Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Implementation of LMNtal Model Checkers: a Metaprogramming Approach

Yutaro Tsunekawa[a] Taichi Tomioka[a] Kazunori Ueda[a]

a. Dept. of Computer Science and Engineering, Waseda University

Abstract

LMNtal is a modeling language based on hierarchical graph rewriting, and its implementation SLIM features state space search and an LTL model checker. Several variations and extensions of the SLIM have been developed, and all of them achieve their functionalities by modifying SLIM written in C. If a model checker is implemented in the modeling language itself, it should be easy to develop prototypes of various model checkers without changing the base implementation of the model checker. This approach is called metaprogramming which has been taken extensively in Lisp and Prolog communities.

In this paper, we design a framework for implementing extendable model checkers. First, we define first-class rewrite rules to extend a modeling language. Second, we design an API to operate on the states of programs. These features enable programmers to handle state transition graphs as first-class objects and implement diverse variants of a model checker without changing SLIM. We demonstrate it by implementing an LTL model checker and its variant and a CTL model checker. Furthermore, we show how easy it is to extend these model checkers in our framework by extending the CTL model checker to handle fairness constraints. The overhead of metainterpretation is around an order of magnitude or less. All these results demonstrate the viability of the resulting framework based on meta-interpreters that handle explicit state space in a flexible manner.

Keywords Model Checkers, Meta-Interpreters, Graph Rewriting, State-space Search

1 Introduction

A graph rewriting system consists of graphs and graph rewrite rules. Graph rewriting systems allow us to model various state transition systems with their highly expressive data structures that subsume lists, trees and multisets. For instance, they allow us

Yutaro Tsunekawa, Taichi Tomioka, Kazunori Ueda. *Implementation of LMNtal Model Checkers: a Metaprogramming Approach*. In Journal of Object Technology, vol. 17, no. 1, 2018, pages 1:1–28.
doi:10.5381/jot.2018.17.1.a1

to model concurrent systems whose configurations are made up of dynamically reconfigurable network of processes. Some graph rewriting tools provide model checkers to explore the state space of nondeterministic (and not necessarily confluent) rewriting systems. Examples include Groove [Ren03] and SLIM [GHU11][UAH+09] (standing for Slim Lmntal IMplementation), of which the latter uses LMNtal [Ued09] as a modeling language.

There are different model checkers depending on transition systems and temporal logics used to describe system properties: LTL (Linear-time Temporal Logic) model checking for discrete transition systems, TCTL (Timed Computation Tree Logic) model checking for timed automata, PCTL (Probabilistic CTL) model checking for Markov decision processes, and so on. Indeed, several variations and extensions of the SLIM model checker have been developed, for example by introducing real-time model checking to SLIM. However, all those extensions were done by modifying and extending SLIM which is a complex program with fifty thousand lines of C code. It is not easy to change complicated and large software such as model checkers, which has been an obstacle to the rapid prototyping of new model checkers.

In programming languages such as Lisp and Prolog, a metaprogramming approach has been taken to change the syntax or the semantics of those languages without modifying their implementations [Bra11]. Such a metaprogramming approach is accomplished by implementing and modifying a meta-interpreter of the language. It enables those languages to provide prototypes of other experimental programming languages or to add domain-specific features. For example, the initial implementation of Erlang, a fine-grained concurrent programming language, was developed from a meta-interpreter of Prolog [Arm96].

If a program in a programming language can be expressed as first-class data structure and access the functionalities of the implementation, it is easy for programmers to create variants of the interpreted language by implementing and modifying a meta-interpreter. However, such a meta-interpreter cannot readily evolve into model checkers because it usually does not treat nondeterministic state transitions of programs as first-class.

Our goal is to add the ability to handle state transitions of LMNtal programs to LMNtal so that state space search and state space construction strategies for various model checking algorithms can be explicitly specified in high-level languages. Furthermore, by making state transitions explicit, one can attach additional information to individual states and compare different states for applications involving heuristic search. We first designed and implemented (i) rewrite rules that can be treated as first-class data structures and (ii) API to use SLIM's basic functionalities of operating on model states. To demonstrate this ability of state handling, we implemented prototype model checkers of LMNtal in LMNtal. We show first-class rewrite rules and API that enable programmers to operate on model states of LMNtal programs. We also discuss implementations of model checkers in LMNtal using the API. The results indicate a framework that is sufficient for a meta-interpreter to explore the state space of systems.

This paper extends our results presented at the META'16 workshop [TTU16]. First, we redesigned and reimplemented the API to access and manipulate state space to ensure better performance and computational complexity. Second, we evaluated the performance of state space construction. Third, we experimented on the extension of model checkers based on meta-interpreters. The source code of all the programs described in this paper is available at https://github.com/lmntal/McLMNtal.

```
prove(true).
prove((Goal1, Goal2)) :- prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

Figure 1 – Simple big-step meta-interpreter in Prolog

$$(\textit{Process})\ P ::= 0 \mid p(X_1, \ldots, X_m)\ (m \geq 0) \mid P, P \mid \{P\} \mid T\texttt{:-}T$$
$$(\textit{Template})\ T ::= 0 \mid p(X_1, \ldots, X_m)\ (m \geq 0) \mid T, T \mid \{T\} \mid T\texttt{:-}T \mid \texttt{@}p \mid \$p$$

Figure 2 – Syntax of LMNtal

1.1 Metaprogramming

In this paper, *metaprogramming* means tools and techniques for changing the syntax and the semantics of a language without modifying the language's implementation. Languages for symbolic computation, such as Lisp and Prolog, have promoted this metaprogramming approach conventionally. This approach is used to implement a prototype of a new language or features of the language adapted for a specific problem, as will be discussed in further detail in Section 2.1.

As an example of metaprogramming, Fig. 1 shows a simple meta-interpreter of Prolog based on big-step semantics. It is well-known that, by modifying the meta-interpreter, we can modify the operational semantics of the interpreted language. For example, we can easily modify the above interpreter to construct and return proof trees, and a meta-interpreter based on small-step semantics to implement an interpreter that enables tracing execution.

Prolog meta-interpreters as shown in Fig. 1 enumerate reachable final states by exploring all the execution paths. On the other hand, they handle states explored in different branches of nondeterministic search independently and cannot construct *the set of* states or final results as first-class values. This is exactly why so-called metapredicates such as `findall` are provided to collect information from otherwise independent search paths.

1.2 LMNtal

The modeling language we use is a rule-based graph rewriting language LMNtal [Ued09] (and its extension HyperLMNtal [UO12]; we call both of them LMNtal in this paper), which we will describe briefly in this section.

An LMNtal program is composed of four elements: *atoms*, *links*, *membranes* and *rewrite rules*. Atoms, links, and membranes constitute hierarchical graphs which are a basic data structure in LMNtal. Intuitively, atoms and links correspond to *nodes* and *(undirected) edges* in graph theory. Membranes group atoms and form a hierarchical structure. A rewrite rule is made of a left-hand side (LHS) and a right-hand side (RHS).

Figure 2 shows the syntax of LMNtal. P, called a *process* reflecting the view of LMNtal as a model of concurrency, is either of the following: 0 is an *empty* process; $p(X_1, ..., X_m)$ is an m-ary atom with m totally ordered links; P, P is called a *molecule* which forms a multiset of processes; $\{P\}$ is called a *cell* which groups the process P

with a *membrane* {}; and T`:-`T is a rewrite rule that rewrites a graph matching the LHS template into the RHS template.

An atom name begins with lowercase letters and numbers, or it may be a single-quoted string, while a link name begins with uppercase letters and represents either end of the link. Links occurring just once in a process are called *free links*, and links occurring exactly twice are called *local links*. If a link name is prefixed by "`!`", it represents an end of a *hyperlink* that may have an arbitrary number of endpoints.

T is called a *process template* and may occur in the LHS and the RHS of a rewrite rule. Templates may contain a *rule context* `@`p and a *process context* `$`p in addition to atoms, links, membranes and rewrite rules. A rule context matches all rewrite rules in a membrane, while a process context matches all processes that are not rewrite rules and are not explicitly specified in the membrane it belongs to. For instance, `{a(X),$p[],@q}` stands for a cell containing an atom `a(X)`, zero or more other atoms with no free links (matched by `$p[]`, where the `[]` stands for the absence of free links), and zero or more rules (matched by `@q`). Both contexts and links are sort of "variables," but they play very different roles: Contexts act as wildcards for rules and processes, while links represent graph edges.

LMNtal provides a predefined infix binary atom name `=` called a *connector*. An atom `X=Y` means that the link `X` and the link `Y` are interconnected. This interconnection is handled as structural congruence (i.e., 0-step rewriting); for instance, the process `p(X),q(Y),X=Y` is equivalent to `p(X),q(X)`.

For notational convenience, LMNtal provides the following syntactic scheme: Writing an atom p without its final link instead of the n^{th} link of an atom q means that there is a link between the n^{th} link of q and the final link of p. Using this syntax, a process `f(X,Y),3(X)` can be written as `f(3,Y)` by embedding the unary atom `3` to the first argument of `f`. Also, from the predefined semantics of the connector, the above process is equivalent to `Y=Z,f(X,Z),3(X)`, to which we can apply the above rule twice to obtain `Y=f(3)` by embedding `3` to `f` and `f` to the second argument of `=`. This form is frequently used to represent graphs representing terms and function calls, where the left-hand side stands for a link representing the whole term or the result of a call. It is worth noting that numbers in LMNtal are represented as unary atoms as the above `3(X)` which is also written as `X=3`. Lists are represented using a Prolog-like syntax; for instance, `L=[H|T]` is a ternary *cons* atom with the head `H`, the tail `T`, and the link `L` representing the whole atom. The form `p({...})` stands for a graph `p(X), {'+'(X), ...}`, where a unary atom `'+'(X)` is used to refer to the cell from outside.

There are several extensions of LMNtal: *typed process contexts* and *guarded rewrite rules*. Typed process contexts may occur in process templates and represent constraints on the forms of graphs. Guarded rewrite rules rewrite graphs only when constraints specified in *guards* are satisfied. Constraints that can be specified in guards include (in)equality constraints and type constraints.

For example, a program for the Euclidean algorithm can be written as follows:

```
n(20), n(8).
n($x), n($y) :- $x>$y | n($x-$y), n($y).
```

Here, the rule with the guard "`$x>$y`" is applied if there are subgraphs of the form `n($x)` and `n($y)`, where the typed process contexts `$x` and `$y` represent graphs representing integer atoms and the value of the former is greater than the latter. The `$x-$y` evaluates to an integer atom standing for the difference of the two values. Ap-

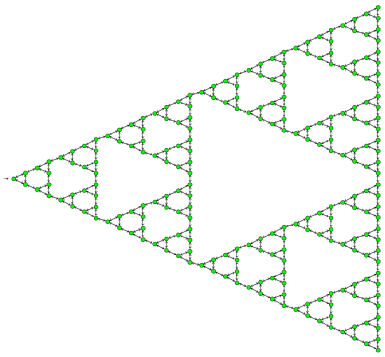

Figure 3 – State transition graph of five-disc *Tower of Hanoi*

plication of the rule is repeated until the arguments of the two unary atoms `n` becomes the same.

Type constraints in guards include `unary($x)` which specifies that `$x` represents a graph representing a unary atom and `ground($x)` which specifies that `$x` represents a connected graph with a single free link.

Let us consider another example.

```
A=p([$x|T]), B=p([$y|S]) :- $x<$y | A=p(T), B=p([$x,$y|S])
```

The rule is applied if there are subgraphs of the form `A=p([$x|T])` and `B=p([$y|S])` and (the integer value represented by) `$x` is smaller than `$y`. In fact, the above rule represent a single-rule program of the Tower of Hanoi. Given the initial graph

```
pegs(p([1,2,3,4,5,99]),p([99]),p([99])),
```

the above rule chooses two of the three available pegs, compares their top discs, and moves the smaller disk to the other peg. As can be seen from both of the examples, the data structure and the rewrite rules of LMNtal allow us to exploit the *symmetry* of the problem: In the Euclidean algorithm example, the two numbers are indistinguishable by their container names; in the Tower of Hanoi example, although the three pegs are totally ordered, the above single rewrite rule covers three possible choices of the two pegs. More examples in which symmetries are exploited will be discussed in Section 4.1.3.

LMNtal has been used to describe diverse computational models with various forms of concurrency and nondeterminism, including the (strong reduction) λ-calculus and the π-calculus, as well as state transition systems from diverse areas, which are included in our software distribution.[1]

1.3 SLIM: the model checker

SLIM is a model checker that takes a model described in LMNtal and a property described as an LTL formula. It verifies whether the model satisfies the property or

[1] http://www.ueda.info.waseda.ac.jp/lmntal/ (portal site),
https://github.com/lmntal/ (source repository)

not. SLIM constructs a state transition graph whose states are hierarchical graphs (henceforth we call them *LMNtal graphs* or simply *graphs*) and whose transitions are applications of rewrite rules.

State transition graphs are constructed in a standard manner using a stack of open states. Successor states are LMNtal graphs obtained from an original state by applying a rewrite rule. In LMNtal, rewrite rules are applied nondeterministically when multiple rewrite rules are applicable to the same graph or a single rewrite rule is applicable to different subgraphs. SLIM computes successor states by applying rewrite rules in all possible ways.

SLIM manages states with hash tables of which keys are hash values of graphs [GHU11]. Since each state of a state transition graph is a graph, (hyper)graph isomorphism checking is in general necessary to construct the state transition graph. However, SLIM's general graph isomorphism algorithm is invoked only when the hash values conflict.

SLIM can also be used without LTL formulas, in which case SLIM explores the whole state space and constructs a state transition graph. For instance, given the Tower of Hanoi program described above, LaViT (LMNtal Visual Tool) invokes SLIM and visualizes its state transition graph as shown in Fig. 3.

2 Frameworks for the metaprogramming approach to model checking

In this section, we discuss a framework for constructing useful meta-interpreters, and describe our design and implementation of the framework in LMNtal.

2.1 Useful meta-interpreters

Useful meta-interpreters should be easily modifiable and implemented. Although meta-interpreters can be implemented in any Turing-complete programming languages, the interpreters are not useful unless the size and the level of abstraction are appropriate. Excessively complicated meta-interpreters cannot be modified easily, and excessively simple meta-interpreters cannot be extended. For example, a meta-interpreter of (a subset of) C could easily exceed 1000 lines of code (LOC), while that of Python can be only one LOC using *exec*. In contrast, Prolog meta-interpreters are about 10 LOC, and Lisp meta-interpreters are about 100 LOC. They are at an appropriate level of abstraction for the flexible modification of their functionalities.

Lisp and Prolog share two remarkable features. First, programs are expressed as fundamental data structures of the language, which is called *homoiconicity*. Second, the basic functionalities of the underlying implementation for program executions are available to programmers. In Prolog, programs are expressed as *terms* and the functionalities are available with `clause` and `call`, while in Lisp, programs are expressed as lists and the functionalities are available with `eval` and `apply`.

We suggest an experimental framework to implement useful meta-interpreters in LMNtal. First, we design and implement first-class rewrite rules to give LMNtal homoiconicity. Second, we implement an API to access SLIM's features that enable LMNtal programs to execute LMNtal programs.

- `state_space.react_nd_set(`*RuleMem*`,` *GraphMem*`,` *RetRule*`,` *Ret*`)`
- `state_space.state_map_init(`*Ret*`)`
- `state_space.state_map_find(`*Map*`, {$`*key*`[]},` *Res*`,` *Ret*`)`
- `state_space.state_map_find(`*Map*`, $`*key*`,` *Res*`,` *Ret*`)`

Figure 4 – Syntax of the state space API

2.2 First-class rewrite rules

An LMNtal program is composed of (hierarchical) graphs and rewrite rules. If rewrite rules are treated as graphs, the whole LMNtal program becomes first-class, i.e., it can be created and manipulated by LMNtal programs. We introduced *first-class rewrite rules* which are composed of graphs and behave like rewrite rules. A first-class rewrite rule is defined as a graph with a ternary atom `':-'`

`':-'({`*Head*`}, {`*Guard*`}, {`*Body*`})`,

where *Head*, *Guard*, and *Body* are processes other than rewrite rules. Such graphs represent rewrite rules *Head* `:-` *Guard* `|` *Body* as first-class citizens. For example, a first-class rule

```
':-'({a(!X,!Y), b(!X), c(!Z)}, {}, {b(!W,!Y), c(!W), a(!Z)})
```

expresses the rewrite rule

```
a(X,Y), b(X), c(Z) :- b(W,Y), c(W), a(Z).
```

First-class rewrite rules do not belong to the syntactic category T `:-` T of ordinary rules, but they act on processes just as the corresponding ordinary rules do. Furthermore, being first-class mean that they can be constructed and manipulated by rewrite rules. This is achieved by dynamically compiling first-class rewrite rules to SLIM's abstract machine code and associating them in the membrane they belong to.

2.3 API to use SLIM's internal functionalities

We designed and implemented an API that enables LMNtal programs to construct state transition graphs of programs. It is shown in Fig. 4 and consists of four atoms whose behavior is implemented using the foreign-language interface of SLIM to access SLIM's internal functionalities. The first one, `state_space.react_nd_set`, is to compute possible destinations of nondeterministic state transition, and the rest are member functions of the `state_map` collection for state management. All users have to do is to import a library module `state_space` to use the API.

Each atom in Fig. 4 has links such as *RetRule* in order to return given rules. Such links are necessary for later use of rules because, in LMNtal, data given as input to API are consumed as resources unless they are explicitly returned through another argument.

2.3.1 state_space.react_nd_set

`state_space.react_nd_set` applies rewrite rules in a membrane *RuleMem* to a process in a membrane *GraphMem*. It calculates all possible rewritings. After rewriting, a list of rewritten processes is connected to *Ret*. If no possible rewritings exist, an empty list is connected.

For example, `state_space.react_nd_set` rewrites a graph

```
state_space.react_nd_set({':-'({a(!X)}, {}, {b(!X)})},
                         {a(1),a(2),a(3)},
                         retrule,
                         ret).
```

and the result is as follows:

```
retrule({':-'({a(!X)}, {}, {b(!X)})}),
ret([{b(1),a(2),a(3)}, {a(1),b(2),a(3)}, {a(1),a(2),b(3)}]).
```

2.3.2 state_map collection

A `state_map` records a set of states, each represented as a membrane containing an individual state, and assigns an ID to each state that is unique in the map. The unique IDs act as handles of possibly complex LMNtal graphs and are used in representing state transition graphs in an efficient manner.

We have two operations on this mapping: `state_map_init` and `state_map_find`. `state_map_init` creates an empty `state_map` and returns it through *Ret*. `state_map_find` returns different graphs depending on the type of the second argument: If the second argument is a cell, this API returns the ID of the state through *Res*. If the second argument is an integer atom expressing the ID of a state, the API returns the cell associated with the ID through *Res*. *Map* must be connected to a `state_map` and the `state_map` is returned through *Ret* .

For example, `state_map_init` and `state_map_find` behave as shown in the following rewriting steps:

```
    state_map_find(state_map_init, {a(1),a(2),a(3)}, res, ret).
⟶ state_map_find(<state_map>, {a(1),a(2),a(3)}, res, ret).
⟶ res(13458), ret(<state_map>).
```

In the first step, `state_map_init` returns a `state_map`. Next, `state_map_find` records the cell `{a(1),a(2),a(3)}` and returns its ID (say `13458`) and a new `state_map`.

As described in the end of Section 1, computing an ID of an LMNtal graph efficiently is highly nontrivial and involves graph hashing and the checking of graph isomorphism. The point of our `state_map` is that it makes these functionalities inside SLIM available to LMNtal programmers without letting them re-implement those. Technically, this is achieved by SLIM's mechanism called *special atoms*. Like ordinary unary atoms, special atoms can be created and passed around by links connected to them, but have no atom names visible to programmers. Instead, they represent abstract data and can be created and passed around in LMNtal programs only by using foreign-language interface rather than ordinary rules.

2.3.3 Design space of the APIs

We have introduced four operations in this section. These operations are mainly used to construct state spaces in LMNtal. In our previous version [TTU16], we introduced and used `membrane.eq` to check the isomorphism of two graphs for state space construction, which was expressive enough but not necessarily efficient. In the present version, we add `state_map` operations to access SLIM's internal functionalities of managing states in order to exploit various optimizations. Unlike `membrane.eq` to compare two explicit graphs, the SLIM model checker checks the isomorphism of an explicit graph and a graph encoded as a byte string and stored in `state_map` with a hash value. The performance of our interpreter was improved without losing the functionality of state space construction and management.

3 LMNtal meta-interpreters

Now we are ready to present an LMNtal meta-interpreter. An LMNtal meta-interpreter constructs a state transition graph from an LMNtal program. We implemented LMNtal meta-interpreters using first-class rewrite rules and the API described in Section 2.

3.1 Algorithm

State transition graphs are composed of nodes representing states and edges representing transitions between states. An LMNtal meta-interpreter constructs a state transition graph from an LMNtal program. Algorithm 1 shows an algorithm for constructing state transition graphs. A state transition graph is represented as a pair of a set of states S and a set of transitions T. At first, S contains only the initial state s_0 and T is an empty set. *Stack* is a stack of unexpanded states and contains only the initial state in the beginning. The algorithm has inner and outer loops and constructs state transition graphs by depth-first search. The first state on *Stack* is expanded in every iteration of the outer loop. The function *expand* takes an unexpanded state and returns the set of all possible transition destinations. The inner loop confirms the freshness of newly expanded states and their transitions. New states are added to S, and new transitions are added to T.

3.2 Implementation

The LMNtal meta-interpreter we have implemented works on an LMNtal program and computes (i.e., rewrites it to) a state transition graph of the program. State transition graphs represent all states and their transition relations generated by nondeterministic execution of programs. Each state of an LMNtal program is an LMNtal graph, while a state transition is the application of a rewrite rule to an LMNtal graph.

The input (i.e., initial form) of the LMNtal meta-interpreter is expressed by a graph of the form:

$$\mathit{Ret} = \texttt{run(}\mathit{RuleSet}\texttt{,}\ \mathit{Init}\texttt{)}.$$

RuleSet is connected to[2] a cell containing first-class rewrite rules. *Init* is a cell containing an initial state. *Ret* is a link to this `run` atom. The output (i.e., the final

[2]Henceforth we omit the phrase "connected to" when it is clear that the two entities are interconnected by links.

Algorithm 1 Constructing state transition graphs

```
S := {s_0}; T := ∅; Stack := ∅
push s_0 Stack
while Stack ≠ ∅ do
  s := pop Stack
  succ := expand(s)
  for all s' ∈ succ do
    if s' ∉ S then
      S := S ∪ {s'}
      T := T ∪ {(s, s')}
      push s' Stack
    else if (s, s') ∉ T then
      T := T ∪ {(s, s')}
    end if
  end for
end while
```

form) of the LMNtal meta-interpreter is a state transition graph expressed as follows:

$$\mathit{Ret} = \texttt{state\_space(}\mathit{Init}\texttt{, }\mathit{Map}\texttt{, }\mathit{States}\texttt{, }\mathit{Transition}\texttt{)}$$

Init is an atom expressing an ID of the initial state. *Map* is a `state_map` which maps all states to IDs. *States* is a hash table of ids mapping to states. *Transitions* is a hash table of pairs of ids expressing transitions. *Ret* is a reference to this `state_space` atom.

States of an LMNtal program are graphs, but they are abstracted to IDs by `state_map`. The abstraction of states enables LMNtal meta-interpreters to run more efficiently with less memory. A transition between states is expressed as a graph $\mathit{Ret} = \texttt{'.'(}\mathit{From}\texttt{,}\mathit{To}\texttt{)}$, where *From* is an ID of a graph expressing the source state and *To* is an ID of a graph expressing the destination state. The hash table of state transitions is used to remove multiple edges between nodes: our meta-interpreters create *minimal* state transition graphs by removing duplication of transitions.

Figure 5 shows the source code of the LMNtal meta-interpreter, where each rule is prefixed by a rule name and a `@@`.

The rules `run` and `exp0` initialize an LMNtal meta-interpreter. These rules create empty sets of states and of state transitions and a stack containing an initial state, and initializes a state map. The rules `exp` and `exp'` correspond to the outer loop of Fig. 1. If the stack is non-empty, `exp` expands an unexpanded state by `react.nd_set`; otherwise `exp'` returns the computed state space, and the meta-interpreter halts. The rules `suc` and `suc'` correspond to the inner loop of Fig. 1. If there is an unchecked transition destination s' of a source state s, `suc` finds an ID of s' by `state_space.state_map_find`. Otherwise, `suc'` breaks the inner loop. The rules `ns` and `ns'` check whether s' is a new state. If s' is new, `ns` adds s' to the set of states and adds the transition from s to s' to the set of transitions. The rules `nt` and `nt'` check whether the transition from s to s' is new. If the transition is new, `nt` adds it to the set of transitions. We used LMNtal's *set* library to manage states and transitions. `set.empty` expresses an empty set. `set.find` looks up an element from a set, and `set.insert` inserts an element to a set.

For example, suppose we apply the LMNtal meta-interpreter to the following

```
run @@  Ret = run(Rs, {$ini[]}):-
        Ret = exp0(Rs, s(ID,{$ini[]}),
          state_space.state_map_find(state_space.state_map_init, {$ini[]}, ID),
          set.empty, set.empty).

exp0@@  Ret = exp0(RS, S0, Map, Ss, Ts), S0 = s($id,{$ini[]}) :- int($id) |
        Ret = exp(RS, [s($id,{$ini[]})], Map, set.insert(Ss, $id), Ts), ini($id).

exp @@  Ret = exp(RS, S0, Map, Ss, Ts), S0 = [s($id,{$src[]})|Stk] :- int($id) |
        Ret = suc(R, Stk, Exp, p($id,{$src[]}), Map, Ss, Ts),
        Exp = state_space.react_nd_set(RS, {$src[]}, R).
exp'@@  Ret = exp({$rs[],@rs}, [], Map, Ss, Ts), ini(I) :-
        Ret = state_space(I, Map, Ss, Ts).

suc @@  Ret = suc(RS, Stk, [{$dst[]}|Suc], Src, Map, Ss, Ts) :-
        M = state_space.state_map_find(Map, {$dst[]}, ID),
        Ret = ns0(RS, Stk, Suc, Src, p(ID,{$dst[]}), M, Ss, Ts).
suc'@@  Ret = suc(RS, Stk, [], p($id,{$src[]}), Map, Ss, Ts) :- int($id) |
        Ret = exp(RS, Stk, Map, Ss, Ts).

ns0 @@  Ret = ns0(RS, Stk, Suc, Src, p($d,D), Map, Ss, Ts) :- int($d) |
        Ret = ns(RS, Stk, Suc, Res, Src, p($d,D), Map, S, Ts),
        S = set.find(Ss, $d, Res).
ns  @@  Ret = ns(RS, Stk, Suc, some, p($s,Src), p($d,Dst), Map, Ss, Ts) :-
        int($s), int($d) |
        Ret = nt(RS, Stk, Suc, Res, p($s,Src), p($d,Dst), Map, Ss, T),
        T = set.find(Ts, '.'($s, $d), Res).
ns' @@  Ret = ns(RS, Stk, Suc, none, p($s,Src), p($d,Dst), Map, Ss, Ts) :-
        int($s), int($d) |
        Ret = suc(RS, [s($d,Dst)|Stk], Suc, p($s,Src), Map, S, T),
        S = set.insert(Ss, $d), T = set.insert(Ts, '.'($s,$d)).

nt  @@  Ret = nt(RS, Stk, Suc, some, Src, p($d, {$dst[]}), Map, Ss, Ts) :-
        int($d) |
        Ret = suc(RS, Stk, Suc, Src, Map, Ss, Ts).
nt' @@  Ret = nt(RS, Stk, Suc, none, p($s,Src), p(D, {$dst[]}), Map, Ss, Ts) :-
        int($s) |
        Ret = suc(RS, Stk, Suc, p($s,Src), Map, Ss, set.insert(Ts, '.'($s,D))).
```

Figure 5 – LMNtal meta-interpreter

graph.

```
ret = run({':-'({a(!X)}, {}, {b(!X)})}, {a(1), a(2), a(3)}).
```

The resulting state space is obtained as an above-mentioned quadruple named `state_space`. To manipulate the state space further, we can easily convert it into concrete LMNtal processes using the following two rules:

```
Ret = state_space(I, M, S, T) :-
    Ret = ss(I, M, set.to_list(S), set.to_list(T)).
Ret = ss(I, M, [$x|S], T) :- int($x) |
    Ret = ss(I, state_space.state_map_find(M, $x, Res), S, T),
    state($x, Res).
```

Applying these two rules to the result of the meta-interpreter, we obtain the following:

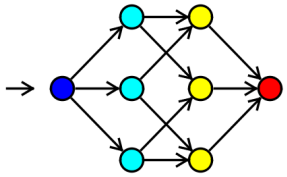

Figure 6 – State transition graph rendered by SLIM and LaViT

Table 1 – The environment of experiments

CPU	Intel Xeon E5-4620 v2
CPU frequency	2.6GHz
Memory	512 GiB

```
ret(ss(5056,<state_map>,[],
      [[6336|6208],[5568|6208],[5440|6336],[5056|5312],[5056|4928],[5312|5696],
       [5440|5568],[4928|5696],[5312|6336],[4928|5568],[5696|6208],[5056|5440]])).
state(5056,{a(1). a(2). a(3). }). state(5696,{a(3). b(1). b(2). }).
state(5312,{a(1). a(3). b(2). }). state(6336,{a(1). b(2). b(3). }).
state(4928,{a(2). a(3). b(1). }). state(5568,{a(2). b(1). b(3). }).
state(6208,{b(1). b(2). b(3). }). state(5440,{a(1). a(2). b(3). }).
```

This is a first-class version of the state transition graph generated and rendered by SLIM and LaViT shown in Fig. 6.

3.3 Performance

We describe the performance of the LMNtal meta-interpreter for state space construction, which is an important measure of our model checkers described in Section 4. We experimented with the computer shown in Table 1. Figure 7 shows the graphs comparing the LMNtal meta-interpreter with SLIM for the following four examples with varying parameters.

- Dining Philosophers Problem,
- Tower of Hanoi,
- Euclidean Algorithm, and
- Sort.

The *Dining Philosophers Problem* is a model with a deadlock state in which all philosophers keep waiting for their left forks holding their right forks. *Sort* is a one-rule program

```
L = [$x,$y|L2] :- $x > $y | L = [$y,$x|L2].
```

which nondeterministically finds and swaps unordered adjacent elements anywhere in a list of integers. Note that the link `L` in the rule may match *any* link at the top

or inside of a list as long as the link is followed by at least two elements. After the exchanges, the list is sorted in ascending order.

We further measured the performance of 20 more problem instances, which are shown in Table 2. The execution time of each instance was measured three times. The models were written as briefly and intuitively as possible rather than keeping efficient execution in mind.

The results show that the LMNtal meta-interpreter runs within an order of magnitude compared to SLIM for those instances except for one instance whose memory footprint is large. To further analyze the overhead of our interpreter, we show in Fig. 8 breakdown of the total execution time into that of the interpreter core and of the API (left) and the breakdown of the execution time of the API (right), using the example of the Tower of Hanoi. The breakdown shown in the two graphs in Fig. 8 indicates that the interpretation overhead and the API overhead are more or less balanced and that the time of state expansion and the time of equivalence checking is more or less balanced; i.e., there is no single bottleneck whose removal would improve the overall performance significantly. Nonetheless, our current implementation of the callback interface is not necessarily lightweight and there is room for further improvement.

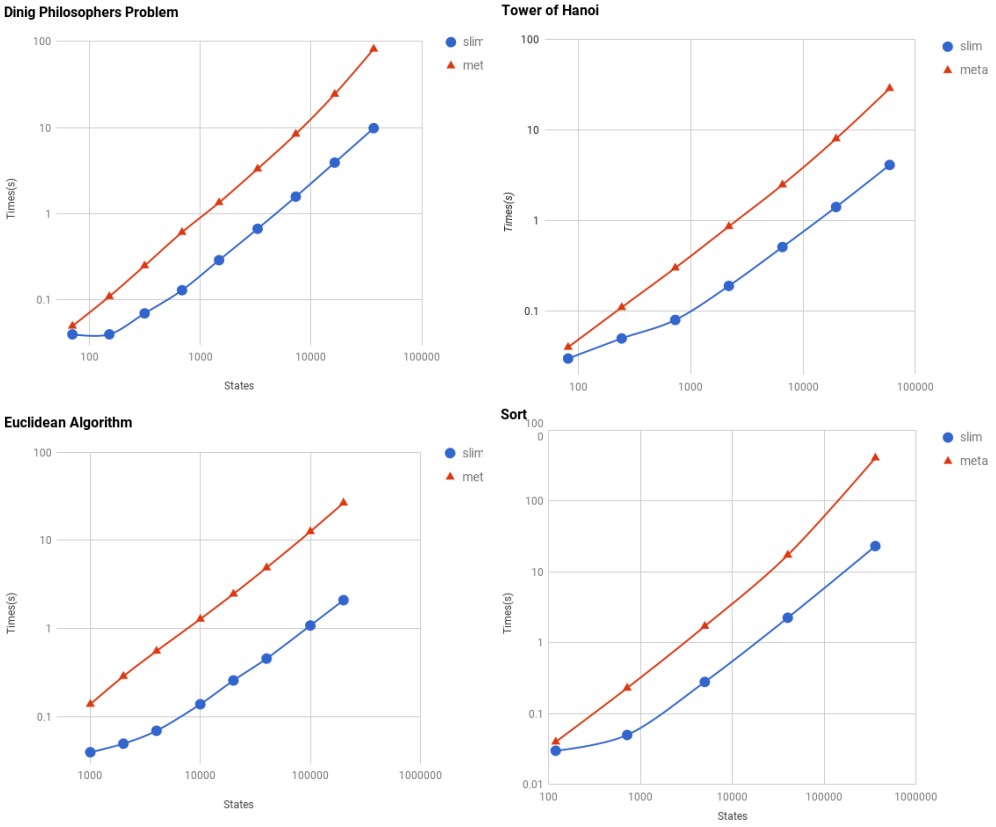

Figure 7 – Comparison of the running times of the LMNtal meta-interpreter with SLIM

Table 2 – Performance of state space construction

Instance Name	States	Time(s) (SLIM)	Time(s) (Meta)	Ratio
Knight_4	1657	0.06	0.28	4.66
Knight_5	508493	15.42	137.31	8.90
Peterson_2	115	0.03	0.06	2.00
Peterson_3	7779	0.70	5.17	7.38
PhiM_5	1370	0.22	1.68	7.68
PhiM_6	5785	1.12	9.21	8.22
PhiM_7	24484	5.96	52.57	8.82
PhiM_8	103691	31.25	308.90	9.88
Qlock_5	657	0.07	0.35	5.00
Qlock_6	3920	0.34	2.33	6.85
Qlock_7	27407	2.55	19.39	7.60
Qlock_8	219210	22.59	203.27	8.99
Queen_8	2057	0.09	0.46	5.11
Queen_9	8394	0.32	1.85	5.78
Queen_10	35539	1.37	7.98	5.82
Queen_11	166926	6.56	46.46	7.08
Rabbit_10	22052	1.20	4.75	3.95
Rabbit_12	92020	5.63	23.77	4.22
Rabbit_14	377234	25.36	159.66	6.29
Rabbit_16	1531664	110.04	1570.23	14.26

3.4 Extensions of the LMNtal meta-interpreter

3.4.1 User-defined Structural Congruence

We can construct an implementation of a semantically extended version of LMNtal with a meta-interpreter. We introduce an interpreter that allows users to define structural congruences. This interpreter runs in simulation (as opposed to nondeterministic search) mode: it chooses and applies one applicable rule in every step and halts if there are no applicable rules. Assume an LMNtal program that reduces a polynomial such as $2x + x^2 + 3x + 1$. If we express a term Ax^B as `term(`A`,`B`)` the polynomial above could be represented as

```
ans = add(add(add(term(2,1), term(1,2)), term(3,1)), term(1,0))
```

Now we can write a rule to calculate the sum:

```
R = add(term($a,$x), term($b,$y)) :- $x=:=$y | R = term($a+$b,$x).
```

However, this rule does not rewrite the polynomial because `term(2,1)` and `term(3,1)` are not siblings in the expression tree. We now define associativity and commutativity of addition as structural congruences:

$$
\begin{array}{rrcl}
(Assoc) & \texttt{R = add(add(A, B), C)} & \equiv & \texttt{R = add(A, add(B, C))} \\
(Comm) & \texttt{R = add(A, B)} & \equiv & \texttt{R = add(B, A)}
\end{array}
$$

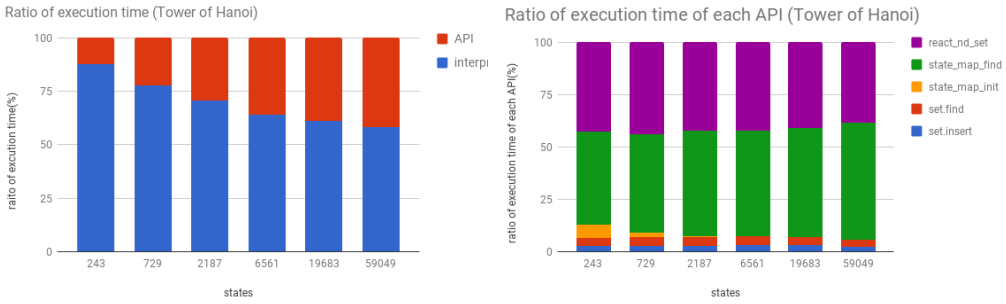

Figure 8 – Breakdown of execution time

These congruences enable the following computation:

$$
\begin{array}{lll}
 & \texttt{ans = add(add(add(term(2,1), term(1,2)), term(3,1)), term(1,0))} & \\
\equiv & \texttt{ans = add(add(add(term(1,2), term(2,1)), term(3,1)), term(1,0))} & (Comm) \\
\equiv & \texttt{ans = add(add(term(1,2), add(term(2,1), term(3,1))), term(1,0))} & (Assoc) \\
\rightarrow & \texttt{ans = add(add(term(1,2), term(5,1)), term(1,0))} &
\end{array}
$$

Our implementation achieves this in the following manner. First, the implementation expands each of (Comm) and (Assoc) into rewrite rules from the LHS to the RHS and from the RHS to the LHS. Second, with these rules from structural congruences, the implementation constructs a state space starting from the current state. Third, the implementation finds a state from the states space and applies an ordinary rule to the state. These steps are repeated while there is an applicable rule.

Although rewriting with associativity and commutativity deserves optimized dedicated implementation, we took this example to illustrate a straightforward implementation applicable to any laws giving finite sets of equivalent graphs. Our interpreter is made of 26 rules.

3.4.2 Introducing A* Search into State Space Exploration

We describe another variant of an LMNtal meta-interpreter. Assume that there is a goal state and we want to obtain the shortest path to the goal from the initial state. Applications of a model checker in this direction are found in the field of automated planning. In order to perform *A* search* to obtain the shortest path, our new interpreter takes a heuristic function, rewrite rules with costs, and an initial state. A heuristic function consists of rewrite rules which calculate the score of a state. The cost of state transitions is obtained from the rewrite rules on state expansion. The interpreter checks an equivalence of states using `state_map` API. Supplemental information such as the score and the accumulated cost of states is associated with the ID of a state. The main part of our implementation consists of 21 rewrite rules.

4 LMNtal model checkers

In this section, We describe an LTL model checker and a CTL model checker that use the API to construct state transition graphs.

4.1 LTL model checker

LTL model checking is a method to verify properties described in linear-time temporal logic (LTL) on state transition systems. LTL model checkers take LTL formulae and model descriptions as inputs. They return true if a model satisfies the LTL formula; otherwise, they return a sequence of states that violates the LTL formula as a counterexample. When LTL model checkers verify whether a model satisfies an LTL formula p, they search exhaustively a sequence of states which satisfies $\neg p$. An LTL formula p semantically corresponds to a set of allowed infinite sequences of non-temporal propositions $p_0, p_1, \ldots$. A sequence of states $\pi = s_0, s_1, \ldots$ satisfies an LTL formula p, denoted $\pi \models p$, iff there is a path $p_0, p_1, \ldots$ such that $s_k \models p_k$ holds for all $k \geq 0$. State transition graphs explored by a model checker are obtained by synthesizing a model and a Büchi automaton derived from an LTL formula. Such a Büchi automaton satisfies the following property: given an LTL formula p, there is an infinite sequence π including accepting states such that $\pi \models p$ holds.

An LTL formula is used to specify temporal properties [CGP99]. Figure 9 shows a graph representation for LTL formulae. LTL formulae have temporal operators $\Box$ (globally), $\Diamond$ (in the future) and U (until). In Fig. 9, these operators are defined as `g`, `f` and `u`. A formula `f(`ϕ`)` means that ϕ holds at some state in the future; `g(`ϕ`)` means that ϕ holds at every state in the execution path; `u(`ϕ_1`,`ϕ_2`)` means that ϕ_1 holds until ϕ_2 holds; and `x(`ϕ`)` means that ϕ holds at the next state. There exists a Büchi automaton corresponding to an LTL formula.

Our implementation of an LTL model checker takes (i) a model description as first-class rewrite rules, (ii) an initial state, and (iii) a Büchi automaton derived from the negation of an LTL formula represented as a hierarchical graph. The model checker generates an atom `no_acceptance_cycle_exists` when the model satisfies the LTL formula. Otherwise, it generates a list of states leading to the violation of p.

4.1.1 Property description

We explain how we express a Büchi automaton in LMNtal before we describe our LTL model checker. Each state of a Büchi automaton is represented by a unique integer, and a Büchi automaton is expressed using a 5-ary atom

$$\mathit{Ret} = \texttt{ba(}S\texttt{,}\mathit{Delta}\texttt{,}S_0\texttt{,}F\texttt{)}$$

where S is a list of integer atoms representing a set of states; *Delta* is a list of atoms *Ret* = `d(`*From*`,`*Prop*`,`*To*`)` representing a set of transition relations; S_0 is an integer atom representing the initial state; F is a list of integer atoms representing a set of accepting states.

An atom *Ret* = `d(`*From*`,`*Prop*`,`*To*`)` expresses a transition relation that a state *From* can be rewritten to a state *To* if a proposition *Prop* is satisfied. It is a transition augmented with additional information about the requirements. *From* and *To* are integer atoms representing states, and *Prop* is a graph that represents a proposition defined in Fig. 10.

Predicate in Fig. 10 is a hyperlink to a graph expressing the property of each state, which is expressed as a ternary atom

$$\mathit{Ret} = \texttt{pred(\{}\mathit{Head}\texttt{\}, \{}\mathit{Guard}\texttt{\})}.$$

A state satisfies a predicate *Ret* = `pred({`*Head*`},{`*Guard*`})` if the state has a subgraph that matches *Head* and satisfies the constraints *Guard*. When there are multiple copies

(*LTL Formula*) $\phi ::=$
`true` | `false` | *Predicate* | `not(`ϕ`)` | `or(`ϕ_1`,`ϕ_2`)`
| `and(`ϕ_1`,`ϕ_2`)` | `imply(`ϕ_1`,`ϕ_2`)` | `g(`ϕ`)` | `f(`ϕ`)` | `x(`ϕ`)` | `u(`ϕ_1`,`ϕ_2`)`

Figure 9 – Graph representation for LTL formulae

(*Proposition*) $P ::=$
`true` | `false` | *Predicate* | `not(`P`)` | `and(`P_1`,`P_2`)` | `or(`P_1`,`P_2`)` | `imply(`P_1`,`P_2`)`

Figure 10 – Graph representation for propositions

of a predicate, the predicate is shared using a hyperlink instead of a normal link *Ret*. The API `state_space.react_nd_set` described in Section 2.3 is used to check if the predicate is satisfied: if a rewrite rule *Head*`:-`*Guard*`|`*Head* is applicable to a state, the state satisfies the predicate *Ret* = `pred({`*Head*`},{`*Guard*`})`.

For example, let a predicate P be "a process `a(X)` exists" and Q be "a process `b(Y)` exists." A proposition $P \wedge (\neg Q \vee P)$ is expressed by the graph shown in Fig. 11, where circles are atoms, rectangles are membranes, solid lines are links and broken lines are hyperlinks. Arrows around circles illustrate the order of links.

4.1.2 Implementation

We have implemented an LTL model checker in LMNtal. The algorithm used is based on *Nested Depth-First Search* [CVWY91].

The input of our LTL model checker is expressed in an atom `mc` as follows:

$$Ret = \texttt{mc(}A\texttt{,}Rs\texttt{,}Init\texttt{)}$$

A is a Büchi automaton given as a `ba` atom described in Section 4.1.1; *Rs* is a cell containing first-class rewrite rules; and *Init* is a cell containing an initial state. If the model does not satisfy the specification expressed in the `ba` atom, our LTL model

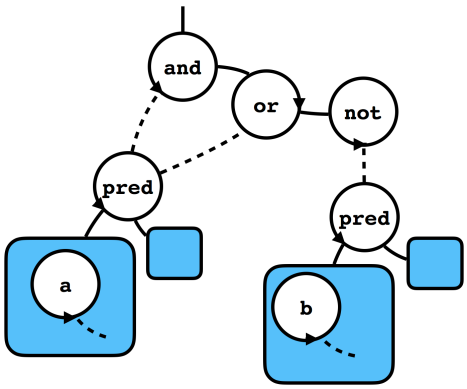

Figure 11 – Hierarchical graph representation of proposition $P \wedge (\neg Q \vee P)$, where P stands for "a process `a(X)` exists" and Q stands for "a process `b(Y)` exists."

```
dfs1, stack1([['.'($s,$q)|T]|T0]), hash1(H0), on_stack(H1) :- int($s), int($q) |
    dfs1_foreach(succ($s, $q)), stack1([['.'($s, $q)|T]|T0]),
    hash1(set.insert(H0, '.'($s, $q))), on_stack(set.insert(H1,'.'($s,$q))), st_([]).

dfs1, stack1([[], ['.'($s,$q)|T]|T0]) :- int($s), int($q) |
    dfs1_acc($q, [], f), stack1([['.'($s,$q)|T]|T0]).

dfs1, stack1([[]]) :-
    no_acceptance_cycle_exists.

dfs1_acc(t), stack1([['.'($s,$q)|T]|T0]):- int($s), int($q) |
    dfs2, stack2([['.'($s,$q)]]), stack1([['.'($s,$q)|T]|T0]).

dfs1_acc(f) :- dfs1_pop.

dfs1_pop, stack1([['.'($s,$q)|T]|T0]), on_stack(H) :- int($s),int($q) |
    dfs1, stack1([T|T0]), on_stack(set.erase(H,'.'($s,$q))).

dfs1_foreach(['.'($s,$q)|T]), hash1(H0) :- int($s), int($q) |
    dfs1_foreach_inner(['.'($s,$q)|T],Res), hash1(set.find(H0, '.'($s,$q), Res)).

dfs1_foreach([]), st_(St_), stack1(St) :-
    dfs1, stack1([St_|St]).

dfs1_foreach_inner(['.'($s,$q)|T], none), st_(St) :- int($s), int($q) |
    dfs1_foreach(T), st_(['.'($s,$q)|St]).

dfs1_foreach_inner(['.'($s,$q)|T], some) :- int($s), int($q) |
    dfs1_foreach(T).
```

Figure 12 – First DFS of the LTL model checker

checker outputs a counterexample:

`counterexample(`*Path*`)`

where *Path* is a list of states leading to an *acceptance cycle* which is a cycle with accepting states. Otherwise, the output is an atom `no_acceptance_cycle_exists`. A state of state transition graphs explored by our LTL model checker is expressed as a graph $Ret = \texttt{'.'}(Sm, Sa)$, where Sm is a unique id of the model's state and Sa is a unique id of the automaton's state.

We show the core part of the LTL model checker implemented in LMNtal in Fig. 12 and Fig. 13. The whole LTL model checker including the rest of the rules consists of 72 rewrite rules. Considering that each state is a combination of a model state and an automaton state and that the search algorithm is more complicated than the standard interpreter shown in Section 3.2, this size seems to be quite reasonable for a model checker.

Nested DFS takes advantage of the post-order traversal for detecting an acceptance cycle and creating a counterexample. We use a somewhat tricky data structure to implement post-order *Nested DFS* in LMNtal. Each of `stack1` and `stack2` holds a stack containing lists of unexpanded successors. If the top of the stack is an empty list, all of the descendants of a node have been visited in post-order. Otherwise, rewrite rules with `succ` expand the head node of the list. Each of `hash1` and `hash2` holds a set of visited nodes for the first DFS and the second DFS, respectively. `on_stack`

```
dfs2, stack2([['.'($s,$q)|T]|T0]), hash2(H0) :- int($s), int($q) |
    dfs2_foreach(succ($s,$q)),stack2([['.'($s,$q)|T]|T0]),
    hash2(set.insert(H0, '.'($s,$q))), st_([]).

dfs2, stack2([[]]),  :- dfs1_pop.

dfs2, stack2([[], ['.'($s,$q)|T]|T0]) :- int($s), int($q) |
    dfs2, stack2([T|T0]).

dfs2_foreach(['.'($s,$q)|T]), on_stack(H) :- int($s), int($q) |
    dfs2_foreach_inner0(['.'($s,$q)|T],Res),on_stack(set.find(H,'.'($s,$q),Res)).

dfs2_foreach([]), st_(St_), stack2(St) :-
    dfs2, stack2([St_|St]).

dfs2_foreach_inner0(S, none) :- dfs2_foreach2(S).

dfs2_foreach_inner0(['.'($s,$q)|$g], some) :- int($s), int($q), ground($g) |
   terminate0([]).

dfs2_foreach2(['.'($s,$q)|T]), hash2(H0) :- int($s), int($q) |
    dfs2_foreach_inner(['.'($s,$q)|T],Res), hash2(set.find(H0, '.'($s,$q), Res)).

dfs2_foreach_inner(['.'($s,$q)|T], none), st_(St) :- int($s), int($q) |
    dfs2_foreach(T), st_(['.'($s,$q)|St]).

dfs2_foreach_inner(['.'($s,$q)|T], some) :- int($s), int($q) |
    dfs2_foreach(T).
```

Figure 13 – Second DFS of the LTL model checker

holds a set of nodes on the searching path of the first DFS. A cycle is found when the second DFS visits a node in `on_stack`.

Our LTL model checker is composed of rewrite rules with an atom `dfs1` and with `dfs2`. The former constructs a state transition graph until an accepting state is visited in post-order, and the latter searches a cycle through a reachable accepting state. The successors are obtained by synthesizing both the successors of model states and of automaton states. The rewrite rules with `succ` (not shown in Fig. 12 and Fig. 13) calculate the successors using our state space API. In these rules, `state_map` and `state_space.react_nd_set` are used to obtain the successors. The API `state_space .react_nd_set` is used to obtain the successors of model states and to check the propositions of Büchi automata. The successors of automaton states are obtained if there is a transition in the automaton and the transition source satisfies the proposition for the transition. The states are managed with `state_map` in the same way as shown in Section 3.2.

Table 3 shows the execution time of SLIM and the LTL model checker implemented in LMNtal. LTL formulae used in experiments include *safety*, *recurrence* and *response* properties. For instances with no acceptance cycles (i.e., those which explore the whole state space), the ratio is quite constant, while other cases see big variance in their performance compared to that of SLIM. This is because nested depth-first search of the two model checkers may not necessarily explore the state space exactly in the same order, finding different counterexamples.

Table 3 – Performance of the LTL model checkers

Instance Name	Results	States (SLIM)	Time(s) (SLIM)	States (Meta)	Time(s) (Meta)	Ratio
Byzantine_10	counterexample	557	0.11	1434	4.35	39.5
Byzantine_11	counterexample	688	0.12	1918	6.95	57.9
Byzantine_12	counterexample	833	0.14	2500	12.08	86.3
Mutex_10	no accepting cycle	6144	0.82	6144	9.81	11.96
Mutex_11	no accepting cycle	13312	1.83	13312	23.38	12.77
Mutex_12	no accepting cycle	28672	4.71	28672	56.67	12.03
PhiM_5	counterexample	332	0.07	65	0.03	0.43
PhiM_6	counterexample	665	0.10	93	0.04	0.40
PhiM_7	counterexample	2073	0.20	126	0.05	0.25
Rabbit_8	counterexample	1612	0.15	1457	1.71	11.4
Rabbit_9	counterexample	3268	0.25	2857	3.63	14.54
Rabbit_10	counterexample	6839	0.50	5831	8.77	17.54

```
// system rules
':-'({p_thinking(!Lx0, !Rx0), fork_free(!Rx1, !Lx0)},
     {},
     {p_one_fork(!Lx0, !Rx0), fork_used(!Rx1, !Lx0)}),
':-'({p_one_fork(!Lxx0, !Rxx0), fork_used(!Rxx1, !Lxx0), fork_free(!Rxx0, !Lxx1)},
     {},
     {p_eating(!Lxx0, !Rxx0), fork_used(!Rxx1, !Lxx0), fork_used(!Rxx0, !Lxx1)}),
':-'({p_eating(!Lxxx0, !Rxxx0), fork_used(!Rxxx0, !Lxxx1), fork_used(!Rxxx1, !Lxxx0)},
     {},
     {p_thinking(!Lxxx0, !Rxxx0), fork_free(!Rxxx0, !Lxxx1), fork_free(!Rxxx1, !Lxxx0)}),
// init state
p_thinking(L0, R0), fork_free(R0, L1).
p_thinking(L1, R1), fork_free(R1, L2).
p_thinking(L2, R2), fork_free(R2, L0).
```

Figure 14 – Model of dining philosophers problem

4.1.3 Example: Dining Philosophers Problem

The Dining Philosophers problem is about the synchronization of processes. Philosophers spend their lives thinking and eating, sitting around a table. There is a fork between each pair of adjacent philosophers. A philosopher holds two forks, first on the left, then on the right. A philosopher with two forks can start eating. After eating, the philosopher puts the forks back on the table. We verify whether the situation "one of the philosophers eats" happens infinitely often. Figure 14 shows a model description as first-class rewrite rules and the initial state as a graph. Atoms in this graph correspond to philosophers and forks. All these atoms and links form a circular graph. In this way, LMNtal nicely manages the symmetry of the model; that is, LMNtal identifies symmetric variants of graphs as isomorphic and reduces the size of the state transition graph of the symmetric model. In the case of five Dining Philosophers, the resulting state space consists of 18 states if the philosophers are indistinguishable from each other, but once each philosopher is given a name, the number of states becomes 82.

```
counterexample([
  '.'({p_thinking(fork_free(p_thinking(fork_free(p_thinking(fork_free(L8)))),L8)}, 0),
  '.'({p_thinking(fork_free(p_one_fork(fork_used(p_thinking(fork_free(L9)))),L9)}, 1),
  '.'({p_thinking(fork_free(p_one_fork(fork_used(p_one_fork(fork_used(L10)))),L10)},1),
  '.'({p_one_fork(fork_used(p_one_fork(fork_used(p_one_fork(fork_used(L11)))),L11)},1),
  '.'({p_one_fork(fork_used(p_one_fork(fork_used(p_one_fork(fork_used(L12)))),L12)},1)
            ])
```

Figure 15 – Counterexample of the dining philosophers problem

Let a proposition p represent the existence of a process `p_eating(L,R)`. An LTL formula $\Box\Diamond p$ expresses the property that the situation "one of the philosophers eats" happens infinitely often in this model. The negation of $\Box\Diamond p$ translates to the following graph representation of a Büchi automaton

```
ba([0, 1],
   [d(0,true,0), d(0,not(!P),1), d(1,not(!P),1)],
   0,
   [1]),
pred({p_eating(!X,!Y)}, {}, !P).
```

When the LTL model checker runs on the model in Fig. 14 and a graph representing the LTL formula $\Box\Diamond p$, it returns a counterexample that every philosopher holds one fork, i.e., a situation of deadlock shown in Fig. 15.

4.1.4 Extension: Depth-Limited Search

We have adapted the LTL model checker in Fig. 12 and Fig. 13 into a depth-limited version of Nested DFS. The depth of a state is defined as the minimal number of transitions to reach it from the initial state. We add a depth parameter to the elements of the stacks for Nested DFS (`stack1` and `stack2`) and the sets for visited nodes (`hash1` and `hash2`). The depth values are incremented when expanding new states and decremented when shorter paths to existing states are found. The new state is expanded only if its depth is lower than a given depth limit. Otherwise, the model checker merely stops the search of this path. The depth of a state is passed from the first DFS loop to the second because the depth of a state must be consistent through both of the loops. On the basis of the depth-limited search, we can also obtain the iterative deepening depth-first search version of the LTL model checker.

4.2 CTL model checker

CTL model checking is a method to verify properties described in computation tree logic (CTL) on state transition systems. CTL model checkers return true if the initial state of a model is contained in a set of states that satisfy a given CTL formula. Otherwise, they return false.

Figure 16 shows a graph representation for CTL formulae. A modal operator in CTL formulae is a pair of a path quantifier E or A and an LTL-like temporal operator F, G or U. E means that there exists a execution path satisfying a CTL formula, while A means that a CTL formula holds for all execution paths. For example, $EG\phi$ specifies that there is a path of which all states satisfy ϕ.

(*CTL Formula*) ϕ ::=
`true` | `false` | `p(`*Predicate*`)` | `not(`ϕ`)` | `or(`ϕ_1`,`ϕ_2`)` | `and(`ϕ_1`,`ϕ_2`)` | `imply(`ϕ_1`,`ϕ_2`)`
| `ax(`ϕ`)` | `ex(`ϕ`)` | `ag(`ϕ`)` | `eg(`ϕ`)` | `af(`ϕ`)` | `ef(`ϕ`)` | `au(`ϕ_1`,`ϕ_2`)` | `eu(`ϕ_1`,`ϕ_2`)`

Figure 16 – Graph representation for input CTL formulae

CTL model checking is achieved by the calculation of S_ϕ, a set of states that satisfy the input CTL formula ϕ. S_ϕ can be computed inductively upon the structure of ϕ. For example, let p be a proposition and ϕ be a CTL formula $EX\neg p$. We first find $S_p = \{s \in S \mid s \models p\}$ from the set S of the model's whole states. We next calculate the complement of S_p, that is, a set $S_{\neg p}$. Finally, we find $S_{EX\neg p}$ from $S_{\neg p}$ as follows:

$$S_{EX\neg p} = \{s \in S \mid \text{there is a transition from } s \text{ to } s' \in S_{\neg p}\}.$$

4.2.1 Implementation

We implemented a CTL model checker for models described in LMNtal. The algorithm described in [CE82] is the base of our CTL model checking algorithm. Our implementation takes first-class rewrite rules and an initial state as a model description and takes a CTL formula as a specification. The input of the CTL model checker is expressed by an atom

$$\mathit{Ret} = \texttt{mc(ctl(}\mathit{Ctl}\texttt{),}\ \mathit{Rs}\texttt{,}\ \mathit{Init}\texttt{)}$$

where *Rs* is a cell containing first-class rewrite rules; *Init* is a cell containing an initial state; and *Ctl* is a graph expressing a CTL formula defined in Fig. 16.

We show the core part of the CTL model checker implemented in LMNtal in Fig. 17. The CTL model checker was constructed as a natural extension of the meta-interpreter described in Section 3.2. It consists of a total of 105 rewrite rules, of which 11 rules are the LMNtal meta-interpreter described in Fig. 5, and the remaining 94 rules are for computing sets of states satisfying individual CTL operators.

The rule `finish` checks whether an initial state belongs to a set of states that satisfy the input CTL formula. If the initial state is contained in the set, our CTL model checker returns an atom `true` and halts; otherwise, the model checker returns an atom `false` and halts. The rule `pred` calculates a set of states that satisfy the input predicate. Atom names starting with `s_` mean computing sets of states satisfying properties starting with individual CTL operators. The other rewrite rules correspond to logical operators. For example, if the input CTL formula is $\phi_1 \vee \phi_2$, the rule `or` takes S_{ϕ_1} and S_{ϕ_2}, and returns $S_{\phi_1 \vee \phi_2} = S_{\phi_1} \cup S_{\phi_2}$. The calculation of $S_{EG\phi}$ starting from the rule `eg` is a bit complicated. Such states must appear in a strongly connected component (SCC) of a state transition graph, because they appear infinitely often in the path of the finite state transition graph. In other words, $s \models EG\phi$ means that there is an SCC in a state transition graph and there is a finite path from a state s to any state in the SCC. So, in the calculation of $S_{EG\phi}$, the model checker calculates SCCs of the state transition graph and states that are reachable to a state in an SCC.

4.2.2 Example: model of an oven

Figure 18 shows an example model of an oven. Labels in each circle express the properties of the oven in a state. The prefix "$\sim$" of labels means negation. We verify

```
finish@@Ret = mc1(Ini, ctl(sat(S)), SS) :-
    Ret = result(set.find(S, Ini, Res), Res, SS).

true@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_=true :-
    Ret = mc1(Ini, Ctl, state_space(M, set.copy(S, S_), T)), Ret_=sat(S_).

p@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), R = p($x), pred({$h[]}, {$g[]}, $y):-
    hlink($x), hlink($y), $x==$y |
    Ret = mc_(Ini, Ctl),
    R = s_p(set.init, set.to_list(S_), {':-'({$h[]}, {$g[]}, {$h[]})},
        state_space(M, set.copy(S, S_), T)), pred({$h[]}, {$g[]}, $y).

not@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_=not(sat(S_)) :-
    Ret = mc_(Ini, Ctl, state_space(M, set.copy(S, R), T)),
    Ret_=s_not(set.diff(R, R_), set.copy(S_, R_)).

or@@Ret = mc1(Ini, Ctl, SS), Ret_=or(sat(S0), sat(S1)) :-
    Ret = mc1(Ini, Ctl, SS), Ret_=sat(set.union(S0, S1)).

ex@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_ = ex(sat(S_)) :-
    Ret = mc_(Ini, Ctl),
    Ret_ = s_ex(set.init, set.to_list(T_), S_, state_space(M, S, set.copy(T, T_))).

eu@@Ret = mc1(Ini, Ctl, state_space(M, S, T)), Ret_=eu(sat(S0), sat(S1)) :-
    Ret = mc_(Ini, Ctl),
    Ret_=s_eu(set.copy(S1, S1_), S0, set.to_list(S1_), set.to_list(T_),
              state_space(M, S, set.copy(T, T_))).

eg@@Ret=mc1(Ini, Ctl, state_space(M, S, T)), Ret_=eg(sat(S0)) :-
    Ret = mc_(Ini, Ctl, state_space(M, S, set.copy(T, T_))),
    Ret_=s_eg0(S0, set.to_list(T_), [], []).
```

Figure 17 – Core part of the CTL model checker

a CTL formula `ag(ef(p(`*Init*`)))` that means "the initial state is reachable from any state." This model is based on [CGP99], with atomic proposition *Init* appended to all states.

Figure 19 shows a model description as first-class rewrite rules and an initial state. A CTL formula `ag(ef(p(`*Init*`)))` is equivalent to

`not(eu(true, not(eu(true, p(`*Init*`)))))`

. When the CTL model checker runs on the model in Fig. 19 and a CTL formula `not(eu(true, not(eu(true,p(`*Init*`)))))`, the model checker returns true. Figure 18 shows that State `3` satisfying *Init* is reachable from any states. Thus the CTL model checker finished successfully.

Table 4 shows the execution time of the CTL model checker implemented in LMNtal. The performance of Mutex instances that satisfy the safety specification is somewhat better than their LTL counterparts. In addition to *set* library, we use *hashmap* library to manage state transition graphs. It enables us to implement the traversal of state transition graphs efficiently. On the other hand, some of the other instances are not as efficient. This is partly due to the implementation of predicate checking that involves the decoding of states and application of the `react_nd_set` API to the decoded states.

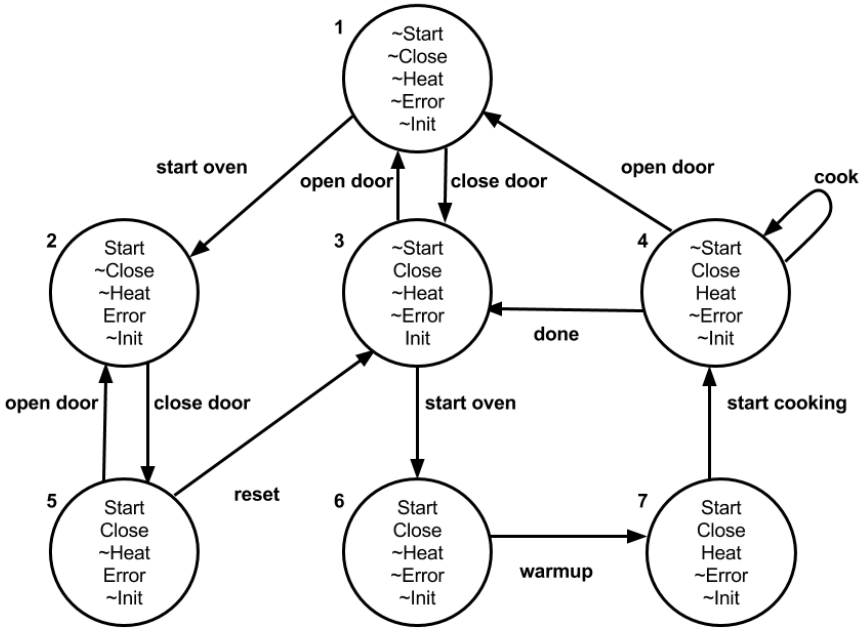

Figure 18 – State transition graph of an oven

4.2.3 Extension: Fairness

We can easily extend our CTL model checker to handle fairness constraints that can be expressed in LTL but not as CTL formulae. Fairness constraints rule out unrealistic paths of models such as "one philosopher keeps eating." The extended CTL model checker only checks paths satisfying fairness constraints. Fairness constraints FC are expressed as a list in LMNtal as follows:

$$Ret = \texttt{[pred\_fair}(\{Head\}, \{Guard\})\texttt{, ...]}$$

The atom `pred_fair` expresses a predicate for states in the same way as the `pred` atom described in Section 4.1.1. Paths satisfying fairness constraints FC have an infinite number of occurrences of states satisfying all predicates in the FC.

We mainly extend the calculation of $S_{EG\phi}$. Let *fair scc* be an SCC containing, for each predicate p in FC, a state satisfying that predicate. In the calculation of $S_{EG\phi}$, we rule out SCCs that are not *fair scc*. We also extend the rules `ex` and `eu`. In the basic CTL model checker, `ex` takes S_ϕ for the calculation of $S_{EX\phi}$. On the other hand, in this extended CTL model checker, states in the set taken by `ex` must be included in *fair scc* of state transition graphs. So we extend `ex` to take a set $S_\phi \cap \{s \mid s \in fair\ scc\}$ as well. We have extended `eu` in a similar manner.

5 Related work

There are a number of model checkers whose modeling languages and implementation languages are the same, including Java Pathfinder (for Java bytecode) and CBMC

```
//system rules
':-'({-start, -close, -heat, -error, -init}, {}, {start, -close, -heat, error, -init}),
':-'({-start, -close, -heat, -error, -init}, {}, {-start, close, -heat, -error, init}),
':-'({start, -close, -heat, error, -init}, {}, {start, close, -heat, error, -init}),
':-'({-start, close, -heat, -error, init}, {}, {-start, -close, -heat, -error, -init}),
':-'({-start, close, -heat, -error, init}, {}, {start, close, -heat, -error, -init}),
':-'({-start, close, heat, -error, -init}, {}, {-start, -close, -heat, -error, -init}),
':-'({-start, close, heat, -error, -init}, {}, {-start, close, heat, -error, -init}),
':-'({-start, close, heat, -error, -init}, {}, {-start, close, -heat, -error, init}),
':-'({start, close, -heat, error, -init}, {}, {start, -close, -heat, error, -init}),
':-'({start, close, -heat, error, -init}, {}, {-start, close, -heat, -error, init}),
':-'({start, close, -heat, -error, -init}, {}, {start, close, heat, -error, -init}),
':-'({start, close, heat, -error, -init}, {}, {-start, close, heat, -error, -init}),
//init state
-start, close, -heat, -error, init
```

Figure 19 – Model of the oven

Table 4 – Performance of the CTL model checker

Instance Name	States	Property	Results	Time(s)
Byzantine_3	420	recurrence	false	0.49
Mutex_10	6144	safety	true	6.60
Mutex_11	13312	safety	true	15.67
Mutex_12	28672	safety	true	37.90
PhiM_4	327	response	false	0.75
PhiM_5	1370	response	false	3.67
PhiM_6	5785	response	false	20.52
Rabbit_4	200	safety	false	1.35
Rabbit_5	482	safety	false	6.69
Rabbit_6	1096	safety	false	32.77

(for C and C++); however, model checkers whose approaches are more or less close to ours are those for declarative languages, which we describe below.

XMC is a model checker for process calculi implemented in XSB, a tabled logic programming language [CDD+98]. It is based on a top-down interpreter of a given model and a specification and exploits the tabling mechanism of XSB to handle state space. It is concise due to the interpreter-based approach and is also efficient due to the optimized indexing techniques for tabling. How to incorporate such indexing techniques in our state space implementation for graphs is highly non-obvious. The experiences of the tabled logic programming approach to model checking is reported in [LM00] which points out the affinity of CTL model checking with the interpreter approach, the finding we also observed. A concise survey and report of the use of declarative languages for verification can be found in [Leu08].

Compared to CTL model checkers implemented in XSB, our CTL model checker code is larger because we explicitly explore state space while they use builtin backtracking mechanism of Prolog. To our knowledge, the tabling mechanism is not provided as a first-class citizen to programmers via an API. We have made our state space explicit because complicated efficient algorithms in underlying implementations, such as graph isomorphism, are worth making accessible from programmers because

they are useful in their own right. Another difference between LMNtal and Prolog is that LMNtal is an inherently concurrent language which enables rather concise modeling of diverse computational models, while the modeling of concurrency in Prolog is via interleaving.

McErlang [FS07] is a model checker for Erlang implemented in Erlang. Functions in Erlang are first-class and thus can be manipulated by Erlang programs as can be done in other functional languages. However, Erlang functions are not expressed as data structures and cannot be modified. On the other hand, the extension of LMNtal in this work allows the change of rewrite rules themselves.

Maude [CDE+07] provides both full-fledged metaprogramming constructs, with which it seems to be possible to describe model checkers (though Maude provides direct support of LTL model checking). Unlike LMNtal, Maude requires various declarations. This contributes to performance, but the source code tends to be significantly longer.

OPEN/CAESAR [Gar98] provides a core platform for the verification and testing for various modeling languages in order to enable the sharing of various functionalities needed to build model checkers. Its open architecture based on callback achieves modularity and orthogonality, but the whole framework is centered around mainstream languages (such as C), and states are assumed to be expressible as fixed-length byte strings. Our approach, in contrast, is centered around a platform supporting highly complex data structures, namely dynamically evolving graphs.

6 Conclusion and future work

We have proposed a metaprogramming approach to developing prototypes of various model checkers. We described first-class rewrite rules designed for metaprogramming and API to access and use the functionality of LMNtal's underlying implementation. We implemented an LMNtal meta-interpreter for state space construction that runs almost within an order of magnitude compared to SLIM. We successfully implemented and extended model checkers using these features for manipulating state transition graphs. This seems to demonstrate that our design of the API extracted the essence of model checkers.

The primary goal of this work is to propose a way of rapid prototyping of model checkers. LMNtal model checkers pursue high-level model description and flexibility for playing with models, making it complementary to other model checkers (such as SPIN) that pursue performance. Considering LMNtal's rather heavyweight data structure for concise and intuitive modeling, the performance seems acceptable at least for proof of concept model checkers. To improve scalability, however, we plan to enhance our minimalistic API and reduce the overhead of the callback mechanism.

Our future work includes implementing model checkers for various transition systems and modal logics other than LTL and CTL model checkers. We are currently working on a TCTL model checker. Furthermore, now equipped with an extendable syntax, we plan to build domain-specific languages for state space search with various strategies and heuristics.

Unlike the SLIM model checker that features scalable multi-core model checking, our state space API does not yet allow concurrent accesses. It might be possible to make the API thread-safe, but to build a parallel version of metaprogramming-based model checkers is a highly nontrivial task because that means exposing and handling parallelism in the model checking algorithm expressed in LMNtal. It is a

highly challenging future work to extend our framework in this direction.

References

[Arm96] J. Armstrong. Erlang—a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.

[Bra11] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2011.

[CDD+98] B. Cui, Y. Dong, X. Du, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *Proc. 10th International Symposium on Principles of Declarative Programming (PLILP/ALP'98)*, volume 1490 of *LNCS*, pages 1–20. Springer-Verlag, 1998. doi:10.1007/BFb0056604.

[CDE+07] M. Clavel, F. Dura, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. *All About Maude—A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer-Verlag, 2007. doi:10.1007/978-3-540-71999-1.

[CE82] E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1982. doi:10.1007/BFb0025774.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[CVWY91] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Proc. CAV 1990*, volume 531 of *LNCS*, pages 233–242. Springer-Verlag, 1991. doi:10.1007/BF00121128.

[FS07] L.-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. *ACM SIGPLAN Notices*, 42(9):125–136, 2007. doi:10.1145/1291151.1291171.

[Gar98] H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In *Proc. TACAS 1998*, volume 1384 of *LNCS*, pages 68–84. Springer-Verlag, 1998. doi:10.1007/BFb0054165.

[GHU11] M. Gocho, T. Hori, and K. Ueda. Evolution of the LMNtal runtime to a parallel model checker. *Computer Software*, 28(4):137–157, 2011. doi:10.11309/jssst.28.4_137.

[Leu08] M. Leuschel. Declarative programming for verification: Lessons and outlook. In *Proc. PPDP'08*, pages 1–7. ACM, 2008. doi:10.1145/1389449.1389450.

[LM00] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *Proc. LOPSTR'99*, volume 1817 of *LNCS*, pages 62–81. Springer-Verlag, 2000. doi:10.1007/10720327_5.

[Ren03] A. Rensink. The groove simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 479–485. Springer-Verlag, 2003. doi:10.1007/978-3-540-25959-6_40.

[TTU16] Y. Tsunekawa, T. Tomioka, and K. Ueda. Implementation of LMNtal model checkers: a metaprogramming approach. In *Proc. Meta-Programming Techniques and Reflection, http://2016.splashcon.org/event/meta2016-implementation-of-lmntal-model-checkers-a-metaprogramming-approach*, 2016.

[UAH+09] K. Ueda, T. Ayano, T. Hori, H. Iwasawa, and S. Ogawa. Hierarchical graph rewriting as a unifying tool for analyzing and understanding nondeterministic systems. In *Proc. ICTAC 2009*, volume 5684 of *LNCS*, pages 349–355. Springer-Verlag, 2009. doi:10.1007/978-3-642-03466-4_24.

[Ued09] K. Ueda. LMNtal as a hierarchical logic programming language. *Theoretical Computer Science*, 410(46):4784–4800, 2009. doi:10.1016/j.tcs.2009.07.043.

[UO12] K. Ueda and S. Ogawa. HyperLMNtal: An extension of a hierarchical graph rewriting model. *Künstliche Intelligenz*, 26(1):27–36, 2012. doi:10.1007/s13218-011-0162-3.

About the authors

Yutaro Tsunekawa is a Research Associate and a PhD student at Waseda University in Japan. He is interested in programming languages, graph rewriting systems, and model checking. He is currently working on optimized model checking for graph rewriting systems. Contact him at tsunekawa@ueda.info.waseda.ac.jp.

Taichi Tomioka is a Master's student in Computer Science at Waseda University, Tokyo. He is studying graph rewriting systems, model checking, and abstract interpretation.

Kazunori Ueda is Professor in Computer Science and Engineering at Waseda University, Tokyo. His research interests include design and implementation of programming languages, concurrency and parallelism, high-performance verification, and hybrid systems. Contact him at ueda@ueda.info.waseda.ac.jp, or visit http://www.ueda.info.waseda.ac.jp/~ueda/.

Acknowledgments The authors are indebted to the present and past members of the LMNtal group on which the present work was based. In particular, Kota Nara gave the authors useful ideas in the beginning of this research, Shota Matsumoto discussed the techniques of verification related to this research, and Takahiro Yanagawa discussed the implementation of the LTL model checker. The authors would like to thank anonymous reviewers for their useful comments and pointers to the literature. This work was partially supported by Grant-in-Aid for Scientific Research ((B) JP26280024, JP18H03223), JSPS, Japan.