情報処理学会論文誌 Vol.60 No.4 1025–1039 (Apr. 2019)

環境変化時に保証可能な安全性を特定するための ゲーム分析アルゴリズム

相澤 和也[1,a) 鄭 顕志[1,2,b) 本位田 真一[1,2,c)

受付日 2018年8月1日, 採録日 2019年1月15日

概要：ソフトウェアシステムの安全性は，通常，開発時に想定した実行環境の前提下において保証される．この前提が実行時の環境変化等によって崩れると，システムの安全性は保証されない．実行時に起こる環境変化に対して可能な限りの安全性を維持・保証するためには，変化した環境下でどのような安全性が保証可能かを実行時に分析する必要がある．実行時の情報を用いた分析手法は環境情報の分析にかかる計算時間オーバヘッドが課題となる．本論文では，(1) 環境変化の差分情報から効率的に安全性保証の判定を行うアルゴリズムを提案し，(2) アルゴリズムの効率性に関する評価と (3) 安全性保証に関する証明を行う．このアルゴリズムは安全性を構成する要素ごとの保証可否と，環境変化によって生じる差分の 2 つの観点に基づいて分析している．これら 2 つの観点を組み合わせることによって既存技術を用いた分析と比べて計算時間を最大 0.2%程度にまで削減できることが実験結果によって確認できた．

キーワード：自己適応システム，離散制御器合成，安全性保証

Identifying Guaranteeable Safety Property in Changed Environment by Game Analysis

Kazuya Aizawa[1,a) Kenji Tei[1,2,b) Shinichi Honiden[1,2,c)

Received: August 1, 2018, Accepted: January 15, 2019

Abstract: Safety properties of software systems are typically verified and guaranteed at development time under environmental assumptions put by developers. The guarantees become invalid if a change in the environment breaks the assumptions. Runtime analysis to identify guaranteeable safety properties under the changed environment is necessary to enable self-adaptation to maintain guarantees on safety properties as much as possible. However, it cannot be applied if computation time is too long to use at runtime. In this paper we propose a fast runtime analysis algorithm based on difference caused by changes in the environment. Experimental results show that the computation time of the algorithm is fast enough for practical use. We also provide the reason of the efficiency and formal guarantee to our algorithm. We confirmed that our algorithm could reduce the execution time to 0.2% in some case of our evaluation.

Keywords: self-adaptive system, discrete controller synthesis, safety property guarantee

1. はじめに

ソフトウェアシステムの振舞いには安全性が求められる．動作環境におけるシステムの誤った振舞いは重大な損失につながりかねない．ゆえに開発者は安全性を保証するような仕様を策定しようとする．安全性を保証するために，開発者は動作環境に対して前提条件を置いて仕様を策定する [1], [2]．しかし，システム開発時に置かれたこれら

1 早稲田大学
Waseda University, Shinjuku, Tokyo 162–0042, Japan
2 国立情報学研究所
National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
a) k.a.s-s-i-t-w@ruri.waseda.jp
b) ktei@aoni.waseda.jp
c) honiden@nii.ac.jp

© 2019 Information Processing Society of Japan

の前提が実行時に崩れた場合，システムの安全性は保証できない．

環境が変化しても安全性等の要求を保つために，環境変化に合わせて仕様を切り替える自己適応システムに関する研究がなされてきた [3]．環境が変化し，前提が崩れても，その環境で可能な限りの安全性を保証する仕様に切り替えればシステムの信頼性を保てる．この自己適応システムにおける課題の 1 つが，変化した環境で満たせる要求をどのように分析し，仕様を準備するかである．

要求の分析と仕様の準備における課題に対して，環境変化が要求に与える影響を開発時に分析して仕様を準備するアプローチがとられてきた [1], [4], [5], [6], [7]．開発者は開発時にありうる環境の変化を予測し，それぞれの環境で保証可能な要求を特定し，仕様を策定する [4], [5], [6]．開発時の開発者の作業負担を軽減するために環境と要求の分析および仕様の合成を自動化する研究も行われてきた [1], [7]．しかし，いずれも開発時の予測に依存した分析であるため，予測と異なる環境の変化が起きた場合，本来は保証が可能な要求を保証することができない．

これに対して実行時に自動で環境変化を分析し，要求を保証できるように仕様を合成するアプローチも研究されてきた [2], [8], [9], [10]．実行時に環境変化の情報を用いて満たしうる要求を特定し，それらの要求を満たす仕様を自動で合成する．環境を実行時に分析するため，開発時のアプローチに比べて，より豊富な要求を保証する仕様に切り替えることが可能である．しかし，主に非機能要求が実行時の分析対象として扱われており，機能要求を実行時に分析することは，我々の知る限り，これまで行われてこなかった．

ここで機能要求とは，ソフトウェアシステムの動作環境における事象の発生に対する要求を指している．本論文における安全性とは Alpern ら [11] が述べた「誤った事象が起こらないこと」という機能要求を指している．たとえば「荷物を運ぶロボットは荷物を持っていないときに荷物を下ろす動作を行わない」等が該当する．これに対して非機能要求はソフトウェアシステムが振舞いを行うときに満たすべき性能や品質を指しており，たとえば「荷物を運ぶロボットの荷物を下ろす動作は 5 秒以内に完了する」等が該当する．機能要求を保証するためにはモデル検査のように振舞いを網羅的に調べる必要があることから時間的な課題があり，設計段階で機能要求を保証するような仕様を準備するアプローチ [1] がとられてきた．

我々は環境変化時に保証可能な安全性を効率的に分析するアルゴリズムを提案する．このアルゴリズムは安全性を保証するための分析を環境の振舞いモデルと形式化された安全性から構築されるゲーム上で行う．このアルゴリズムが扱う環境変化は，環境の振舞いモデルに新しい振舞いが追加されるような場合である．これは開発時に置かれた前提が崩れることによって期待されたものと異なる振舞いが発生することを想定した環境変化であり，安全性は特にこのような振舞いが追加されたときに保証不能となりうるからである．実行時分析を行うにあたって計算時間が課題となる．我々は分析時に安全性を構成する要素ごとの保証可否と，環境変化によって生じる差分の 2 つに着目することで計算時間を削減する．

提案するアルゴリズムの初期のアイディアは先行論文 [12] で報告されている．文献 [12] では前述した 2 つの観点に基づくアルゴリズムとそのアルゴリズムによる計算時間の評価を行った．それに加えて本論文では 2 つの観点のそれぞれが効率化に与える効果を評価し，またアルゴリズムによって特定される安全性が保証されることを証明することでこのアルゴリズムの有用性を示す．

本論文の評価では (1) 提案する実行時分析の手法と開発時の分析手法とで，保証可能な安全性を 30 の実験ケースで比較し，18 のケースで実行時分析の手法がより高い安全性を保証することを確認した．また (2) 提案するアルゴリズムの計算時間を観点ごとに測定し，既存の開発時分析手法 [1] を拡張した手法と比較した．環境変化によって生じる差分に着目したことによって最大 0.8%程度にまで計算時間が削減できたことを確認し，さらに要素ごとの保証可否に着目することで最大 0.2%程度にまで削減できることを確認した．

本論文の構成は次に示すとおりである．2 章で関連研究について詳述し，3 章では背景技術について説明する．4 章で本論文が扱う問題を例示し，5 章では安全性を分析するアルゴリズムを詳述する．6 章でアルゴリズムの評価を行い，7 章で研究のまとめと将来研究について記述する．

2. 関連研究

自己適応システムにおける仕様の準備については様々なアプローチがとられてきた．開発時に開発者が準備するアプローチとしては次のような研究がなされている．Cailliau ら [4] は要求とそれを阻害する要因との関係をゴールモデルを用いて表現し，阻害要因に対抗するための仕様に切り替えるためのフレームワークを提案した．彼らは機能要求の阻害要因となる事象とその対抗手段を開発時に形式化して分析している．Calinescu ら [6] は振舞い仕様のモデルに実行時に測定する確率パラメータを付与することで，確率モデルチェッキングにより最も要求を満たす確率の高い仕様を動的に選択する手法を提案した．Fredericks ら [5] は開発時に仕様とそのテストスイートを用意し，ランタイムテストを行うことで切り替える仕様に保証を与える手法を提案した．彼らはテストスイートと実行環境の関係性を保つために，遺伝的アルゴリズムによってテストスイートを適応させた．

上記のアプローチでは開発者に依存する部分が大きく，

© 2019 Information Processing Society of Japan

その負担や人為的ミスが課題であった．この課題を解決するために仕様の自動合成に関する研究がなされてきた．Cámara ら [7] はシステムが扱える振舞いとそれらの振舞いが環境に与える影響のみを開発者に記述させ，非機能要求に対して確率モデルチェッキングによる保証をともなった仕様を自動合成できるようにした．D'Ippolito ら [1] は前提が崩れた環境のモデルを開発時に段階的に用意し，それぞれの段階の環境で保証可能な機能要求を分析したうえで，離散制御器合成の技術によって仕様を準備し，実行時に切り替える手法を提案した．

開発時に準備するアプローチはいずれも環境変化を予測する必要がある．予測が誤っていた場合，本来満たせる要求が満たせなくなる．この問題に対して実行時に環境情報を分析し，要求が満たせるように仕様を自動合成するアプローチが考えられてきた．Ghezzi ら [2] はシステムがとりうる振舞いのすべてを記述した確率遷移モデルを用いて，実行時に非機能要求を満たす確率が最も高くなるように振舞い仕様を自動合成する手法を提案した．Cámara ら [10] はシステムの構造に関する適応戦略を自動合成する技術を提案した．適応戦略が与える影響を確率ゲームとしてモデル化し，そのゲームの中で得られる報酬が最大となる状態を目指す戦略を実行時に自動合成した．Qian ら [8] は実行時に起きた非機能要求の違反に対してゴールモデルを用いた推論による仕様の合成を行い，さらに効果的な仕様は環境情報等とともにケースとして蓄積することで，実行時の最適な振舞いを学習することを可能にした．Incerto ら [9] はキューによって構成されたシステムを対象として非機能要求を保証するパラメータ制御器を実行時に合成する手法を提案した．彼らは非線形な常微分方程式によってモデル化されたシステムを離散近似と線形近似によって変形することで制御器合成時の計算時間を削減した．

実行時に機能要求の保証に着目した仕様の自動合成は我々の知る限り，取り組まれてこなかった．我々は開発時に行われてきた機能要求を保証する仕様の自動合成を実行時にも実現できるように，安全性要求の保証可否を分析するアルゴリズムを提案する．

3. ロボットシステムによる問題例

本論文の目的の説明のために倉庫内自動荷物運搬ロボットシステム “Kiva システム [13]” を例として用いる．Kiva システムとは倉庫内での荷物出荷作業における生産性を向上させるために提案された倉庫管理システムである．Kiva システムの具体的な構成と振舞い [14] について説明する．

Kiva システムは商品の在庫保存および注文された商品の出荷準備を行うための倉庫管理システムである．倉庫は 4 つの作業エリアとそれらのエリアをつなぐように位置する在庫保存のエリアが存在する．それぞれのエリアの役割は以下である．

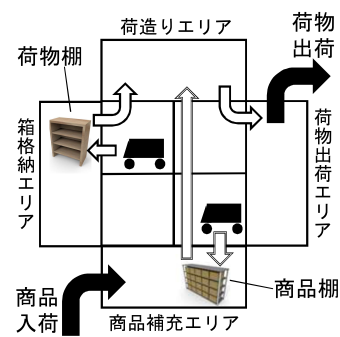

図 1 Kiva システムにおけるロボットの動き
Fig. 1 Behaviors of the Kiva system.

- 箱格納エリア：荷物を詰め込む箱を準備するエリア
- 荷作りエリア：商品を箱に詰める作業をするエリア
- 荷物出荷エリア：荷造りした荷物を出荷するエリア
- 商品補充エリア：入荷した商品を補充するエリア
- 在庫保存エリア：在庫商品を保存するエリア

上記の 5 つのエリアにロボットが作業用の棚を運ぶことで荷物の出荷作業を行う．人手の作業は 4 つの作業エリアに集約されているため作業効率が上がる．作業用の棚には，出荷する荷物を運ぶための荷物棚と商品を保存しておくための商品棚の 2 種類が存在し，いずれもふだんは在庫保存エリアに置かれている．

Kiva システムにおけるロボットの動きを図 **1** に示す．荷物棚を運ぶロボットは荷物棚を在庫保存エリアから箱格納エリアに移動させ，荷物を詰める箱を受け取る．その後，荷造りエリアに移動して商品棚と待ち合わせ，商品を箱に詰める．そして出来上がった荷物を荷物出荷エリアに運ぶ．一方，商品棚を運ぶロボットは商品棚を在庫保存エリアから商品補充エリアに移動して商品を補充した後，荷造りエリアに移動して商品棚と待ち合わせる．

Kiva システムではそれぞれの棚を運ぶロボットが各エリアに正しい順番で入ることや，在庫保存エリアを移動する際に他のロボットと衝突しないこと等が安全性として求められる．これらの安全性はロボットの移動動作が正しく機能するという前提の下に保証することができる．しかし，ロボットのセンサやアクチュエータの劣化や倉庫内の路面状況によっては移動動作が期待と異なる結果になることが考えられる．本論文ではこのような環境変化時にも可能な限りの安全性を維持することを目的としている．

4. 背景技術

本論文で説明するアルゴリズムは離散制御器合成 [15], [16] を背景としている．離散制御器合成は与えられた環境下で与えられた要求を満たす制御器を自動で合成する技術である．システムと環境との相互作用は Labeled transition system（LTS）[17] によってモデル化され，システムが満たすべき安全性は Fluent linear temporal logic（FLTL）[18]

© 2019 Information Processing Society of Japan

```
const MaxX=1 const MaxY=1 range N=0..1 set Direction={e,w,n,s}
set Controllable  = {[N].move[Direction], [N].wait, [N].pick, [N].put, start, end}
INITIAL(I = 0) = (when (I == 0) [I].arrive[0][1]->start -> STORAGE[0][1]
             |when (I ==1) [I].arrive[1][1] ->start-> STORAGE[1][1]
STORAGE[x:X][y:Y] = (
    when (x < MaxX ) [I].move['e] -> [I].arrive[x+1][y] -> STORAGE[x+1][y]
    |when (x > 0 ) [I].move['w] -> [I].arrive[x-1][y]-> STORAGE[x-1][y]
    |when (x == MaxX) [I].move['e] -> [I].arriveShipping[y]->[I].wait-> SHIPPING[y]
    |when (x == 0) [I].move['w] -> ([I].arriveInduction[y]->[I].wait-> INDUCTION[y])
    |when (y < MaxY ) [I].move['s] -> ([I].arrive[x][y+1]-> STORAGE[x][y+1] )
    |when (y > 0 ) [I].move['n] -> [I].arrive[x][y-1]->STORAGE[x][y-1]
    |when (y == MaxY) [I].move['s] -> [I].arriveReplenish[x]->[I].wait-> REPLENISH[x]
    |when (y == 0) [I].move['n] -> [I].arrivePicking[x]->[I].wait-> PICKING[x]
    |[I].pick -> [I].picksucc -> STORAGE[x][y]
    |[I].put -> [I].putsucc -> STORAGE[x][y]
    |end ->reset ->INITIAL),
SHIPPING[y:Y] = ([I].move['w] ->([I].arrive[MaxX][y]-> STORAGE[MaxX][y]
                |when(I==0&&y==0)[I].arriveShipping[y]->SHIPPING[y])//追加遷移
                |end ->reset -> INITIAL),
REPLENISH[x:X] =(...),
INDUCTION[y:Y]=(...),
PICKING[x:X]=(...).
||ENV=(forall [i: N](INITIAL(i))).
```

図 2　Kiva システムのモデルの一部

Fig. 2　Part of the model of the Kiva system.

によって記述される．離散制御器合成ではこれら環境のモデルと要求の記述からゲーム空間を構築し，そこから制御器が合成可能であれば制御器を LTS モデルとして出力する．アルゴリズムはこのゲーム空間を分析することで，与えられた環境下で与えられた安全性の保証可否を分析する．

定義 1. LTS は $E = (S, A, \Delta, s_0)$ で表現される．S は有限の状態集合，$A = A_C \cup A_U$ は環境下で発生するアクションの集合，$\Delta \subseteq (S \times A \times S)$ は遷移関係，そして $s_0 \in S$ は初期状態である．ここで A_C はシステムが制御可能なアクションの集合であり A_U はシステムが制御不可能なアクションの集合である．

LTS による環境モデルの例として，図 1 に示した Kiva システムのモデルを図 **2** に示す．モデルの記述には Finite State Process（FSP）[17] を用いている．モデル化するのは作業用の棚を持つロボットの倉庫内での振舞いである．「INITIAL」はロボットの初期配置を示しており，「STORAGE」,「SHIPPING」,「REPLENISH」,「INDUCTION」,「PICKING」はロボットの 5 つのエリアを示している．「STORAGE」は他の 4 つのエリアをつなぐ 2×2 マスの在庫保存エリアであり，ロボットの移動と置かれている棚の持ち上げ，持ち下げのアクションが記述されている．他の 4 つのエリアには在庫保存エリアとの行き来に関するアクションと，それぞれのエリアでの作業を待つ「wait」の動作が記述されている．「end」は荷造りと出荷の完了通知のアクション，「reset」は新たな荷造りと出荷作業の開始通知のアクションである．後述するゲームを構築するうえでは環境モデル上で制御器が制御可能なアクションの集合を定義する必要があり，ここでは「Controllable」として表している．

環境に対して置かれた前提が崩れた場合，システムのアクションに対して，環境から期待と異なる応答が返される [1]．したがって本論文では環境モデルに対してそのような遷移や状態が追加されるような環境変化に対して焦点を当てる．

```
SHIPPING[y:Y] = ([I].move['w] ->([I].arrive[MaxX][y]-> STORAGE[MaxX][y]
                |when(I==0&&y==0)[I].arriveShipping[y]->SHIPPING[y])//追加遷移
                |end ->reset -> INITIAL),
```

図 3　Kiva システムのモデル更新の例

Fig. 3　An example of model update of the Kiva system.

```
fluent FlagforTask1 = <[0].arriveInduction[0], reset>
fluent ArriveStorage[a:A][x:X][y:Y] = <[a].arrive[x][y], {[a].move[Direction], reset}>

ltl_property GOAL03 = [] (!(ArrivePicking[0][0]) W (FlagforTask1))
ltl_property InvalidReturn1 = [](FlagforTask1 -> (!X [0].arriveInduction[0] W (reset)))
ltl_property StopConflict010T0 =[]!((ArriveStorage[0][0][0] && ArriveStorage[1][0][0]))
```

図 4　Kiva システムの安全性の一部

Fig. 4　Part of safety properties of the Kiva system.

Kiva システムにおける環境モデルの更新例を図 **3** に示す．ロボットが出荷エリアから在庫保存エリアに移動しようとしたとき，路面の劣化等から移動が失敗し，出荷エリアに留まる場合が発生したとする．そのような環境変化では図 3 に示すように，モデルに新たな遷移が追加される．我々はこのような環境変化を対象とする．

また，FLTL による Kiva システムが満たすべき安全性の一部を図 **4** に示す．安全性の要素は環境モデル上のアクション名と fluent と呼ばれる要素から記述される．たとえば「FlagforTask1」は荷物棚を運ぶロボットが箱格納エリアに到達してから作業が完了し，再び作業を開始するまでの間，真となるような論理要素である．安全性の要素として定義されている「GOAL03」と「InvalidReturn1」はエリアに入る順序の一部である．前者は荷物棚を運ぶロボットは箱格納エリアを訪れるまでは出荷エリアに訪れてはならないことを表し，後者は荷物棚を運ぶロボットが箱格納エリアで作業をした後に，箱格納エリアに戻ってくることを禁止している．また，「StopConflict010T0」は他のロボットと衝突しないための要素の一部であり，2 体のロボットがある場所に同時に入らないことを表している．

本論文では，安全性の要素を ϕ で表し，その要素の集合 Φ によってシステムに対する安全性を定義する．また，ϕ には優先度が与えられ，Φ の優先度は以下のように ϕ の優先度の総和により計算される[*1]．

定義 2. ある安全性の要素の集合 Φ が与えられたとき，安全性の要素 $\phi \in \Phi$ の安全性は実数を返す関数 $p : \Phi \rightarrow \mathcal{R}$ によって表現され，Φ の優先度は p を用いて $p(\Phi) = \sum_{\phi \in \Phi} p(\phi)$ と計算する．また，ある 2 組の安全性の集合 Φ_i，Φ_j が与えられたとき，$p(\Phi_i) > p(\Phi_j)$ であれば Φ_i は Φ_j よりも優先度が高いという．$p(\Phi_i) = p(\Phi_j)$ の場合，それぞれの安全性要素 $\phi_i \in \Phi_i$，$\phi_j \in \Phi_j$ の優先度を最大のものから順に 1 対 1 で比較し，$p(\phi_i) > p(\phi_j)$ となるような ϕ_i が発見されれば Φ_i は Φ_j よりも優先度が高いという．

安全性の要素に対する優先順位付けについては，たとえばゴールモデルを用いた手法 [19] を用いることができる．

離散制御器合成では環境モデル上で安全性が保証できる

[*1] 本提案は安全性の優先度計算自体には非依存のため，アプリケーションに応じて優先度計算の方法を変更することが可能．

© 2019 Information Processing Society of Japan

かを分析するために，次のような 2 人対戦型のゲームが構築される [15], [16]．ここでゲームのプレイヤは一方が制御器であり，もう一方が環境である．

定義 3. 2 人対戦型のゲームは $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ で表現される．S_{sg} は有限の状態集合，$\Gamma^-, \Gamma^+ \subseteq S_{sg} \times S_{sg}$ は Γ^- が環境が，Γ^+ がシステムの制御器が制御できる状態遷移関係で，$s_{sg0} \in S_{sg}$ は初期状態，そして $X \subseteq 2^{S_{sg}}$ は勝利条件である．ある $s_{sg} \in S_{sg}$ から Γ^- によって遷移できる状態の集合を $\Gamma^-(s_{sg}) = \{s'_{sg} | (s_{sg}, s'_{sg}) \in \Gamma^-\}$ と表現し，Γ^+ についても同様に表現する．ゲーム SG のプレイを Γ^-, Γ^+ に従うような遷移列 $p = s_{sg0}, s_{sg1}, \ldots$ で表現する．ある有限のプレイ $p = s_{sg0}, s_{sg1}, \ldots, s_{sgn}$ が与えられたとき，$\Gamma^-(s_{sgn}) \neq \emptyset$ であれば，環境の手番であり，環境が任意の $s_{sgn+1} \in \Gamma^-(s_{sgn})$ を，$\Gamma^+(s_{sgn}) \neq \emptyset \wedge \Gamma^-(s_{sgn}) = \emptyset$ であれば制御器の手番であり，制御器が任意の $s_{sgn+1} \in \Gamma^+(s_{sgn})$ を選択して p の末尾に追加できる．プレイに $s_{sg} \in \{x | x \in X\}$ が含まれるとき，環境側の勝利となり，そうでない場合はシステムの制御器側の勝利となる．

ここで，与えられた安全性の要素 ϕ と構築されたゲームの勝利条件の要素 $x_\phi \in X$ には対応関係が存在し，x_ϕ は ϕ を違反する状態の集合である．たとえば，図 4 の「InvalidReturn1」に対する勝利条件 $x_{InvalidReturn1}$ は荷物棚を運ぶロボットが箱格納エリアでの作業後に戻ってきてしまったことを表す状態の集合となる．

このようなゲームにおいては環境側，制御器側のそれぞれがつねに自身の勝利条件を満たせるような領域を次のように定義できる [20]．

定義 4. ゲーム $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ が与えられたとき，すべての $s_{sg} \in w_E$ から制御器による遷移の選択に非依存に環境が勝利できるとき，w_E を環境の勝利領域と呼ぶ．また，すべての $s_{sg} \in w_C$ において，環境による遷移の選択に非依存に制御器が勝利できるとき，w_C をシステムの制御器の勝利領域と呼ぶ．

離散制御器合成では w_C を特定することで保証可否が判断される．ここで，SG は到達可能性ゲーム [20] であるため，次の定理が成り立つことが知られている．

定理 1. ゲーム $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ および環境の勝利領域 $w_E \subseteq S_{sg}$ が与えられたとき，システムの制御器側の勝利領域は $w_C = S_{sg} \backslash w_E$ である．

すなわち，w_C は w_E を特定することで導き出せるのである．

ゲーム SG における遷移関係には LTS のようなアクションが存在しないが，SG を構築するときに入力となった環境モデルの遷移関係と対応関係が存在する [16]．すなわち，SG 上の遷移が環境モデル上のどの遷移であるかを知ることのできる関数 $Inv : (\Gamma^- \cup \Gamma^+) \to \Delta$ が存在するのである．離散制御器合成ではこの関係を用いてゲーム空間の制御器側の勝利領域から制御器モデルを構築している．

5. 実行時環境モデル分析手法

本章では実行時に変化した環境を分析し，保証可能な安全性を最大化するアルゴリズムについて説明する．このアルゴリズムは以下の 3 つを入力とする．

- 環境変化を反映したモデル
- 安全性の集合の候補
- 実行中の制御器モデル

環境変化を反映したモデルは環境学習の手法 [21], [22] によって実行時に取得する．安全性の集合の候補は開発時に開発者によって用意される．集合の候補は優先度に基づく全順序で与えられる．実行中の制御器モデルは，変化する前の環境モデルとそこで保証されていた安全性の集合から合成された制御器のモデルある．我々は制御器モデルを安全に切り替えるために，実行中の制御器モデルを合成された制御器モデルがシミュレートできること [1] を条件としている．このアルゴリズムの出力は与えられたモデル下で保証可能な安全性の集合の候補のうち，最大のものである．本章では 2 つのアルゴリズムを説明する．1 つは離散制御器合成 [16] を利用して開発時に仕様を準備し，実行時に切り替える既存手法 [1] の仕様準備も実行時に行うようにしたアルゴリズムである．もう 1 つは，提案する要求の要素単位，および，環境変化の差分に着目した分析アルゴリズムである．

5.1 離散制御器合成を用いた安全性の実行時分析

図 5 は離散制御器合成を利用した分析アルゴリズムの概観を示している．このアルゴリズムは開発時に離散制御器合成を利用して仕様を準備して実行時に切り替える既存手法 [1] を拡張し，分析と仕様準備を実行時に行うようにしたアルゴリズムである．既存手法では仕様の準備を開発時に行っているが，仕様の準備は離散制御器合成を利用した自動合成であることから，入力となる環境モデルと安全性の集合を用意できれば実行時にも仕様が合成可能である．そこで，開発時に安全性の集合の候補を準備し，実行時に適切な仕様が得られるまで安全性の候補から逐次的に選んで合成を試みるのがこのアルゴリズムである．

図中の安全性の集合の候補は n 個与えられており，$G(n)$

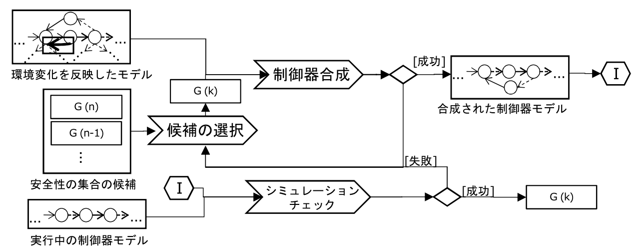

図 5　離散制御器合成による安全性分析

Fig. 5　Analysis of safety properties with discrete controller synthesis.

© 2019 Information Processing Society of Japan

から $G(n-1), G(n-2), \ldots, G(1)$ と，優先度順に並べている．このアルゴリズムでは与えられたモデル下で保証可能な最大の安全性の集合を，逐次探索によって特定する．まず，候補の中から最も優先度の高い集合を選択し，環境変化を反映したモデル下での制御器合成を試みる．合成が成功した場合には，得られたモデルが実行中の制御器モデルをシミュレート可能か，シミュレーションチェックを行う．シミュレーションチェックに成功した場合，その集合をアルゴリズムの出力とする．制御器合成，シミュレーションチェックのどちらかに失敗した場合，優先度を1つ下げた集合を用いて同様の操作を行う．

このアルゴリズムでは保証可能な最大の安全性の集合が見つかるまで制御器合成を繰り返し実行するが，1回の制御器合成にかかる計算時間は小さくないため，繰り返し実行することによって計算時間が膨大となる．我々の評価では最大で20分かかるケースが存在した．したがってこのアルゴリズムでは，計算時間の問題から適用可能なアプリケーションが限られてしまう．

5.2 要素単位かつ差分に着目した安全性分析

図6は本論文で提案するアルゴリズムの概観を示している．我々のアルゴリズムは開発時と実行時の2つに分かれている．開発時には環境モデルと安全性の集合の候補から保証可能な安全性を分析するためのゲーム空間を構築し，安全性の各要素について分析する．このため候補からすべての要素を抽出している．この分析によって，環境モデル上のどの状態がどの要素を保証可能かを特定することができる．実行時には，環境変化を反映したモデルからゲームの差分更新を行い，各状態で保証可能な要素についても差分分析を行う．その後，更新されたゲーム空間が実行中の制御器モデルをシミュレートするときに，必要となる状態を確認し，すべての状態で保証可能な要素の集合を特定する．ただし，環境の状態によっては保証可能な要素が競合する場合があるため，保証可能な安全性の集合には複数の候補が存在しうる．したがってこの候補の中から最大のものを選択してアルゴリズムの出力とする．

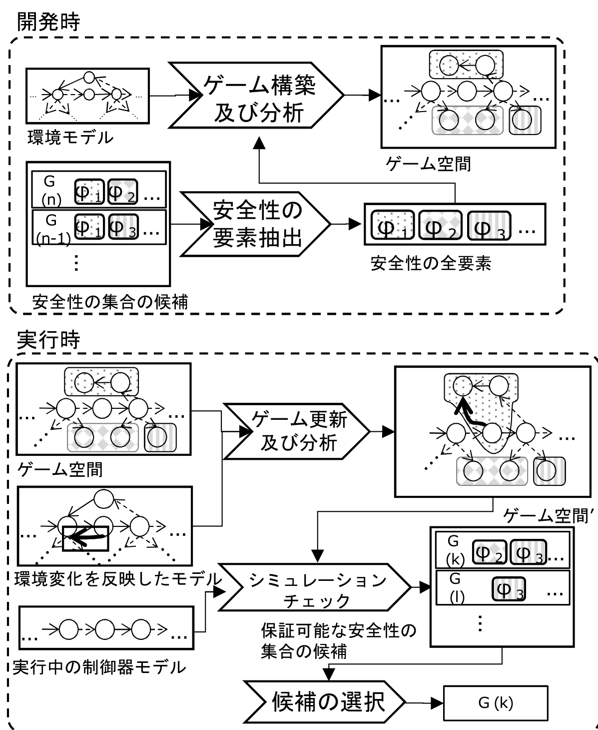

図6 要素単位かつ差分に着目した安全性分析の概観
Fig. 6 Overview of difference analysis of safety properties.

ゲーム空間の構築および更新については既存の離散制御器合成 [16] における構築アルゴリズムによって構築可能である．したがって本論文ではゲームにおける安全性の要素単位での分析アルゴリズムと環境変化の差分に着目した安全性分析アルゴリズムを説明する．

5.2.1 分割型勝利領域による安全性の要素単位での分析

ゲーム空間において保証可能な安全性を要素単位で分析するためには，どの要素がどの状態で保証できなくなるかを特定する必要がある．したがって次のような領域を定義する．

定義 5. ゲーム $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ が与えられたとき，ある $X_\Phi \subseteq X$ を勝利条件とするゲーム $SG_G = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X_\Phi)$ の環境の勝利領域 w_{X_Φ} を SG の X_Φ における環境の部分勝利領域と呼ぶ．また，すべての X_Φ に対して $w_{X_\Phi} \in W$ であるような W を SG における環境の分割型勝利領域と呼ぶ．

この定義から，X による部分勝利領域 w_X は SG 自身の環境の勝利領域でもある．以下，環境の部分勝利領域および分割型勝利領域を単に部分勝利領域，分割型勝利領域と呼ぶ．この定義から次の定理が導き出せる．

定理 2. ゲーム $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ が与えられたとき，任意の $x \in X$ に対するゲーム $SG' = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X \backslash \{x\})$ の分割型勝利領域は SG の分割型勝利領域から $x \in X_\Phi$ であるようなすべての $X_\Phi \subseteq X$ の部分勝利領域を取り除いたもの，すなわち $W' = W \backslash \{w_{X_\Phi} | x \in X_\Phi, X_\Phi \subseteq X\}$ である．

証明． SG の $x \notin X_{\Phi'}$ であるような $X_{\Phi'} \subseteq X$ による部分勝利領域は定義5より SG' の部分勝利領域でもあるため SG' の分割型勝利領域に含まれる，つまり，$w_{X_{\Phi'}} \in W'$ である．一方で定義5と $\{X_\Phi \subseteq X | x \notin X_\Phi \wedge X_\Phi \not\subseteq X \backslash \{x\}\} = \emptyset$ から $x \notin X_{\Phi'}$ であるような $X_{\Phi'}$ による部分勝利領域以外は SG' の分割型勝利領域には含まれない．したがって $x \in X_\Phi$ であるような $X_\Phi \subseteq X$ による部分勝利領域は $X_\Phi \not\subseteq X \backslash \{x\}$ より，SG' の部分勝利領域ではないため，SG' の分割型勝利領域にも含まれない，つまり，$w_{X_\Phi} \notin W'$ である．したがって，W' は W から $X_\Phi \subseteq X$ による部分勝利領域を取り除いたものに一致する．以上より，$W' = W \backslash \{w_{X_\Phi} | x \notin X_\Phi, X_\Phi \subseteq X\}$ である． □

© 2019 Information Processing Society of Japan

定理 2 より，ゲーム空間中で実行中の制御器モデルをシミュレートするときに必要となる状態が，ある $X_\Phi \subseteq X$ による部分勝利領域に含まれるとき，任意の $x \in X_\Phi$ を取り除いた $X_{\Phi'} \subseteq X$ を勝利条件とするようなゲーム SG' であれば，実行中の制御器モデルをシミュレートしながら，$X_{\Phi'}$ に対応する安全性の要素を保証する制御器モデルが構築可能ということである．

5.2.2　分割型勝利領域の構築アルゴリズム

あるゲーム $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ における分割型勝利領域は到達可能性ゲームにおける勝利領域の特定アルゴリズム [20] を拡張することで特定可能である．具体的なアルゴリズムを **Algorithm 1** に示す．

分割型勝利領域の特定はゲーム SG 内の勝利条件 X が起点となる．部分勝利領域 $w_{\{x_\phi\}}$ を対応する勝利条件 x_ϕ で初期化（4–7 行目）し，以後，この部分勝利領域に含まれる状態に遷移可能な状態に，部分勝利領域を伝播させる形で特定していく（9–13 行目）．n 回目の伝播が終わったとき，$n-1$ 回目と分割型勝利領域がまったく変わっていなければ，それ以上伝播させる状態が存在しないため終了する．また，具体的な部分勝利領域の伝播のさせ方は，対象の状態が環境の手番か制御器の手番かによって異なる．それぞれのアルゴリズムを **Algorithm 2** および **Algorithm 3** に示す．

Algorithm 2 はある時点で特定されている分割型勝利領域 W に遷移できる状態 s_{sg} のうち，環境の手番であるような状態に対して部分勝利領域を伝播させる（4 行目）．環境の手番においては制御器が遷移を選択できないため，環境の選択による遷移先の部分勝利領域 w を避けることができない．したがって，そのような w に s_{sg} を追加する．

Algorithm 3 はある時点で特定されている分割型勝利領域 W に遷移できる状態 s_{sg} のうち，制御器の手番であるような状態に対して部分勝利領域を伝播させる（4 行目）．制御器の手番においては制御器が遷移を選択できるため，すべての遷移先がいずれかの部分勝利領域に含まれるような s_{sg} を対象に部分勝利領域を伝播させる（4 行目）．このとき，s_{sg} は遷移先のそれぞれが属している部分勝利領域 w_{x_Φ} を発生させている勝利条件の和集合による部分勝利領域に含まれる．ここで，遷移先の状態が複数の部分勝利領域に含まれている場合を考える．このような場合，s_{sg} は遷移先のそれぞれの状態が属している勝利領域の組合せを変えてできる全勝利領域に属することになる．このため，部分勝利領域を構築する勝利条件の集合に対して s_{sg} の遷移先の状態についての直積集合を構築しているのである（5 行目）．ただし，本論文では直積集合を $\prod_{\lambda \in \Lambda} A_\lambda = \{\{a_\lambda\}_{\lambda \in \Lambda} | a_\lambda \in A_\lambda, \forall \lambda \in \Lambda\}$ によって定義する．これによって得られる集合族から，それぞれの勝利条件の集合を取り出し，集合内の勝利条件すべての和集合をとることで求める勝利条件を取得し，対応する勝利領域に s_{sg} を加える．

開発時にはこのようにして，構築したゲームの部分勝利領域と分割型勝利領域を特定する．なお，提案したアルゴリズムによって特定される領域によって得られる保証については付録に記載する．

5.3　分割型勝利領域の差分更新アルゴリズム

実行時の環境変化に基づいてゲームが更新されたとき，分割型勝利領域もまた更新が必要となる．更新されたゲームにおいて分割型勝利領域を更新するためのアルゴリズムを **Algorithm 4** に示す．

Algorithm 4 は **Algorithm 1** の初期化を更新前の分

Algorithm 1 分割型勝利領域の生成

1: INPUT: $S_{sg}, \Gamma^-, \Gamma^+, X$
2: OUTPUT: W_n
3: $W_0 \Leftarrow \{\}$
4: **for all** $x_{\phi_i} \in X$ **do**
5: 　$w_{\{x_\phi\}} \Leftarrow x_{\phi_i}$
6: 　$W_0 \Leftarrow W_0 \cup w_{\{x_{\phi_i}\}}$
7: **end for**
8: $n = 0$
9: **while** $W_n \neq W_{n-1}$ **do**
10: 　$n \Leftarrow n+1$
11: 　$W_n \Leftarrow W_{n-1} \cup$ **Algorithm 2**(W_{n-1}, Γ^-)
　　　$\cup$ **Algorithm 3**(W_{n-1}, Γ^+)
12: **end while**
13: **return** W_n

Algorithm 2 環境の手番における部分勝利領域の特定

1: INPUT: W, Γ^-
2: OUTPUT: W'
3: $W' \Leftarrow \{\}$
4: **for all** $s_{sg} \in S_{sg} | y \in \Gamma^-(s_{sg}), y \in \bigcup_{w \in W} w$ **do**
5: 　**for all** $w \in W | \Gamma^-(s_{sg}) \cap w \neq \emptyset$ **do**
6: 　　$w' \Leftarrow w \cup \{s_{sg}\}$
7: 　　$W' \Leftarrow W' \cup \{w\}$
8: 　**end for**
9: **end for**
10: **return** W'

Algorithm 3 制御器の手番における部分勝利領域の特定

1: INPUT: W, Γ^+
2: OUTPUT: W'
3: $W' \Leftarrow \{\}$
4: **for all** $s_{sg} \in S_{sg} | \Gamma^-(s_{sg}) = \emptyset, \forall y \in \Gamma^+(s_{sg}),$
　　$y \in \bigcup_{w \in W} w$ **do**
5: 　$F^{F^{X_\Phi}} \Leftarrow \prod_{s'_{sg} \in \Gamma^+(s_{sg})} \{X_\Phi \in 2^X | s'_{sg} \in w_{X_\Phi}, w_{X_\Phi} \in W\}$
6: 　**for all** $F^{X_\Phi} \in F^{F^{X_\Phi}}$ **do**
7: 　　$X_\Psi \Leftarrow \bigcup_{X_\Phi \in F^{X_\Phi}} X_\Phi$
8: 　　$w_{X_\Psi} \Leftarrow w_{X_\Psi} \cup \{s_{sg}\}$
9: 　　$W' \Leftarrow W' \cup \{w_{X_\Phi}\}$
10: 　**end for**
11: **end for**
12: **return** W'

© 2019 Information Processing Society of Japan

Algorithm 4 分割型勝利領域の更新

```
1: INPUT: S_sg, Γ−, Γ+, X, W
2: OUTPUT: W_n
3: W_0 ⇐ W
4: n = 0
5: while W_n ≠ W_{n−1} do
6:     n ⇐ n + 1
7:     W_n ⇐ W_{n−1} ∪ Algorithm 2(W_{n−1}, Γ−)
                     ∪ Algorithm 3(W_{n−1}, Γ+)
8: end while
9: return W_n
```

割型勝利領域 W に変更したものである．本論文では環境モデルに新たな遷移や状態が追加される場合に焦点を当てている．このとき，ゲーム空間の更新内容は新しい遷移と状態の追加である．したがって分割型勝利領域 W の更新内容はこの追加された状態と遷移が含まれる部分勝利領域の特定である．ゆえに W によって初期化した状態で，部分勝利領域の特定を開始すれば分割型勝利領域の更新が可能である．すでに特定済みの W は再計算の必要がないため，計算時間の削減が図れるのである．

5.4　分割型勝利領域を用いた保証可能な安全性の特定

分割型勝利領域を用いて実行時に保証可能な安全性を特定するアルゴリズムについて説明する．変化後の環境で安全性を保証し，かつ，実行中の制御器と安全に切り替えられる制御器を合成するためには合成される制御器が実行中の制御器をシミュレート [1] する，すなわち，プロセスを模倣する必要がある．この関係を保てるように，まずは実行中の制御器がゲーム SG によってシミュレート可能かを確認する．しかしながら，SG の遷移関係にはアクションが存在しないためそのままでは LTS モデルである制御器をシミュレートすることは不可能である．そこで SG 上の遷移関係と対応する環境モデル上の遷移関係を取得する関数 Inv を用いてゲーム上の遷移のアクションを補完する．そのうえで SG と制御器とのシミュレーション関係を確認する．その過程で到達したゲーム上の状態をすべて記憶する．記憶された状態のうち，SG の部分勝利領域に含まれる状態が存在するならば，その状態が含まれる部分勝利領域をすべて取り出す．この部分勝利領域が実行中の制御器をシミュレートするような新しい制御器を合成するためには回避できない領域である．このようにして求めた部分勝利領域を構築する安全性の集合のうち，いずれか 1 つが保証できなくなるため，優先度計算に基づいて保証できる安全性が最大となるように除外する安全性を決定する．

6.　評価

本章では提案手法の有用性を，分析結果と分析にかかる計算時間を測ることによって評価する．この評価にあたって 2 つの研究課題を設定する．

研究課題 1　実行時分析の手法は開発時の分析手法に比べてより高い安全性を保証することが可能か．

研究課題 2　要素単位での分析，および環境変化の差分分析は安全性の実行時分析の計算時間をどの程度効率化できるのか．

研究課題 1 は実行時に保証可能な安全性を分析することの有用性を確認することを目的としている．本研究の目的は計算時間を効率化したアルゴリズムを提案することで開発時にのみ行われてきた機能要求の分析を実行時にも可能にすることである．したがって，開発時の分析と実行時の分析とで保証できた安全性を比較することで本研究の目的の有用性を確かめている．

研究課題 2 では実行時分析の計算時間が提案手法によって削減されているかを確認することが目的である．本研究では開発時の分析手法 [1] を拡張し，仕様準備を実行時に行う手法と，計算時間の削減を意図した提案手法の 2 つの実行時分析について説明をした．この提案手法が開発時の分析手法の拡張よりも時間効率が良くなっていることを確認する．

評価時に利用した計算機の CPU は Intel(R) Core(TM) i7-4790K CPU @4.00 GHz，メモリは 16.0 GB RAM，OS は Windows10 Home 64 bit である．また，モデルや安全性の要素の記述には MTSA [23] を用いた．MTSA は離散制御器合成 [16] も実装されており，制御器合成を用いた分析手法では MTSA を利用した．MTSA は Java で実装されていることから，条件を揃えるために評価に用いたアルゴリズムについては Java で実装している．

6.1　研究課題 1 の評価設定

研究課題 1 を評価するにあたっては，開発時の分析手法として，機能要求を対象としている D'Ippolito ら [1] の手法と比較し，保証できる安全性の要素数を評価する．ただし，安全性の要素間には優先度が存在する場合がしばしば存在することより，それぞれの手法で保証できた安全性の要素自体についても確認する．

評価については小規模なモデルと中規模なモデルの 2 つを用いる．小規模なシナリオとして，前章で説明した Kiva システムのシナリオを用いる．ただし，本評価においては商品棚を運ぶロボットの 1 台を対象としてモデル化している．これは生成されるゲーム空間が状態爆発することを防ぐためである．この設定は本手法の限界によるものであり，これについては 6.5 節で後述する．この環境モデルは小規模であり，状態数は 66 であり，安全性の要素の個数は 13 個である．

もう 1 つは製品加工工場におけるロボットシステムのシナリオである．材料をトレイから取り出し，オーブンで焼きドリルで穴を空け，プレス機で成形してトレイに戻す，という加工作業を行う．それぞれの加工機械の間はロボットアームによって材料が運ばれる．この環境モデルは中規

© 2019 Information Processing Society of Japan

表 1　各ケースで遷移が追加された順番（Kiva システム）
Table 1　The order in which transitions were added in each case (Kiva system).

ケース番号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
追加遷移	A	B	C	D	D-E	B-D	B-D-E	B-C	C-B	D-B	A-C	D-B-C	D-B-E	D-C-E	D-B-C-E

表 2　各ケースで遷移が追加された順番（製品加工）
Table 2　The order in which transitions were added in each case (production cell).

ケース番号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
追加遷移	A	B	C	D	E	D-C	E-A	D-B	C-E	D-A	A-C	D-A-E	D-C-B	D-E-A	D-A-E-C

模であり，状態数は 6,944 である．また，想定される環境変化としてはロボットアームの移動が失敗し，正しい位置に到着できなくなる，材料を持ち上げたり下したりする動作に失敗するようになる，等がある．このシステムに対する安全性として，材料を正しい手順で加工することや，作業中の加工機械の下に材料を運ばないこと等があげられる．このシステムの安全性の要素は合計で 22 個である．

開発時の分析手法 [1] では環境変化の発生順を予測したうえで，予測に沿って段階的に環境変化を追加した複数の環境モデルを用意する必要がある．たとえば製品加工工場のロボットアームについて，トレイからドリルに移動する動作と，トレイからオーブンに移動する動作の 2 つが失敗する可能性がある場合を考える．開発時の分析手法はどちらが先に失敗するのかを予測して，先に発生すると予測した動作の失敗を組み込んだ環境モデルと，両方の動作の失敗を組み込んだ環境モデルの 2 つを用意し，それぞれの環境モデルで保証可能な安全性の分析と仕様の合成を行う必要がある．実行時には発生した失敗が含まれている環境モデルと仕様に切り替わるのである．

本評価では，それぞれのシナリオにおいて動作の失敗を想定した遷移を 5 つ用意し，環境モデルにこのうちのいくつかを順番に追加するケースを準備し，最後の遷移を追加したときに安全性の要素がいくつ保証できたかを測定する．

開発時の分析手法ではあらかじめ 5 つの動作の失敗の発生順序の予測と，それぞれが発生したときに保証できなくなる安全性の特定，そして離散制御器合成による仕様の合成を行っておき，それぞれのケースにおいて適用される仕様と保証される安全性を特定する．実行時分析では，それぞれのケースにおいて最後の遷移が追加されるタイミングで分析を実行し，保証可能な安全性を特定する．実行時分析で用いる安全性の集合の候補は，5 つの動作の失敗によって保証できなくなる可能性のある安全性を開発時に特定し，それらを全体の集合から組み合わせながら取り除いて構築する．Kiva システムでは 17，製品加工では 16 の候補が構築され，候補は優先度に従って全順序で並んでいる．

各ケースで実際に遷移が追加された順番を**表 1**，**表 2** に示す．追加される遷移は A から E までのラベルで表記し，追加された順番に-（ハイフン）でつないでいる．ケースは

表 3　開発時に準備したモデルに追加された遷移と保証される安全性の要素数（Kiva システム）
Table 3　Transitions added to the design-time models and safety properties guaranteed there (Kiva system).

モデル	1	2	3	4	5
追加遷移	-	A, D	A, D, B	A, D, B, C	A, D, B, C, E
保証要素数	13	12	11	10	9

表 4　開発時に準備したモデルに追加された遷移と保証される安全性の要素数（製品加工）
Table 4　Transitions added to the design-time models and safety properties guaranteed there (production cell).

モデル	1	2	3	4	5
追加遷移	-	D	D, A, B	D, A, B, E	D, A, B, E, C
保証要素数	22	21	20	19	18

開発時分析の予測どおりに遷移が追加されるケースを 4 つ準備し，残りは任意に選択して生成して各シナリオで 15 ケースずつ，合計 30 ケース実施する．予測どおりに遷移を追加したケースは Kiva，製品加工ともにケース 4，10，12，15 である．開発時分析で予測に基づいて遷移を追加して構築された環境モデルとそこで保証される安全性の要素数を**表 3**，**表 4** に示す．遷移を追加しないモデル 1 からすべての遷移を追加したモデル 5 までの 5 段階となっている．開発時の分析では遷移の追加によって保証されなくなる安全性の要素数が最小となるようにモデルを構築しているため，準備した環境モデルごとに 1 つずつ保証される安全性の要素数が少なくなっている．また異なる遷移が同じ安全性を保証できなくする場合は同時に 1 つのモデルに追加している．開発時分析ではこの 5 つの環境モデルとそこで保証される安全性，そしてそれらから合成される仕様を構築した．

6.2　研究課題 2 の評価設定

研究課題 2 を評価するにあたっては，研究課題 1 で扱った 2 つのシナリオ，30 のケースを対象に前章で説明した制御器合成を用いた分析手法と分割型勝利領域による分析および勝利領域の差分更新による分析の計算時間を評価する．ここで，分割型勝利領域による分析と勝利領域の差分更新による計算時間への削減効果がどの程度であるかを明

© 2019 Information Processing Society of Japan

確にするため，2つの手法を組み合わせた場合に加えてそれぞれの手法に分割したときの計算時間も計測する．それぞれの手法への分割方法は次のとおりである．分割型勝利領域による分析については図6の開発時に行うゲーム構築を実行時に行うものとする．勝利領域の差分更新による分析については，1つのゲームについての分割型勝利領域を生成する代わりに，安全性の集合の候補の組合せごとにゲームを構築し，その勝利領域を差分更新するものとする．

また，安全性の実行時分析は実行時に新しい仕様を自動合成するために行うことより，制御器合成によって新しい仕様が合成されるまでの時間も含めた評価も行う．提案手法による分析の計算時間に制御器合成によって新しい仕様を合成するのにかかった時間を加えたうえで，制御器合成を用いた分析手法でかかった時間と改めて比較する．

評価には2つのシナリオに対して，環境変化を想定した遷移が追加されたモデル上で保証可能な最大の安全性を特定し，その結果と分析にかかった計算時間を測定する．なお，実行中の制御器のモデルは遷移が追加される前の環境モデルと安全性の集合から離散制御器合成 [16] を用いて合成したものを利用する．

本評価においては環境変化をモデルに反映するまでの時間を評価していないが，Tanabeらの手法 [22] を用いれば小規模であれば10 ms，大規模な場合も100 ms程度でモデルに反映できる．この時間は数分かかる場合がある仕様の自動合成に比べて無視できる程小さいため，全体の性能評価には影響を与えないものと判断し，評価対象から外している．

6.3 研究課題1の評価結果

それぞれのシナリオのそれぞれのケースにおいて保証された安全性の個数を**表5**に示す．なお，製品加工のシナリオにおける結果については過去の評価結果 [12] からの引用である．いずれのケースにおいても実行時分析によって保証された安全性の要素数は開発時分析によって保証された安全性の要素数以上であった．開発時分析の予測どおりに遷移を追加したケースについては開発時分析も実行時分析も同じ数の安全性の要素を保証していた．

開発時分析は予測が正しければ，環境の変化に即したモデルが適用できるため，そのときに保証可能な安全性はすべて保証可能である．実行時分析でも開発時分析と同様にすべての安全性を保証することができる．一方で，開発時分析の予測と異なるケースでは実行時分析が開発時分析と同じか，より多くの安全性の要素を保証していた．開発時分析では後に追加されると予測した遷移が先に追加されると，本来は追加されていない遷移まで追加されたモデルが適用される．これにより本来は保証できたはずの安全性の要素が保証できなくなる場合がある．実行時分析では追加された遷移に基づいてゲームを更新し分析するため，これを防ぐことができるのである．

表5 保証された安全性の要素数
Table 5 The number of guaranteed safety properties.

ケース番号	シナリオ1 Kivaシステム		シナリオ2 製品加工	
	開発時分析	実行時分析	開発時分析	実行時分析
1	12	12	20	21
2	11	12	20	21
3	10	12	18	21
4	12	12	21	21
5	9	11	19	21
6	11	11	18	20
7	9	10	19	20
8	10	10	20	20
9	10	10	18	20
10	11	11	20	20
11	10	11	18	20
12	10	11	19	19
13	9	10	18	19
14	9	10	19	20
15	9	9	18	18
平均	10.13	10.80	19.00	20.07

つまり，実行時分析によって保証された安全性は開発時分析によって保証された安全性を包含するものであった．実行時分析の方が多くの安全性を保証できたケースはいずれも，ある特定の場所で移動が失敗するようになることを想定したケースであり，保証できなくなった安全性は移動の順番に関するものであった．

以上の結果から，実行時分析の手法は開発時の分析手法において，開発時の予測が的中した場合と比べて同じだけの安全性を保証することが可能であり，また予測が外れた場合と比べるとより多くの安全性を保証できる可能性があるといえる．開発時の分析手法では遷移が追加される順番は1つに絞らなければならないが，複数の順番を扱えるように拡張すれば実行時分析と同様な分析は可能である．しかしながら準備するモデルの数が膨大となってしまう．仮に環境が変化する順番をすべて網羅しようとした場合，必要なモデルの数は追加される遷移の集合の冪集合の個数に一致する．今回の実験では5つの遷移がモデルに追加される想定であったが，遷移が追加される順番のすべてを再現するには32のモデルが必要となる．必要となるモデルの個数が追加される遷移の数に対して指数関数的に増加することから開発時にすべての環境変化の過程を網羅することは困難である．したがって，環境変化の過程を予測することが難しい場合，本手法は開発時の分析手法に比べて多くの安全性を保証する可能性が十分にあると考えられる．

6.4 研究課題2の評価結果

本実験における手法ごとの計算時間と制御器合成を用いた分析手法に対する各手法の計算時間の割合を**図7**, **図8**に

© 2019 Information Processing Society of Japan

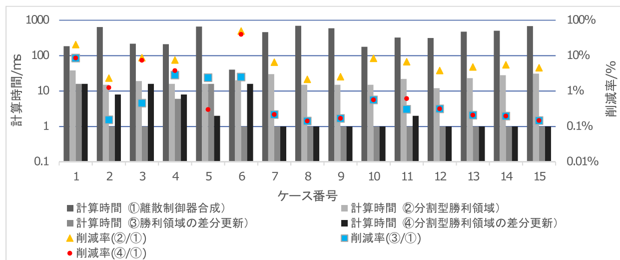

図 7　安全性分析の実行時間（Kiva システム）

Fig. 7　Execution time of analysis (Kiva system).

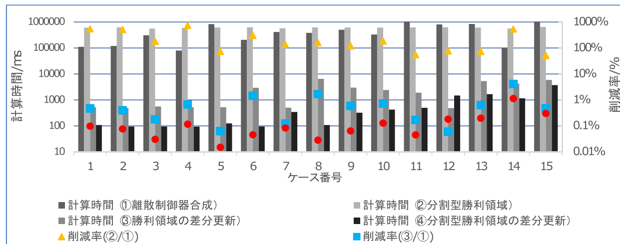

図 8　安全性分析の実行時間（製品加工工場）

Fig. 8　Execution time of analysis (production cell).

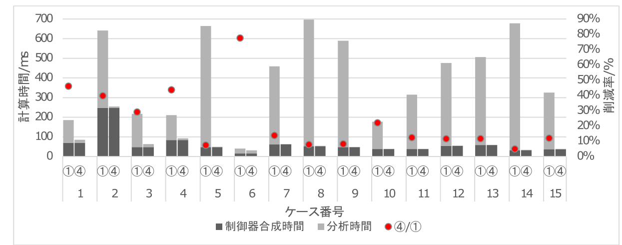

図 9　仕様合成まで含めた実行時間（Kiva システム）

Fig. 9　Execution time including synthesis (Kiva system).

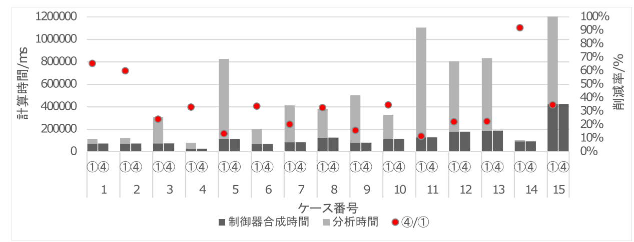

図 10　仕様合成まで含めた実行時間（製品加工工場）

Fig. 10　Execution time including synthesis (production cell).

示す．棒グラフの縦軸は左側であり，各ケースにおける 4 つの分析手法の計算時間を表している．グラフ上の 3 種類のプロットの縦軸は右側であり，各ケースにおける手法ごとの計算時間の割合を表している．軸はいずれも対数軸である．

Kiva システムについての結果を図 7 に示す．離散制御器合成を用いた分析に比べて，分割型勝利領域による分析で中央値 5.5%，最悪時で 50%に，勝利領域の差分更新で中央値 0.3%，最悪時で 8.7%に，2 つを組み合わせた分割型勝利領域の差分更新で中央値 0.3%，最悪時で 40%にまで計算時間を削減できた．これより，いずれの手法も削減に対して効果があることが確認できた．一方で，5 つのケース（2，3，4，6，11）では勝利領域の差分更新よりも分割型勝利領域の差分更新の方が計算時間がかかっていた．これは本シナリオにおけるゲーム空間の更新箇所が小規模であったため，多くのゲーム空間の勝利領域を更新するコストよりも，分割型勝利領域を更新するコストの方が高くなってしまったためと考えられる．

製品加工についての結果を図 8 に示す．なお，離散制御器合成を用いた分析と分割型勝利領域の差分更新による分析については過去の評価結果 [12] の引用である．離散制御器合成を用いた分析に比べて，勝利領域の差分更新で中央値 0.5%，最悪時で 4.2%，分割型勝利領域の差分更新で中央値 0.1%，最悪時で 1.1%にまで計算時間を削減できた．しかし，Kiva システムの場合と違い，分割型勝利領域では中央値 160%，最悪時で 750%にまで計算時間が増加した．これはゲーム空間の規模が大きくなり，また安全性の要素間での関係性が複雑になったために部分勝利領域の特定に時間を要しているためであると考えられる．一方で，勝利領域の差分更新に比べて分割型勝利領域の差分更新の方が計算時間が短くなっている．これは Kiva システムのときと比べてゲーム空間の更新箇所が大きくなったため，複数のゲーム空間の勝利領域を更新するコストが大きくなり，分割型勝利領域の更新コストを上回るようになったものと考えられる．

分割型勝利領域の差分更新による分析の計算時間とモデルの規模については次のように考察する．分割型勝利領域の実行時更新では，状態が属している部分勝利領域の更新が計算時間の大部分を占めており，計算時間はこの部分勝利領域がどの程度更新されたかに依存している．Kiva システムのシナリオで構築されたゲームの状態数は約 2,500 で，実行時に部分勝利領域が更新された状態数は 100 から 1,000 程度であり，全体に占める割合は平均で 17%，最悪時で 40%であった．これに対し，製品加工のゲーム空間の状態数は約 89 万であったが，実行時に部分勝利領域が更新された状態数は 200 から 1 万程度で全体に占める割合は平均で 0.3%，最悪時でも 1.1%であった．2 つのシナリオのゲーム空間の規模は約 350 倍程の差があるが，実行時に更新される状態の数は最悪時どうしの比較で 10 倍に収まっていた．これよりゲーム空間の規模が増大しても実行時に環境モデルに追加される遷移が同程度であれば，更新される状態数はそれほど増大しないと考えられる．一方で追加される遷移が多ければ実行時に更新される状態数が増加し，計算量は大きくなると考えられる．これについては後述の 6.5 節で議論する．

次に分割型勝利領域の差分更新による分析において新しい仕様の合成までを行った場合の計算時間と制御器合成を用いた分析手法に対する各手法の計算時間の割合を**図 9**，**図 10** に示す．棒グラフの縦軸は左側であり，各ケースに

© 2019 Information Processing Society of Japan

おける，①離散制御器合成を用いた分析での制御器合成までの時間と④分割型勝利領域の差分更新による分析での制御器合成までの時間を表している．棒グラフの明るい部分は安全性の分析に要した時間であり，暗い部分は制御器合成に要した時間である．①の結果は図 7，図 8 と同じもので，最後に制御器合成を行った時間が暗い部分で表されている．グラフ上のプロットの縦軸は右側であり，各ケースにおける計算時間の割合を表している．軸はいずれも線形軸である．

新しい制御器を合成するまでの計算時間を含めた場合，制御器合成を用いた分析に対する分割型勝利領域の差分更新による分析の計算時間の割合は Kiva システムで中央値 32.7%，最悪時で 92.1%，製品加工で中央値 12.4%，最悪時で 77.5%であった．

製品加工のシナリオにおいて，制御器合成を用いた分析手法では計算時間の最大値はケース 15 の 20 分であった．環境変化から仕様切替えまでの間システムが止まった場合，その分だけ製品加工のスケジュールに遅延が生じ，完成した製品の出荷等に影響を及ぼす．この影響を少なくするために，環境変化から仕様切替えまでにかかる時間は少なくとも 1 製品あたりの加工時間を下回らなければならないと考えられる．こうした状況において制御器合成を用いた分析手法では，たとえば 1 製品あたりの加工時間が 10 分の工場には適用が困難である．これに対して分割型勝利領域の差分更新による分析の計算時間の最大はケース 15 の 7 分であることから適用が可能であると考えられ，提案手法による高速化は適用可能なアプリケーションを広げることができたといえる．しかしながら，このような時間制約が 5 分以下であるような工場では適用できないため，さらなる高速化は今後の課題である．

分割型勝利領域の差分更新による安全性の分析にかかる計算時間に比べて新しい仕様の合成にかかる計算時間が著しく大きく，全体の計算時間はほとんど新しい仕様の合成にかかる時間であるといえる．これ以上の高速化を行うためには仕様の合成についても高速化が必要となる．提案手法では制御器合成で用いるゲームに対して差分更新を行い分析の高速化を実現した．この発想を仕様合成に対しても適用し，仕様の差分合成が可能となればさらなる高速化が期待でき，適用可能なアプリケーションをさらに広げられるものと考えられる．この仕様の差分合成については将来研究として今後取り組んでいく．

6.5 提案手法の限界と妥当性への脅威

本評価で用いた Kiva システムの環境モデルおよび安全性はゲーム空間が状態爆発を起こさないように意図的に変更している．開発時の分析手法では安全性と環境モデルの分析範囲を限定して行えば，提案手法ではゲーム空間が状態爆発するような場合でも適用可能であることから，提案手法では開発時にゲーム空間が状態爆発する場合に適用できないという限界がある．この限界については開発時の分析と同様に，分析範囲の限定を行うことで低減できる可能性があることから将来研究として今後解決を図る．

また，本実験で扱った環境変化は遷移を 1 つ追加するような小さなものであることから，本評価に外的妥当性への脅威が存在する．多くの遷移を追加するような大きな環境変化が起こる場合，実行時に部分勝利領域が更新される状態数が増加し，それにともなって分析にかかる計算時間も増大する可能性がある．あるいは非常に大きな環境変化によって実行時分析中にゲーム空間が状態爆発する可能性もわずかながら存在する．しかしながら，実システムにおける環境の変化は多くの場合は小さなものである [24] ことから，適用可能性を著しく下げるものではないと考えられる．また，状態爆発の有無については開発時に予測されうるすべての環境変化を含めた環境モデルを対象にゲームの構築と分割型勝利領域の生成を試みることで事前にある程度検知することが可能である．今後，モデル上の評価だけでなく実システムを交えた評価も実施し，有効性を確認したい．

7. おわりに

本論文では環境変化時に保証可能な安全性を効率的に特定するアルゴリズムを提案した．そしてそのアルゴリズムが基づく 2 つの観点が計算時間に対してどのような効果を与えているのか，そして，それら 2 つを組み合わせた場合にどのような効果が得られるのかを評価し，特に規模の大きくなった場合において，提案したアルゴリズムが有効であることを明らかにした．また，アルゴリズムによって特定される安全性が保証されることについても証明を与えることで明らかにした．

将来研究としては次の 4 つをあげる，1 つは前章であげたゲーム空間の構築における状態爆発を抑え，アプリケーションの適用範囲を広げることである．2 つ目は今回扱わなかった環境モデル上の遷移が消えるような環境変化に対応することである．3 つ目が安全性に加えて，「つねにいつかは成立する」という活性要求も取り扱えるようにすることである．そして 4 つ目は実行時差分更新のアプローチを制御器合成にまで拡張することである．また，これと並行して評価するアプリケーションも，IoT 等に範囲を広げていく．

謝辞 本研究の一部は JSPS 科研費 18H03225，17H00732，および独立行政法人情報通信研究機構（NICT）の委託研究「欧州との連携によるハイパーコネクテッド社会のためのセキュリティ技術の研究開発」の成果です．

参考文献

[1] D'Ippolito, N., Braberman, V., Kramer, J., Magee, J., Sykes, D. and Uchitel, S.: Hope for the best, prepare

© 2019 Information Processing Society of Japan

for the worst: Multi-tier control for adaptive systems, *Proc. 36th International Conference on Software Engineering, ICSE 2014*, pp.688–699, ACM (2014).

[2] Ghezzi, C., Pinto, L.S., Spoletini, P. and Tamburrelli, G.: Managing non-functional uncertainty via model-driven adaptivity, *2013 35th International Conference on Software Engineering (ICSE)*, pp.33–42 (May 2013).

[3] Salehie, M. and Tahvildari, L.: Self-adaptive software: Landscape and research challenges, *TAAS*, Vol.4, pp.14:1–14:42 (2009).

[4] Cailliau, A. and van Lamsweerde, A.: Runtime monitoring and resolution of probabilistic obstacles to system goals, *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp.1–11 (May 2017).

[5] Fredericks, E.M. and Cheng, B.H.C.: Automated generation of adaptive test plans for self-adaptive systems, *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp.157–167 (May 2015).

[6] Calinescu, R., Ghezzi, C., Kwiatkowska, M. and Mirandola, R.: Self-adaptive software needs quantitative verification at runtime, *Comm. ACM*, Vol.55, pp.69–77 (Sep. 2012).

[7] Cámara, J., Schmerl, B., Moreno, G.A. and Garlan, D.: MOSAICO: Offline synthesis of adaptation strategy repertoires with flexible trade-offs, *Automated Software Engineering*, Vol.25, No.3, pp.595–626 (May 2018).

[8] Qian, W., Peng, X., Chen, B., Mylopoulos, J., Wang, H. and Zhao, W.: Rationalism with a dose of empiricism: Case-based reasoning for requirements-driven self-adaptation, *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pp.113–122 (Aug. 2014).

[9] Incerto, E., Tribastone, M. and Trubiani, C.: Software performance self-adaptation through efficient model predictive control, *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp.485–496 (Oct. 2017).

[10] Cámara, J., Garlan, D., Schmerl, B. and Pandey, A.: Optimal planning for architecture-based self-adaptation via model checking of stochastic games, *Proc. 30th Annual ACM Symposium on Applied Computing, SAC '15*, pp.428–435, ACM (2015).

[11] Alpern, B. and Schneider, F.B.: Recognizing safety and liveness, *Distributed Computing*, Vol.2, No.3, pp.117–126 (1987).

[12] Aizawa, K., Tei, K. and Honiden, S.: Identifying safety properties guaranteed in changed environment at runtime, *The 3rd IEEE International Conference on Agent (ICA '18)* (July 2018).

[13] Wurman, P.R.. D'Andrea, R. and Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses, *Proc. 19th National Conference on Innovative Applications of Artificial Intelligence - Volume 2, IAAI '07*, pp.1752–1759, AAAI Press (2007).

[14] Enright, J.J. and Wurman, P.R.: Optimization and coordinated autonomy in mobile fulfillment systems, *Proc. 9th AAAI Conference on Automated Action Planning for Autonomous Mobile Robots, AAAIWS '11-09*, pp.33–38, AAAI Press (2011).

[15] Piterman, N., Pnueli, A. and Sa'ar, Y.: Synthesis of reactive(1) designs, *Verification, Model Checking, and Abstract Interpretation*, Emerson, E.A. and Namjoshi, K.S. (Eds.), pp.364–380, Springer Berlin Heidelberg (2006).

[16] D'Ippolito, N.R., Braberman, V., Piterman, N. and Uchitel, S.: Synthesis of live behaviour models, *Proc. 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10*, pp.77–86, ACM (2010).

[17] Magee, J. and Kramer, J.: *Concurrency: State Models and Java Programs*, 2nd ed., Wiley Publishing (2006).

[18] Giannakopoulou D. and Magee, J.: Fluent model checking for event-based systems, *SIGSOFT Softw. Eng. Notes*, Vol.28, pp.257–266 (Sep. 2003).

[19] Angelopoulos, K., Souza, V.E.S. and Mylopoulos, J.: Dealing with multiple failures in zanshin: A control-theoretic approach, *Proc. 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014*, pp.165–174, ACM (2014).

[20] Grädel, E., Thomas, W. and Wilke, T. (Eds.): *Automata Logics, and Infinite Games: A Guide to Current Research*. Springer-Verlag New York, Inc. (2002).

[21] Sykes, D.. Corapi, D., Magee, J., Kramer, J., Russo, A. and Inoue, K.: Learning revised models for planning in adaptive systems, *Proc. 2013 International Conference on Software Engineering, ICSE '13*, pp.63–71, IEEE Press (2013).

[22] Tanabe, M., Tei, K., Fukazawa, Y. and Honiden, S.: Learning environment model at runtime for self-adaptive systems, *Proc. Symposium on Applied Computing, SAC '17*, pp.1198–1204, ACM (2017).

[23] Braberman, V., D'Ippolito, N., Piterman, N., Sykes, D. and Ucriitel, S.: Controller synthesis: From modelling to enactment, *2013 35th International Conference on Software Engineering (ICSE)*, pp.1347–1350 (May 2013).

[24] Gerasimou, S., Calinescu, R. and Banks, A.: Efficient runtime quantitative verification using caching, look-ahead, and nearly-optimal reconfiguration, *Proc. 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014*, pp.115–124, ACM (2014).

付　　録

A.1 分割型勝利領域を特定するアルゴリズムによって得られる保証の証明

本付録では本論文で説明したアルゴリズムによって得られる保証について記載する．ここで，保証とは安全性の各要素に対応する勝利条件を含んだゲームに制御器がつねに勝利できることを指している．本付録ではアルゴリズムによって特定した領域から保証できなくなった安全性に対応する部分勝利領域を取り除くことで，他の安全性が保証できることを証明する．

証明に先立って，**Algorithm 2** および **Algorithm 3** によって特定される領域が分割型勝利領域の一部であることを証明する補題を 2 つ設ける．補題 1 は **Algorithm 2** に，補題 2 は **Algorithm 3** にそれぞれ対応している．

補題 1. *ゲーム* $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ *が与えられたとき，*$s_{sg} \in S_{sg}$ *が部分勝利領域* w_{X_Φ} *に含まれるなら* $s_{sg} \in \Gamma^-(s'_{sg})$ *を満たす* s'_{sg} *もまた* w_{X_Φ} *に含まれる．*

証明. $s_{sg} \in \Gamma^-(s'_{sg})$ より $\Gamma^-(s'_{sg}) \neq \emptyset$ である．したがっ

© 2019 Information Processing Society of Japan

て定義 4 より s'_{sg} では環境が遷移を選択でき，制御器による遷移の選択に非依存に $s_{sg} \in w_\Phi$ を選択できる．定義 5 より s_{sg} は $SG_\Phi = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X_\Phi)$ において制御器による遷移の選択に非依存に環境が勝利できる．以上より，s'_{sg} は制御器による遷移の選択に非依存に環境が勝利できるため，$s'_{sg} \in w_{X_\Phi}$ である． □

補題 2. ゲーム $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ が与えられたとき，$\Gamma^+(s_{sg}) \neq \emptyset \wedge \Gamma^-(s_{sg}) = \emptyset$ であるような $s_{sg} \in S_{sg}$ を考える．すべての $s'_{sg} \in \Gamma^+(s_{sg})$ のいずれかの部分勝利領域 w_{X_Φ} に含まれるならば，s_{sg} はすべての $X_\Psi = \bigcup_{X_\Phi \in F^{X_\Phi}} X_\Phi$，$F^{X_\Phi} \in \prod_{s'_{sg} \in \Gamma^+(s_{sg})} \{X_\Phi \in 2^X | s'_{sg} \in w_{X_\Phi}, w_{X_\Phi} \in W\}$ による部分勝利領域 w_{X_Ψ} に含まれる．

証明． すべての $s'_{sg} \in \Gamma^+(s_{sg})$ が部分勝利領域に含まれることより，s'_{sg} を含む部分勝利領域を s'_{sg} ごとに 1 つずつ要素に持つ集合族 $F^{W'} = \prod_{s'_{sg} \in \Gamma^+(s_{sg})} \{w_X \in W | s'_{sg} \in w_X\}$ を考える．定義 4 より $\Gamma^+(s_{sg}) \neq \emptyset \wedge \Gamma^-(s_{sg}) = \emptyset$ であるような $s_{sg} \in S_{sg}$ では制御器が遷移を選択できる．$s'_{sg} \in \Gamma^+(s_{sg}) | s'_{sg} \in w_{X_\Phi}$ を含むプレイには制御器の選択に非依存に環境が勝利できることより，$s''_{sg} \in \bigcup_{x \in X_\Phi} x$ が制御器の選択に非依存にプレイに含まれる．$W' \in F^{W'}$ に対して，s_{sg} を含むプレイには $s'''_{sg} \in \bigcup_{w_{X_{\Phi'}} \in W'} \{x \in X_{\Phi'}\}$ が制御器の選択に非依存に含まれる．したがってすべての W' に対して s_{sg} は $X_\Psi = \bigcup_{w_{X_\Phi} \in W'} X_\Phi$ による部分勝利領域に含まれる．以上より s_{sg} はすべての $X_\Psi = \bigcup_{X_\Phi \in F^{X_\Phi}} X_\Phi$，$F^{X_\Phi} \in \prod_{s'_{sg} \in \Gamma^+(s_{sg})} \{X_\Phi \in 2^X | s'_{sg} \in w_{X_\Phi}, w_{X_\Phi} \in W\}$ による部分勝利領域 w_{X_Ψ} に含まれる． □

上記 2 つの補題を使って次の定理を証明する．

定理 3. ゲーム $SG = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X)$ が与えられたとき，任意の $x_\phi \in X$ を除いたゲーム $SG' = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X \backslash \{x_\phi\})$ の勝利領域 w'_E は **Algorithm 1** によって得られる W_n からすべての $x_\phi \in X_\Phi$ であるような $X_\Phi \in X$ による部分勝利領域を取り除いた領域，すなわち $w'_E = \bigcup_{w \in W \backslash \{w_{X_\Phi} | x_\phi \in X_\Phi, X_\Phi \subseteq X\}} w$ である．

証明． **Algorithm 1** の W_n（n は 0 以上の整数）において，$w'_n = \bigcup_{w \in W_n \backslash \{w_{X_\Phi} | x_\phi \in X_\Phi, X_\Phi \subseteq X\}} w$ が $SG' = (S_{sg}, \Gamma^-, \Gamma^+, s_{sg0}, X \backslash \{x_\phi\})$ の勝利領域 w'_E の一部であることを証明する．$n = 0$ のとき，W_0 はすべての $x_\psi \in X$ に対して $w_{\{x_\psi\}} = x_\psi$ であるような $w_{\{x_\psi\}}$ の集合であるため，分割型勝利領域の一部である．また，定義 3 より $W_0 \backslash \{w_\phi\}$ は SG' の勝利領域 w'_E の一部である．$n \geq 1$ のとき，$n = k-1$ で W_{k-1} が分割型勝利領域の一部であり，$w'_{k-1} = \bigcup_{w \in W_{k-1} \backslash \{w_{X_\Phi} | x_\phi \in X_\Phi, X_\Phi \subseteq X\}} w$ が SG' の勝利領域 w'_E の一部であると仮定する．このとき，補題 1，補題 2 より **Algorithm 1** によって得られる W_k も分割型勝利領域の一部である．また，定理 2 より $W'_{k-1} = W_{k-1} \backslash \{w_{k-1X_\Phi} | x \notin X_\Phi, X_\Phi \subseteq X\}$ は SG' の分割型勝利領域の一部であることから，SG' の勝利領域 w'_E の一部でもある．以上より，すべての n について $w'_n = \bigcup_{w \in W_n \backslash \{w_{X_\Phi} | x_\phi \in X_\Phi, X_\Phi \subseteq X\}}$ は SG' の勝利領域 w'_E の一部である．以下，$W_n = W_{n+1}$ となる最小の n を η とする．SG' の勝利領域 w'_E のうち，$s'_{sg} \notin w'_\eta$ であるような $s'_{sg} \in S_{sg}$ が存在すると仮定する．このとき，到達可能性ゲームの勝利領域の性質 [20] より $\Gamma^-(s'_{sg}) \cap w'_\eta \neq \emptyset$ または $\Gamma^-(s'_{sg}) = \emptyset \wedge \Gamma^+(s'_{sg}) \subseteq w'_\eta$ であるが，$\Gamma^-(s'_{sg}) \cap w'_\eta \neq \emptyset$ であるならば **Algorithm 2** により，$\Gamma^-(s'_{sg}) = \emptyset \wedge \Gamma^+(s'_{sg}) \subseteq w'_\eta$ であるならば **Algorithm 3** により SG の分割型勝利領域の一部 W_η に含まれる．s'_{sg} は SG' の勝利領域に含まれるので定理 2 より $w'_\eta = W_\eta \backslash \{w_{\eta X_\Phi} | x \notin X_\Phi, X_\Phi \subseteq X\}$ に含まれることになるためこれは矛盾．以上より，SG' の勝利領域は $w'_n = \bigcup_{w \in W_n \backslash \{w_{X_{\Phi'}} | x_\phi \in X_{\Phi'}, X_\Phi \subseteq X\}} w$ である． □

定理 3 より，ゲームにおいて制御器がある部分勝利領域を避けられなくなったときに，その部分勝利領域を構築する安全性の要素のうち，いずれか 1 つを除外することで対応する部分勝利領域を取り除くことができ，他の安全性については引き続き保証可能であることが証明された．また，定理 3 を再帰的に適用することで，複数の要素を除外する場合についても残りの安全性については保証される．たとえば本来，安全性の要素単体によって構築される部分勝利領域も重複して含んでいないといけないが，これは含まれていない．しかし，ある安全性の要素 ϕ を除外する際に ϕ を含むすべての部分勝利領域は取り除かれるため，**Algorithm 1** によって特定された領域があれば十分なのである．定理 3 ではこのことについてもあわせて証明されている．

相澤 和也

2014 年早稲田大学基幹理工学部情報理工学科卒業．2016 年同大学大学院基幹理工学研究科情報理工・情報通信専攻修士課程修了．みずほ情報総研（株）を経て 2018 年より早稲田大学大学院基幹理工学研究科情報理工・情報通信専攻博士課程．現在に至る．

© 2019 Information Processing Society of Japan

鄭 顕志 （正会員）

2008年早稲田大学大学院理工学研究科情報・ネットワーク専攻博士課程修了．2006年同大学理工学部助手．2007年同大学基幹理工学部助手．2008年同大学メディアネットワークセンター助教．2010年国立情報学研究所アーキテクチャ科学研究系助教．2015年同准教授．2018年より早稲田大学理工学術院総合研究所研究院准教授/主任研究員．現在に至る．博士（工学）（早稲田大学）．自己適応ソフトウェア，ソフトウェアアーキテクチャ，モデル駆動工学の研究に従事．

本位田 真一 （正会員）

1978年早稲田大学大学院理工学研究科修士課程修了．（株）東芝を経て2000年国立情報学研究所教授，2012年同研究所副所長を兼務．2001年東京大学大学院情報理工学系研究科教授を兼任，2018年より早稲田大学理工学術院教授，現在に至る．現在，英国UCL客員教授ならびに国立情報学研究所GRACEセンター長を兼任．2005年度パリ第6大学招聘教授．2015年度リヨン第1大学招聘教授．日本学術会議連携会員．本会フェロー．

© 2019 Information Processing Society of Japan