such as Maude offer a high-level, elegant, and efficient approach to programming and analyzing complex, highly nondeterministic software systems. Maude's symbolic capabilities are based on equational unification and narrowing in rewrite theories, and provide Maude with advanced logic programming capabilities such as unification modulo user-definable equational theories and symbolic reachability analysis in rewrite theories. Intricate computing problems may be effectively and naturally solved in Maude thanks to the synergy of these recently developed symbolic capabilities and classical Maude features, such as (i) rich type structures with sorts (types), subsorts, and overloading, (ii) equational rewriting modulo various combinations of axioms such as associativity, commutativity, and identity, and (iii) classical reachability analysis in rewrite theories. However, the combination of all these features may hinder the understanding of Maude symbolic computations for non-experienced developers. The purpose of this system description is to describe how programming and analysis of Maude rewrite theories can be made easier by providing a sophisticated graphical tool called Narval that supports the fine-grained inspection of Maude symbolic computations.

KEYWORDS: Symbolic reachability analysis, narrowing, equational unification, Maude, rewriting logic

1 Introduction

Maude (Clavel et al. 2007) is a high-performance implementation of rewriting logic, a simple extension of equational logic that models concurrent systems (Meseguer 2012). Maude seamlessly integrates: (i) functional, logic, concurrent, and object-oriented computations; (ii) rich type structures with sorts, subsorts, and operator overloading; and (iii) equational reasoning modulo axioms such as associativity (A), commutativity (C), and unity (U) of functions. With regard to the language performance, Maude was ranked second (after Haskell) as the best performance language in a recent benchmarking of well-known algebraic, functional, and object-oriented languages[1] carried on in (Garavel et al. 2018). Because of its efficient rewriting engine and its meta-language capabilities, Maude is an excellent instrument for creating rich executable environments for various logics, programming languages, and tools (e.g., (Alpuente et al. 2016)).

∗ This work has been partially supported by the Spanish MCIU under grants TIN2015-69175-C4-1-R and RTI2018-094403-B-C32, and by Generalitat Valenciana under grant PROMETEO/2019/098.

[1] CafeOBJ, Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, SML, Stratego / XT, and Tom; see references in (Garavel et al. 2018).

A rewrite theory (or Maude program) is a triple $\mathscr{R} = (\Sigma, E, R)$ where Σ is a signature that contains the program operators together with their type definition, E is a collection of (possibly conditional) Σ-equations (so that (Σ, E) is an equational theory) that models system states as terms of an algebraic data type (initial algebra), $\mathscr{T}_{\Sigma/E}$, and R is a set of (possibly conditional) rewrite rules that define concurrent transitions between states. The equational theory E is generally decomposed as a disjoint union $E = E_0 \uplus Ax$, where the set E_0 consists of (conditional) equations and membership axioms (i.e., axioms that assert the type or *sort* of some terms) that are implicitly oriented from left to right as rewrite rules (and operationally used as simplification rules), and Ax is a set of commonly occurring algebraic axioms such as associativity, commutativity, and identity that are implicitly expressed as function attributes and are mainly used for Ax-matching (e.g., assuming a commutative binary operator $*$, the term $s(0) * 0$ matches within the term $X * s(Y)$ *modulo* the commutativity of symbol $*$ with matching substitution $\{X/0, Y/0\}$).

The rewrite theory $\mathscr{R}$ specifies a concurrent system that evolves by rewriting states using *equational rewriting*, i.e., rewriting with the rewrite rules in R *modulo* the equations and axioms in E (Meseguer 1992). For instance, consider the sort *Int* for integer numbers that are expressed by using the constant 0, the successor operator s, and the predecessor operator p, with the commutative addition operator $+$, and the (partial) specification of integer numbers given by the equations $E_0 = \{X + 0 = X, X + s(Y) = s(X + Y), p(s(X)) = X, s(p(X)) = X\}$ where variables X, Y are of sort *Int*. Consider a binary state constructor operator $\langle \_, \_ \rangle$: *Int Int* $\rightarrow$ *State* for a new sort *State* that models a system of processes waiting to enter a critical section in the first argument and inside the critical section in the second argument. The system state $t = \langle s(0), s(0) + p(0) \rangle$ can be rewritten with the following rule (denoting that a waiting process is entering the critical section)

$$\langle s(X), Y \rangle \Rightarrow \langle p(s(X)), s(Y) \rangle \tag{1}$$

which yields the state $\langle 0, s(0) \rangle$. This is essentially done in Maude by first simplifying the input state t (with the equations E_0 modulo Ax) to its irreducible form[2] $t_{\downarrow_{E_0,Ax}} = \langle s(0), 0 \rangle$; then $t_{\downarrow_{E_0,Ax}}$ is rewritten into $t' = \langle p(s(0)), s(0) \rangle$ by applying the rewrite rule (1), and, finally, t' is also normalized with the equations E_0 (modulo Ax) to its irreducible form $t'_{\downarrow_{E_0,Ax}} = \langle 0, s(0) \rangle$ (by applying the third equation in E_0 to the subterm $p(s(0))$ of t'). In symbols, a rewrite step $t \xrightarrow{r}_{\mathscr{R}} s$ consists of the rewrite sequence $t \rightarrow^*_{E_0,Ax} (t_{\downarrow_{E_0,Ax}}) \rightarrow_{\{r\},Ax} t' \rightarrow^*_{E_0,Ax} (t'_{\downarrow_{E_0,Ax}})$, with $t'_{\downarrow_{E_0,Ax}} = s$, and denotes a transition (modulo E) from state t to state s via the rewrite rule r of R.

System computations (also called execution traces) correspond to equational rewrite sequences $t_0 \xrightarrow{r_0}_{\mathscr{R}} t_1 \xrightarrow{r_1}_{\mathscr{R}} \ldots$, where each $t_i \xrightarrow{r_i}_{\mathscr{R}} t_{i+1}$ denotes a transition (modulo E) from state t_i to t_{i+1} via the rewrite rule r_i of R. The transition space of all computations in $\mathscr{R}$ from the initial state t_0 can be represented as a *computation tree* whose branches specify all of the system computations in $\mathscr{R}$ that originate from t_0.

Rewriting and equational rewriting are of course symbolic reasoning methods, but Maude supports symbolic reasoning in a stronger sense by means of *(equational) unification and narrowing in the rewrite theory* $\mathscr{R} = (\Sigma, E, R)$. Narrowing is a generalization of term rewriting that allows free variables in terms (as in logic programming) and handles them by using unification (instead of pattern matching) to non-deterministically reduce these terms. Originally introduced in the context of theorem proving, narrowing is complete in the sense of logic programming (com-

[2] Note that the term $t = \langle s(0), s(0) + p(0) \rangle$ is first simplified into $\langle s(0), s(p(0) + 0)) \rangle$ (by reducing the subterm $s(0) + p(0)$ of t with the second (implicitly oriented) equation in E_0 *modulo* the commutativity of $+$), which is then further simplified into $\langle s(0), s(p(0)) \rangle$ (by reducing the subterm $s(p(0) + 0)$ with the first equation in E_0), and then finally simplified into the irreducible term $t_{\downarrow_{E_0,Ax}} = \langle s(0), 0 \rangle$ (by reducing the subterm $s(p(0))$ with the fourth equation in E_0).

putation of answers) and functional programming (computation of irreducible forms) so that efficient versions of narrowing have been adopted as the operational principle of so-called multi-paradigm (constraint, functional, and logic) programming languages (see, e.g., (Hanus 2013)). In the last few years, there has been a resurgence of narrowing in many application areas such as equational unification, state space exploration, protocol analysis, termination analysis, theorem proving, deductive verification, model transformation, testing, constraint solving, and model checking of infinite-state systems. To a large extent, the growing interest in narrowing is motivated by the recent takeoff of symbolic execution applications and the availability of efficient narrowing implementations. Narrowing-based, symbolic reasoning methods and applications in rewriting logic and Maude are discussed in (Meseguer 2018).

Similarly to equational rewriting, where matching modulo E (or E-matching) is used, *equational* unification (or E-unification) is adopted (instead of standard, syntactic unification) in (R,E)-narrowing (i.e., narrowing with the rules in R modulo the equations and axioms in E). More precisely, (R,E)-narrowing in a rewrite theory $\mathscr{R} = (\Sigma, E, R)$, with $E = E_0 \uplus Ax$, is supported in Maude by means of a *three-level* machinery.

1. A (R,E)-narrowing step from t_1 to t_2 with a rule $l \Rightarrow r$ in R is carried out by first performing E-unification between a subterm s of t_1 and the left-hand side l of the rule; then t_2 is obtained from t_1 by replacing s in t_1 with the right-hand side r and instantiating the yielded term with the computed E-unifier. Note that the rule may have extra variables in its right-hand side.
2. In turn, the E-unification problem $s =^?_E l$ is solved by using *folding variant* narrowing (in short, FV-narrowing) in the equational theory (Σ, E), which is an equational narrowing strategy that computes a finite and complete set of E-unifiers for $s =^?_E l$ under suitable requirements (Escobar et al. 2012). The idea of folding variant narrowing is to *equationally* narrow the term $eq(s,l)$ (that encodes the unification problem $s =^?_E l$) to a special constant tt in the "rewrite theory" $\mathscr{R}_0 = (\Sigma, Ax, \vec{E}_0 \cup \{\varepsilon\})$, where $\vec{E}_0$ results from explicitly orienting the equations of E_0 as rewrite rules and the extra rule $\varepsilon = (eq(X,X) \Rightarrow tt)$ is added to $\vec{E}_0$ in order to mimic unification of two terms (modulo Ax) as a narrowing step that uses ε (Middeldorp and Hamoen 1992); e.g., by using equation ε, the term $eq(s(0) * 0, U * s(V))$ FV-narrows to `tt` (modulo commutativity of $*$), and the computed narrowing substitution does coincide with the unifier modulo C of the two argument terms, i.e., $\{U/s(0), V/0\}$.
3. For each folding variant narrowing step using a rule in $\vec{E}_0$ modulo Ax in Point 2, Ax-unification algorithms are employed, allowing any combination of symbols having any combination of associativity, commutativity, and identity (Durán et al. 2018).

(R,E)-narrowing computations are natively supported[3] by Maude version 2.8 for unconditional rewrite theories. Following the previous example, the input state $\langle Z, s(Z) + 0 \rangle$ (R,E)-narrows to $\langle X, s(s(X)) \rangle$ with substitution $\{Y/s(s(X)), Z/s(X)\}$ (i.e., an E-unifier of $\langle Z, s(Z) + 0 \rangle$ and the left-hand side $\langle s(X), Y \rangle$ of the rewrite rule (1)) in the theory $\mathscr{R}$ of the example above. Note that the reduced state $\langle X, s(s(X)) \rangle$ signals a possible programming error in rule (1) since it shows that multiple processes might enter a critical section, simultaneously.

Analogously to rewriting, the search space of (R,E)-narrowing computations in (Σ, E, R) (respectively, FV-narrowing computations in (Σ, E)) can be represented as a tree-like structure that we call (R,E)-narrowing (respectively, FV-narrowing) tree. When it is clear from the context, we simply write narrowing instead of (R,E)-narrowing or FV-narrowing.

[3] Maude 2.8 is currently available as Maude alpha version 120 for developers, and it will be officially released soon.

This paper describes Narval (*Narrowing variant-based tool)*, a visual system for exploring all three levels of symbolic computations in Maude programs. This includes not only the inspection of partially computed substitutions, but also the intestines of *Ax*-matching and equational simplification sequences as well as *Ax*-unification and equational unification sequences that are concealed within rewriting and narrowing algorithms and are jealously hidden within Maude's symbolic execution machinery. This contribution is important because Maude lacks any symbolic tracing facility that can help the user to advance through a given symbolic execution stepwisely. Indeed, in order not to jeopardize the language performance, many of the state transformations (using E) described above are internal and are never recorded explicitly in the symbolic trace; hence, any erroneous intermediate result is difficult to debug. Furthermore, Maude narrowing traces are either directly displayed or written to a file (in both cases, in plain text format) and are thus only amenable for manual inspection by the user. This is in contrast with the enriched views provided by Narval, which are complete (every single transition is recorded by default) and can be either graphically displayed or delivered in its meta-level representation, which is very useful for further automated manipulation. Also, the displayed view can be abstracted when deemed appropriate to avoid cluttering the display with unneeded details. Finally, important insights regarding the programs/theories can be gained from controlling the narrowing space exploration.

This paper summarizes our experience as follows: i) identifying what to visualize in terms of narrowing computations and how to represent each element; ii) showing how visualization can enhance program analysis, debugging and evolution; and iii) implementing the components of Narval. The rest of the paper is structured as follows. Section 2 introduces a leading example that will be used throughout the paper for describing the narrowing-based, symbolic reasoning capabilities of Narval. In Section 3, we explain the core functionality of Narval that supports both symbolic search space exploration and interactive reachability analysis for Maude programs. We also show how such features can be used to diagnose and correct the Maude programs as well as to infer new formal descriptions that satisfy the user's intent. Section 4 describes some additional tool features that allow the user to glimpse into the technical details of narrowing computations such as the fine-grained inspection of narrowing steps, the computation of equational unifiers, the exploration of different representations of Maude's narrowing and rewriting search spaces, and interactive visualization with source code inspection. In Section 5, we provide a description of the tool architecture and we overview the main implementation choices. Finally, some related work and further applications are briefly discussed in Section 6.

2 Software Systems as Maude programs: a Generic Grammar Interpreter

Nondeterministic as well as concurrent software systems can be formalized through Maude *system modules* whose syntax is `mod <NAME> is <SPEC> endm`, where `<NAME>` is the module name and `<SPEC>` encodes a given rewrite theory $\mathscr{R} = (\Sigma, E_0 \uplus Ax, R)$. Maude's syntax is almost self-explanatory, and here we just highlight the most relevant keywords that are used to specify Σ, E_0, Ax, and R (for further details, please refer to (Clavel et al. 2016)). Sorts and operators of the signature Σ are respectively declared by means of the keywords `sort(s)` and `op(s)`. Both prefix and mixfix notation can be used to specify operators: in the latter case, the special wildcard `_` is used as argument placeholder. Algebraic axioms in Ax are attached to operator declarations via attributes: operator attributes `assoc`, `comm`, and `id` respectively stand for associativity, commutativity, and identity. Subtyping relations are introduced by means of the `subsort` keyword. Equations in E are denoted by the `eq` keyword, while the equational attribute `variant` distinguishes those equations in E that can be used for equational unification (while the rest of equations in

E are only used for equational simplification). Similarly, keyword `rl` defines rewrite rules in R, and the rule attribute `narrowing` characterizes all and only those rewrite rules that can be used to perform (R,E)-narrowing steps (while the rest of rules in R are only used for rewriting).

```
 1 mod GRAMMAR-INT is
 2   sorts Symbol NSymbol TSymbol String Production Grammar Conf .
 3   subsorts TSymbol NSymbol < Symbol < String .
 4   subsort Production < Grammar .
 5   ops 0 1 2 eps : -> TSymbol .
 6   ops S A B : -> NSymbol .
 7   op _@_ : String Grammar -> Conf .
 8   op _->_ : String String -> Production .
 9   op __ : String String -> String [assoc] .
10   op mt : -> Grammar .
11   op _;_ : Grammar Grammar -> Grammar [assoc comm id: mt] .
12   vars L1 L2 U V : String .
13   var G : Grammar .
14   var N : NSymbol .
15   var T : TSymbol .
16   eq [EQ1] : eps V = V [variant] .
17   eq [EQ2] : U eps = U [variant] .
18   eq [EQ3] : U eps V = U V [variant] .
19   rl [apply] : ( L1 U L2 @ (U -> V) ; G) => ( L1 V L2 @ (U -> V) ; G) [narrowing] .
20 endm
```

Fig. 1: The `GRAMMAR-INT` Maude module

To illustrate the novel features of the Narval tool, we consider, as a leading example, the Maude module `GRAMMAR-INT` of Figure 1, which encodes an elegant, generic grammar interpreter (Durán et al. 2018). Interpreter states are specified by (possibly non-ground) terms of the form `T @ G`, where `G` is an input grammar and `T` represents the input string of terminal and non-terminal symbols to be recognized. For the sake of simplicity, we provide three non-terminal symbols of sort `NSymbol`: `A`, `B`, and `S` (the grammar axiom), and four terminal symbols of sort `TSymbol`: `0`, `1`, `2`, and the finalizing mark `eps` (i.e., the empty string). We also declare sort `Symbol` (that includes both `NSymbol` and `TSymbol`) as a subsort of `String` so that strings can be simply built by using the (associative) juxtaposition operator `__` which also has `eps` as unity element, meaning it is redundant to write it. Note that the unity axiom is explicitly formalized via the equations in lines 16–18 to offer AU-unification capabilities over strings by means of the FV-narrowing strategy.

The input grammar `G` is defined by means of the associative and commutative operator `_;_` with identity `mt` (the empty grammar), which allows `G` to be concisely specified as a multiset of productions $\mathtt{U}_1 \to \mathtt{V}_1; \ldots; \mathtt{U}_n \to \mathtt{V}_n$, where each $\mathtt{U}_i \to \mathtt{V}_i$ denotes a production rule of `G`. Note that $\mathtt{U}_i, \mathtt{V}_i$ in each production $\mathtt{U}_i \to \mathtt{V}_i$ can be any arbitrary string of symbols; thus, any kind of grammar of Chomsky's hierarchy can be formalized within our very compact interpreter —from the simple Type-3 grammars that generate regular languages to the unrestricted Type-0 grammars that denote recursively enumerable languages.

The interpreter behaviour is specified by a single rewrite rule implements state transitions (namely, the rule identified by label `apply` in line 19). More specifically, given a state `W @ G`, this rule non-deterministically applies the production rules of `G` to the string of symbols `W`, thus yielding a new state; this is done by rewriting any substring `U` of `W` by using the production `U -> V` of `G` (with `W` being non-deterministically decomposed as `L1 U L2` thanks to matching modulo associativity in strings, and the production `U -> V` being automatically identified thanks to ACU-matching in the multiset that represents the grammar `G`), and then proceeding with `L1 V L2`. Generating a string *st* that belongs to the language of `G` consists of rewriting the initial state

`S @ G` until the final state *st* `@ G` is reached. Moreover, the very same rule can be also used to recognize that a given string belongs to the language of `G`. Parsing a word `w` according to `G` can be obviously defined by doing rewriting in the opposite direction, e.g., by defining a new rule

```
rl [parsing] : (L1 V L2 @ (U -> V) ; G) => (L1 U L2 @ (U -> V) ; G)
```

However, there is no need for introducing this rule in Maude since recognizing `w` can be simply achieved by solving the *reachability goal* `S @ G` $\longrightarrow^*$ `w`.

Example 1
Consider the following Type-2 grammar `G`

$$\texttt{S} \rightarrow \texttt{0S1} \mid \texttt{10}$$

that generates the language $\{0^n101^n \mid n \geq 0\}$. Then, the language string `001011` can be generated by the following rewriting sequence in module `GRAMMAR-INT`: `S @ G` $\longrightarrow$ `0S1 @ G` $\longrightarrow$ `00S11 @ G` $\longrightarrow$ `001011 @ G`. Also, solving the reachability goal *S* `@ G` $\longrightarrow^*$ `001011` recognizes the string is grammatically correct, and moreover, solving *S* `@ G` $\longrightarrow^*$ `001W` binds variable `W` with the string value `011`. Note that the form of reasoning given by (classical) reachability goals $t \longrightarrow^* \exists X t'$, with X being the set of variables appearing in t', does not involve any unification as no variables in the input term t get instantiated (but only E-matching of subsequently rewritten forms of t within the term t' is performed); that is, no *narrowing* on t is performed. This is in contrast with the *symbolic reachability analysis* based on narrowing that we describe in Section 3.

3 Narrowing-based Symbolic Reachability Analysis with Narval

Given a Maude program that encodes a rewrite theory $\mathscr{R} = (\Sigma, E_0 \cup Ax, R)$, and a (possibly) non-ground term t, the search space of $\mathscr{R}$ that originates from t can be symbolically explored by using (R,E)-narrowing. Indeed, t represents an abstract characterization of the (possibly) infinite set of all of the concurrent states $[\![t]\!]$ (i.e., all the ground substitution instances of t, or, more precisely, the $E_0 \cup Ax$-equivalence classes associated to such ground instances) within $\mathscr{R}$. In this scenario, each (R,E)-narrowing computation $\mathscr{C}$ subsumes all of the rewrite computations that are "instances" of $\mathscr{C}$ modulo $E_0 \cup Ax$. Therefore, the narrowing tree that stems from t offers a compact, symbolic representation of the program behaviors for the different instances of $[\![t]\!]$.

More importantly, an exhaustive exploration of the (R,E)-narrowing tree of t allows one to prove existential reachability properties of the form

$$\exists X\ t \longrightarrow^* t' \tag{2}$$

with t and t' being two (possibly) non-ground terms —called the *input* term and *target* term, respectively— and X being the set of variables appearing in t and t'. Roughly speaking, proving the logic formula (2) amounts to determining whether there exists a state in $[\![t]\!]$ from which we can *reach* a state in $[\![t']\!]$ after a finite (possibly zero) number of narrowing steps with the rules of R modulo the equational theory $E_0 \uplus Ax$. Solving this problem means searching for a symbolic solution to it within $\mathscr{R}$'s narrowing tree that originates from t in a hopefully *complete* way (so that, for any existing solution, a more general answer modulo $E_0 \uplus Ax$ will be found).

Completeness of (R,E)-narrowing is guaranteed for topmost rewrite theories (i.e., all rewrites take place at the root of a term). This class of theories is of primary importance in the Rewriting Logic framework since it has many practical applications (e.g., the analysis of security protocols (Meseguer and Thati 2005)). Furthermore, more complex theories (e.g., topmost modulo Ax

theories, and Russian dolls theories) can be easily transformed into equivalent, topmost rewrite theories (Alpuente et al. 2019).

The latest Maude distribution provides the built-in `vu-narrow` command that allows the symbolic search space of a given term t to be explored as well as sophisticated reachability properties to be investigated by incrementally visiting, in a breadth-first manner, the narrowing tree for t. Since the narrowing search may either never terminate and/or find an infinite number of solutions, two *bounds* can be added to the `vu-narrow` command: a bound for the number of solutions requested, and another bound for the number of narrowing steps from the initial input term t (i.e., a threshold depth on the (R,E)-narrowing tree is set to make the search finite). Unfortunately, `vu-narrow` outputs are given in a raw, often giant, text format that can be difficult to inspect and understand for non-experienced users.

The Narval system described in this paper gracefully overcomes this drawback by providing a rich, graphical environment, where narrowing trees can be exhaustively and stepwisely explored by means of an intuitive point-and-click strategy, and solutions for reachability problems are automatically highlighted when progressively hit during the incremental construction of the narrowing tree.

In the sequel, we illustrate the core symbolic analysis features of Narval by using the generic grammar interpreter of Section 2.

3.1 Interactive Search Space exploration

Narval offers an interactive exploration facility for the incremental construction and visualization of narrowing trees. By simply selecting a node (i.e., state) t in the frontier of the (current) tree $\mathscr{T}$, all of the (R,E)-narrowing steps from t are automatically computed and added to $\mathscr{T}$, thereby providing an incremental expansion in amplitude of the original tree fragment. This feature can be particularly convenient when debugging or analyzing Maude programs. Let us see an example.

Example 2

Consider again the `GRAMMAR-INT` Maude module of Figure 1, together with the following Type-1 grammar `G`

$$\texttt{S} \rightarrow \texttt{0A2} \mid \texttt{02}$$
$$\texttt{0A} \rightarrow \texttt{00A2} \mid \texttt{02}$$

which fails to generate the following language $\{\texttt{02}\} \cup \{\texttt{0}^{\texttt{n}}\texttt{12}^{\texttt{n}} \mid \texttt{n} > \texttt{1}\}$ since one of its productions contains a small bug. The bug can immediately be detected by feeding Narval with the input interpreter state s_1

```
N:NSymbol @ (S -> 0 A 2) ; (S -> 0 2) ; (0 A -> 0 0 A 2) ; (0 A -> 0 2)
```

where `N` is a "logic" variable of sort `NSymbol`, and generating the narrowing tree fragment $\mathscr{T}$ of Figure 2. Indeed, each state `w @ G` in $\mathscr{T}$, with `w` being a string of terminal symbols, indicates that `w` is a word in the language accepted by `G`, whenever the variable `N` is bound to the nonterminal symbol `S`. Now, observe that the (R,E)-narrowing step from state s_9 to state s_{14} yields the undesired string `022` (with computed narrowing substitution $\{\texttt{N/S}\}$) due to the application of the production $\texttt{0A} \rightarrow \texttt{02}$, which replaces the nonterminal symbol `A` with the erroneous terminal symbol `2` (which actually should be `1`). A simple fix is thus obtained by replacing the faulty production with $\texttt{0A} \rightarrow \texttt{01}$.

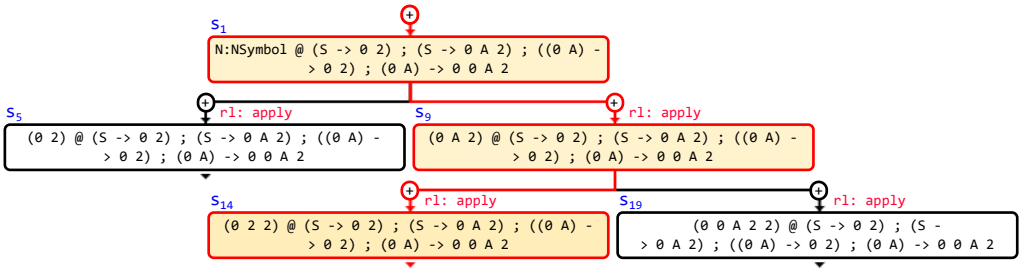

Fig. 2: Fragment of the narrowing tree computed by Narval in Example 2.

3.2 Narrowing-based reachability

A reachability property $\exists X\, t \to^* t'$ is specified in Narval by simply providing the input and target terms, t and t', in the input phase. The property is then incrementally checked while expanding the (R,E)-narrowing tree of t by simply E-unifying all of the nodes reached in the expanded tree fragment with the target term t'. Each branch in the tree that reaches a node u that E-unifies with t' is a narrowing computation that represents a constructive proof of the given property. Indeed, instantiating t with the composition of the sequence of E-unifiers that enable each step in the narrowing computation from t to u (i.e., the computed narrowing substitution σ), plus a E-unifier γ for $u = t'$, gives us a concrete rewrite sequence witnessing the existential reachability formula. To highlight the proof, Narval automatically colours the node u in green, and the *reachability answer substitution* $\sigma\gamma$ is delivered.

As anticipated in Example 1, the Maude module `GRAMMAR-INT`, which was used as a pure, nondeterministic, word generator in Example 2, can also be employed as a parser that recognizes whether or not a word is in the language of a given grammar `G`. Furthermore, more sophisticated reachabilty analyses than in Example 1 can be achieved by using (R,E)-narrowing and its inherent logical program inversion capabilities as shown in the following examples.

Example 3
Consider the following Type-2 grammar `G`

$$\texttt{S} \to \texttt{0S0} \mid \texttt{1S1} \mid \texttt{eps}$$

that generates the language of all even palindromes over the alphabet $\{\texttt{0},\texttt{1}\}$. Now, to show that the word `0110` is in the language of `G`, we just need to feed Narval with the input term

```
N:NSymbol @ (S -> 0 S 0) ; (S -> 1 S 1) ; (S -> eps)
```

and the target (ground) term

```
0 1 1 0  @ (S -> 0 S 0) ; (S -> 1 S 1) ; (S -> eps)
```

By exploring the (R,E)-narrowing tree, after a few tree expansions, we get the tree fragment of Figure 3 in which the green node s_{23} shows that the word `0110` can be derived in `G`. Moreover, by inspecting the details of the (R,E)-narrowing step from s_{20} to s_{23}, we can discover that the reachability answer substitution includes the binding `N:NSymbol/S`, which is correct and expected, since the word `0110` can be only generated starting from the grammar non-terminal symbol `S`.

Reachability symbolic analysis in Narval can also be a valuable tool for deriving new information from a given Maude program in order to automatically complete or mutate a given description w.r.t. the user's intent.

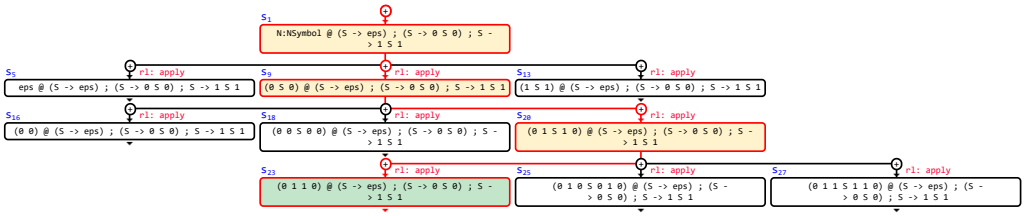

Fig. 3: Fragment of the narrowing tree computed by Narval in Example 3.

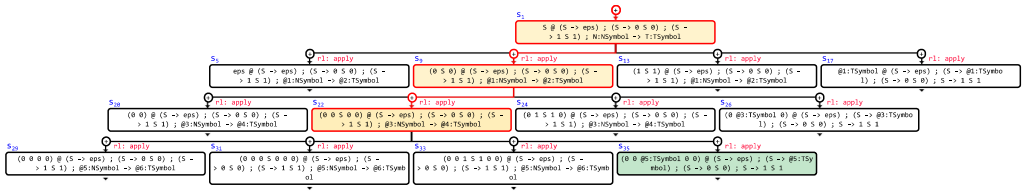

Fig. 4: Fragment of the narrowing tree computed by Narval in Example 4.

Example 4

Consider the grammar `G` of Example 3 and the palindrome `00100`, which does not belong to the language of `G` since its length is odd. One may want to know what missing production is needed so that `00100` derives from the non-terminal symbol `S`. This question can be automatically answered by feeding Narval with the input term

```
S @ (N:NSymbol -> T:TSymbol) ; (S -> 0 S 0) ; (S -> 1 S 1) ; (S -> eps)
```

in which the original grammar specification has been augmented with a new, fully generic production `N:NSymbol -> T:TSymbol`, and the target term

```
0 0 1 0 0 @ (N:NSymbol -> T:TSymbol) ; (S -> 0 S 0) ; (S -> 1 S 1) ;
            (S -> eps).
```

By using Narval, the tree fragment of Figure 4 is generated that includes the green node s_{35}. This node provides a solution for the considered reachability problem: indeed, the E-unification of the term in s_{35} and the target term succeeds and computes the reachability answer substitution $\sigma = \{\mathtt{N} : \mathtt{NSymbol}/\mathtt{S}, \mathtt{T} : \mathtt{TSymbol}/\mathtt{1}\}$ in Figure 5, which allows the missing production `S -> 1` to be inferred from the instantiation with σ of production `N:NSymbol -> T:TSymbol`.

4 Additional Features of Narval

Execution modalities. Besides the core, (R,E)-narrowing functionality that we discussed in Section 3, which is available through the *Narrowing in a rewrite theory* execution mode, Narval supports three additional execution modalities: the *Rewriting* mode, the *FV-narrowing* mode, and the *Equational unification* mode.

The rewriting mode allows the user to interactively explore the computation tree generated by Maude's *rewrite* engine that performs state transitions by non-deterministically rewriting system states modulo equations and axioms instead of using the much more involved (R,E)-narrowing relation. This option turns Narval into a program stepper that can be conveniently used to animate programs w.r.t. concrete inputs (i.e., ground input terms).

The FV-narrowing mode implements an inspection modality, based on the folded variant

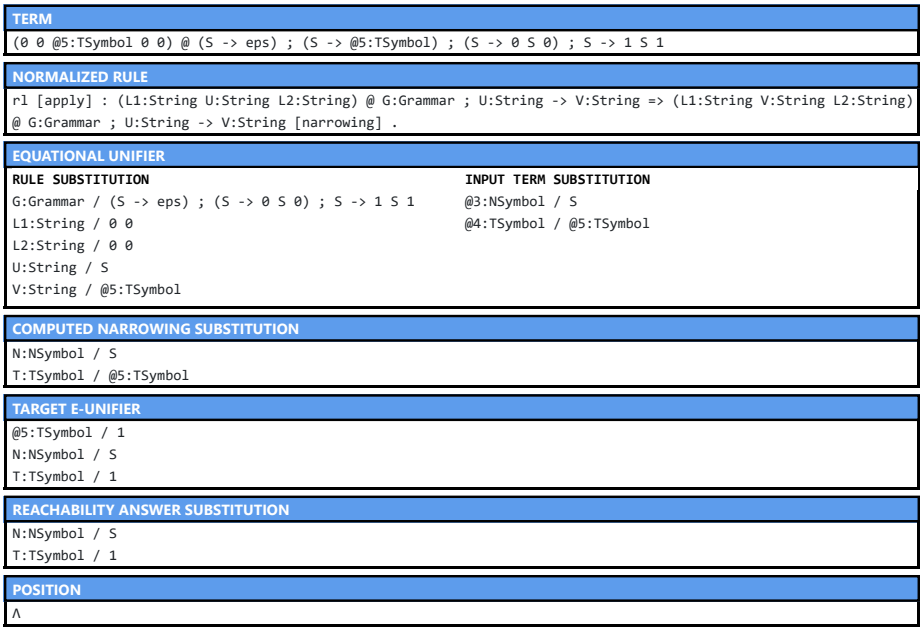

Fig. 5: Details of the narrowing step from s_{22} to s_{35} of Figure 4.

narrowing strategy of (Escobar et al. 2012), that only uses equations and axioms to narrow terms, thereby providing a means to analyze the purely equational search space of Maude programs. This modality serves also as a basis for the equational unification mode that inspects the insights of $E_0 \uplus Ax$-unifiers which are computed by FV-narrowing and Maude's built-in Ax-unification algorithms. In fact, as explained in Section 1, given a rewrite theory $\mathscr{R} = (\Sigma, E, R)$, with $E = E_0 \uplus Ax$, each (R,E)-narrowing step requires performing $(E_0 \uplus Ax)$-unification between the term t to be narrowed and the left-hand side l of the applied rewrite rule. More precisely, this is done by executing a new Narval instance that explores the FV-narrowing tree rooted at $eq(t,l)$. Computed narrowing substitutions in successful tree branches (i.e., branches that end with the success constant tt) represent $(E_0 \uplus Ax)$-unifiers for the unification problem $t =^{?}_{E_0 \uplus Ax} l$. Narval automatically highlights, within the inspected FV-narrowing tree fragment, the tree branch that corresponds to the computation of the $(E_0 \uplus Ax)$-unifier involved in the considered (R,E)-narrowing step. Let us see an example.

Example 5
Consider the (R,E)-narrowing step from state s_1 to state s_9 in the narrowing tree of Figure 3. The step uses the `apply` rewrite rule of the `GRAMMAR-INT` module and computes the equational unifier

```
{ G:Grammar / (S -> eps) ; (S -> 1 S 1), L1:String / eps,
  L2:String / eps, U:String / S, V:String / 0 S 0, N:NSymbol / S }
```

To inspect the computation of such E-unifier, it suffices to right-click the state s_9 and select Inspect unifier from the contextual menu. This action starts the exploration of the FV-narrowing tree that solves the equational unification problem between the state s_1 (that (R,E)-narrows into s_9) and the left-hand side of the `apply` rule. Figure 6 shows a fragment of the FV-narrowing tree that includes the FV-narrowing computation of the equational unifier under examination (that is, the FV-narrowing computation from the root node s_1 to the green node s_8).

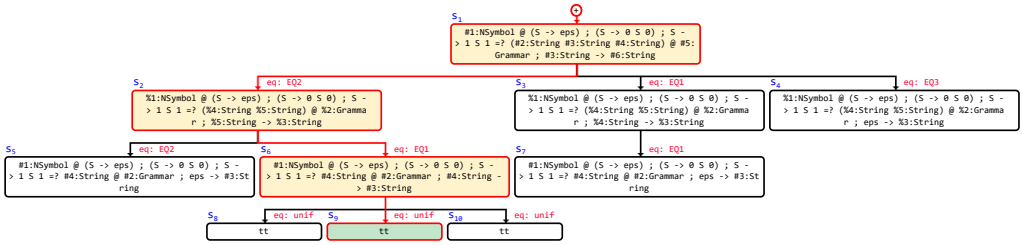

Fig. 6: Fragment of the FV-narrowing tree that computes the equational unifier in Example 5.

Transition and computation information. By clicking any rule (edge) label in the narrowing tree, the user can obtain the complete information of the associated (R,E)-narrowing step. Specifically, the following information is provided: (i) the term t that is yielded by the narrowing step, (ii) the (normalized version of) the rewrite rule applied, (iii) the position in t where the narrowing step occurred, (iv) the computed $(E_0 \uplus Ax)$-unifier, (v) the computed narrowing substitution, (vi) the target E-unifier (if any), that is, the E-unifier between t and the target term of a reachability goal, and (vii) the reachability answer substitution (this field is only present if t is a solution of a specified reachability property). For instance, Figures 5 shows transition information for the narrowing step from state s_{22} to state s_{35} of the narrowing tree of Figure 4. Similarly, the user can also access the details of FV-narrowing steps that are performed by means of dedicated Ax-unification algorithms.

Narval also offers the possibility of isolating any computation within the (R,E)-narrowing and FV-narrowing trees. By selecting a tree node, the computation from the root to the chosen node is visualized in a tabular form in a separate window. The selected computation can also be exported into meta-level representation, so the user can easily transfer it to any other Maude tool for further analysis or manipulation.

Enriched views. Narval supports two distinct views of the (R,E)-narrowing and FV-narrowing trees: namely, the standard view and the instrumented view. The standard view (which is the default mode) focuses on the narrowing steps, whereas the instrumented view completes the picture with all of the internal reduction steps that are performed, using equations, axioms, and Maude built-in operations up to reaching the normalized form of each term. The instrumented view can be locally enabled on a selected narrowing step by pressing the $+$ symbol labelling the associated edge in the tree. For instance, Figure 7 illustrates the instrumented view for the (R,E)-narrowing step from s_1 to s_5 that appears in the narrowing tree of Figure 3. Observe that the sequence of lightblue nodes in Figure 7 rewrites the term `eps eps eps` into its normalized form `eps` by using the equations 16–18 of the `GRAMMAR-INT` module of Figure 1.

Additional navigation capabilities. Narval encompasses some additional features to improve the user experience while navigating (narrowing) trees. The user can automatically explore multiple levels of the tree by right-clicking with the mouse on a node s and then selecting Expand Subtree from the contextual menu. This option allows the user to automatically unfold, up to a given depth k, for $k \leq 5$ (with default depth $k = 3$), the subtree hanging from the considered node s by following a breadth-first strategy. Dually, a subtree rooted at s can be folded into s by means of the Fold Node option.

Common actions like dragging or moving the tree via arrow keys are supported. Also, when a tree node is selected, the position of the tree on the screen is automatically rearranged to keep the chosen node at the center of the scene. Finally, several data that decorate the narrowing tree

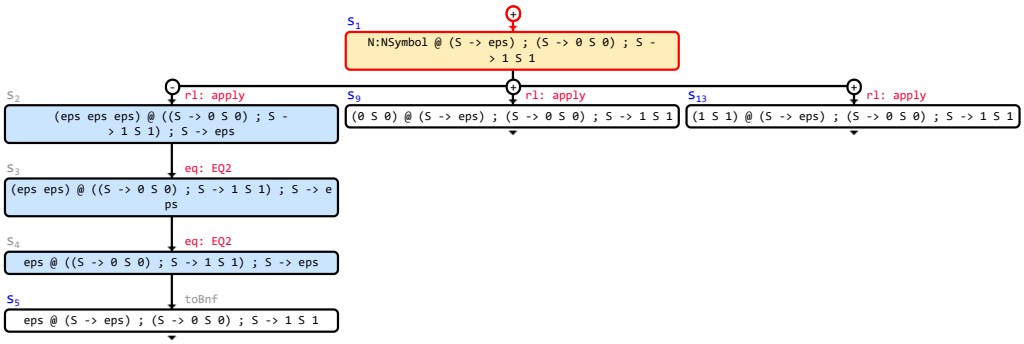

Fig. 7: Instrumented view for the (R,E)-narrowing step from s_1 to s_5 appearing in Figure 3.

(e.g. state/rule labels and substitution information) could be toggled on and off to focus the user attention on selected aspects of the explored search space.

Finally, even if the narrowing search space for a given input term is hierarchically organized as a tree in order to systematize its exploration, Narval additionally supports a graph-like representation where equivalent system states (i.e., states that are equal modulo variable renaming and algebraic axioms) are grouped together in a single state representative.

5 Implementation

Narval does not simply process Maude's output for the different symbolic features (rewriting, narrowing, and equational unification) to draw an exploration tree. Instead, it builds on top of the numerous reflective capabilities provided by Maude for reproducing in full detail every rewrite step, FV-narrowing step, and (R,E)-narrowing step in a rewrite theory $\mathscr{R}$. There are many aspects of dealing with symbolic computations, such as proper variable renaming, handling substitutions modulo axioms, or avoiding variable capturing (by providing different families of variable indices) that are made completely explicit in Narval, while Maude hides them inside its narrowing machinery.

Furthermore, Narval performs a program transformation to support the in-depth analysis of equational unification. Equational unification in Maude is available by means of the `variant unify` command and its corresponding meta-level operation `metaVariantUnify` (see Chapter 12.9 of (Clavel et al. 2016)). However, these commands provide poor insights into the internal reasoning yielding their results. Narval automatically transforms the user input program in order to explicitly encode the equational unification problem to be solved within the program and then uses the standard Maude functionality to carry out the inspection of the equational unification computation.

Specifically, the input program is complemented by adding the binary, polymorphic operator `=?` that is used to specify unification problems, and the constant operator `tt` that represents success in computing an equational unifier. The Maude declarations for such operators are as follows:

```
op _=?_ : Universal Universal -> [Bool] [poly (1 2)] .
op tt : -> [Bool] .
```

where `Universal` denotes a placeholder for any known sort, `[Bool]` is the *kind* for the Boolean sort[4], and `poly (1 2)` specifies that both arguments of `=?` are polymorphic.

Finally, the input program is also augmented with a set of unification equations (one for each kind in the program) of the form: `eq [unif] : X =? X = tt [variant]` where `X` is a variable of the appropriate kind.

Architecture of Narval. The Narval tool has been implemented as a web application and is publicly available at `http://safe-tools.dsic.upv.es/narval`. Narval's architecture consists of three main components (i.e., Narval's core, client, and web services) that are implemented by combining a number of different technologies.

Firstly, the underlying rewriting and narrowing machinery of Narval's core has been implemented in a custom version of Maude, named Mau-Dev (Mau-Dev 2016), which provides extensions for some critical operators such as `metaReducePath` (see (Alpuente et al. 2015)) or `metaGetVariant` (see (Alpuente et al. 2017)). These extensions are necessary to fully reproduce in detail the internal reasoning modulo axioms of Maude, which is only retrievable as raw text by using the interpreter's built-in (rewriting) debugger. Narval's core consists of approximately 1800 lines of Maude and C++ source code.

Secondly, the user-friendly graphical user interface of Narval's client has been implemented by using CSS, HTML5, and Javascript. Specifically, Narval's graphic controls have been implemented by using the latest available version of Bootstrap 4, and the graph visualization feature is powered by the D3 for Data-Driven Documents library, which provides a representation-transparent approach to data visualization for the web. Without including these libraries, Narval's client consists of approximately 4400 lines of original HTML, CSS, and Javascript source code.

Finally, Narval's web service connects the core component of the system to the client user interface and consists of 10 RESTful Web Services that are implemented by using the JAX-RS API for developing Java RESTful Web Services (around 900 lines of Java source code).

6 Conclusion and Related Work

Our main motivation for developing Narval was to assist users in analysing complex software models described in Maude; however, the tool can also be used in training and education by showing the process and result of symbolic executions in a stepwise manner.

There are hardly any tools in the literature for visualizing symbolic execution trees or narrowing trees. Symbolic execution (King 1976) is a program analysis technique that is based on the interpretation of a program with symbolic values. Hahnle et al. implemented a tool, called visual symbolic state debugger, which can be used to debug sequential Java applications visually with the usage of a symbolic execution tree (Hähnle et al. 2010). This makes it easy for the bug hunter to comprehend intermediate states and the actions performed on them in order to find the origin of a bug. SEViz (Honfi et al. 2015) is another tool for interactively visualizing symbolic executions in a form of symbolic execution trees of simple .NET Framework programs. The visualization serves as a quick overview of the whole execution and helps to enhance the test input generation.

In (Alpuente et al. 2017), a graphical tool for exploring FV-narrowing computations in an equational theory (Σ, E) is described that can be used for inspecting selected parts of the folding variant tree. It can also be used to analyze whether a given theory satisfies the finite variant

[4] A kind can be seen as an error supersort of a given sort.

property, which is a fundamental requirement for the termination of FV-narrowing (and hence termination of equational unification). However, the tool of (Alpuente et al. 2017) provides little support for debugging and understanding (three-level) narrowing computations in a rewrite theory $\mathscr{R} = (\Sigma, E, R)$ since it does not provide any (R,E)-narrowing symbolic reasoning functionality. To our knowledge, Narval is the first graphical tool for the symbolic analysis of Maude rewrite theories.

References

ALPUENTE, M., BALLIS, D., FRECHINA, F., AND SAPIÑA, J. 2015. Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Computation 69*, 3–39.

ALPUENTE, M., BALLIS, D., FRECHINA, F., AND SAPIÑA, J. 2016. Assertion-based Analysis via Slicing with ABETS. *In Proc. ICLP 2016, Theory and Practice of Logic Programming 16,* 5–6, 515–532.

ALPUENTE, M., BALLIS, D., AND SAPIÑA, J. 2019. Static Correction of Maude Programs with Assertions. *Journal of Systems and Software 153*, 64–85.

ALPUENTE, M., CUENCA-ORTEGA, A., ESCOBAR, S., AND SAPIÑA, J. 2017. Inspecting Maude Variants with GLINTS. *In Proc. ICLP 2017, Theory and Practice of Logic Programming 17,* 5-6, 689–707.

CLAVEL, M., DURÁN, F., EKER, S., ESCOBAR, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. 2016. Maude Manual (Version 2.7.1). Tech. rep., SRI International Computer Science Laboratory. Available at: `http://maude.cs.uiuc.edu/maude2-manual/`.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. 2007. *All About Maude: A High-Performance Logical Framework*. Springer.

DURÁN, F., EKER, S., ESCOBAR, S., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. 2018. Associative Unification and Symbolic Reasoning Modulo Associativity in Maude. In *Proceedings of the 12th International Workshop on Rewriting Logic and its Applications (WRLA 2018)*. Lecture Notes in Computer Science, vol. 11152. Springer, 98–114.

ESCOBAR, S., SASSE, R., AND MESEGUER, J. 2012. Folding Variant Narrowing and Optimal Variant Termination. *The Journal of Logic and Algebraic Programming 81,* 7–8, 898–928.

GARAVEL, H., TABIKH, M., AND ARRADA, I. 2018. Benchmarking implementations of term rewriting and pattern matching in algebraic, functional, and object-oriented languages - the 4th rewrite engines competition. In *Proceedings of the 12th International Workshop on Rewriting Logic and Its Applications (WRLA 2018)*. Lecture Notes in Computer Science, vol. 11152. Springer, 1–25.

HÄHNLE, R., BAUM, M., BUBEL, R., AND ROTHE, M. 2010. A Visual Interactive Debugger based on Symbolic Execution. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*. ACM, 143–146.

HANUS, M. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*. Lecture Notes in Computer Science, vol. 7797. Springer, 123–168.

HONFI, D., ANDRÁS, V., AND ZOLTÁN, M. 2015. SEViz: A Tool for Visualizing Symbolic Execution. In *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*. IEEE, IEEE Computer Society Press.

KING, J. C. 1976. Symbolic Execution and Program Testing. *Communications of the ACM 19,* 7, 385–394.

Mau-Dev 2016. The Mau-Dev Website. Available at: `http://safe-tools.dsic.upv.es/maudev`.

MESEGUER, J. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science 96,* 1, 73–155.

MESEGUER, J. 2012. Twenty Years of Rewriting Logic. *The Journal of Logic and Algebraic Programming 81,* 7-8, 721–781.

MESEGUER, J. 2018. Symbolic Reasoning Methods in Rewriting Logic and Maude. In *Proceedings of the 25th International Workshop on Logic, Language, Information, and Computation (WoLLIC 2018)*. Lecture Notes in Computer Science, vol. 10944. Springer, 25–60.

MESEGUER, J. AND THATI, P. 2005. Symbolic Reachability Analysis Using Narrowing and its Application to Verification of Cryptographic Protocols. *Electronic Notes in Theoretical Computer Science 117*, 153–182.

MIDDELDORP, A. AND HAMOEN, E. 1992. Counterexamples to Completeness Results for Basic Narrowing. In *Proceedings of Algebraic and Logic Programming (ALP 1992)*. Lecture Notes in Computer Science, vol. 632. Springer, 244–258.