

# MethodsX

## A Partial Evaluation Methodology for Optimizing Rewrite Theories Incrementally --Manuscript Draft--

<b>Keywords:</b>	Concurrent and non-deterministic system modeling; Algebraic specification; Code optimization; Narrowing-based partial evaluation; Symbolic reasoning; Rewriting logic; Maude
<b>Authors:</b>	María Alpuente
	Demis Ballis
	Santiago Escobar
	Julia Sapiña
	Daniel Galán

## ABSTRACT

Partial evaluation (PE) is a branch of computer science that achieves code optimization via specialization. This article describes a PE methodology for optimizing rewrite theories that encode concurrent as well as nondeterministic systems by means of the Maude language. The main advantages of the proposed methodology can be summarized as follows:

- An automatic program optimization technique for rewrite theories featuring several PE criteria that support the specialization of a broad class of rewrite theories.
- An incremental partial evaluation modality that allows the key specialization components to be encapsulated at the desired granularity level to facilitate progressive refinements of the specialization.
- All executability theory requirements are preserved by the PE transformation. Also the transformation ensures the semantic equivalence between the original rewrite theory and the specialized theory under rather mild conditions.

## SPECIFICATIONS TABLE

<b>Subject Area</b>	Computer Science
<b>More specific subject area</b>	Program optimization
<b>Method name</b>	Narrowing-based program specialization for rewrite theories
<b>Name and reference of original method</b>	M. Alpuente, D. Ballis, S. Escobar, J. Sapiña. Optimization of rewrite theories by equational partial evaluation. In <i>Journal of Logical and Algebraic Methods in Programming</i> , vol. 122, 2022. DOI: 10.1016/j.jlamp.2021.100729
<b>Resource availability</b>	The <i>iPresto</i> system, available at: <a href="http://safe-tools.dsic.upv.es/ipresto/">http://safe-tools.dsic.upv.es/ipresto/</a>

## 1. Short Introduction

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input that are known statically [7]. Partial evaluation is accomplished by detecting program fragments depending exclusively on specialized variables whose values are fixed, and by symbolically precomputing these fragments. The residual or specialized program runs faster (and yields the same result as running the original program on all of its input data) because the aforementioned fragments have been removed or compressed. As a classic simple example consider the power function that calculates  $x^n$  for natural numbers:

```
power(0, x) = 1
power(n, x) = if n is even then square(power(n/2, x))
              else x * power(n-1, x)
```

Assuming that `n` is set to 5, PE is able to reduce this program to the following one

```
power5(x) = x * square(square(x))
```

which is far more efficient, since the time-expensive call in the else branch of the original if-statement has been completely removed.

In the literature there exist few attempts to apply partial evaluation to concurrent languages (see, e.g., [8]). This paper presents a methodology, which is based on the partial evaluation framework originally presented in the companion paper [2], for specializing concurrent software systems modeled as Maude rewrite theories. The methodology is fully automatic and has been implemented in the *iPresto* system that can be remotely used via a user-friendly interface at <http://safe-tools.dsic.upv.es/ipresto>. Figure 1 shows *iPresto*'s basic workflow. Once the user has loaded the program into *iPresto*, some checks and transformations are automatically performed to ensure the applicability of the method, then the partial evaluation process starts and automatically produces a specialized Maude rewrite theory according to a chosen specialization strategy.

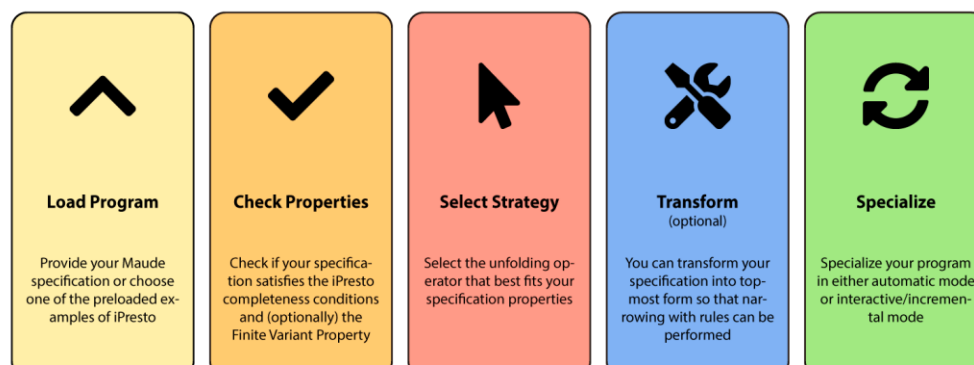


Figure 1. *iPresto* workflow

## 2. On Maude Rewrite Theories

Maude is a high-performance language and system that efficiently implements Rewriting Logic [9], which is a logic of change that seamlessly unifies a wide variety of models of concurrency. A Maude rewrite theory  $\mathcal{R}$  is essentially made up of two components, E and R, where

- E is a canonical equational theory that models system states as terms of an algebraic data type by means of equations defining the system's deterministic functionality, and
- R is a set of rewrite rules that specify transitions between states and which are assumed to be coherent w.r.t. the equations in the set E.

Canonicity of E and coherence between R and E are fundamental executability properties that guarantee the soundness and completeness of Maude's evaluation mechanism [5]. Note that the rewrite rules in R may be non-confluent as well as non-terminating; hence, a Maude rewrite theory provides an adequate computation model for the specification of non-deterministic as well as concurrent software systems exhibiting infinite behaviors.

Algebraic structures often involve axioms like associativity, commutativity, and/or identity (also known as unity) of function symbols, which cannot be handled by ordinary term rewriting but instead are handled implicitly by working with congruence classes of terms. More precisely, the equational theory E is decomposed into a disjoint union  $E = D \cup Ax$ , where

- the set D consists of equations that are implicitly oriented from left to right as rewrite rules (and operationally used as simplification rules), and
- Ax is a set of algebraic axioms that are implicitly expressed as function attributes and are mainly used for Ax-matching.

Rewrite theories are executed by rewriting states using equational rewriting, i.e., rewriting with the rewrite rules in R modulo the equations D and axioms Ax in E. We consider *topmost* Maude rewrite theories, that is, Maude rewrite theories in which rewrites can only happen at the state top-level position. This implies that no local state changes are allowed: in other words, each rewrite step completely replaces a state  $s_1$  with a new term representing the derived state  $s_2$ .

The symbolic engine of Maude's equational theories is based on narrowing. Roughly speaking, narrowing can be viewed as a generalization of term rewriting that allows free variables in terms (as in logic programming) and that non-deterministically reduces these partially instantiated function calls by using unification (instead of pattern-matching) at each reduction step. For instance, the input call `power(N, X)` narrows to 1 with computed substitution  $\{N \rightarrow 0\}$ . Besides rewriting with rules modulo equations and axioms, Maude has provided full native support for narrowing computations in rewrite theories since Maude version 3.0 (2020). Narrowing computations can be systematically represented by a (possibly infinite) finitely branching tree, which we call *narrowing tree*.

A rewrite theory  $\mathcal{R} = (E, R)$ , with  $E = D \cup Ax$ , can be symbolically executed in Maude by using narrowing at *two levels*: (i) narrowing with the equations D (explicitly oriented as rewrite rules) modulo the axioms Ax; and (ii) narrowing with the (typically non-confluent and non-terminating) rules of R modulo the equational theory E. Completeness of level (ii) narrowing in Maude requires topmost rewrite theories. Nevertheless, *iPresto* implements a novel transformation called topmost extension that automatically achieves in one shot the coherence of rules with respect to equations and axioms and the topmost requirement on rules [1].

Our PE scheme is based on level (i) narrowing, which is efficiently implemented in Maude by means of the folding variant narrowing (FVN) strategy [6]. Completeness of FVN is only guaranteed when the equational theory satisfies the finite variant property (FVP), that is, every term t has a finite number of most general variants so that the folding variant narrowing tree for t is finite. Equational theories that satisfy (resp., do not satisfy) the FVP are called *finite variant* (resp., *non-finite variant*) equational theories.

The FVP property is semi-decidable. A semi-decision procedure for the FVP is given in [10] that works by computing the variants of all flat terms  $f(X_1, \dots, X_n)$  for any n-ary operator f in the theory and pairwise-distinct variables  $X_1, \dots, X_n$  (of the corresponding sort); the theory does have the FVP iff there is a finite number of most general variants for every such term.

### 2.1. An Example of a Maude Rewrite Theory: a CTL Model-checker

We consider a Maude rewrite theory  $\mathcal{R}$  for model-checking systems against CTL formulas. The systems to be verified are formally represented as Kripke structures, that is, transition graphs equipped with a labeling function that maps each node in the graph to a set of atomic formulas that hold in the corresponding node. Edges in the graph define system transitions.

For instance, Figure 2 shows the Kripke structure of an industrial oven that contains 5 nodes and whose atomic formulas, representing some properties of the oven in a given state, are `open`, `working`, and `hot`. Arrows represent edges, i.e., system transitions.

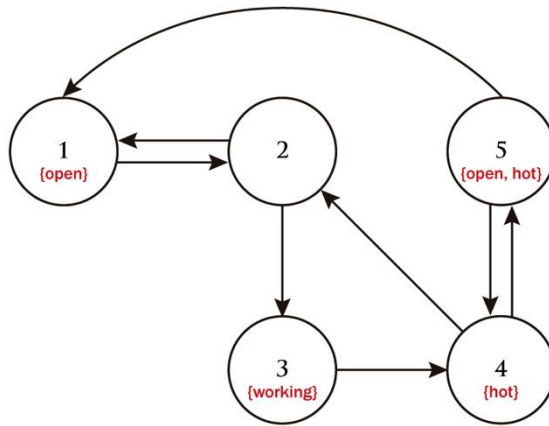


Figure 2. Kripke structure of an industrial oven

In our setting, a Kripke structure is modeled as a term of the form `< transitions ; labels >` where `transitions` is a list of system transitions, and `labels` is a list that represents the labeling function. The Kripke structure of Figure 1 can thus be encoded by means of the following Maude term:

```

< 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ;
    [1 : open], [3 : working], [4 : hot], [5 : open], [5 : hot] >

```

CTL formulas are also represented as Maude terms that may include propositional logic operators as well as CTL modal operators such as `AG` (from now on) and `EX` (in a successor state). For example the following term:

```

AG (Not (open And (open Implies (EX working))))

```

defines a CTL formula that specifies that “the oven cannot work if it was open in the previous state.”

The rewrite theory  $\mathcal{R}$  includes

- i) an equational theory `E` which consists of about 60 equations that specify the CTL semantics as well the satisfaction predicate `( M , S ) |= F` that checks whether a CTL formula `F` holds w.r.t. a Kripke structure `M` and an initial node `S` of `M`;
- ii) a singleton `R` that only includes the following rewrite rule

```

rl [check] : { M | S | F } => if (( M , S ) |= F) then
    ok
  else
    fail
  fi .

```

which takes a state of the form `{ M | S | F }`, where `M` is a Kripke structure, `S` is an initial node, and `F` is the CTL formula to be checked on `M`, and rewrites it to either `ok` or `fail` according to the result of the model-checking predicate `( M , S ) |= F`.

The full Maude specification of our CTL model-checker is available in *iPresto* as a preloaded example that can be fully inspected and partially evaluated, as illustrated in Figure 3.

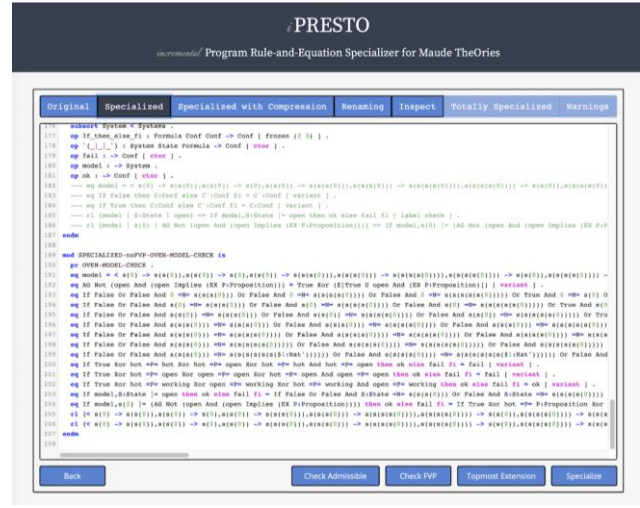


Figure 3. Specialization of the CTL model checker for an industrial oven

### 3. Specializing Maude Equational Theories: the EqNPE algorithm

Traditional uses of partial evaluation have focused on the specialization of entire programs. This includes EqNPE (a partial evaluation algorithm, based on folding variant narrowing), which allows a Maude equational theory  $E$  to be specialized w.r.t. a given set of input function calls  $Q$ . Similarly to the partial evaluation of pure logic programs (also called partial deduction (PD)), EqNPE not only allows inputs to be instantiated with constant values, but it also deals with terms (i.e., function calls) that may contain logic variables, thus providing extra capabilities for program specialization.

The EqNPE algorithm (fully described in [2]) follows the classic control strategy of logic specializers with two separate components:

1. *local control* (managed by an unfolding operator), which avoids infinite evaluations and is responsible for the construction of the residual function definitions (equations) for each call in  $Q$ ;
2. *global control* or control of *polyvariance* (managed by an abstraction operator), which avoids infinite iterations of the partial evaluation algorithm and decides which specialized functions appear in the partially evaluated rewrite theory. Abstraction guarantees that only finitely many expressions are evaluated, thus ensuring global termination.

More specifically, partial evaluation of  $E$  w.r.t.  $Q$  is achieved by iterating two steps:

- i) *Symbolic execution (Unfolding)*. A finite, possibly partial folding variant narrowing tree for each input call in  $Q$  is generated. To handle both finite-variant and non-finite variant equational theories, two unfolding strategies are available. More specifically, for theories that satisfy the Finite Variant Property (FVP), every term  $t$  has a finite folding variant narrowing tree. Hence, for every input call in  $Q$ , the whole narrowing tree can be unfolded. For theories that do not satisfy the FVP, any branch in the folding variant narrowing tree is stopped whenever a term is reached that embeds (modulo  $Ax$ ) any unfolded ancestor occurring in the same branch. In both cases, the algorithm terminates delivering a sound and complete partial evaluation of the input equational theory provided that the correct unfolding strategy is selected.
- ii) *Search for regularities (Abstraction)*. In order to ensure that all calls that may occur at runtime are covered by the specialization, it must be guaranteed that every (sub-)term in any leaf of the narrowing tree is equationally closed w.r.t.  $Q$ . Equational closedness extends the classical PD closedness (i.e., being a subsumption instance) by: 1) considering  $Ax$ -equivalence of terms; and 2) recursing over the term structure (in order to handle nested function calls). Equational closedness ensures that leaves in the narrowing tree are subsumed by some calls in  $Q$ . To properly add the non-closed (sub-)terms to the set of already partially evaluated calls, an abstraction operator  $A$  is applied that yields a new set of terms which may need further evaluation.

Steps (i) and (ii) are iterated as long as new terms are generated until a fixpoint is reached, and the augmented, final set  $Q'$  of closed specialized calls is yielded. The specialization of  $E$  is finally derived from  $Q'$  by computing the partially evaluated equations  $t\sigma = t'$  associated with the derivations in the narrowing tree from the root  $t \in Q'$  to the leaf  $t'$  with computed substitution  $\sigma$ .

### 4. Specializing Maude Rewrite Theories: the extended NPER algorithm

The EqNPE algorithm can be effectively extended to the specialization of Maude rewrite theories by means of our novel specialization methodology which consists of a two-phase algorithm called NPER. Let  $\mathcal{R}$  be a Maude rewrite theory that is made up of a set of rewrite rules  $R$  and an equational theory  $E$ . NPER sequentially executes the following two phases:

- i) *Partial Evaluation.* The key idea for this phase is to apply EqNPE to the underlying equational theory E. This is done by partially evaluating E with respect to the maximal (or outermost) function calls that occur in the rules of R in such a way that E gets rid of any possible over-generality. Indeed, E is transformed into a specialized theory E' that aims at optimizing the performance of the maximal function calls that appear in R.
- ii) *Compression.* On top of that, the narrowing-driven partial evaluation algorithm compacts the functional computations of E' occurring in R, while keeping every system state in the concurrent computations of R as reduced as possible, yet semantically equivalent to the original system. This is achieved by first renaming common expressions in E and R via suitable renaming functions. Next, a refactoring transformation is applied to remove redundant conditions from the rewrite rules in R. Indeed, the partial evaluation process may produce specialized function calls included in rule conditions that can be safely removed without changing the original program semantics. Finally, the computed specialized theory is cleaned up by deleting any function symbols (and their corresponding axioms) that do not occur in the transformed equations and rules.

The NPER algorithm comes with two modalities of execution: monolithic and incremental. The former takes as input the rewrite theory  $\mathcal{R}$  and executes the NPER algorithm until a complete specialization of R is achieved.

The latter allows the user to stop or pause the partial evaluation process so that they can inspect any intermediate specialization results, making it easier to correct, on-the-fly, any faulty optimizations that might result from a violation of the specialization requirements or from fixing inadequate specialization criteria.

It is worth pointing out that the program transformation performed by the NPER algorithm preserves the executability conditions of the input rewrite theory  $\mathcal{R}$ . Furthermore, when  $\mathcal{R}$  is topmost and strongly normalizing,  $\mathcal{R}$  and its partially evaluated version are semantically equivalent (as proven in [2]). Since not all theories are topmost, a topmost extension is implemented by iPresto that automatically transforms a rewrite theory into an equivalent, topmost one. The extension works for several classes of relevant and well-studied theories.

## 5. The NPER Algorithm in Action

The rewrite theory  $\mathcal{R}$  of Section 2.1 allows a Maude user to model-check an arbitrary Kripke structure M w.r.t. an initial state S and a CTL formula F. In this section, we show how NPER can be used to generate an optimized specialization of  $\mathcal{R}$  for some fixed inputs. Roughly speaking, the idea is to automatically produce a specialized model-checker for the given Kripke structure of Section 2.1 that is optimized for the verification of a restricted class of CTL properties.

Specifically, we fixed the following inputs:

```
M: < 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ;
    [1 : open], [3 : working], [4 : hot], [5 : open], [5 : hot] >
S: 1
F: AG (Not (open And (open Implies (EX P:Proposition))))
```

Note that AG (Not (open And (open Implies (EX P:Proposition)))) is a pattern that represents an infinite numbers of CTL formulas, since it contains the variable P of sort Proposition that can be instantiated by any possible well-formed CTL formula.

We then execute the NPER algorithm on  $\mathcal{R}'$ , which is a slight mutation of  $\mathcal{R}$  in which the check rewrite rule has been replaced by the following rule where M, S, and F have been instantiated by using the terms above.

```
rl [check'] : { < 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ;
    [1 : open], [3 : working], [4 : hot], [5 : open], [5 : hot] >
    | 1 | AG (Not (open And (open Implies (EX P:Proposition))))
=>
    if (( < 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ;
        [1 : open], [3 : working], [4 : hot], [5 : open], [5 : hot], 1 )
        |= AG (Not (open And (open Implies (EX P:Proposition)))) then
        ok
    else
        fail
    fi .
```

The execution of NPER on  $\mathcal{R}'$  first partially evaluates  $\mathcal{R}'$  yielding the following equations:

```
eq True Xor hot == hot Xor hot == open Xor hot == hot And hot == open = False [ variant ] .
eq True Xor hot == open Xor open == open Xor hot == open And open == open = False [ variant ] .
eq True Xor hot == working Xor open == working Xor hot == working And open == working = True [ variant ] .
eq (< 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 1, 5 -> 4 ; [1 : open], [3 : working],
    [4 : hot], [5 : hot], [5 : open] >, 1) |= (AG Not (open And (open Implies (EX P:Proposition))))
=
```

```

True Xor hot == P:Proposition Xor open == P:Proposition Xor hot == P:Proposition And open == P:Proposition [ variant ] .
eq {< 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ;
  [1 : open],[3 : working],[4 : hot],[5 : hot],[5 : open] > | 1 | AG Not (open And (open Implies (EX P:Proposition)))}
=
{< 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ;
  [1 : open],[3 : working],[4 : hot],[5 : hot],[5 : open] > | 1 | True Xor (E[True U open And (EX P:Proposition)])} [
variant ] .

```

and the following rewrite rule:

```

rl [check'-pe]: {< 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ;
  [1 : open],[3 : working],[4 : hot],[5 : open],[5 : hot] > | 1 | True Xor
  (E[True U open And (EX P:Proposition)])}
=>
  if True Xor open == P:Proposition Xor hot == P:Proposition Xor open == P:Proposition
  And hot == P:Proposition then
    ok
  else
    fail
  fi .

```

After the partial evaluation phase, NPER proceeds with the compression phase to deliver an even more compact specialized theory:

```

eq < 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ; [1 : open],
  [3 : working],[4 : hot],[5 : hot],[5 : open] >,1 |= (AG Not (open And (open Implies (EX P:Proposition))))
= f0(P:Proposition) [ variant ] .

eq {< 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ; [1 : open],[3 : working],[4 : hot],[5 : hot],[5 : open] >
  | 1 | AG Not (open And (open Implies (EX P:Proposition)))}
= f1(P:Proposition) [ variant ] .

eq f0(hot) = False [ variant ] .
eq f0(open) = False [ variant ] .
eq f0(working) = True [ variant ] .

rl [check'-c] : f1(P:Proposition) => if f0(P:Proposition) then ok else fail fi.

```

Note that two auxiliary functions `f0` and `f1` have been automatically introduced to define the following renaming:

```

True Xor hot == P:Proposition Xor open == P:Proposition Xor hot == P:Proposition And open == P:Proposition
→ f0(P:Proposition)

{< 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ; [1 : open],[3 : working],[4 : hot],[5 : hot],[5 : open] >
  | 1 | True Xor (E[True U open And (EX P:Proposition)])}
→ f1(P:Proposition)

```

It is worth noting that the resulting specialization has been greatly optimized. Indeed, the specialized rule `check'-c`, which is obtained after the compression phase, completely removes the need for evaluating the time-expensive modal operators `AG` and `EX` originally included in the formula `F` to be model-checked, thereby providing a more efficient specialized model-checker that reduces the problem of model-checking the CTL formula `F` to model-checking the simpler formula `P`.

A thorough experimental evaluation of the methodology has been conducted by using the *iPresto* system with the aim of measuring the degree of optimization that our partial evaluation method can achieve on several software systems of different technical nature (e.g., model-checkers, network protocols, client-server applications). Our example set also includes a Maude implementation for the controller of an unmanned space probe orbiting Earth. This model results from several efforts conducted by the European Space Agency to improve mission planning and scheduling of several operations, including the Mars Express mission. Our figures show that the specialized systems achieve a significant improvement in execution time when compared to the original systems, with an average speedup of 53,74, that is, the specialized theory runs 53,74 times faster than the original one on average. Full details of our experiments can be found at the url: <http://safe-tools.dsic.upv.es/ipresto/benchmarks.html>.

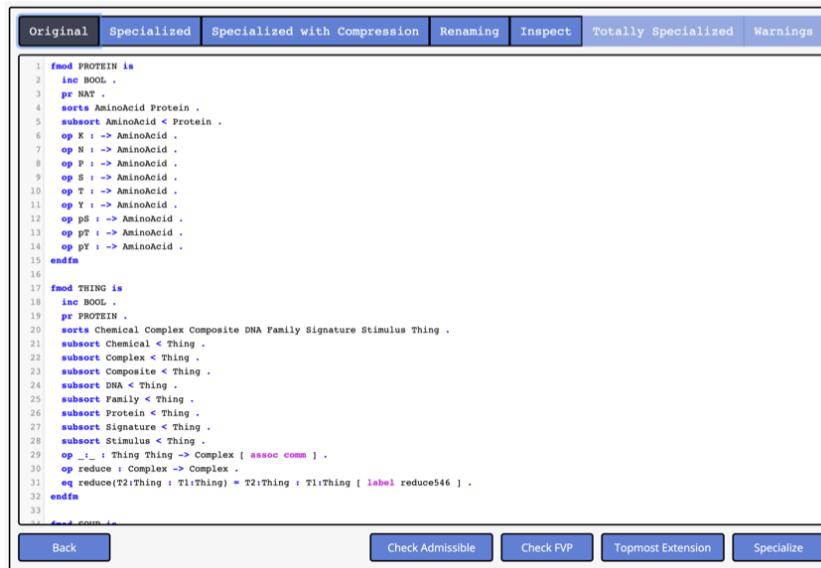
*iPresto* implements both the EqNPE algorithm and the NPER algorithm. The former allows one to partially evaluate equational theories w.r.t. a set of external, user-defined function calls; the latter partially evaluates rewrite theories w.r.t. function calls that occur in the rewrite rules. A quick start guide of *iPresto* is available at <http://safe-tools.dsic.upv.es/ipresto/quickstart.pdf>.

## 6. FURTHER APPLICATIONS

The methodology described in this article can be particularly useful for specializing and debugging a complex, overly general equational theory  $E$  when being plugged into a host rewrite theory  $\mathcal{R}$  as happens, for instance, in protocol analysis, where sophisticated equational theories for cryptography are used [3].

Another interesting application domain for our methodology lays in the optimization of biological systems. Biological systems have been represented in rewriting logic and Maude using different approaches. As shown in [4], a rewriting logic framework for operational semantics of membrane systems can be easily formulated where cells are seen as parallel and distributed processing units that communicate by passing objects through their membranes like chemicals traverse those of biological cells. The membrane system is modeled as a collection of cells, objects playing the role of chemicals, and evolution rules describing their reactions and communication. All of them are assumed to be contained inside a topmost skin. Cells can be populated by a multiset of other nested cells so that multisets of objects and the nested structure of membranes are naturally represented in Maude by terms with associative-commutative constructor operators. Like any other Maude theory, biological systems can be analyzed and model-checked in Maude, but moreover, the models themselves as well as their formal verification can be easily optimized in *iPresto* by straightforwardly using

the methodology described in this paper. An example is provided in <http://safe-tools.dsic.upv.es/iPresto/> where we use *iPresto* to optimize a system that models biological pathways for a mammal cell (see Figure 4).



```
1 fmod PROTEIN is
2   ine BOOL .
3   pr NAT .
4   sorts AminoAcid Protein .
5   subsort AminoAcid < Protein .
6   op X :> AminoAcid .
7   op N :> AminoAcid .
8   op P :> AminoAcid .
9   op S :> AminoAcid .
10  op T :> AminoAcid .
11  op Y :> AminoAcid .
12  op p8 :> AminoAcid .
13  op pT :> AminoAcid .
14  op pY :> AminoAcid .
15 endfn
16
17 fmod THING is
18   ine BOOL .
19   pr PROTEIN .
20   sorts Chemical Complex Composite DNA Family Signature Stimulus Thing .
21   subsort Chemical < Thing .
22   subsort Complex < Thing .
23   subsort Composite < Thing .
24   subsort DNA < Thing .
25   subsort Family < Thing .
26   subsort Protein < Thing .
27   subsort Signature < Thing .
28   subsort Stimulus < Thing .
29   op _ : Thing Thing -> Complex [ assoc comm ] .
30   op reduce : Complex -> Complex .
31   eq reduce(T2:Thing : T1:Thing) = T2:Thing : T1:Thing [ label reduce546 ] .
32 endfn
33
```

Figure 4. Specializing metabolic pathways of a mammal cell with *iPresto*.

**Acknowledgements:** This work was partially supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation programme under GA No 952215, grant PID2021-122830OB-C42 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", and by the Generalitat Valenciana under grant PROMETEO/2019/098.

**Declaration of interests:**

X The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

**\*References:**

[1] M. Alpuente, D. Ballis, and J. Sapiña. Static correction of Maude programs with assertions. *J. Syst. Softw.* 153: 64-85, 2019.

[2] M. Alpuente, D. Ballis, S. Escobar, and J. Sapiña. *Optimization of rewrite theories by equational partial evaluation*. *Journal of Logical and Algebraic Methods in Programming*, 124:100729, 2022.

[3] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. *Debugging Maude programs via runtime assertion checking and trace slicing*, *Journal of Logical and Algebraic Methods in Programming*, 85(5):707--736, 2016.

[4] O. Andrei, G. Ciobanu, and D. Lucanu. *A rewriting logic framework for operational semantics of membrane systems*. *Theoretical Computer Science*, 373(3):163-181, 2007.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify Program and Verify Systems in Rewriting Logic*. *Lecture Notes in Computer Science* 4350, Springer 2007.

[6] S. Escobar, R. Sasse, and J. Meseguer. *Folding Variant Narrowing and Optimal Variant Termination*. *Journal of Logic and Algebraic Programming*, 81(7-8):898-928, 2012.

[7] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[8] M. Marinescu, B. Goldberg. *Partial Evaluation Techniques for Concurrent Programs*. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, Amsterdam, The Netherlands, June 1997, pp. 47-62. ACM, New York.

[9] J. Meseguer. *Conditional Rewriting Logic as a Unified Model of Concurrency*. *Theoretical Computer Science* 96(1):73-155, 1992.

[10] J. Meseguer. *Variant-Based Satisfiability in Initial Algebras*. In *Proc. of the 4th Int'l Workshop for Safety-Critical Systems (FTSCS 2015)*, volume 596 of *Communications in Computer and Information Science*, pages 3-34. Springer-Verlag, Berlin, 2015.



