実行パスの動的解析によるハイブリッドシステム処理系 HyLaGI の最適化

佐藤 柾史　上田 和紀

時間進行に依る振る舞いを表す連続変化と, 状態の変化を表す離散変化を交互に繰り返す系をハイブリッドシステムという. ハイブリッドシステムのモデリング言語として HydLa がある. HydLa は制約プログラミングのパラダイムで設計された宣言型言語であり, モデルの時間変化である微分方程式や離散変化を制約を用いて記述する. HyLaGI は HydLa 言語の意味論を実装した処理系であり, 記号実行により誤差のない, 不確定値を扱ったシミュレーションを実現する. バックエンドのソルバを用いてプログラムとして記述された制約を繰り返し解くが, それに費やす時間が全実行時間の多くを占めることが珍しくない. 本研究では, 制約解決の手続きを動的に解析し, 過去に解いた結果を再利用することで処理系を最適化する手法を提案し, 処理系への実装と評価を行った.

1 はじめに

ハイブリッドシステム [3] とは, 時間進行に依る振る舞いを表す連続変化と, 離散変化を交互に繰り返す系である. 制御工学や物理学の幅広い分野のモデルとして応用可能である.

ハイブリッドシステムのモデリング言語として HydLa [7] がある. HydLa は制約プログラミングのパラダイムで設計された宣言型言語であり, モデルの時間変化である微分方程式や離散変化を制約として記述する. 一般に用いられる数式と論理式によって構成される言語であるため, 広範の記述能力と有りのままのモデルの記述が可能である.

記述されたプログラムは, 処理系 HyLaGI [5] によって実行される. HyLaGI は記号実行により誤差の発生しないシミュレーションが可能であり, 不確定値を残した実行が可能であることが特徴である. HydLa はハイブリッドシステムをシミュレーションにより検証, 解析を行うことが目的とされており, 実行可能なモデルの広さ, 実行速度, 記述の容易さは処理系に求められる重要な要素である.

ハイブリッドシステムのモデルは同じ挙動を繰り返すものが多く, HyLaGI はそのようなモデルに対して同じ数式処理を繰り返し実行する. 本研究では, 制約解決の手続きの実行パスを動的に解析し, 同じ計算を繰り返さないように処理系の最適化を行った.

2 HydLa 言語

HydLa は, ハイブリッドシステムを数式, 論理式と制約階層 [1] を用いてモデリングする宣言型言語である. HydLa 言語では時刻の関数である関数変数に制約を与えることで, ハイブリッドシステムを記述する. 制約とは, 等式や不等式などの原子制約を論理記号により組み合わせた式である. 論理記号には, 時刻による振る舞いを記述するために時相演算子 □ が含まれる. 時相演算子に修飾された制約は always 制約と呼ばれ, 恒久的に成立する制約として保持される.

また, 離散変化した変数の変化前の値を参照するために, 変数に対する後置演算子 “-” が用いられる.

2.1 HydLa プログラムの例

図 1 は, HydLa で記述したサーモスタットのモデルである. 解軌道は, 図 2 によって与えられる.

1-6 行目は制約モジュールの定義であり, 原子制約

Optimization of A Hybrid System Implementation HyLaGI Using Dynamic Path Analysis
Masashi Sato　Kazunori Ueda, 早稲田大学基幹理工学術院情報理工・情報通信専攻, Dept. of Computer Science and Engineering, Waseda University.

```
INIT <=> p = 65 & [](k1 = 1) & [](k2 = 2)
  & mode = 0.
MODE <=> [](mode' = 0).
OFF <=> [](mode = 0 => p' = -k2).
ON <=> [](mode = 1 => p' = k1*(70-p)).
SWITCHOFF<=>[](p- = 68 & mode- = 1
    => mode = 0).
SWITCHON <=>[](p- = 62 & mode- = 0
    => mode = 1).

INIT, MODE << (SWITCHON, SWITCHOFF),
OFF, ON.
```

図 1: サーモスタットの HydLa プログラム

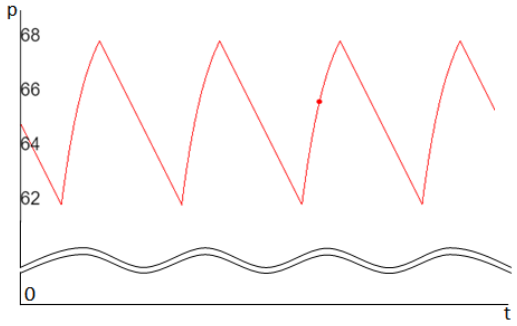

図 2: サーモスタットプログラムの解軌道

の連言をモジュールとして命名している．3-6 行目は条件付き制約と呼ばれ, 前件が満たされた際に後件の制約が発行されることを示す．定義したモジュールは, 8-9 行目で宣言される．記号 “<<”は, モジュール SWITCHON, SWITCHOFF が MODE より優先度の高い制約であることを表す．優先度の高い制約と低い制約が矛盾した場合, 弱い制約を打ち消すことで制約の解決を行う．

変数 p は温度と対応する．温度が 62/68 度に達した際, 条件付き制約 SWITCHON/SWITCHOFF の前件が成立し, それに伴って ON/OFF の前件の成立が切り替わり, 同様に微分方程式が切り替わる．

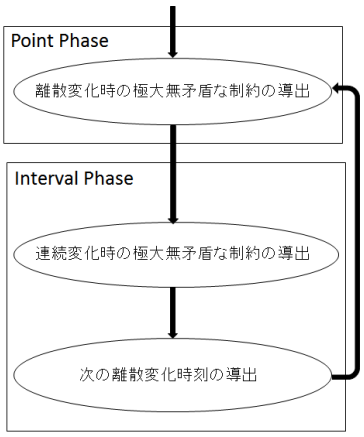

図 3: HyLaGI のシミュレーション 概略図

2.2 HydLa の処理系 HyLaGI

HyLaGI は HydLa の処理系であり, HydLa プログラムを記号計算を用いて誤差のないシミュレーションを行う．不等式としてパラメタを記述することができ, パラメタの値によってシステムが定性的に異なる振る舞いをする場合は全てのケースを実行する．フロントエンドは C++, バックエンドのソルバは Mathematica [8] によって実装されている．図 3 は, HyLaGI におけるシミュレーションの概略図である．

HyLaGI のシミュレーションでは, 離散変化時の制約解決フェーズであるポイントフェーズ (PP) と, 連続変化時のインターバルフェーズ (IP) を繰り返す．PP では, 宣言された各々のモジュール同士が無矛盾かつ極大である制約集合 (極大無矛盾集合) を導出する．実処理系では, 閉包計算によるガード条件の成立判定と, 各々のガードの成立が決定した後の無矛盾性の確認の 2 ステップで構成される．この手続きは, 連立不等式の求解を繰り返すことで実現される．

IP では, PP と同様に極大無矛盾集合を導出する．変数の時間変化の式を導出する必要があるので, 微分方程式の求解を行う．

3 極大無矛盾集合導出手続きの解析

図 3 における極大無矛盾集合導出の手続きは, HyLaGI の記号シミュレーションの要であり, 問題の性質にもよるが, この手続きにコストがかかることが多い．図 2 に示す解軌道を持つシステムは温度が 62 度

に到達 → 温度の増加 → 68 度に到達 → 温度の減少を繰り返す系であり, HyLaGI 処理系の内部でも全く同じ計算手続きが何度も呼び出される. 本研究では, PP および IP における極大無矛盾集合の導出手続きの実行経路を解析することによって, 一度行った計算を将来のフェーズで再利用する手法を提案する. そのために, この手続きの性質と動作を明らかにする.

3.1 導出手続きの性質

まず, 入出力と内部状態の概念的な関係を図 4 に示す.

手続きの入力は, 前のフェーズで求められた制約集合である. 各変数に唯一の値, あるいは不確定値が割り当てられるように前のフェーズで計算されており, 以降変更されることは無い. 入力の制約集合は, PP では内部の恒久的な制約に左極限値として明示的に参照されるほか, 左連続である変数によって暗黙的に参照される. IP においては, 入力の制約集合は微分方程式の初期値として参照される. 最初のポイントフェーズは例外であり, □ 記号の無い制約が代わりとして入力される.

内部状態として, モジュールの集合とそれらの優先度関係が保存されている. 制約モジュールの集合は always 制約を保存しており, これらと入力の制約, 暗黙的に追加される左連続性に関する制約を併せて解くことで, 出力を決定する.

出力は変数に割り当てられる値や式である. これは, 採用された制約モジュールの集合と成立した条件付き制約の組と言い換えることもできる.

本研究においては, この手続きは基本的には内部状態を変更しない純粋な関数である点が重要となる. 唯一の例外は後件に always 制約を持つ条件付き制約であり, この条件付き制約のガードが成立したら恒久的な制約が発行されて内部状態が変更される.

3.2 導出手続きの動作

極大無矛盾集合導出手続きの出力結果は, 以下の 2 つの要素によって決定づけられる.

1. ガード条件の成立有無
2. 採用候補の制約同士の無矛盾性

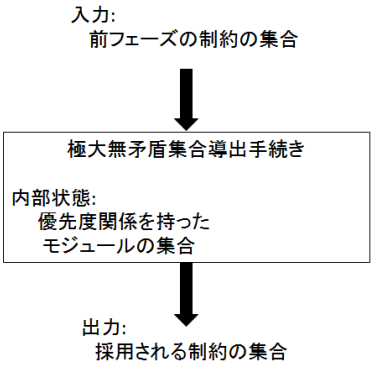

図 4: 極大無矛盾集合導出手続きの入出力関係, および内部状態

```
A <=> [](y=1).
B <=> [](0<= x- <=2 => y=x-).
A << B.
```

図 5: 極大無矛盾集合導出手続きの確認例題

図 5 に示す簡単な例題で, PP における動作を確認する.

最初のフェーズ以外の任意のポイントフェーズにおいて, x の入力値に応じて 3 つのケースに分岐し得る.

1. $0 \leq x < 1 \vee 1 < x \leq 2$ の場合
   入力値は, モジュール B の前件と後件によって明示的に参照される. モジュール B のガード条件が成立し, 後件が発行されることによって A と B が矛盾する.
2. $x = 1$ の場合
   B のガードが成立することは 1. の場合と同じであるが, 発行された制約は A と矛盾しない. 実質的な変数の割当ては 1. と同じであるが, 採用されるモジュールの集合が異なるために異なる出力とみなされる.
3. $x < 0 \vee x > 2$ の場合
   ガードが成立せず, 後件は発行されない.

入力が複数の条件にまたがった場合は, 入力値を区間値で分割して該当するケースを全て実行することになる.

IP においては, ガード条件と無矛盾性が現在時刻か

入力	出力
$0 \leq x < 1 \vee 1 < x \leq 2$	$y = x$
$x = 1$	$y = x \wedge y = 1$
$x < 0 \vee x > 2$	$y = 1$

表 1: プログラム 5 における, 極大無矛盾集合導出手続きの入出力対応関係

ら連続的に成立するかを判定する手続きが行われる.

3.3 導出手続きの最適化手法

節 3.2 の動作をまとめると, 手続きの入出力関係は表 1 のようになる.

入出力対応関係を一度保存しておくことにより, 以降のフェーズでは入力条件との照合と代入で済ますことができ, 計算コストを削減することができる. このような手法は計算再利用 (Computation Reuse) [2] と呼ばれ, プログラミングテクニックやコンパイラの最適化手法として知られている. 入力に局所性があり, かつ頻繁に呼び出される手続きに有効である. 同じ動作を繰り返す系においてはこの手続きの入力には局所性が有り, 計算結果は効果的に再利用可能である.

HydLa プログラムに明示的な条件文は存在しないが, 入力の割当てによってガードの成立の有無, あるいは無矛盾であるか否かで分岐が発生する. 最適化手法では, 手続きの実行フローにおける, ある入力に対して同じパスを通るための分岐の条件 (path constraints [6]) を実行時に保存し, 1 のような入出力の対応関係の表 (再利用表) を動的に作成する.

4 極大無矛盾集合導出手続きにおける, 計算再利用アルゴリズム

本章では, 節 3.3 で述べた極大無矛盾集合導出手続きの最適化手法を組み込んだ HyLaGI の実行アルゴリズムの説明を行う. 実行アルゴリズムの詳細は文献 [5] にて述べられているため, 本節では実装に当たって理解の必要な手続きのみを表示, 説明することとする.

4.1 提案手法を組み込んだ実行アルゴリズム

提案手法では, 以下の 3 つの処理が HyLaGI の実行アルゴリズムに追加される.

1. 毎回のガード条件の判定, 及び無矛盾性の判定の結果に応じて, 条件分岐に関係する制約を保存 (図 7 の 3 行目).
2. 極大無矛盾集合が解決された時点で, 1. で保存した制約の連言と出力の制約, その他シミュレーションに必要な情報を保存.
3. 再び極大無矛盾集合導出の手続きが呼び出され

Require: HydLa プログラム $HydLa$,
最大時刻 $MaxT$

1: $MS := TopologicalSort(HydLa)$
2: $V := GetVariables(HydLa)$
3: $T := 0$
4: $S := true$
5: $P := true$
6: $E := \emptyset$
7: $CP := \emptyset$
8: $CI := \emptyset$
9: **while** $T < MaxT$ **do**
10: $S := Subst(S, T)$
11: $(S, P, _, _) := GetCacheResult(S, P, CP)$
12: **if** $S = false$ **then**
13: $(S, P, E, CP, _, _) := MCS(S, MS, E, P, T, CP, CheckConsistencyPP)$
14: **end if**
15: **if** $S = false$ **then**
16: Break
17: **end if**
18: $(S, P) := AddParameters(S, P, V)$
19: $(S, P, A_-, A_+) := GetCacheResult(S, P, CI)$
20: **if** $S = false$ **then**
21: $(S, P, E, CI, A_-, A_+) := MCS(S, MS, E, P, T, CI, CheckConsistencyIP)$
22: **end if**
23: $S := SolveDifferentitalEquation(S)$
24: **if** $S = false$ **then**
25: Break
26: **end if**
27: $(MinT, P) := GetElement(CompareMinTime((\bigcup_{(g \Rightarrow c) \in A_-} FindMinTime(Subst(g, S), P)) \cup (\bigcup_{(g \Rightarrow c) \in A_+} FindMinTime(Subst(\neg g, S), P)) \cup \{(MaxT - T, true)\}))$
28: $T := MinT + T$
29: **end while**

図 6: 提案手法を組み込んだ HyLaGI の実行アルゴリズム

Require: 前フェーズの制約ストア S_{prev}
制約ストア S
記号定数の条件 P
path constraints PC
Ensure: 充足の有無
記号定数の条件 S
新しい path constraints

1: $V := GetVariables(S)$
2: $P_{tmp} := \exists V(S \wedge S_{prev} \wedge P)$
3: $newPC := removeUnnessesaryCondition(S)$
4: **if** $P_{tmp} = false$ **then**
5: **return** $(false, P, PC \wedge \neg newPC)$
6: **else if** $P_{tmp} = P$ **then**
7: **return** $(true, P, PC \wedge newPC)$
8: **else**
9: **return** $GetElement($
$\{(true, P_{tmp}, PC \wedge newPC),$
$(false, P \wedge \neg P_{tmp}, PC \wedge \neg newPC)\})$
10: **end if**

図 7: 提案手法を組み込んだ checkConsistencyPP のアルゴリズム

た時点で, 2. で保存した制約の連言が成立するかを判定. 成立しているならば, 対応させて保存してある出力の制約に前フェーズの変数の値を代入する (図 6 の 11 行目, 及び 19 行目).

図 6 は, 提案記法を組み込んだ HyLaGI 全体の記号実行の手続きである. HydLa プログラムを受け取り, 図 4 で表される手続きを繰り返す. 関数 MCS は, 極大無矛盾集合を求める手続きである. 1 つ 1 つのガード条件, 及び制約同士の無矛盾性を判定する関数である CheckConsistencyPP, CheckConsistencyIP を複数回呼び出すことで, 極大無矛盾集合を導出する. 提案手法では, CheckConsistencyPP, 及び CheckConsistencyIP において, 小節 3. 3 で述べた path constraints を保存する. CP, CI は, PP と IP における再利用表に対応する.

path constraints を保存する CheckConsistencyPP の手続きを図 7 に示す. この関数では, 今まで保存した path constraints に同じ分岐をするための前フェーズの制約ストアに関する条件を追加する. 3 行目の removeUnnessesaryCondition では, 無矛盾性を判定する制約の中から分岐に寄与しない制約を除去する.

Require: 前フェーズの制約ストア S_{prev}
記号定数の条件 P
入力条件と出力のリスト C
Ensure: 採用される制約
記号定数の条件
成立した条件付き制約
成立しない条件付き制約

1: $V := GetVariables(S)$
2: $P_{tmp} := false$
3: $R := \emptyset$
4: **for all** (PC, S, A_+, A_-) in C **do**
5: $P_{true} := \exists V(PC \wedge S_{prev} \wedge P)$
6: **if** $P_{true} \neq false$ **then**
7: $R := Append(R, (S, P_{true}, A_+, A_-))$
8: $P_{tmp} := P_{tmp} \vee P_{true}$
9: **end if**
10: **end for**
11: **if** $P_{tmp} \neq P$ **then**
12: $R := Append(R, (false, P \wedge \neg P_{tmp}, _, _))$
13: **end if**
14: **return** $GetElement(R)$

図 8: GetCacheResult のアルゴリズム

例えば, $y = 10$ といった些細な制約は, 入力にかかわらず必ず充足可能であるため, 分岐を決定づけず保存する必要がない。この操作は必ずしも行う必要は無いが, path constraints の肥大化を防ぐために行う. また, 限定子除去を行うことで最小の path constraints を抽出できるが, ソルバに負担がかかり, 実行時間が増大する恐れがあるため行わない. 判定結果が充足可能である場合は, path constraints をそのまま追加する (図 7, 7 行目). 充足不可能である場合は, その否定を追加する (図 7, 5 行目). パラメタによって充足の可否が別れる場合は, 非決定的に選択される.

CheckConsistencyIP では, 微分方程式を解くという点と, 無矛盾性判定をその時刻の正の近傍で行うという点で異なる. 本質的には同じなので, 省略する.

関数 MCS が終了した時点で, 保存してきた path constraints と出力の制約ストア, また, 成立した/成立しない条件付き制約のタプルを保存する. これは, 再利用表の列に対応するものである. このときの path constraints は, 極大無矛盾集合導出の関数が同じ出力をするための条件である. また, always 制約が発行

された場合は, 今までの入出力対応関係をリセットする. 今後再び MCS が呼び出される時, まず保存した path constraints と前フェーズの制約ストアの充足性を確認し, 充足可能ならば保存した出力の制約ストアと, IP なら次の最小時間導出の手続きに必要な成立した/成立しない条件付き制約を取得し, MCS の呼び出しを短絡してシミュレーションを続ける (図 8). この時, 記号定数の条件によって複数の path constraints が満たされる, あるいは条件によっては満たす場合がある. この場合は, 本質的に非決定性が伴うため, 記号定数の条件の分割が行われ, 出力が非決定的に選択される.

4.2 HyLaGI への実装

文献[4] で述べられているように, 現在の HyLaGI は前フェーズとの差分情報を保持して, 変化しない変数に関する制約の再計算を行っていない. 本アルゴリズムはその上に実装されているため, 前フェーズとの文脈 (成立していた条件付き制約や変化した変数等) をキーとして, 各々のキーに対応した再利用表が保存される. そのため, オブジェクトとガードが多数存在するモデルでも部分問題に分割されるため, path constraints の無駄な照合が発生しにくくなっている.

また, 最初の PP は左極限値の扱いが特殊であるため, 再利用表の作成は行わない. 続く IP も同様であり, 提案手法は 3 フェーズ目から適用する.

5 実験による評価

表 3 で示す性質の異なる例題[9] を対象に, 実装の評価実験を行った. 実験環境を表 2 に示す.

OS	CentOS 7
CPU	AMD Ryzen Threadripper 1950X 16-Core Processor
Memory	64Gbyte
Backend	Mathematica 11.3

表 2: 実験環境

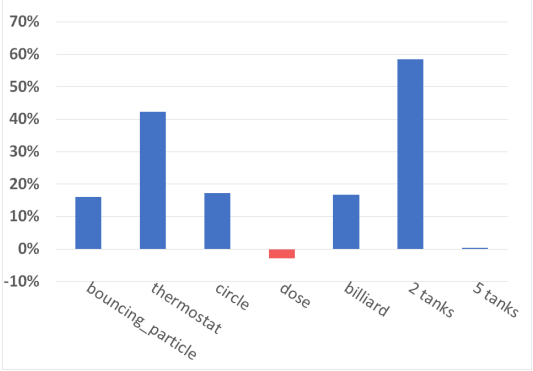

図 9: 実装による実行時間削減率

5.1 実験結果と考察

実装前後の実行時間削減率を図 9 に示す. dose を除いて実行時間が最大で 60%程度削減されていることが分かる.

simple_water_tanks (5 tanks) を除いたプログラムは無矛盾性判定関数の入力の局所性が高く, 提案手法により多くのフェーズの判定を短絡可能である. 表 4 に, 計算を短絡した割合 (短絡したフェーズ数/合計フェーズ数) を示す.

bouncing_particle や thermostat, circle, simple_water_tanks は, 単純な繰り返しの例題であり, 本手法で効果が期待できる問題である. 極大無矛盾集合導出手続きの局所性が高く, 数フェーズで通り有るパスが列挙される. そのため, バックエンドの呼び出し回数が減り, 充足判定の試行錯誤のコストが削減されて効率化が実現できた. simple_water_tanks (2 tanks) は無矛盾性判定におけるバックエンドの呼び出し回数が多いため, 最も性能が向上した.

dose も問題としては単純な繰り返しであるが, 超越関数が発生することで計算式の簡約に多くの時間を費やす問題である. バックエンドでの無矛盾性判定の試行錯誤と, それに伴う数式の簡約処理が減ることで大幅な性能の向上を期待したが, 反対に実行時間が増加した. あるクリティカルな数式の簡約処理が大きなボトルネックであり, 単純な代入処理でも行う必要があるため, 結果としてバックエンドの数式処理の実装に大きく依存する問題であり, 本手法では効果が期待

プログラム名	フェーズ数	説明	特徴
bouncing_particle	100	床を跳ねる質点	繰り返しの基本的な例題
thermostat	100	室温の調整	周期的な繰り返し
circle	30	円の中を跳ねる質点	肥大化する数式
dose	40	定期的な投薬	解析の難しい数式
billiard	100	線上で 20 個の質点の衝突	多数のオブジェクト
simple_water_tanks (2 tanks)	100	2 つのタンクの水量制御	無矛盾性判定のコストが高い
simple_water_tanks (5 tanks)	50	5 つのタンクの水量制御	複雑な path constraints, 非周期的な動作

表 3: 実験対象のプログラム

プログラム名	bouncing_particle	thermostat	circle	dose	billiard	2 tanks	5 tanks
計算を短絡した割合	0.96	0.94	0.87	0.92	0.54	0.94	0.28

表 4: 計算を短絡したフェーズの割合

できない.

billiard はオブジェクトが多く, 無矛盾性判定の入力である変数の数も多い. 現在の処理系ではこのような問題は部分問題に分割されており, 本実装では再利用表も分割される. フェーズ数を増やして実行パスの解析が進むにつれて, 殆どの計算を再利用できることとなる.

simple_water_tanks(5 tanks) は PP の無矛盾性の判定がボトルネックであるが, 非周期的で条件分岐がかなり複雑になる. 結果として, 計算再利用による利潤と path constraints の比較や再利用表作成のコストが釣り合う結果となった.

6 関連研究

具体的な入力値に基づいて関数やプログラムの実行パスを検査する手法として, Concolic Execution [6] がある. 具体値をベースにプラクティカルなパスを列挙する点で類似しているが, こちらはプログラムのテスト手法であり本研究とは目的が異なる. 関数の入力条件と出力を対応付けるのはソフトウェアの検証手法としては一般的であるが, 計算再利用による最適化という観点では, 入力条件の照合のコストや入力の局所性が必要なため, 適用範囲が限られている. HyLaGI が数式ソルバを備え, 入力に局所性のあり計算コストの高い手続きを持つことに注目した研究と言える.

HyLaGI の最適化では, 制約を独立の部分問題に分割する手法や再計算の必要な制約を特定する手法 [4] が提案されている. 本研究における手法は, 再計算の必要の有る制約の計算処理を短絡できる点で新規性がある.

7 まとめと今後の課題

同じ動作を繰り返す問題では極大無矛盾集合導出手続きの入力に局所性がある. 実行時の具体的な入力値に対して実際に通るパスの条件 (path constraints) を保存し, 入出力の対応関係の表を作成することで, 繰り返しの計算を短絡するように実装を施した. 実験結果として, 繰り返しの多い問題, かつ極大無矛盾集合の導出において試行錯誤を幾度も繰り返す問題で性能の向上が確認できた.

今後の課題としては, 動的に問題の性質を分析して計算再利用アルゴリズムを無効化する処理の実装である. 無矛盾性判定関数の入力のローカリティが無い場合や, path constraints が肥大化する場合は実行時間の短縮が見込めず, むしろオーバーヘッドが発生する可能性がある. 再利用表の利用率やオーバーヘッドを分析し, 適合しない問題は無効化してしまうのが望ましい.

本研究では, 最適化という観点で実行パスの分析を利用したが, HyLaGI を用いたモデル検査への応用が発展課題として挙げられる. HyLaGI のモデル検査では入力を抽象化することで行われるが, パラメタによる実行は実行パスの爆発や計算の困難さが伴うので, プログラマによる適切な抽象化が必要となる. 本研究での手法では, 具体的な入力値をベースに定性的に等しい振る舞いをするための入力条件を抽出するため, 無矛盾性判定の手続きにおいては入力を適切に抽象化可能であると考えられる.

本研究の一部は, 科学研究費基盤研究 (B)18H03223 の助成を受けて実施しました.

参 考 文 献

[1] Borning, A., Freeman-Benson, B. and Wilson, M.: Constraint Hierarchies, Lisp and Symbolic Computation, Vol. 5, No. 3, 1992, pp. 223–270.

[2] Ding, Y., Li, Z. : A Compiler Scheme for Rusing Intermediate Computation Results, Proceedings of the International Symposium on Code Generation and Optimization, 2004, pp. 277–288.

[3] Lunze, J : Handbook of Hybrid Systems Control: Theory, Tools, Applications, Cambridge University Press, 2009.

[4] 河野文彦, 小林輝哉, 松本翔太, 上田和紀: 数式処理に基づくハイブリッドシステムシミュレータ Hyrose の大規模モデルシミュレーションに向けた拡張，日本ソフトウェア科学会第 31 回大会, 2014.

[5] 松本翔太, 上田和紀： ハイブリッド制約言語 HydLa の記号実行シミュレータ Hyrose, コンピュータソフトウェア, Vol. 30, No. 4, 2013, pp.18–35.

[6] Sen, K., Marinov, D., and Agha, G.: CUTE: A Concolic Unit Testing Engine for C, *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, New York, NY, USA, ACM, 2005, pp. 263–272.

[7] 上田和紀, 石井大輔, 細部博史： ハイブリッド制約言語 HydLa の宣言的意味論, コンピュータソフトウェア, Vol. 28 No. 1, 2011, pp. 306–311.

[8] Wolfram Research, Inc., Mathematica, `http://www.wolfram.com/mathematica/index.html`.

[9] HyLaGI examples, `https://github.com/HydLa/HyLaGI/tree/master/examples`.