

Alloyによるタスクスケジューリング解析

中堂園 貴幸[†] 結 縁 祥 治[†]

本論文では、ハードリアルタイムシステムにおける周期タスクのスケジューリングを形式仕様言語 Alloy によって記述する。タスク処理モデルは Rate Monotonic スケジューリングと Earliest Deadline First スケジューリングを含む制約の集合として表現する。プリエンブション可能で資源制約を含むタスクスケジューリング可能性判定を、Alloy Analyzer による制約の解消として示す。資源制約を含むスケジューリング問題は一般に NP 困難な問題として知られているが、Alloy Analyzer による制約解消は実用的な時間で解を得ることができる場合も多い。Alloy によってタスク処理モデルを表現することで、'Light-weight' な形式手法により柔軟で正確な記述でスケジューリング可能性を検証し、システムの信頼性を向上させることができる。

Task Scheduling Analysis by Alloy

TAKAYUKI NAKADOZONO [†] and SHOJI YUEN[†]

This paper represents an Alloy description for scheduling periodic tasks in hard real-time systems. We give an Alloy description of task scheduling model as a set of constraints with the rate monotonic policy and earliest deadline first policy. The Alloy Analyzer solves the constraint to check the schedulability with preemption and resources. Although the problem is known to be NP-hard in general, the analyzer may often give answers in a practical amount of time. The merit to represent task scheduling model in Alloy is to give a flexible description precise enough to analyze the schedulability by a 'Light-weight' formal method for improving system reliability.

1. はじめに

形式仕様による振舞いの検証によってソフトウェアの信頼性を向上させる技術が活発に研究されている。リアルタイムシステムにおいては予測可能性が前提であり、リアルタイムシステムで動作するソフトウェアは設計段階で解析・検証を行う必要がある。リアルタイムシステムの複雑化に応じて、複数のタスクは資源を共有しながら実行される。このようなシステムにおけるスケジューリングは、一般には複雑で最適な解を得ることが困難な問題である^{?)}。

この問題に対して本研究では、周期的に発生するタスクを処理するハードリアルタイムシステム、及びそのスケジューリングポリシーをタスク処理モデルとして、形式手法の一つである Alloy^{?)?)} により表現する。Alloy を用いることによって柔軟で正確な定義を与える。特定のシステムに対してスケジューリング可能性を設計段階で厳密に検証することによって、システムの効

率を高めるとともに信頼性を向上させることができる。

Alloy はモデル規範型形式仕様言語と、それによって記述されたソース記述を解析する AlloyAnalyzer で構成される。言語としての Alloy は手続きの機能仕様やデータの静的な構造の表現に適しており、その論理は一階述語論理と関係計算をベースにした一階関係論理と集合論からなり、高い表現力を持つ。Alloy による仕様のソース記述は AlloyAnalyzer により自動的に連言標準形の述語論理式へと変換し、検証エンジンである SAT ソルバによりその充足性を判定する。AlloyAnalyzer は解析結果を充足可能性判定とともにモデル図として表示するため、抽象化・モデル化が正しいかを視覚的に判断できる。一階関係論理による表現力の高さと AlloyAnalyzer によるテストの容易さから、Alloy は Light-weight な形式手法（以下、LFM と書く）として期待されている^{?)}。

タスク処理モデルからエンティティとリレーションを抽出する。「タスクを処理した/された」という関係を、タスクとタスク処理システムを表すエンティティ間のリレーションと定義することで、Alloy による形式仕様でタスク処理モデルを記述する。スケジューリ

[†] 名古屋大学 大学院情報科学研究科
Graduate school of Information Science, Nagoya University

ングポリシーは「タスクを処理した/された」という関係を付与する際を守るべき制約として定義することで表現する。タスクへの優先度の与え方として、静的優先度である Rate Monotonic^{?)} (以下, RM と書く) と動的優先度である Earliest Deadline First^{?)} (以下, EDF と書く) とに従った制約を表現する。さらに, 共有資源によるタスク間の同期による制約を加えてスケジュール可能性を解析する。Alloy による記述手法を提案し, システム毎のタスクの振舞いを単純に記述して解析することにより, 検証のターンアラウンドタイムを短縮することを目標とする。

本論文の構成は以下の通りである。まず, 2 章でモデル規範型形式仕様言語 Alloy について簡単に説明する。続いて, 3 章で資源を共有しない場合の, 4 章では資源を共有する場合のタスク処理モデルを Alloy による形式仕様で記述する。5 章では Alloy Analyzer を用いて 4 章で記述した Alloy モデルを解析, 比較する。最後に第 6 章で結論と今後の課題を述べる。

2. Alloy

Alloy は手続きの機能仕様やデータの静的な構造の表現を目的とする LFM ツール^{?)} である。Alloy では一階述語論理と関係計算をベースとした一階関係論理と集合論により, 仕様を厳密に記述する。述語, 表明を記述し, それらが充足可能であるかを判定することで不具合の発見が可能である。

Alloy 仕様言語は構造化された述語論理式の集まりである。Alloy によるシステムの仕様記述は, 検査対象であるシステムの仕様記述と性質検査からなる。仕様は以下の手順で記述する。

- (1) モデルを表現するための「集合」, 「データ構造」をシグネチャ (sig) で定義する
- (2) 関係 (field) によりシグネチャ間の関係を作る
- (3) 関係に対して制約 (fact) を作る
- (4) 関数 (function) を作る
- (5) 述語, 性質を作る: 動作命令を述語 (pred), 検査する性質を表明 (assert) で記述する
- (6) スコープ (scope) を与えて実例・反例を探索する: 動作命令の実行は run コマンド, 性質の検査には check コマンドを記述する

Alloy Analyzer は Alloy 仕様記述を自動解析・制約解消するツールである。Alloy ソース記述を CNF 標準ファイル形式に変換し, その充足性を検証エンジンである SAT ソルバにより判定する。動作命令の実行・性質の検証は, 検証範囲として与えられたスコープの範囲内における網羅的な反例の探索である。検証

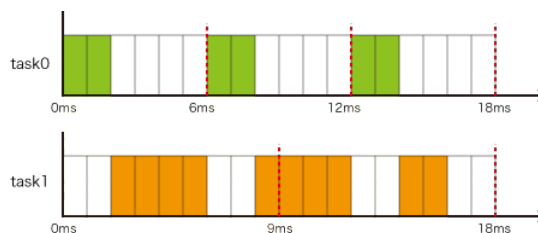


図 1 タスクスケジューリングの例
Fig. 1 example of Task scheduling

は「小スコープ仮説 (Small Scope Hypothesis)」と呼ばれる, 初期状態からの探索の深さを制限し, 最も欠陥の発生しやすい範囲を網羅的に調べることで不具合は発見できるという仮説に基づく。この仮説はほとんどの欠陥は小さい探索範囲であっても反例として発見されるという, 不具合発見の経験から立てられたものである。Alloy Analyzer は与えられたスコープにおいて充足可能性を判定し, その実例・反例を Alloy モデルとして表示する。

3. Alloy によるタスク処理モデルの表現

タスクは実行条件として発生する周期と処理に必要な実行時間をもつ。タスク処理時間スロットはタスクの実行系列を離散的に表現する。タスク処理時間スロットのひとつのスロットを離散時間の最小単位とする。実行時間と周期の長さはスロットの数で表す。図 1 は周期 6 実行時間 2 の task0 と周期 9 実行時間 5 の task1 を, 各タスクの周期をそのデッドラインとし, デッドラインの時刻までに実行時間分の処理を完了するようスケジューリングした例を示す。本論文ではタスクとタスク処理時間スロット間の「タスクを処理した/された」という関係に着目し, タスクからタスク処理時間スロットへの関係付けを示す一本の矢印を「タスクが 1 実行分処理された」と解釈して Alloy によって図 2 のようにタスク処理モデルを表現する。

3.1 タスクとタスク処理時間スロットの表現

タスクとタスク処理時間スロットは以下のように表現する。

```
sig Task{period: set Time}
sig Time{nexttime: set Time}
```

タスクを表すシグネチャ Task は, タスク処理時間スロットを表すシグネチャ Time を関係付ける field period をもつ。タスク処理時間スロットを表す Time は, 次の Time への時間遷移関係を表す field nexttime をもつ。図 3 のように関係 period は Task とその Task が処理された Time を関係付ける。タスク処理時間ス

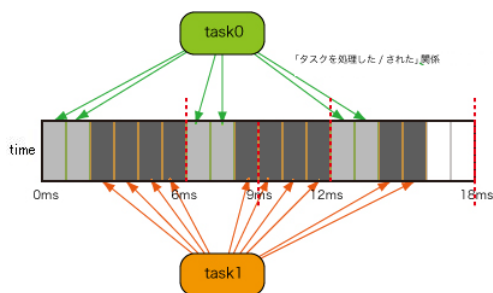


図 2 タスクとタスク処理時間スロットの表現例
Fig. 2 Example of Expression Task and Task scheduling time slot

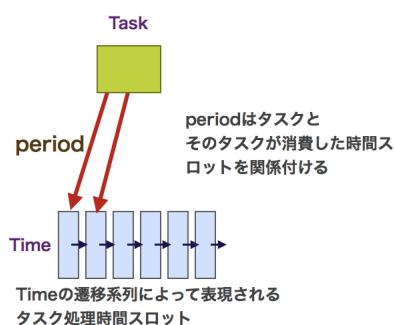


図 3 関係 period
Fig. 3 Relation Period

ロットを表現する Time の遷移系列に、関係 period によって Task をひとつひとつ重複なく順に関係付けることでタスクスケジューリングを表現する。

モデルがその構成・機能として満たすべき仕様、検証したい振舞いを再現するために守るべき制約は fact によって記述する。関係 period には、以下のような制約がある。

```
fact periodTimeRule01{
    period in Task lone -> one Time}
```

関係 period は高々ひとつの Task を唯一の Time に関連付けるものである。

関係 nexttime には、以下のような制約がある。

```
fact nextTimeRule01{
    nexttime in Time lone -> lone Time}
fact nextTimeRule02{
    all k:Time | k.nexttime !=none
    or k.^nexttime != none}
fact nextTimeRule03{
    no k: Time | k in k.^nexttime}
```

関係 nexttime は高々ひとつの Time を高々ひとつの Time に関連付けるものであり、前後に時間遷移関係を

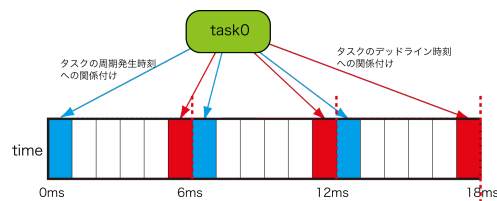


図 4 周期タスクの表現
Fig. 4 Expression of Periodic Task

もたないような Time は存在しない。タスク処理時間スロットは時間遷移であるため非循環であるから、関係演算子[^]で記述する推移閉包を用いて、任意の Time k は関係 nexttime による自身からの推移関係に含まれない、という制約を与える。以上の記述により、タスク処理モデルを表現する。

3.2 周期的に発生するタスクの表現

実行条件である周期に従って発生するタスクを表現するために、図 4 のようにタスク処理時間スロットに対して各タスクの周期発生時刻を関係付ける、関係 prd を定義する。周期発生時刻はタスク処理時間スロットの初期時間から始まり、各タスクの周期数間隔で関係 prd によって関連付けされる。同様にタスクのデッドラインを示す関係 dl を定義する。周期発生時刻の直前の時刻をデッドラインとし、タスク処理時間スロットの最終時刻まで各タスクの周期数間隔で関係 dl を関連付ける。ひとつの周期におけるタスクの処理回数は実行時間分であるから、各タスクからタスク処理時間スロットに対して関係 period によって関連付けされる回数は実行時間に等しい、という制約を与える。以上により周期タスクの振舞いを表現する。

同時刻におけるタスクの重複処理を防ぐために、1 タスク処理時間スロットで 2 つ分以上の処理をしてはならないという制約を加える (制約 1)。ある時刻 k において全タスクの処理が実行されていないにもかかわらず、時刻 k-1 までに処理が終了していないタスクがあってはならない、という制約を加えることで、実行可能であるタスクが順次処理されることを保証する (制約 2)。

以下に 3.1 で表現したタスクとタスク処理時間スロットを基にした、周期タスクを処理するタスク処理モデルの仕様とモデルの振舞いに対する制約を示す。

仕様 1 周期タスクとタスク処理時間スロットが存在する

```
abstract sig Task{
    prd: set Time, dl: set Time}
one sig task_0, task_1 extends Task{}
sig Time{nexttime: set Time}
```

(シグネチャによって宣言することで、モデルにその名前の集合を導入できる。abstract は、そのシグネチャ自身は要素を持たず、その拡張シグネチャだけが要素をもつことを示す。one はそのシグネチャの要素がひとつだけであることを示す)

仕様 2 各周期タスクごとに周期 (Prd) と最悪実行時間 (Wcet) が存在する

```
abstract sig Prd0, Wcet0{}
abstract sig Prd1, Wcet1{}
```

仕様 3 タスク処理時間スロットは非循環である

```
fact nextTimeRule01 {
  nexttime in Time lone -> lone Time}
fact nextTimeRule02 {
  all k: Time | k.nexttime != none
  or k.^nexttime != none }
fact nextTimeRule03 {
  no k: Time | k in k.^nexttime}
```

制約 1 1 タスク処理時間スロットにおいて、2 つ以上の周期タスクが同時に処理されることはない

```
fact TaskSchedulingRule01{
  all k: Time, t: Task |
  #k.(tperiod[t]) <= 1}
fact TaskSchedulingRule02{
  all k: Time, disj t, t': Task |
  let num = k.(tperiod[t]) + k.(
    tperiod[t']) |
  #num <= 1}
```

(関数 tperiod[t] は、タスク t が関係 period によって関連付けられている Time を返す)

制約 2 タスク処理時間スロットにおいてタスク処理が実行されていないとき、それ以前の時間スロットですべてのタスク処理が完了している

```
fact TaskSchedulingRule03{
  all disj t, t': Task,
  k: Time - Time0 / first [] |
  let num = k.(tperiod[t]) + k.(
    tperiod[t']) |
  #num = 0 =>
  #periodTimes[recentPrd[k, t], k.prev,
  t] = wcetNum[t] and
  #periodTimes[recentPrd[k, t'], k.
  prev, t'] = wcetNum[t']}
```

(関数 periodTimes[t, k, k'] は、タスク t が関係 period によって関連付けられている、時刻 k から k' の間の Time を返す)

(関数 recentPrd[k, t] は時刻 k 以前の時間で最近の、タスク t の周期発生時刻を返す) (関数 wcetNum[t] はタスク t の実行時間を返す)

3.3 スケジューリング・アルゴリズムの表現

スケジューリング・アルゴリズムとは、与えられたタスクに対して、それを処理するリソースをポリシーに従い分配するアルゴリズムである。スケジューリング・アルゴリズムはシステムが時間要件を満たせるかどうかを決定づけるものであり、タスク間の処理の優先度を決定付けるアルゴリズムの研究が盛んである。本研究では静的優先度の RM と動的優先度の EDF とによる優先度付けの方法に従ったスケジューリングポリシーを制約として Alloy により表現する。

3.3.1 RM スケジューリング

RM スケジューリングでは周期をデッドラインとし、周期のより短いタスクにより高い優先度を与える、静的な優先度付けの方法である。図 1 は $task_0$ により高い優先度を、 $task_1$ により低い優先度を与えて RM に従ってスケジューリングした例である。RM を表現するために、タスク間の優先度を示す関係 prt を追加する。関係 prt は高々ひとつのタスクから高々ひとつのタスクへの関係であり、非循環である。関係 prt で結ばれた 2 つのタスクにおいて親がより高い優先度を、子がより低い優先度をもつ。より高い優先度のタスクの処理が終了していないにも関わらず、より優先度の低いタスクの処理は実行されることはない、という制約は、ある時間 k においてより優先度の低いタスクが関係 period により関係付けられ、かつ、k 以前の最近の周期発生時刻から時刻 k-1 までの間で処理されたより優先度の高いタスクの数が実行時間分ではないという状況を否定することで RM に従ったスケジューリングを表現する。3.2 で構築したタスク処理モデルに以下のような仕様と制約を追加する。

仕様 3 各周期タスクごとに優先度が存在する

```
abstract sig Task{
  prt: set Task, prd: set Time,
  dl: set Time}
```

仕様 4 優先度は非循環である

```
fact prtRule01{
  prt in Task lone -> lone Task}
fact prtRule02{
  no t: Task | t in t.^prt}
fact prtRule03{
  no t: Task |
  t.prt = none and t.^prt = none}
pred addPrt(t, t': Task){
  t.prt = t.prt + t'}
```

(述語 addPrt(t, t') は、タスク t が関係 prt によって関係付けられている Task に、タスク t' を加える。これによりタスク t, t' 間に $t > t'$ の優先度関

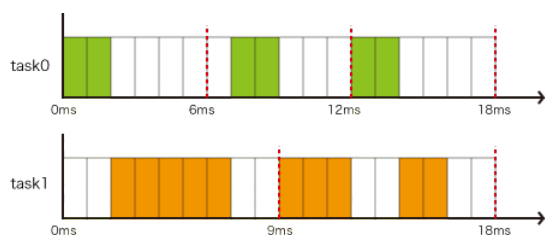


図 5 EDF によるタスクスケジューリングの例
Fig.5 Example of Task scheduling by EDF

係を与える)

制約 3 より優先度の高い周期タスクの処理が完了していないならば、より優先度の低い周期タスクの処理は実行されない

```
fact RateMonotonicRule{
  all k:Time, disj t,t': Task |
    t' = t.prt =>
    not (k.(tperiod[t']) != none and
    #(periodTimes[recentPrd[k,t],k.
    prev,t]) != wcetNum[t]))}
```

3.3.2 EDF スケジューリング

EDF スケジューリングでは周期をデッドラインとして、デッドラインまでの時間が最短のタスクを最優先に処理する、動的な優先度付けの方法である。周期を分母、実行時間を分子とし、各タスクの実行時間/周期の総和が1以下であればスケジューリング可能であることが保証されている。例えば図1と同様の実行条件のタスクセットを EDF に従ってスケジューリングすると図5のようになる。EDF を表現するために、デッドラインまでの時間を返す関数 *deadline* を定義する。関数 *deadline* はある時刻 *k* において、*k* から *k* 以降の最近のデッドラインまでの時間数を返す。デッドラインのより短いタスクの処理が終了していないにも関わらず、よりデッドラインの長いタスクの処理は実行されることはない、という制約は、ある時間 *k* における各タスクのデッドラインまでの時間数を比較し、よりデッドラインの長いタスクが関係 *period* により時刻 *k* におけるタスク処理スロットに関係付けられ、かつ、*k* 以前の最近の周期発生時刻から時刻 *k-1* までの間で処理されたよりデッドラインの短いタスクの数が実行時間分ではないという状況を否定することで EDF に従ったスケジューリングを表現できる。

制約 3 よりデッドラインの短い周期タスクの処理が完了していないならば、よりデッドラインの長い周期タスクの処理は実行されない

```
fact EarliestDeadlineFirstRule{
  all k:Time-Time0/first [],
  disj t,t': Task |
```

```
let dt=deadline[k,t],
dt'=deadline[k,t'],
num= k.(tperiod[t']) |
dt' > dt =>
not(num != none and
#(periodTimes[recentPrd[k,t],k.
prev,t]) != wcetNum[t]))}
```

4. 資源制約を含むタスクスケジューリング

マルチタスクにおいて、複数のタスクが同一のデータや資源を共有する際、同時に複数のタスクがその資源に対してアクセスすることは、データの読み間違いや書き間違いの可能性があり、危険である。したがって、共有資源へアクセスする際は他のタスクのアクセスを禁止する必要がある。複数のタスクによって使用される共有資源があり、排他アクセスを実現せねばならない制約を資源制約 (Resource Constraint) という。周期的に発生するタスクのスケジューリングにおいて資源制約があるときのスケジュール可能性判定は NP 困難 (NP-hard) な問題²⁾である。本章では資源を共有するタスク処理モデルを RM, EDF の優先度の与え方によって Alloy によって表現した。

4.1 資源制約を含んだタスク処理モデルの表現

3章で表現した周期タスクのタスク処理モデルに資源制約を含めるために、新たに共有資源をシグネチャとして定義する。また、周期タスクに共有資源に対するロック・アンロックの操作命令を与えるために、新たに操作命令をシグネチャとして定義する。操作命令には共有資源をロックする P 操作、アンロックする V 操作、そのどちらでもない操作が含まれる。ひとつの操作命令を処理・実行するためには1処理時間スロット分を必要とする。ひとつの周期における操作命令で P 操作と V 操作は対になって行われ、各操作命令の総数はそのタスクの実行時間と等しい。

各操作はタスクの処理と同期する。P 操作、V 操作が処理されるとタスク処理スロットを通して共有資源に対してロック・アンロックの関係が関係付けされる。共有資源がロックされている状態では、新たに P 操作を処理することはできない。これにより待ち状態が発生する。以下が資源制約を表現するための仕様と制約である。

仕様 1 共有資源が存在する

```
sig Resource{nextreso: set Resource}
```

仕様 2 周期タスクには操作命令セット (OperationSet) があり、操作命令には P 操作 (POperation), V 操作 (VOperation), どちらでもな

い操作 (Operations) が含まれる

```
sig OperasionSet{
  orner: set Task,
  nextopeset: set OperationSet,
  job: set Operations}
sig Operations{
  period: set Time,
  nextjob: set Operations}
sig POperation in Operations{
  lock: set Time}
sig VOperation in Operations{
  unlock: set Time}
```

仕様 3 各操作は1 処理単位時間であり、ひとつの周期タスクにおける操作命令の総数は実行時間に等しい

```
fact operationRule01{
  all s: OperationSet | #s.job = #Wcet}
```

仕様 4 ひとつの周期における操作命令の最初は P 操作, 最後は V 操作である

```
fact operationRule02{
  all o: Operations, p: POperation,
  v: VOperation |
  let os=o.^job, ps=p.^job, vs=v.^job |
  os = ps and os = vs =>
  o in (p+p.^nextjob) and
  v in (o+o.^nextjob)}
```

制約 1 周期タスクの操作命令が P 操作であるならば, 共有資源はロックされる

```
fact LockRule01{
  all k:Time, s: OperationSet |
  k in sPoperations[s].period =>
  k in sPoperations[s].lock}
fact LockRule02{
  all k:Time, s: OperationSet |
  k !in sPoperations[s].period =>
  k !in sPoperations[s].lock}
```

(関数 sPoperations[s] は操作命令セット s が関係 job によって関係付けられている P 操作を返す)

制約 2 周期タスクの操作命令が V 操作であるならば, 共有資源はアンロックされる

```
fact unLockRule01{
  all k:Time, s: OperationSet |
  k in sVoperations[s].period =>
  k in sVoperations[s].unlock}
fact unLockRule02{
  all k:Time, s: OperationSet |
  k !in sVoperations[s].period =>
  k !in sVoperations[s].unlock}
```

(関数 sVoperations[s] は操作命令セット s が関係 job によって関係付けられている V 操作を返す)

制約 3 共有資源がロックされているならば, 新たに P 操作は処理されない

```
fact lockRule05{
  all k:Time-Time0/first [],
  disj t, t': Task |
  let p=tperiod[t][tPOperation[t]]+
  tperiod[t'][tPOperation[t']] |
  (k.prev).slock !=none => k !in p}
```

(関数 tPOperation[t] はタスク t の操作命令に含まれている P 操作を返す)

4.2 RM スケジューリング

4.1 で表現したタスク処理モデルに RM に従ったスケジューリングポリシーを追加する. 共有資源がロックされている場合, より優先度が高いタスクであってもその操作が P 操作であるならば実行されないという制約を追加することで資源制約を満たした RM によるスケジューリングアルゴリズムを表現する.

制約 4 共有資源がロックされておらず, より優先度の高い周期タスクの処理が完了していないならば, より優先度の低い周期タスクの処理は実行されない

```
fact RateMonotonicRule2{
  all k:Time-Time0/first [],
  disj t, t': Task |
  let num = k.^(tperiod[t']) |
  t'=t.prt=>(k.prev).slock=none=>
  not (num != none and #(periodTimes
  [recentPrd[k,t],k.prev,t]) !=
  wcetNum[t])}
```

4.3 EDF スケジューリング

4.1 で表現したタスク処理モデルに EDF に従ったスケジューリングポリシーを追加する. 共有資源がロックされている場合, よりデッドラインが短いタスクであってもその操作が P 操作であるならば実行されないという制約を追加することで資源制約を満たした EDF によるスケジューリングアルゴリズムを表現する.

制約 4 共有資源がロックされておらず, よりデッドラインの短い周期タスクの処理が完了していないならば, よりデッドラインの長い周期タスクの処理は実行されない

```
fact EarliestDeadlineFirstRule2{
  all k:Time-Time0/first [],
  disj t, t': Task |
  let dt=deadline[k,t],
  dt'=deadline[k,t'],
  num= k.^(tperiod[t']) |
  dt'>dt=>(k.prev).slock=none=>
  not(num != none and #(periodTimes
  [recentPrd[k,t],k.prev,t]) !=
  wcetNum[t])}
```


表 1 タスクセット 1

Table 1 Task Set 1

周期タスク	周期	実行時間
$task_0$	4	2
$task_1$	7	3

表 3 タスクセット 2

Table 3 Task Set 2

周期タスク	周期	実行時間
$task_0$	4	1
$task_1$	4	1
$task_2$	5	2

表 5 タスクセット 3

Table 5 Task Set 3

周期タスク	周期	実行時間
$task_0$	5	2
$task_1$	7	4

表 2 タスクセット 1 の解析結果

Table 2 Result of Analysis for Task Set 1

優先度	変数	主変数	節	CNF 変換時間	充足可能性判定時間	判定
RM	663366 個	1742 個	1581562 節	37749ms	1309ms	充足可能
EDF	654467 個	1738 個	1536311 節	41632ms	1086ms	充足可能

表 4 タスクセット 2 の解析結果

Table 4 Result of Analysis for Task Set 2

優先度	変数	主変数	節	CNF 変換時間	充足可能性判定時間	判定
RM	258753 個	992 個	467841 節	19346ms	434ms	充足可能
EDF	265337 個	983 個	498768 節	20631ms	641ms	充足可能

表 6 タスクセット 3 の解析結果

Table 6 Result of Analysis for Task Set 3

優先度	変数	主変数	節	CNF 変換時間	充足可能性判定時間	判定
RM	1352123 個	2666 個	3196501 節	76990ms	1646ms	充足不可能
EDF	1337946 個	2662 個	3121594 節	87562ms	1873ms	充足可能

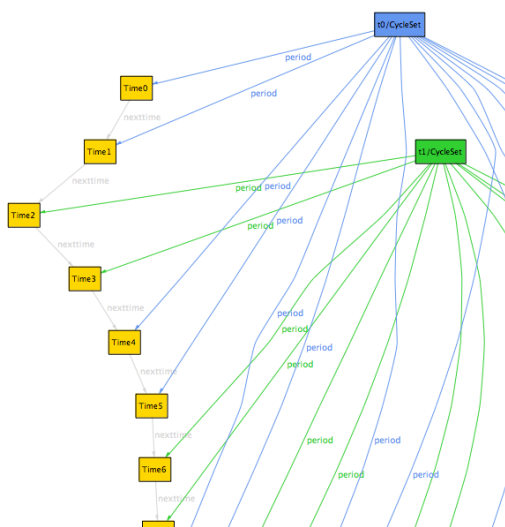


図 6 RM によるタスクセット 1 のスケジューリング例
Fig. 6 Scheduling of Task Set1 by RM

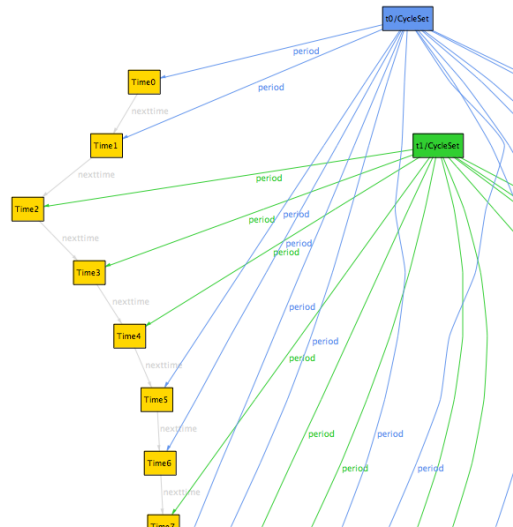


図 7 EDF によるタスクセット 1 のスケジューリング例
Fig. 7 Scheduling of Task Set1 by EDF

5. Alloy Analyzer によるタスクスケジューリング解析

本論文で記述したタスク処理モデルの Alloy ソース記述を Alloy Analyzer により解析し、優先度付けの違いについて比較、評価した。Alloy Analyzer は解析結果として、Alloy ソース記述を CNF 形式ファイルに変換した際の変数、主変数、節数、変換時間、使用した SAT ソルバによる充足可能性判定時間と、充足可能性判定を出力する。本研究の解析で使用した SAT ソルバは SAT4J である。

5.1 資源制約を含まないタスクスケジューリング解析

2 個あるいは 3 個のタスクを RM, EDF に従ってスケジューリングするモデルに対して表 1, 表 3, 表 5 のようなタスクセットを実行条件として用意し、これをスコープに設定し、スケジューリング解析を行った。Alloy Analyzer の解析により、表 2, 表 4, 表 6 に示すような解析結果が得られた。タスク処理モデルにおいて与えたタスクセットがスケジューリング可能であるならばスケジューリングの実例を Alloy モデル図により確認できる。図 6, 図 7 は表 1 のスケジューリングの実例の Alloy モデル図の一部である。Alloy モデ

表 7 タスクセット 4

Table 7 Task Set 4

周期タスク	周期	実行時間
$task_0$	4	2
$task_1$	6	2

表 9 タスクセット 5

Table 9 Task Set 5

周期タスク	周期	実行時間
$task_0$	4	2
$task_1$	6	3

表 8 タスクセット 4 の解析結果

Table 8 Result of Analysis for Task Set 4

共有資源	優先度	変数	主変数	節	CNF 変換時間	充足可能性判定時間	判定
なし	RM	56852 個	366 個	11940 節	2382ms	129ms	充足可能
なし	EDF	55047 個	362 個	111738 節	2874ms	109ms	充足可能
あり	RM	141592 個	1996 個	226237 節	5394ms	8942ms	充足可能
あり	EDF	143953 個	1992 個	233443 節	6886ms	9403ms	充足可能

表 10 タスクセット 5 の解析結果

Table 10 Result of Analysis for Task Set 5

共有資源	優先度	変数	主変数	節	CNF 変換時間	充足可能性判定時間	判定
なし	RM	56852 個	366 個	11940 節	2386ms	45ms	充足不可能
なし	EDF	55047 個	362 個	111738 節	2928ms	131ms	充足可能
あり	RM	141592 個	1996 個	226237 節	5839ms	6988ms	充足可能
あり	EDF	143953 個	1992 個	233443 節	7148ms	6377ms	充足可能

ル図より, RM の場合, $task_0$ の方がより優先度が高いため, Time4 において処理されており, EDF の場合, Time4 のタイミングではデッドラインまでの時間が $task_0$ は 4, $task_1$ は 3 であるためより短い $task_1$ が処理されていることが確認できる. 表 5 は RM ではスケジューリング不可能であるが, EDF ではスケジューリング可能な例である.

5.2 資源制約を含むタスクスケジューリング解析

表 7 のようなタスクセットを実行条件として与え, 資源制約を含むタスク処理モデルと含まないタスク処理モデルとでスケジューリング解析を行った. 表 7 のタスクセットの実行系列は資源制約の有無で変化しないが, 表 8 の解析結果が示すように, Alloy による仕様記述から生成される CNF 論理式の変数, 節の数が増加していることがわかる.

資源制約の追加による優先度の逆転は表 9 のようなタスクセットを実行条件として与え, タスクスケジューリング解析を行うことで確認できた. 表 9 のタスクセットにおいて資源制約を含まない RM スケジューリングでは $task_1$ がデッドラインまでに実行時間分の処理を終了できないためスケジューリングは不可能である. 資源制約を含むスケジューリングにおいては $task_1$ が共有資源をロックすることで $task_0$ は待ち状態となり, $task_1$ はデッドラインまでに処理を完了するため, スケジューリング可能となる.

6. おわりに

本研究では周期的に発生するタスクを処理するリアルタイムシステム, 及びそのスケジューリングポリシーをタスク処理モデルとして Alloy により表現した. RM, EDF の優先度の与え方に従ったスケジュール・アルゴリズムを制約によって表現し, スケジュー

リング可能性を AlloyAnalyzer によって解析した. 資源制約がある, 周期発生タスクの処理を EDF に従ってスケジューリングでき, 実用的なターンアラウンドタイムでスケジューリング可能性を SAT ソルバで柔軟に解くことが可能である. 本研究では, これまで活発に研究されているスケジューリング理論の体系を一階述語論理に基づく制約によって記述した. スケジューリング理論は, 主にオペレーションズリサーチの理論に基づいて発展してきた理論であり, タスク自体は単純な命令の列として表現される. これに対して, リアルタイムシステムではタスクをプログラムで実現するため, タスクの振舞いはより複雑であり, プログラムの設計としては精度がそれほど高いとはいえない. 今後の課題として, Alloy 記述によってタスクの構造をより実現プログラムに近いモデルとすることによって, スケジューラビリティの観点からソフトウェア設計のためのモデルを構築することがあげられる. ここで, モデルの精度は制約解消系の能力に依存する.

マルチコアによるタスクスケジューリングへの拡張には, 複数個のタスク処理時間スロット間の関係や制約に加え, 処理するタスクの挙動がより重要になってくる. 多くのマルチコアによるタスクスケジューリングは NP 困難^{?)} であるが, 決定性スケジューリングに対しては特定の問題は NP 完全ではなく, また, 多項式アルゴリズムの利用や最適スケジュールの開発, 伝統的な理論に基づいたオフラインのヒューリスティックな手法の利用が可能であるため, それほど大きな問題ではない. 多くのヒューリスティクスは平均してよく振舞い, 指数関数的に複雑になるのは稀有な場合のみである. 本研究でのタスクにおける操作命令の振舞いは共有資源に対するロック・アンロックの操作のみであったが, 外的要因等で起動される非周期的な

割込みタスクを含むタスクスケジューリングは Alloy によって表現可能であり、各操作命令の振舞いや命令順序などを解析することで、与えられたタスクが一定の性質をもつならばスケジューリング可能であるような、タスク解析を含めたスケジューリング手法が考えられる。

謝辞 本研究の一部は文部科学省科学技術研究費基盤研究 (B) 課題番号 20300009 の助成による。また、本研究を進めるにあたり、熱心にご指導頂いた名古屋大学大学院情報科学研究科 寺内多智弘 准教授、結縁・寺内研究室の皆様に深く感謝いたします。

参 考 文 献

- 1) Alloy.
<http://alloy.mit.edu/>.
 - 2) S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemp-tive scheduling of periodic real-time tasks on one process, 1990.
 - 3) Daniel Jackson. *Software Abstractions*. The MIT Press, 2009.
 - 4) C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, Vol. 20, pp. 46–61, January 1973.
 - 5) 中島震, 鷗林尚靖. Alloy : 自動解析可能なモデル規範形式仕様言語. コンピュータソフトウェア, Vol. 26, No. 3, 2009.
 - 6) J.A. Stankovic. *Deadline scheduling for real-time systems: EDF and related algorithms*. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1998.
 - 7) J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, Vol. 28, pp. 16–25, 1995.
-