The 32nd Annual Conference of the Japanese Society for Artificial Intelligence, 2018

グラフ書き換え言語LMNtalによる容易に拡張可能なモデル検査器の実装

Implementation of LMNtal Model Checkers:a Metaprogramming Approach

恒川雄太郎 *1　上田和紀 *2
Yutaro TSUNEKAWA　Kazunori UEDA

*1早稲田大学大学院基幹理工学研究科情報理工・情報通信専攻
Graduate School of Fundamental Science and Engineering, Waseda University

*2早稲田大学理工学術院情報理工学科
Faculty of Science and Engineering, Waseda University

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, a CTL model checker and a TCTL 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.

1. はじめに

LMNtal[6] とはグラフ書き換えに基づくモデリング言語であり，そのモデル検査器 SLIM[4] は状態空間探索および LTL モデル検査を備えている．これまで SLIM に対しては様々な拡張や変種が開発されてきたが，それらはすべて C 言語で実装された SLIM のソースコードを改変・修正することによって成されてきた．しかし，モデル検査器のような複雑大規模なプログラムを改変・修正することは難しい．そこで，種々のモデル検査器の迅速なプロトタイピングを目的とし，本研究では容易に実装及び拡張可能なモデル検査器のためのフレームワークを提案する．提案するフレームワークを用いることでプログラマは容易に多種多様なモデル検査器を実装したり，修正したりすることが可能となる．

本研究では Lisp や Prolog で伝統的に用いられてきたメタインタプリタを修正・拡張を行うメタプログラミング手法をモデル検査器の実装に応用したフレームワークを設計及び実装し，フレームワークを用いて様々なモデル検査器の実装及び拡張を行った [5].

2. 階層グラフ書き換え言語 LMNtal

LMNtal とは階層グラフとその書き換え規則から成る階層グラフ書き換え言語である．LMNtal の計算はグラフ構造に書き換え規則を適用して，グラフを書き換えてゆくことによって進む．グラフ構造はアトムと呼ばれるノードとリンクと呼ばれるエッジ，階層構造を表す膜によって構成される．`a`，`b` というアトムが `X` というリンクで接続されるようなグラフ構造は `a(X)`，`b(X)` のように記述する．書き換え規則は *Head*`:-`*Guard* | *Body* のように記述し，*Head* にパターンマッチしたグラフ構造が *Guard* 制約を満たしていれば，マッチしたグラフを *Body* に書き換える．

モデル検査器 SLIM は LMNtal をモデル記述言語とした LTL モデル検査器である．SLIM は LMNtal グラフを状態，書き換え規則の適用を状態遷移とした状態遷移グラフを構築し，その探索によってモデルが LTL による仕様記述を充足しているかどうかを検査する．

3. メタプログラミングのためのフレームワーク

本研究で設計したメタプログラミングのためのフレームワークは (1) 第一級の書き換え規則と (2) 処理系内部機能への API から構成される．

3.1 第一級書き換え規則

第一級書き換え規則とは以下のようなグラフによって表現された書き換え規則である．

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

この第一級の書き換え規則は *Head*`:-`*Guard* | *Body* という書き換え規則を表現する．書き換え規則をグラフによって表現することによって LMNtal プログラム全体はグラフ構造として表現され，LMNtal の第一級の値となる．このような性質は同図像性と呼ばれる Prolog や Lisp が持つ特殊な性質であり，この性質によってプログラムから自身のプログラムの操作が容易になる．

連絡先: 恒川雄太郎, 早稲田大学大学院基幹理工学研究科情報理工学専攻, 〒169-8555 新宿区大久保 3-4-1 63 号館 5 階 02 号, 03-5286-3340, tsunekawa(at)ueda.info.waseda.ac.jp

```
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).

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).

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).
Ret=exp({$rs[],@rs}, [], Map, Ss, Ts), ini(I) :-
Ret=state_space(I, Map, Ss, Ts).

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,{$dst[]}), M, Ss, Ts).
Ret=suc(RS, Stk, [], p($id,{$src[]}), Map, Ss, Ts) :- int($id) |
 Ret=exp(RS, Stk, Map, Ss, Ts).

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, $d, Res).
Ret=ns(RS, Stk, Suc, some, p($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, $d), Res).
Ret=ns(RS, Stk, Suc, none, p($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)).

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

図 1: LMNtal メタインタプリタ

3.2 処理系内部機能への API

メタインタプリタの実装にはプログラム実行および状態管理のための処理系内部機能への API が必要となる．本研究では次の 4 つの state_spaceAPI の設計・実装を行った．

react_nd_set(*RuleMem*, *GraphMem*, *RetRule*, *Ret*)

state_map_init(*Ret*)

state_map_find(*Map*, {$*key*[]}, *Res*, *Ret*)

state_map_find(*Map*, $*key*, *Res*, *Ret*)

1 つ目の state_space.react_nd_set は非決定的な状態遷移のすべての遷移先状態を計算するための API であり，残りの API は状態管理のための state_map コレクションの操作関数である．state_map コレクションはそれぞれの状態を管理し，個々の状態にユニークな ID を割り振る．ユニーク ID は複雑な LMNtal グラフを扱いやすくし，状態の遷移関係を効率良く表現するのに適している．ユーザは state_space.state_map_find を用いて状態に ID を付けたり，ID に対応する状態を得たりすることができる．

4. LMNtal メタインタプリタ

本研究でフレームワークを用いて実装を行った LMNtal メタインタプリタを図 1 に示す．実装は 11 本の書き換え規則から成る．このメタインタプリタは入力として第一級書き換え規則と初期グラフを受け取り，その状態遷移グラフを出力する．アルゴリズムはグラフの深さ優先探索であり，状態のユニーク ID を用いて訪問済みノードの管理を行う．状態遷移グラフは state_map コレクション，状態の集合としてのハッシュテーブル，状態遷移の集合としてのハッシュテーブルから成る．状態は state_map によって ID として表現されるため，ハッシュテーブルによって容易に管理することができる．例としてこのメタインタプリタに次のような階層グラフを入力して実行する．

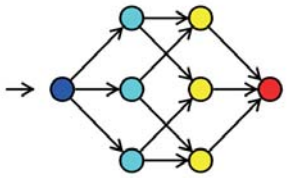

図 2: SLIM と LaViT によって描画される状態遷移グラフ

表 1: 実験環境

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

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

実行の結果 4 価の state_space アトムとして状態空間が得られる．この状態空間は以下の 2 つのルールを用いて状態を簡単に LMNtal プロセスに変換することができる．

```
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).
```

この 2 つのルールを state_space アトムに適用することによって以下を得る

```
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).}).
```

初期状態 a(1), a(2), a(3) からの遷移先はどの a アトムが書き換えられるかによって 3 通りに分岐する．また，最終状態は b(1), b(2), b(3) のみである実行結果出力される階層グラフが表現する状態遷移グラフを図 2 に示す．図 2 の円は状態を表し，矢印は状態遷移を表す．図の最も左に位置する状態は初期状態であり，最も右に位置する状態が最終状態である．

メタインタプリタの実行時間性能に関する実験結果を図 2 に示す．実験環境は表 1 に示す通りである．

図 2 では，食事する哲学者の例題を用いて状態数に対する実行時間を SLIM とメタインタプリタで計測した．実験結果から LMNtal メタインタプリタの性能は SLIM の状態遷移グラフの構築処理と比較して，オーダーに大きな差は見られず，1 桁以下の定数倍の性能低下にとどまっていることを確認した．また，本研究では上述の例題以外の 20 題以上の例題についても同様の実験結果を得た．

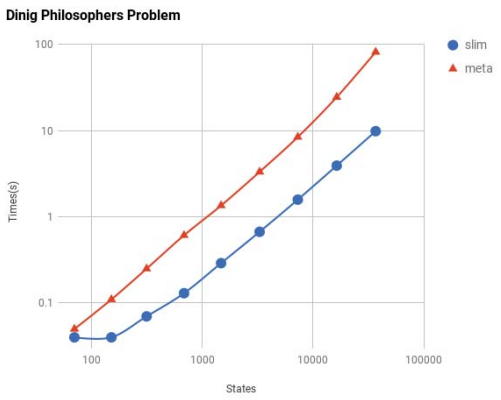

図 3: メタインタプリタと SLIM の実行時間の比較

また，本研究ではメタインタプリタを拡張することで状態遷移グラフの A*探索を行うメタインタプリタの実装を行った．このインタプリタはモデル記述としての LMNtal プログラムとヒューリスティクス関数を入力として，初期状態から目的状態までの最短経路を出力する．ヒューリスティクス関数は状態のスコアを計算するルールの集合であり，インタプリタは状態に対してヒューリスティクス関数を適用することで状態遷移のコストを得る．状態の等価性判定は state map API を用いて実装している．ヒューリスティクス値や初期状態からの最小コストなどの状態の付帯情報は状態の ID をキーとした HashMap で管理することで効率的に付帯情報付きの状態の等価性を判定できた．実装したプログラムの主部分は 20 本ほどのルールから成っている．

5. LMNtal モデル検査器

本研究では前節で紹介したメタインタプリタを基礎にして，LTL モデル検査器，CTL モデル検査器，時間オートマトンに対する TCTL モデル検査器の実装を行い，LTL と CTL についてはそれらの拡張も行った．

LTL モデル検査器は (1) モデル記述としての第一級書き換え規則と (2) 初期状態と (3)LTL 式の否定から得られる LMNtal グラフで表現された Büchi オートマトンを入力として受け取る．モデルを LTL 式が充足した場合はモデル検査器は `no_acceptance_cycle_exists` アトムを返し，そうでない場合は反例として状態のリストを返す．メタインタプリタは入力として与えられる LMNtal プログラムと Büchi オートマトンの同期積オートマトンの状態遷移グラフを構築し，*Nested Depth-First Search*[3] によって仕様の反例を探索する．LTL モデル検査器全体は 72 本の書き換え規則によって実装されている．状態がモデルの状態とオートマトンの状態の組になっている点や探索アルゴリズムが複雑になっている点を考慮すると，実装されたモデル検査器は十分小さいと言える．LTL モデル検査の例として食事する哲学者問題に対して “”哲学者は食事を行う”ことが無限回発生する” という仕様の検証を考える．図 4 で第一級書き換え規則を用いたモデル記述と初期状態のグラフを示す．命題 p をプロセス `p_eating(L,R)` が存在することを表すとする．LTL 式 $\Box\Diamond p$ はモデル中で “”哲学者が食事する”ことが無限回発生する” ことを表現する．$\Box\Diamond p$ の否定を受理する Büchi オートマトンのグラフ表現は以下のようになる．

```
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).
```

図 4 のモデルと LTL 式 $\Box\Diamond p$ のグラフ表現をモデル検査器に入力すると，反例としてすべての哲学者が左手にフォークを持つまでの状態列が返される．反例を図 5 に示す．

CTL モデル検査器はメタインタプリタの自然な発展として実装を行った．モデル検査器は 105 本の書き換え規則で構成され，それらは図 1 で示した 11 本のメタインタプリタと CTL の演算子それぞれについて充足する集合を計算するための 94 本である．モデル検査器への入力を以下に示す．

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

Ctl は CTL 式のグラフ表現，*Rs* は第一級書き換え規則，*Init* は初期状態である．出力は検査の結果を表す真偽値である．モデル検査のアルゴリズムでは，最初にメタインタプリタを用いて状態遷移グラフを構築する．その後入力された CTL 式の構造について帰納的に部分式を満たす状態集合を計算してゆく [2]．

さらに実装した CTL モデル検査器を拡張して公平性制約を扱える CTL モデル検査器の実装を行った．公平性制約を用いることで並行システムにおける非現実的な実行パスを制限することができる．公平性制約は以下のように表現され，CTL モデル検査器に入力される．

$$Ret = \texttt{[pred_fair(}\{Head\},\{Guard\}\texttt{),}\ldots\texttt{]}$$

`pred_fair` で表現されるグラフ構造は状態に対する述語を表現し，制約全体は述語のリストから成っている．公平性制約を満たすパスとはリストのすべての述語を満たすような状態が無限回出現するようなパスである．

本研究では時間オートマトンに対する TCTL モデル検査器の実装も行った．時間オートマトンはクロック変数と呼ばれる時間の経過を計測するための変数を備えており，リアルタイムシステムなどのモデリングが可能である．TCTL とは CTL に対して時間に関する仕様記述が可能になるよう拡張された論理である．実装した TCTL モデル検査器は時間オートマトンによるモデル記述と TCTL 論理式を入力として受け取る．入力は以下のグラフ構造で表現される．

$$Ret = \texttt{mc(tctl(}Tctl\texttt{),}Ta\texttt{)}$$

Tctl には TCTL 論理式のグラフ表現，*Ta* には LMNtal による時間オートマトンのグラフ表現が接続される．LMNtal メタインタプリタは時間オートマトンのグラフ表現を入力として受け取り，領域状態遷移グラフと呼ばれる特殊な状態遷移グラフを深さ優先探索で構築する．モデル検査のアルゴリズムは CTL モデル検査と同様であり，入力された TCTL 式を満たす状態集合を計算してゆく．例として，図 5. に示すような時間制限付きのスイッチの切り替えモデルを表す時間オートマトンを考える．オートマトンの状態は `off` と `on` があり，初期状態は `off` である．`x` はクロック変数であり，`x=1` を満たす時，状態から `off` から `on` へ遷移することが可能である．またこの時 `x` の値を 0 にリセットする．状態 `on` では `x`≤ 1 を満たす限りは状態遷移をしなくても良い．`off` に遷移する際も `x=1` を満たしていなければならない．なおこの例題は [1] か

```
// 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).
```

図 4: 食事する哲学者の LMNtal によるモデリング

```
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)
              ])
```

図 5: 食事する哲学者問題の反例

ら引用したものである．このような時間オートマトンに対して `eu(true, p(!Q), '=<'(1)), pred({on}, {}, !Q)` という “1 秒以内に on になるようなパスが存在する” ことを意味する TCTL 式のモデル検査を行う．その場合時間オートマトンの LMNtal グラフによる表現は以下のようになる．

```
ta({
   ':-'({off}, {reset}, '=:='(x, 1), {on}),
   ':-'({on}, {}, '=:='(x, 1), {off})
   },
   {inv({on}, '=<'(x, 1))},
   {off},
   1))
```

これらの入力に対してモデル検査器は真を返す．確かに初期状態から丁度 1 秒たった時に状態 `off` から状態 `on` へ遷移可能である．

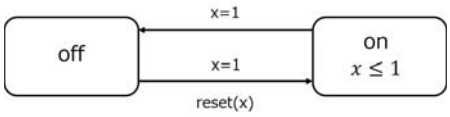

図 6: 時間制限付きスイッチの切り替えモデル

6. まとめと今後の課題

本研究では様々なモデル検査器を容易に実装するためのメタプログラミングを用いたフレームワークを設計および実装した．フレームワークを用いて実装した状態遷移グラフを出力するメタインタプリタを基礎にして，様々なモデル検査器を実装および拡張することに成功した．

今後の課題としては，メタインタプリタを用いて実装したモデル検査器の並列化が挙げられる．本研究の基礎になったモデル検査器 SLIM はスケーラブルなマルチコアモデル検査器であるが，状態空間生成のための API は並行アクセスができない．API のスレッドセーフな実装を行い，複雑な並列モデル検査のアルゴリズムを簡潔に実装することは今後の課題となる．

なお，本研究の一部は科学研究費（基盤 (B)26280024）および早稲田大学特定課題研究助成費 (2017B-134) の助成を得て行った．

参考文献

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

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

[3] 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.

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

[5] Yutaro Tsunekawa, Taichi Tomioka, and Kazunori Ueda. Implementation of lmntal model checkers: a metaprogramming approach. *Journal of Object Technology*, 2018.

[6] K. Ueda. LMNtal as a hierarchical logic programming language. *Theoretical Computer Science*, 410(46):4784–4800, 2009.