On the Implementation of Conditional Narrowing Modulo SMT plus Axioms*

Luis Aguirre, Narciso Martí-Oliet, Miguel Palomino, and Isabel Pita

Departamento de Sistemas Informáticos y Computación
Facultad de Informática, Universidad Complutense de Madrid, Spain
{luisagui, narciso, miguelpt, ipandreu}@ucm.es

In this work we present two prototypes that have been implemented using Maude, as well as several enhancements. We discuss the motivation for each prototype and enhancement and show their effect on the performance of the prototypes.

All the prototypes use a module called `SMTLOGIC`that handles the SMT terms and conditions at the metalevel with the help of the `META-LEVEL` module of Maude. Given a rewrite theory $\mathcal{R} = \{\Sigma, E_0 \cup B, R\}$, all the prototypes begin computing the closed under B-axioms normal form R° of the specification R, needed by the calculus. The rules in R° are the ones used in the computations. The generation of R° makes use of the `acu-coherence-completion` function in the `AX-COHERENCE-COMPLETION` module of full-maude, and an implementation of the *abstract* function in our theory, returning a term and an SMT condition from a given term, which is also used in the implementation of the unification calculus rule.

1 The Prototypes

In Maude, equational theories are defined using functional modules and rewrite theories are defined using modules. We have developed two prototypes, one as an equational theory, where we can define a define a different search engine to the one in Maude, and the other as a rewrite theory, where the optimized Maude search engine does all the in-house work, keeping track of the search tree and discarding the duplicated states that it finds. The first prototype was used to proved a different approach for the search engine that is later discussed. For this approach to work, all SMT terms in the rules and reachability problems have to be abstracted.

1.1 Improvements

Several improvements to the prototypes are discussed here, some for speed up and one for enhancement in the kind of expressible problems within the prototypes. The prototypes without any speed up strategy where also tested to check the improvement achieved with each one of these strategies.

* Partially supported by MINECO Spanish projects TRACES (TIN2015-67522-C3-3-R) and Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731).

- Constant SMT parameters are directly handled by the calculus. It was also desired to include variable SMT parameters in the rules and find feasible values for these parameters given a reachability goal, but rules get new variable names for each fresh instance used, so the relation between each occurrence of the variable parameters gets lost. This problem was solved by turning the variable SMT parameters into new SMT constants in the specification. They are always handled as constants except when the SMT solver has to be invoked. Then they are converted into SMT variables of the same name and type and the transformed condition is passed to the SMT solver. In this way the parameters get never renamed. This non speed up improvement is implemented in all versions of both prototypes.

- The same reachability problem can be generated multiple times. For the Maude engine, although all variables are existentially bounded, if we rename any variable of a term the obtained term is considered as a different one. In order to deal with this multiplicity we tried two different strategies: one renames the variables in the reachability problems that we obtain after each narrowing step, trying to generate a "canonical" version of the problems; the other one, only valid for the implementation as a functional module, keeps a list of all generated reachability problems and tries to identify the new generated reachability problems as already known through matching of the terms in the list against the new ones and checking for SMT equivalence of the, conveniently renamed using the obtained matching substitution, SMT conditions. This second strategy is closer to the desired concept of "equivalent states".

- As calling the SMT solver is an expensive operation, we also checked different approaches regarding the number of calls to the SMT solver, so we developed versions that called the SMT solver after each rewrite or unification step and "lazy" versions that only called the SMT solver after each unification step.

- The output of metaterms are very difficult to read, but metaconditions are even harder to read because the `SMT` module of Maude, that defines the syntax of the SMT terms and holds the hooks to the C++ code, has no axioms for built-in reduction, so all terms are normal forms. Some simplification capabilities where added with a new module `SMTLOGIC`, where each SMT term is converted to a corresponding `Int`, `Rat`, or `Bool` term, to let Maude perform the simplification, and then transformed back into an SMT term. Other benefit of this simplifications is to help the prototypes in the detection of already visited reachability problems. As the first purpose can also be achieved by simplifying the SMT conditions only when a solution was found, we developed versions that checked this fact.

- A call to the SMT solver is only needed when the SMT condition has changed, which may not happen all the times. We developed versions of the prototypes that were aware of this changes.

We have left as last improvements to explain two related improvements that are included in all prototypes but one, that serves as reference:

- Many times the congruence rule may be applied to try to generate a narrowing step from a subterm, having some kind, or any of its proper subterms, having other kinds, and there does not exist any rule having any of these kinds. We modified the prototypes to include a set of kinds for subterms where the congruence rule can be applied. This set is different for every given specification. Currently it is hardcoded into the prototypes, but the set can be derived from the syntax of the rewrite theory as a fixed point using as first set the kinds of the the rules and adding new kinds to the set using the definition of the operators and the rules in the specification.

- After including this set, it was observed during debugging that even if all the defined operators for a kind in the specification were meant to perform narrowing steps only at the top of these kind, if the kind of one parameter in any of these operators was in the set defined above, the congruence rule was applied. Another set of kinds was defined, holding the kinds for terms to which we can try to apply the congruence rule in any of their subterms. The use of this second set is discretional. The constant `allKinds`, holding all the kinds of the specification, can be used for this second set, guaranteeing the completeness of the results.

2 Implementation of the prototypes

We discuss here the details of the implementation of each prototype. First the common parts for both the equational-based and the rule-based are shown. The difference between both approaches is shown later.

2.1 The `SMTLOGIC` module

The `SMTLOGIC` module extends the `META-LEVEL` module of Maude, defining SMT arithmetic metaterms and boolean metaconditions. Other approaches were previously proven, as it was an extension of the syntax for the `SMT` module, to include associative, commutative, and identity axioms in the operators, but the implementation of the hooks for the SMT operators did not recognize this extension. This means that if we only use the `SMT` module in our prototypes, two SMT conditions are only recognized as equal if they are syntactically equal. For instance, the variable $X{:}Integer$ and the term $1.Integer * X{:}Integer$ are not recognized as equal by the Maude search engine. As the states in the state space of the prototypes always include at least one reachability problem, with an SMT condition on it, not detecting duplicated states due to this restriction would hugely increase the state space. We decided then to extend the `META-LEVEL` module to include as much simplification functionality as possible in the prototypes, defining a new module `SMTLOGIC`. As the arithmetic operators $+$, $-$, and $*$ are overcharged, we defined new metaoperators, $+I$, $-I$, $*I$, $+R$, $-R$, and $*R$ in our module for the metaexpressions, also separating integer expressions from real expressions. We defined top sorts `SmtCondi` for `Boolean` metaterms and `SmtTerm`. `Boolean` metavariables may appear in a non-`Boolean` metaterm in the conditional part of a ternary operator *if* ... *then* ... *else*. Thus, the sort `SmtAterm` is reserved for ground arithmetic metaterms and also for arithmetic metaterms having neither `Boolean` metavariables in them nor metaoperators with sort `Real` (this second restriction will be removed in next versions of SMTLOGIC): simplification of metaterms with Boolean metavariables in them proved not to be safe in our experiments, so we decided not to support it. The chosen subsort ordering

```
subsort Term < SmtCondi .
subsort Term < SmtATerm < SmtTerm .
subsort GroundTerm < GroundSmtTerm < SmtATerm .
subsort GroundTerm < GroundSmtCondi < SmtCondi .
```

reflects several already pre-existing relations between META-LEVEL and SMTLOGIC and others that we desired to have:

- variables with sort Boolean become terms with sort Term in the META-LEVEL and with sort SmtCondi in SMTLOGIC,
- non Boolean SMT variables become terms with sort Term in the META-LEVEL and with sort SmtATerm in SMTLOGIC,
- non Boolean SMT constants become terms with sort GroundTerm in the META-LEVEL and with sort GroundSmtTerm in SMTLOGIC, a sort that is needed to characterize the terms with sort GroundSmtCondi, and
- the metaconstants 'true.Boolean and 'false.Boolean have sort GroundTerm in the META-LEVEL and sort GroundSmtCondi in SMTLOGIC, a sort that makes easy for us to remove all redundant ground subexpressions that appear in any satisfiable SMT condition just by checking the sort of the subterms in the SMT condition.

The module has several equations that intend to remove conditions that are subsumed by another condition, simplify conditions and also simplify some simple arithmetic operations like those involving the identity constants for each operation. Also some reordering is done by exchanging the left and right terms in some expressions, with the purpose of achieving a "canonical" representation of the expressions.

The function smtSimplify simplifies SMT conditions. Given an SMTLOGIC metacondition, the function downSmt, that modifies the arithmetic and boolean operators and also modifies the sort of each constant and variable, is called on each arithmetic SMT subterm or ground boolean SMT subterm, generating new metasubterms. When applying the META-LEVEL operator downTerm to each one of these new metasubterms, we obtain terms with sorts (Int, Rat, or Bool) that Maude simplifies automatically. Then the META-LEVEL operator upTerm followed by a call to another SMTLOGIC operator MaudeSort2SmtSort that replaces the sort of each variable or constant with the corresponding one for SMT expressions. Finally, in the case of arithmetic expressions a call to the SMTLOGIC operators term2iExpr, term2rExpr, or term2cExpr restores the SMTLOGIC metaoperators in each metasubterm.

There exist also a pair of functions, constants2variables and variables2constants that are used in the extension of the prototypes to support variable parameters in the rules of the specifications, as explained in Subsection 1.1.

2.2 Common functions for all prototypes

All the prototypes have three constants, except the prototype in cnmsmtpb-FullCongruence.maude that only has one, whose values must be defined before loading the prototype. The value in constant `target`, common to all the prototypes, is the name of the module that holds the specification where we want to solve our reachability goals. The other two constants, `congruenceKinds1` and `congruenceKinds2`, hold the kinds whose terms may get the rule `congruence` applied and the kinds for subterms where the `congruence` rule can be applied, respectively, as explained in Subsection 1.1.

The inclusion of constant `target` in the prototypes allows us to memoize the normal form R° of the specification R and other constant parameters derived from it. In this way, the computations do not have to include the specification, or a transformed version of it, as part of the term that represents the current state when the prototype is a rewrite theory, or as part of the whole computation when the prototype is an equational theory.

The normal form of the rules includes the kind of every rule and every rewrite condition in the rule. As the kind of each reachability subgoal in the original problem is also computed, we only have to select rules with the same kind as the current subgoal, using the matching capabilities of Maude, to try to unify the subgoal with the left side of the rule as a first step towards generating a narrowing step, discarding the rest of the rules. The normal form of the rules also includes a serial number that is used with some memoized operators. The set of normal forms of the rules together with their kind and serial number is memoized in the operator `normalRls`.

A version of the specification without the rules is also memoized. It is used for equational simplification of the metaterms generated by the prototypes. The other memoized set holds the SMT variable parameters that are inferred from the specification. It consists in all the constants with SMT sort.

The reachability problem that we want to solve, P, is represented in our extension of the `META-LEVEL` as a term with sort `OrigPrCond`. The signature for this sort is:

```
op nilOP : -> OrigPr [ctor] .
op _=>*_ : Term Term -> OrigPr [ctor] .
op _&_ : OrigPr OrigPr -> OrigPr [ctor assoc id: nilOP] .
op _&&_ : OrigPr Term -> OrigPrCond [ctor] .
```

For each reachability problem P in our examples, we have obtained each term with sort `Term` using the `upTerm` operator on the corresponding term of P. Each subterm with the form `u =>* v` is the metarepresentation of a subgoal of P, the `&` symbol joins the different subgoals of P, and the term after the `&&` symbol is the metarepresentation of the reachability formula of P.

This term with sort `OrigPrCond` is the processed by operators `op2Rc`, `term2cExpr`, and `getVarListOpc`, producing a term with sort `ReachProblem`, the key sort of the prototypes. The signature for this sort is:

```
op nilR : -> RuleCondi [ctor] .
```

```
op errorR : -> RuleCondi [ctor] .
op _/_=>*_ : Kind Term Term -> RuleCondi [ctor] .
op _/_/_=>0_ : Int Substitution Term Term -> RuleCondi [ctor] .
op _&_ : RuleCondi RuleCondi -> RuleCondi [ctor assoc id: nilR] .
op _||_ : RuleCondi SmtCondi -> ReachProblem [ctor] .
```

Subproblems, together with their kind, are metarepresented in the sort `RuleCondi`. This sort, together with the metarepresentation in `META-LEVEL` of the left and right side of the rule and the SMT condition, is also used in the metarepresentation of the conditional rules:

```
op _&&_ : RuleCondi Term -> FullCondi [ctor] .
op _=>_if_ : Term Term FullCondi -> SmtRule [ctor] .
op kNone : -> KindIntSmtRuleSet [ctor] .
op _/_/_ : Kind Int SmtRule -> KindIntSmtRule [ctor] .
op __ : KindIntSmtRuleSet KindIntSmtRuleSet
        -> KindIntSmtRuleSet [ctor assoc comm id: kNone] .
```

The memoized set `normalRls`, with sort `KindIntSmtRuleSet`, includes the normal form of all the rules, together with their kind and serial number.

Also common to all prototypes are:

- the generation of fresh rules, through the `freshRule` operator.
- The simplification of satisfiable SMT conditions in a problem, through the `simplifySatisfSC` operator. This simplification includes the removal of ground subterms; the removal of temporal variables, of the form $\#n$, from the SMT condition if their value in the computed solution so far is another variable, by applying an inverse substitution to all the problem; the extraction of SMT metaconditions of the form 'metavariable equals metaconstant' as assignments in the computed solution; and the extraction of SMT conditions of the form 'metavariable equals ground smt metacondition' as assignments of the form $v \mapsto \mathit{true}$ or $v \mapsto \mathit{false}$ in the computed solution, by transforming the ground smt metacondition in the metalevel into the corresponding ground smt condition, and letting Maude reduce this ground counterpart to *true* or *false*. When a new assignment is generated, this new assignment is also applied to the rest of the problem, thus taking rid of all the instances of the metavariable being removed.
- The simplification of the `RuleCondi` term of the `Problem`, through the `simplifySatisfRC` operator, which works like `smtSimplify` but in a broader way, since it recognizes the whole signature of the specification and it searches for the SMT subterms in the given `RuleCondi` term. This prototype only simplifies the SMT conditions of the answers and it verifies the satisfiability of an SMT Condition only when it has been changed by the unifier applied to it.

One design decision that it is also shared by all prototypes concerns the position where each unifier is tried in the search tree of the reachability problem, search tree that is generated using the rules in the `target` module and controlled by the prototype, in contraposition with the search tree generated by Maude, using the rules of the prototype, that has a fixed behavior. We have designed the rules of the prototypes in a way that the set of unifiers for any given pair of terms is generated in a depth way instead of in a width way, i.e., if the first unifier is tried at level n of the search tree, then the second unifier will be tried at level $n+1$, and so on, while level n of the search tree can still be used by other narrowing paths. This design allows the search tree of the reachability problem to support potentially infinite unification of terms, like in the associative case, without losing completeness.

The files:

- `full-maude.maude`,
- `smtlogic.maude`, which, in turn, loads the file `smt.maude`, and
- `toasts.maude`, defining the module `TOASTS`, used in the reachability problems of our tests,

are loaded by each prototype test file prior to loading the prototype and running the tests.

2.3 Rule-based prototype

We explain the implementation of the prototype `cnmsmtpbNew2SC`, which gives the best scores in our tests. The test file exampleToasts.maude, loads all the needed files, including `cnmsmtpbNew2SC.maude` that holds the prototype to be used in a module named `CNMSMTPB`.

The file exampleToasts.maude defines a module named `TEST` that includes all loaded modules, so we can formulate our reachability problems in this module.

As the module under test is `TOASTS`, the module `CNMSMTPB` has to be modified prior to its loading, as previously explained. We add the equations

```
eq target = 'TOASTS .
eq congruenceKinds1 = getKind(moduleNoRls, 'System) .
eq congruenceKinds2 = getKind(moduleNoRls, 'Kitchen) .
```

meaning that:

- the target module is `TOASTS`,
- rule congruence has to be applied only to terms whose sort is in the connected component of sort `System`, and
- when rule congruence is applied it will be applied only to subterms whose sort is in the connected component of sort `Kitchen`.

There exists an operator allKinds that returns the name of all connected components of the target module. It can be used in the right side of the last two equations to achieve the standard behavior of rule congruence.

Using the module `TEST` we can formulate reachability problems like:

```
search [9, 90] in TEST :
problem(upTerm(1 / W / 0 ; zt zt / 0) =>* upTerm(N2 / zt / Y ; zt zt / 1) &&
        upTerm(Y < 12)) =>* SOL .
```

This search command asks for solutions to the reachability problem: is it possible from a `State` with just one `Toast` in the `Tray` and another `Toast` (`W`) in the `Tray` to reach a `State` with one well-cooked `Toast` in less than 12 seconds?

The variable `SOL` has sort `Solution`, a subsort of sort `Problem` that only solutions to the reachability problem may have. What we are asking Maude is to find nine rewrite paths from the normal form of the stated problem to a term with sort `Solution`, where each rewrite path may have no more that ninety rewrite steps.

The operator `problem` returns as output a term P with sort `Problem` of the form `K / L =>* R & RC || SC ; OVL ; NV ; AN`, where `OVL` is the list of variables in P, `NV` is the next number of variable, `AN` is `none`, the computed answer so far, and `K / L =>* R & RC || SC` is the reachability problem to solve. Here, `K / L =>* R` represents the first subgoal of P, together with its kind `K`, `RC` holds the rest of subgoals of P and their respective kinds, and `SC` is the reachability formula of P. The SMT solver is called with `metaCheck(smtModule, downSmt(term2cExpr(TC), false))`, to verify that the SMT condition in the original problem is satisfiable, since this is an invariant of all the problems in the search tree generated by the prototypes.

Narrowing rules in the prototype Some rules have two versions: one for constants and another one for operators, forbidding the use of the rule in case that the left term of the subgoal is a variable. The version for operators of these rules is the one being displayed in this subsection.

Given a `Problem` of the form `K / L =>* R & RC || SC ; OVL ; NV ; AN`, there are only two rules that can be applied to it by the Maude search engine: `[t]` and `[u]`.

Rule `[t]` has two versions. The one for operators is:

```
rl [t] : K / F[TL] =>* R & RC || SC ; OVL ; NV ; AN
  => K / F[TL] =>1 newVar(NV, K) &
     K / newVar(NV, K) =>* R & RC || SC ; OVL ; s(NV) ; AN .
```

It is a direct implementation of rule transitivity of the calculus

- $[t]$ transitivity

$$\frac{u \to v\ (\wedge\, \Delta) \mid \phi}{u \to^1 x_k, x_k \to v\ (\wedge\, \Delta) \mid \phi}$$

where $u \notin \mathcal{X}$, $k = [ls(u)]$, and x_k fresh variable

Observe that the prototype wastes no time obtaining the kind of the new variable x_k, since it is precalculated. In the case of complex terms this operation can demand a big number of rewrites. Observe also that this intermediate state, no narrowing step has still been generated by the prototype, is considered by Maude as a new state of its computation.

Rule `[u]`:

```
crl [u] : K / L =>* R & RC || SC ; OVL ; NV ; AN
  => 1 / SU / L =>0 R0 & RC || SC and term2cExpr(TC) ; OVL ; NV'' ; AN
  if (R0 / TC ; NV') := abstract(R, NV) /\
     {SU, NV''} := metaUnify(moduleNoRls, L =? R0, NV', 0) .
```

is an example of the depth oriented approach in the generation of unifiers. It generates the abstract form of R, R0/TC, and the first unifier of L =? R0 in the target module, prior to the appliance of rule unification. Observe that the SMT condition associated to the normal form, SC, is added to the reachability formula using the operator term2cExpr, from SMTLOGIC, that turns a META-LEVEL SMT term into a term in SMTLOGIC, with sort SmtCondi.

Once that rule [u] has been applied to a problem, the syntax of the resulting problem only enables two rules to be applied to it: rule [u1] and rule [u2]. Rule [u1] reflects the choice of using the current unifier in the narrowing path being generated:

```
rl [u1] : NA / SU / L =>0 R0 & RC || SC ; OVL ; NV ; AN
  => u1(NA / SU / L =>0 R0 & RC || SC ;
        OVL ; NV ; AN, (SC <<* su2smtSu(SU, smtParams))) .
```

Following the defined strategy for this prototype, after this rule is applied, one of the following equations is used to reduce the resulting term, depending on the relation between the previous and new SMT conditions, SC and SC':

```
eq u1(NA / SU / L =>0 R0 & RC || SC ; OVL ; NV ; AN, SC)
  = renameVars(simplifySol(RC << SU || SC ;
                           OVL ; NV ; simplifyAnswer((AN .. SU) |>* OVL))) .
eq u1(NA / SU / L =>0 R0 & RC || SC ; OVL ; NV ; AN, SC')
  = if metaCheck(smtModule, downSmt(SC', false))
    then renameVars(simplifySol(simplifySatisfSC(RC << SU || removeGSCs(SC') ;
         OVL ; NV ; simplifyAnswer((AN .. SU) |>* OVL), 'true.Boolean)))
    else nilP fi [owise] .
```

The first equation does not check the SMT condition if it has not changed; the second equation checks the satisfabiability of the new SMT condition, SC', when it differs from the previous one, SC, calls removeGSCs(SC') to remove all the ground SMT conditions from the satisfiable SMT condition SC', and calls the operator simplifySatisfSC, whose simplification behaviour has already been explained. In both cases:

- the new substitution, SU, is composed with the computed answer so far, AN, and a filter |>* is applied to keep only the assignments for the variables that appear in the original variables list, OVL.,
- then the resulting answer is simplified, calling simplifyAnswer, and
- a call to simplifySol is made. If the resulting problem has sort Solution then the SMT condition SC is further simplified.

Rule [u2] reflect the choice of discarding the current unifier and try to use another one to build the narrowing path:

```
crl [u2] : NA / SU / L =>0 RO & RC || SC ; OVL ; NV ; AN
  => s(NA) / SU' / L =>0 RO & RC || SC ; OVL ; NV' ; AN
     if {SU', NV'} := metaUnify(moduleNoRls, L =? RO, NV, NA) .
```

This is a conditional rule that can only be applied if the number `NA` unifier of `L =? RO` exists, with value `{SU', NV'}`. The resulting problem is the same as the previous one with the exception that the next unifier number is `s(NA)` instead of `NA` and the previous substitution, `SU`, is replaced with the new one, `SU'`.

Rules `[u]`, `[u1]`, and `[u2]`, together, implement rule unification of the calculus using the explained depth-oriented approach.

– $[u]$ unification

$$\frac{u \to v \ \wedge \ \Delta \mid \phi}{\Delta\theta \mid \psi}$$

where $abstract_{\Sigma_1}(v) = \langle \lambda\bar{x}.v^\circ; \theta^\circ; \phi^\circ \rangle$, θ in $CSU_B(u = v^\circ)$, $vars(\psi) \subseteq vars((\phi \wedge \phi^\circ)\theta)$, $E_0 \vdash \psi \Leftrightarrow (\phi \wedge \phi^\circ)\theta$, and ψ is satisfiable

If the last rule applied is rule `[t]`, there are two choices now: rule `[c]` and rule `[n]`.

Rule `[c]` has two versions, one for constant subterms and the other one for operator subterms. The one for operator subterms is:

```
crl [c] : K / F[TL, F'[NETL], TL'] =>1 V &
          K' / L =>* R & RC || SC ; OVL ; NV ; AN
  => K'' / F'[NETL] =>1 V' & K' / L << (V <- F[TL, V', TL']) =>* R &
     RC || SC ; OVL ; s(NV) ; AN
     if K in congruenceKinds1 /\ TY := leastSort(moduleNoRls, F'[NETL]) /\
     not (TY in smtSorts) /\  K'' := getKind(moduleNoRls, TY) /\
     K'' in congruenceKinds2 /\ V' := newVar(NV, K'') .
```

It is an implementation of rule congruence of the calculus, with the added optimization of the two sets of kinds that control where this rule may be applied:

– $[c]$ congruence

$$\frac{u|_p \to^1 x_k, u[x_k]_p \to v \ (\wedge \, \Delta) \mid \phi}{u_i \to^1 y_{k'}, u[y_{k'}]_{p.i} \to v \ (\wedge \, \Delta) \mid \phi}$$

where $u|_p = f(u_1, \ldots, u_n)$, $u_i \notin \mathcal{X} \cup \mathcal{T}_{\Sigma_0}(\mathcal{X}_0)$, $k' = [ls(u_i)]$, $1 \leq i \leq n$, and $y_{k'}$ fresh variable

Rule `[n]` is the other example of the depth oriented approach in the generation of unifiers. It lets the search engine choose the rule of the specification to be applied. Then the first unifier between the left term and the head of a fresh version of the rule is computed:

```
crl [n] : K / F[TL] =>1 V & K' / L =>* R & RC || SC ; OVL ; NV ; AN
  => 1 / NV', L0, R', RC' / SU / F[TL] =>1 V &
     K' / L =>* R & RC || SC and term2cExpr(TC) ; OVL ; NV'' ; AN
  if K / RN / SRL KISRLS := normalRls /\
     NV' ; L0 => R' if RC' && TC := freshRule(RN, SRL, NV) /\
     {SU, NV''} := metaUnify(moduleNoRls, F[TL] =? L0, NV', 0) .
```

In a similar way to unification, there exist two choices now, rules [r1] and [r2].

Rule [r1]:

```
crl [r1] : NA / NV', L0, R', RC' / SU / T =>1 V &
           K' / L =>* R & RC || SC ; OVL ; NV ; AN
  => s(NA) / NV', L0, R', RC' / SU' / T =>1 V &
      K' / L =>* R & RC || SC ; OVL ; NV'' ; AN
  if {SU', NV''} := metaUnify(moduleNoRls, T =? L0, NV', NA) .
```

reflects the choice of using another unifier between the head of the rule and the left term, if it exists.

Rule [r2]:

```
rl [r2] : NA / NV', L0, R', RC' / SU / T =>1 V &
          K' / L =>* R & RC || SC ; OVL ; NV ; AN
  => r2(NA / NV', L0, R', RC' / SU / T =>1 V & K' / L =>* R &
        RC || SC ; OVL ; NV ; AN, (SC <<* su2smtSu(SU, smtParams))) .
```

reflects the choice of using the current unifier, invoking operator r2. In this version of the prototype the operator is defined by two equations:

```
eq r2(NA / NV', L0, R', RC' / SU / T =>1 V &
      K' / L =>* R & RC || SC ; OVL ; NV ; AN, SC)
  = renameVars((RC' << SU) & K' / (L << (SU ; V <- (R' << SU))) =>*
               (R << SU) & (RC << SU) || SC ;
                OVL ; NV ; simplifyAnswer((AN .. SU) |>* OVL)) .
eq r2(NA / NV', L0, R', RC' / SU / T =>1 V &
      K' / L =>* R & RC || SC ; OVL ; NV ; AN, SC')
  = if metaCheck(smtModule, downSmt(SC', false))
    then renameVars(simplifySatisfSC((RC' << SU) &
                    K' / (L << (SU ; V <- (R' << SU))) =>* (R << SU) &
                    (RC << SU) || removeGSCs(SC') ; OVL ; NV ;
                    simplifyAnswer((AN .. SU) |>* OVL), 'true.Boolean))
    else nilP fi [owise] .
```

The first equation is applied when the SMT condition has not changed. The second equation calls the SMT solver with the modified SMT condition and simplifies the condition if it is satisfiable.

Rules [n], [r1], and [r2], together, implement rule rewrite of the calculus under the depth-oriented approach.

- $[r]$ rewrite

$$\frac{u|_p \to^1 x_k, u[x_k]_p \to v\ (\wedge\, \Delta) \mid \phi}{(C \wedge u[r]_p \to v\ (\wedge\, \Delta))\theta \mid \psi}$$

where $u|_p \notin \mathcal{X}$, $l^\circ \to r$ *if* $C \mid \phi^\circ$ fresh rule in R°, θ in $CSU_B(u|_p = l^\circ)$, $vars(\psi) \subseteq vars((\phi \wedge \phi^\circ)\theta)$, $E_0 \vdash \psi \Leftrightarrow (\phi \wedge \phi^\circ)\theta$, and ψ is satisfiable

2.4 Equation-based prototype

We explain the implementation of the prototype `cnReduxNew4`, where the search for equivalent problems is used to prun the search tree. The test file exampleToastsRedux.maude, loads all the needed files, including `cnReduxNew4.maude`, that holds the prototype to be used in a functional module named `CNMSMTPB`.

As in the rule-based prototype, the file exampleToastsRedux.maude defines a module named `TEST` that includes all loaded modules, and the same equations defining `target`, `congruenceKinds1`,and `congruenceKinds2` are included in module `CNMSMTPB` prior to loading the test file.

When the search engine of Maude is used, the number of answers and depth of search is included in the `search` command. In the equation-based prototype, this two parameters are now included in the function `problem`, so the equivalent reachability problem in module `TEST` to the one shown in the other prototype is now:

```
red in TEST :
problem(9, 90, (upTerm(1 / W / 0 ; zt zt / 0)
        =>* upTerm(N2 / zt / Y ; zt zt / 1) && upTerm(Y < 12))) .
```

where the operator `problem` is now:

```
  eq problem(ANSWERS, DEPTH, (OP && TC))
   = problem*(ANSWERS, DEPTH, problemCore(OP, nilR, TC, 0),
             getVarListOpc(OP && TC), normalRlsNV) .
```

The operator `problemCore` generates the abstracted version of the problem; the constant `normalRls` holds in this prototype a normalized version of the rules where all the terms in the rules, instead of only the head, have been abstracted. This approach has also been tested in one version of the rule-based prototype.

There are new sorts, aimed to implement the rewrite engine of the functional module:

```
op nilNP : -> NatProblem [ctor] .
op _|_ : Nat Problem -> NatProblem [ctor prec 81] .
op _||_ : NatPrList NatPrList -> NatPrList [ctor assoc id: nilNP prec 83] .
op __ : ProblemSet ProblemSet -> ProblemSet [ctor assoc comm id: nilP prec 81] .
op _/_/_/_/_ : Nat Nat NatPrList NatPrList ProblemSet -> State [ctor] .
```

Sort `NatProblem` associates to each problem the remaining depth of the search. If it reaches zero, the problem is discarded. Sort `ProblemSet` keeps all visited problems. Sort `State` holds the full computation which consists in the state number, the number of answers left to find, a list of current problems and depth of each one, a list of found solutions, and the set of already visited problems. The operator `processState` implements the search engine logic. It uses the operator `processNatPr` to generate the candidate children of a problem in the search tree, which are then pruned with operator `isNewProblem` that searches through the whole list of already visited problems using operator `sameProblem`:

```
ceq sameProblem((K / L' =>* R' & RC' || SC' ; OVL ; NV' ; AN'),
               (K / L =>* R & RC || SC ; OVL ; NV ; AN))
```

```
  = true if {SU, SU', NV''} := metaDisjointUnify(thisModule,
             upTerm(K / L =>* R & RC) =?
             upTerm(K / L' =>* R' & RC'), max(NV, NV'), 0) /\
     isRenaming(SU) /\ isRenaming(SU') /\
     metaCheck(smtModule, downSmt((SC <<* su2smtSu(SU, smtParams)) ===
                (SC' <<* su2smtSu(SU', smtParams)) , false)) .
eq sameProblem(PR, PR') = false [owise] .
```

Identical problems have been previously removed, using the multiset axioms of sort `ProblemSet`, with the equation:

```
eq processState(STATES, ANSWERS, DEPTH | PR || NPO, NPL, SOLS, PR PS)
  = processState(STATES, ANSWERS, NPO, NPL, SOLS, PR PS) .
```

Narrowing rules in the prototype While in the rule-based prototype the narrowing rules are nondeterministically applied by the search engine, in the equation-based prototype a list of problems is generated, for any given problem, in a deterministic way. For instance:

```
eq processNatPr(DEPTH | K / L =>* R & RC || SC ; OVL ; NV ; AN)
 = u(DEPTH | K / L =>* R & RC || SC ; OVL ; NV ; AN) ||
   y(DEPTH | K / L =>* R & RC || SC ; OVL ; NV ; AN) .
```

describes the fact that rules unification and transitivity of the calculus can be applied to a reachability problem, so a list with two problems is generated.

The operator `u` generates one problem of the form `DEPTH | NA / SU / L =>0 R0 & RC || SC ; OVL ; NV ; AN`. When a problem of this form is processed by the rewrite engine, it generates a list with two problems:

```
eq processNatPr(DEPTH | NA / SU / L =>0 R0 & RC || SC ; OVL ; NV ; AN)
 = u1(DEPTH | NA / SU / L =>0 R0 & RC || SC ; OVL ; NV ; AN) ||
   u2(DEPTH | NA / SU / L =>0 R0 & RC || SC ; OVL ; NV ; AN) .
```

The first one reflects the choice of using the unifier `SU`, which is the unifier number `NA`; the second problem reflects the choice of discard this unifier and follow the unification from unifier number `NA+1`. Again, the depth oriented approach in the generation of unifiers is followed.

The operator `t` generates one problem of the form `DEPTH | K / T =>1 V & K' / L =>* R & RC || SC ; OVL ; NV ; AN`. When a problem of this form is processed by the rewrite engine, it generates a list with two problems:

```
eq processNatPr(DEPTH | K / T =>1 V & K' / L =>* R & RC || SC ; OVL ; NV ; AN)
 = c(DEPTH | K / T =>1 V & K' / L =>* R & RC || SC ; OVL ; NV ; AN, empty) ||
   n(normalRlsKind(K), DEPTH | K / T =>1 V &
                              K' / L =>* R & RC || SC ; OVL ; NV ; AN) .
```

Operator `c` reflects the choice of using rule congruence of the calculus. If the term `T` has the form `F[TL]`, a list of problems of the same form is generated. There is one problem in the list for each term in `TL` which is neither a variable nor an SMT term. The sets `congruenceKinds1` and `congruenceKinds2` are used to prun this list.

Operator `n` reflects the choice of using rule rewrite of the calculus. The operator `normalRlsKind` returns the memoized set of all rules having kind `K`. A list of problems is generated, where for each one of these rules a problem is added to the list if there exists one unifier between the head of the rule and the non-variable term `T`.

When a problem of this list is processed by the rewrite engine, it generates a list with two problems:

```
eq processNatPr(DEPTH | NA / NV', LO, R', RC' / SU / T =>1 V &
                        K' / L =>* R & RC || SC ; OVL ; NV ; AN)
 = r1(DEPTH | NA / NV', LO, R', RC' / SU / T =>1 V &
              K' / L =>* R & RC || SC ; OVL ; NV ; AN) ||
   r2(DEPTH | NA / NV', LO, R', RC' / SU / T =>1 V &
              K' / L =>* R & RC || SC ; OVL ; NV ; AN,
              (SC <<* su2smtSu(SU, smtParams))) .
```

The first one reflects the choice of discarding the unifier `SU`, which is the unifier number `NA`, and follow the generation of a narrowing step from unifier number `NA+1`; the second problem reflects the choice of using the unifier `SU`, which has number `NA`, to try to generate a narrowing step. Once more, the depth oriented approach in the generation of unifiers is followed. The operator `r2` checks the satisfiability of the SMT condition only if it has changed.

3 Testing the prototypes

3.1 Versions of the prototypes

The name of the versions for the rewrite-based prototype always begins with `cnmsmtpb`. They are:

- `cnmsmtpbFullCongruence`: version with no optimizations. It serves as reference.
- `cnmsmtpb`: base for the rest of the rewrite-based versions, rule congruence restricted to allowed kinds, simplification of problems and SMT conditions.
- `cnmsmtpbNew`: no simplification of problems, simplification of SMT conditions.
- `cnmsmtpbNew2`: no simplification of problems, simplification of SMT conditions only for solutions.
- `cnmsmtpbNew2SC`: like `cnmsmtpbNew2`, with check for satisfiability only if the SMT condition has changed.
- `cnmsmtpbNew2NoRename`: like `cnmsmtpbNew2`, without renaming of variables.
- `cnmsmtpbNew2Lazy`: like `cnmsmtpbNew2`, with SMT check only in the `unification` rule.
- `cnmsmtpbNew3`: like `cnmsmtpbNew2`, with full abstraction of problems and rules.

The name of the versions for the equation-based prototype always begins with `cnRedux`. Only `cnReduxNew3` and `cnReduxNew4` use the concept of equivalent state to prun the search tree. The others serve as comparison of the performance of both prototypes when the same approach is used. They are:

- `cnRedux`: base for the rest of the equation-based versions, rule congruence restricted to allowed kinds, simplification of problems and SMT conditions.
- `cnRedux2`: no simplification of problems, simplification of SMT conditions only for solutions.
- `cnRedux2Lazy`: like `cnRedux2`, with SMT check only in the `unification` rule.
- `cnReduxNew2`: like `cnRedux2`, with abstraction of problem.
- `cnReduxNew2simpl`: like `cnReduxNew2`, with simplification of SMT conditions.
- `cnReduxNew3`: full abstraction of problems and rules, no renaming and no simplification of problems. Simplification of SMT conditions for solutions. Search tree pruned using the concept of equivalent states.
- `cnReduxNew4`: like `cnReduxNew3`, with check for satisfiability only if the SMT condition has changed.