CC-002

時間制約検証可能なアーキテクチャベース自己適応ソフトウェアプログラミングフレームワーク

Time Constrained Verifiable Architecture-Based Self-Adaptive Software Programming Framework

内藤惇 † 中川博之 ‡ 土屋達弘 †
Atsushi Naito Hiroyuki Nakagawa Tatsuhiro Tsuchiya

概 要

アーキテクチャベースの自己適応では管理対象システムをコンポーネント集合として表し，構成の組み替えにより環境変化に適応する．埋め込みコードベースの適応などと比較してより高い抽象度でシステムの変更を扱うため，ソフトウェアシステムの複雑化を緩和し，開発，運用コストを削減するとされている．時間制約を持つ場合，検証は処理フロー図を用いて行われるが，フロー図とコンポーネント図の境界の曖昧さから 2 つの混同が見られ，システム設計を複雑にしている．本研究では，コンポーネントと処理フローの関係性を定義し，コンポーネント指向のシステム開発が可能なフレームワークと，その上で動作する時間制約検証機能付き自己適応システム実装 API および設計手順を提案する．

1. はじめに

環境が変化したとき，システムは環境の変化に対応し，振る舞いを変更する必要がある．この環境の変化に対して，システム自らが振る舞いを変更し適応をする，自己適応システム構築技法の確立が期待されている [1][2]．アーキテクチャベースの自己適応は，システムをそれぞれ 1 つの責務を持つコンポーネントの集合として表現し，いずれかのコンポーネントを同じ責務を実行することができるコンポーネントと差し替え，コンポーネント構成と接続をシステム実行中に変化させることによって環境に適応するものである [3][4][5]．アーキテクチャベースの自己適応メカニズムはコードベースの適応メカニズムなどと比較して，より高い抽象度で変化した環境への適応方法の推論が可能であるほか，システムの最小単位がコンポーネントとなることからシステムの複雑化を抑えた設計が可能となり，コストを抑えたソフトウェアシステム構築方法として期待されている [6][7]．

本研究ではシステムに課される制約として時間制約を扱う．時間制約は WEB アプリケーションの応答時間のような利用者のユーザビリティに関わることがあるほか，時間制約が設けられたシステムには，組み込みシステムのような物理的な動作を伴うシステムも存在するため，安全面などの理由からも欠かせない制約となっている．

本研究ではまず，各コンポーネントにコンポーネントが持つ責務を示すタイプを設定することで，コンポーネントを差し替える際に，どのコンポーネントと差し替えるべきかをシステムが探索可能な自己適応システム実装フレームワークを提案する．またこのフレームワークの上で動作する時間制約検証機能を実装する．フロー図の各ノード（タスク）に，その処理に関わるコンポーネントタイプを紐付け，同タイプコンポーネント間の性能差とそのタイプのタスク処理における寄与率を設定することで，制約違反時，制約を満たすために，どのコンポーネントを差し替えるべきかの探索と，コンポーネントを差し替えた場合の処理時間の変化予測を可能とする．本フレームワークにより，アーキテクチャ構成図からのシステム設計および実装が体系化されるとともに，コンポーネントの差し替えとフロー図内の処理内容の変化が連携されたことによって，この 2 つの変化の混同を防止したシステムの構築が期待できる．

以降の節では，2 節で本研究で扱う自己適応システムの概要と時間制約検証ついて述べ，3 節では時間制約検証を実現する手法について詳述する．4 節では，提案するフレームワークを使用した実装パターンついて述べる．5 節ではフレームワークを用いて実際に自己適応システムを実装し，動作実験を行った結果を述べ，6 節では実験結果から，フレームワークの有効性について評価し，今後の課題について述べる．最後に 7 節で本研究のまとめを述べる．

2. 研究背景

2.1. アーキテクチャベース適応

本研究で取り扱うアーキテクチャベースの自己適応システムは図 1 のような全体像となっている．管理対象システムはコンポーネント接続図で表される．各コンポーネントには責務が課されており，接続された他のコンポーネントから情報を取得して責務を全うする．管理対象システムを管理するコンポーネントマネージャーは各コンポーネントから情報を受け取り，その情報を元に管理対象システムに課された制約や要件が満たされているかの検証を検証部に依頼する．検証により制約違反が検出されれば，コンポーネントマネージャーは制約を満たすコンポーネント構成を見つけ出し，コンポーネント構成を変更させる．

アーキテクチャベースの自己適応は，現在のコンポーネント構成のうち，いずれかのコンポーネントを同じ責務を持つコンポーネントと差し替えるという適応動作を持つ．同じ責務を持つコンポーネントとは，課程は異なるが結果として提供されるものは同種であるもののことを指している．例えば道を識別し走行するロボットにおいて，コンポーネント ColorSensor は色の種類から道を判別し，ReflectionSensor は光度の違いから道を判別するが，どちらも次にどちらに曲がるべきかという情報を提供するため，代替が可能である．同じ責務を持つコ

†大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University
‡岡山大学, Okayama University

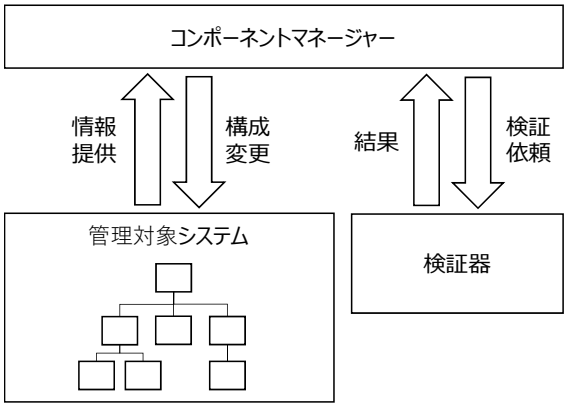

図 1: 自己適応システムの全体像

ンポーネントは，そのうちの 1 つのコンポーネントのみがシステムのコンポーネント構成に含まれ，他のコンポーネントはどのコンポーネントとも接続されず，コンポーネント構成から外された状態（未接続状態）となっている．

しかし，自己適応システムが各コンポーネントが持つ責務を認識していない場合，どのコンポーネントと差し替えるべきか判断することができない．そのため，アーキテクチャベースの自己適応システムは，コンポーネントが持つ責務を認識でき，差し替えられるコンポーネントと同じ責務を持つコンポーネントの探索が可能である必要がある．

2.2. 時間制約検証

本研究における時間制約の検証には，時間制約が設けられた機能を実現するために実行される一連の処理を記述したフロー図を用いる．フロー図のノードに実際の処理時間，あるいは予測される処理時間が当てはめられることで，時間制約の検証が行われる．本研究では機能の実現のために実行される一連の処理，つまりフロー図のノードをタスクと呼称する．

時間制約を持つ自己適応システムは，システム実行中に処理時間を計測し，その値を用いて時間制約検証を行う．制約違反が検出された場合，管理対象システムの処理内容を変更し．そのときに予測される処理時間を算出して再度検証を行い，違反が解消されることが予想される場合には実際に処理内容を変更する．

ここでフロー図のみを考慮し制約違反に対処しようと試みた場合を考える．システムはいずれかのタスクの処理時間を短縮するため，タスクの処理内容を変更することで時間制約を満たそうとする．しかしアーキテクチャベースの自己適応システムは，管理対象システムがコンポーネントで表されており，タスクの処理時間はコンポーネント構成に応じて変化する．そのため時間制約を満たすための適応動作は，タスク単位ではなくそのタスクを実現するコンポーネント単位で考えなくてはならない．

しかし，アーキテクチャベースの自己適応システムにおいてコンポーネント接続図とフロー図をそれぞれ独立して作成した場合，タスクの処理時間を短縮するためにどのコンポーネントを差し替えるべきか，また差し替えによってタスクの処理時間がどのように変化するかが不明である．コンポーネントの差し替えを適応動作とする自己適応システムが時間制約を扱えるようにするには，コンポーネントとタスクの関係を表し，コンポーネントの差し替えを行ったときのタスクの処理時間の変化を予測できるようにする必要がある．

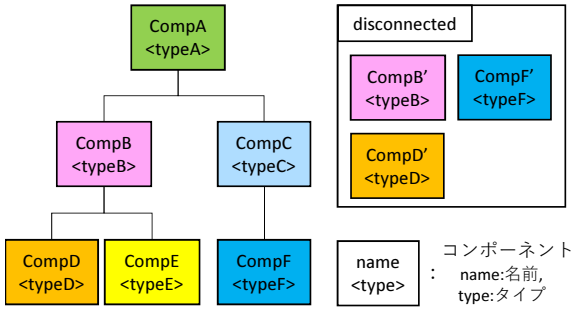

図 2: コンポーネントタイプの設定

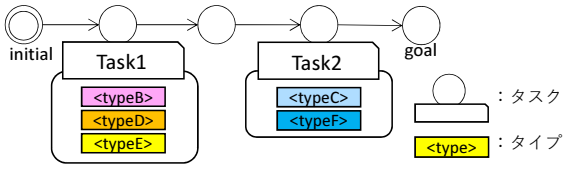

図 3: タスクとコンポーネントタイプを紐付けたフロー図

3. 提案手法

本研究ではコンポーネントの差し替えを行ったときのタスクの処理時間の予測を実現する実装フレームワークを提案する．本手法では，まず各コンポーネントを責務ごとに分類するコンポーネントタイプを定義し，コンポーネントの代替候補の探索のために用いる．またそのコンポーネントタイプとタスクを紐付けることで，タスクの処理時間の短縮のために差し替えるべきコンポーネントの探索を可能とする．そしてコンポーネントを差し替えた場合の処理時間を予測するため，同タイプの 2 つコンポーネント間の性能の違いの程度を表す「処理差倍率」と，タスクに対するコンポーネントタイプの関与の程度を表す「寄与率」を定義する．

3.1. コンポーネントタイプ

図 2 は各コンポーネントにコンポーネントタイプを付けたコンポーネント接続図である．コンポーネントタイプはコンポーネントが持つ責務を表し，同じ責務を持つコンポーネントには同じタイプが割り当てられる．コンポーネントはタイプによって分類され，他のコンポーネントと同じ責務を持つコンポーネントはタイプという観点から 1 つのグループにまとめられる．コンポーネントを差し替える場合は，同じタイプのコンポーネントの中で未接続状態であるコンポーネントから代替候補を選ぶことが可能となる．

次にこのコンポーネントタイプを使用してフロー図とコンポーネントの関係を表現する方法を示す．コンポーネントとタスクの関係性を表現するため，本手法ではタスクとそれを実現するために必要なコンポーネントタ

イプの紐付けを行う．図 3 はタスクにコンポーネントタイプを紐付けたフロー図である．フロー図側はコンポーネントの接続状況を把握する必要がないため、タスクと紐付けられるものを個々のコンポーネントではなくコンポーネントタイプにしている。またコンポーネントタイプを持つことで、フロー図作成後にコンポーネントが追加された場合にフロー図を修正する必要がなくなる。

この紐付けにより，タスクの実現に関与しているコンポーネントタイプを特定することができ，時間制約違反が発生したためにあるタスクの処理時間をより高速にしたい場合は，そのタスクに紐付けられたコンポーネントタイプから，差し替えの候補を絞ることが可能となる。

3.2. 処理差倍率と寄与率

本研究ではコンポーネントを差し替えた際のタスクの処理時間の予測のため，同タイプの 2 つコンポーネント間に対して，それら 2 つを比較したときの性能の違いの程度を表す「処理差倍率」と，あるタスクに対して，それに紐付けられているあるコンポーネントタイプがどれほどそのタスクの処理時間の形成に関与しているかを表す「寄与率」を定義する．

処理差倍率は 2 つの同タイプコンポーネントを比較したとき，課されている責務の処理時間が何倍異なるかを示す値である．例えば，コンポーネント A に対してコンポーネント A’ の処理時間が 1/2 倍になるような場合，A に対する A’ の処理差倍率は 0.5 と設定される．処理差倍率の値はシステム設計時に開発者が設定することを想定している．値を決定する方法は様々であるが，2 つのコンポーネントの処理内容を比較し，その差異から値を導き出す方法や，開発時に実際にコンポーネントを動かし処理時間を計測することで値を算出するといった方法がある．

寄与率はタスクの処理時間の形成に対するコンポーネントタイプの貢献度を表す値である．タスクの処理に複数のコンポーネントが関与している場合，そのコンポーネントが実行する処理以外にも実行される処理があり，それらの処理速度には変化がない．このことを考慮せず処理差倍率のみを使用した場合，例えばそのうちの 1 つのコンポーネントの処理時間が 1/2 倍になるコンポーネントの差し替えを行った場合，タスクの処理時間は 1/2 倍となる．しかし実際にはタスク全体の処理時間は 1/2 倍とはならず，それよりも大きい値となる．このようにコンポーネントの差し替えがタスクの処理時間に与える影響力がどれくらいあるかを示すため，寄与率をコンポーネントタイプとタスクの間に設定する．寄与率は 0 から 1.0 の間で設定される．寄与率に関してもシステム設計時に開発者が設定することを想定しており，タスクを実現するコンポーネント間の関係性などから割り出す必要がある．

本研究では，この処理差倍率と寄与率によってコンポーネントが差し替えられたときのタスクの処理時間の変化を予測する．予測されるタスクの処理時間の計算式は以下のようになる．c（contribution）は寄与率，m（magnification）は処理差倍率，previousTime は入れ替え前のタスクの処理時間，newTime は入れ替え後に予測されるタスクの処理時間である．

$$newTime = previousTime - previousTime * c * (1 - m) \quad (1)$$

4. 自己適応システムの実装パターン

本研究では，3 節で説明した手法を取り入れた実装フレームワークを提案する．提案するフレームワークは，アーキテクチャベースの自己適応システム実装フレームワークと，その上で動作する時間制約検証機能実装 API を合わせた構成になっている．本節では，まずコンポーネント接続図の作成とフレームワークを用いたコンポーネント，コンポーネントマネージャーの実装について説明し，フロー図の作成と時間制約の設定方法，そしてタスクとコンポーネントタイプの紐付けや処理時間の予測を行う API の使用方法について説明する．

4.1. コンポーネント指向フレームワーク

コンポーネント接続図をもとに管理対象システムを実装でき，コンポーネントを差し替えることを適応動作とした自己適応システム実装フレームワークを提案する．このフレームワークの実装は既存のフレームワーク [8] を参考にしており，以下では参考フレームワークと称する．本研究ではこのフレームワークに改変を加え，コンポーネントを差し替えることを適応動作とした自己適応システムが実装できる新たなフレームワークとする．図 4 は提案するフレームワークのクラス関係図と主なメソッドである．このフレームワークは java 言語によって構築されており，コンポーネント，コンポーネント同士を接続するポート，コンポーネントの状態を表す Mode，全コンポーネントを管理する ComponentManager から成る．コンポーネントには Java の Thread クラスが継承されているため，コンポーネントの並列動作が実現される．

本研究では参考フレームワークでも使用されている拡張 Darwin モデル [9] を元に記述したコンポーネント接続図を使用する．図 5 は運搬システムを拡張 Darwin モデルを用いて記述したコンポーネント接続図である．拡張 Darwin モデルでは，コンポーネントは 0 個または 1 個の提供ポートと 0 個以上の要求ポートを持っている．コンポーネント間はこの提供ポートと要求ポートにより接続されており，提供ポートを持つコンポーネントはそのコンポーネントが持つ 1 つの責務の出力結果を，提供ポートを通して，接続された要求ポートを持つコンポーネントに送信する．例えば図 5 の運搬システムにおける，荷物の保管場所まで移動するコンポーネント ApproachObject は，SuperSonicSensor，GyroSensor，ColorSensor の 3 つのセンサーと，タイヤを動かす RotateWheels という 4 つのコンポーネントと接続されており，これらのコンポーネントを用いて責務を実現する．

続いてこのフレームワークを用いたコンポーネントとコンポーネントマネージャーの記述例を示す．

図 6 はコンポーネントの記述例である．各コンポーネントの実装には Component クラスを継承する．perform メソッド内にコンポーネントの処理内容を記述し，perform メソッドはシステムが活性状態のときに繰り返し呼び出される．コンポーネントには名前，タイプ，自身を管理するコンポーネントマネージャー，提供ポートを設定す

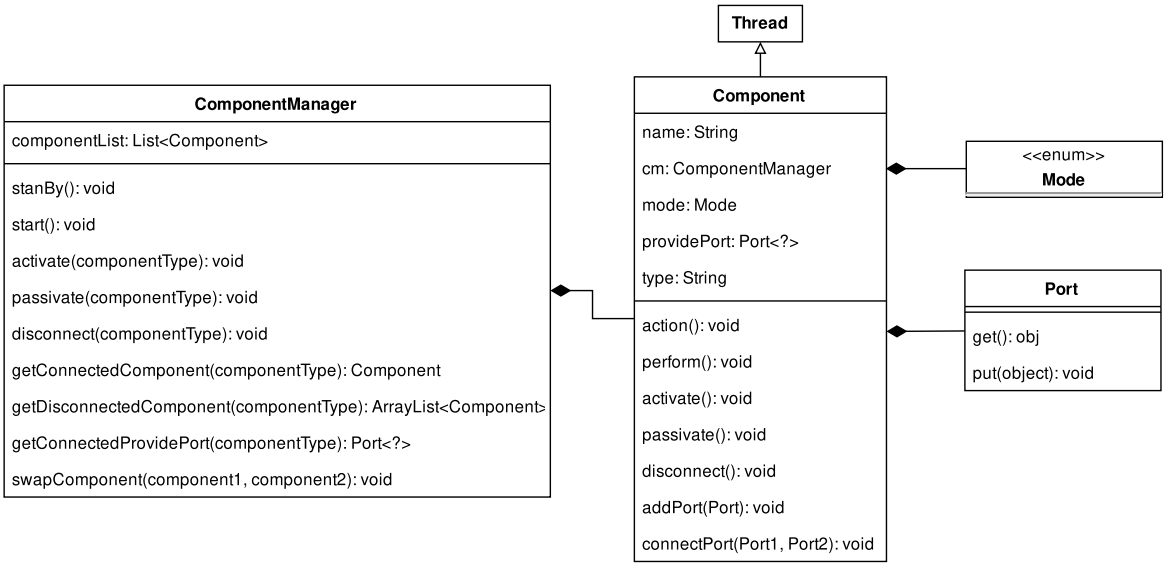

図 4: フレームワークにおけるクラス関係図と主要メソッド

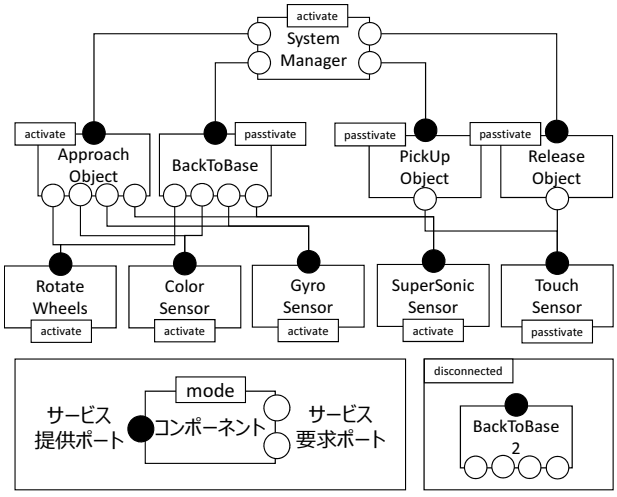

図 5: 運搬システムのコンポーネント接続図

る．またコンポーネントの責務を実現するために必要なコンポーネントタイプの要求ポートを作成する．各コンポーネントが持つ mode 変数には，参考フレームワークから用意されている待機状態（waiting），活性状態（active），達成状態（achieved）に加えて新たに未接続状態（disconnected）を加え，コンポーネントはこの 4 つの状態を取る．待機状態は使用されていない状態を示し，活性状態はコンポーネントが処理中であることを示す．達成状態はコンポーネントが課された責務を達成した状態であることを示し，未接続状態は現在のコンポーネント構成に含まれていない状態，即ち代替コンポーネントであることを示す．

activate，passivate，disconnect メソッドはそれぞれコンポーネントを活性状態，待機状態，未接続状態に遷移させるメソッドであり，遷移時に必要な動作がある場

```
public class ComponentA extends Component {

    public ComponentA(ComponentManager cm, String name, String
type) {
        super(cm, name, type);
        this.cm = cm;
        super.providePort = new Port<Mode>("ComponentAPort",
PROVIDED, this);
        super.addPort(providePort);
    }

    public void activate() {
        super.connectPort(typeBPort,
cm.getConnectedProvidePort("componenttypeB"));
        cm.activate("componenttypeB");
        super.activate();
    }

    public final void action() {
        if (mode == Mode.ACTIVE) {
            perform();
        }
    }

    private void perform() {
        Mode result = typeBPort.get();
        /** process component does **/
        if(true) {
            this.mode = Mode.ACHIEVED;
        }
        ((Port<Mode>) providePort).put(mode);
    }
}
```

図 6: コンポーネント記述例

合はオーバーライドし，記述を加えた後コンポーネントクラスで定義されたメソッドを呼び出す．コンポーネントが活性状態になるときは，ポートを接続し，責務を実現するために必要なコンポ－ネントを活性化させる．参考フレームワークではコンポーネントを活性化させるときは要求側のコンポーネントが直接 activate メソッドを呼び出すが，本手法ではコンポーネントマネージャーの activate メソッドを用いて，タイプ名を入力し現在のコンポーネント構成に含まれているものを活性化させる．これにより，各コンポーネントが現在の接続状況を把握する必要をなくし，安全な設計が可能となる．

```
public class ExampleManager extends ComponentManager {

    public ExampleManager() {
        /** Setup **/
        super.setup();
        compA = new ComponentA(this, “ComponentA”,
“componenttypeA”);
        standBy(compA);
        /** Start **/
        this.activate(" componenttypeA");
        super.start();
    }

    public void controllSystem() {
        if (compA.mode == Mode.ACHIEVED) {
        /** Verify requirements **/
        this.swapComponent(connectedComp, disconnectedComp);
        }
    }
}
```

図 7: コンポーネントマネージャー記述例

図 7 はコンポーネントマネージャーの記述例である．コンポーネントマネージャーの実装には ComponentManager クラスを継承する．コンストラクタ内では管理対象システムを表すコンポーネントを生成し，standBy メソッドによりコンポーネントを List に加えながら待機状態にする．コンポーネントを生成する際に，そのコンポーネントが分類されるコンポーネントタイプを引数として与えることでタイプの設定を行う．また初期状態のコンポーネント構成に含まれるコンポーネントを活性状態に，含まれないコンポーネントを未接続状態に設定し，管理対象システムを稼働させる．その後要件や制約の充足検証を実施し，必要であれば要件を充足できるコンポーネント構成を探索し，swapComponent メソッドを用いて差し替えを行う．swapComponent メソッドは第一引数で与えられた現在接続されているコンポーネントを未接続状態に，第二引数で与えられた未接続状態の代替コンポーネントを待機状態に遷移させる．

4.2. 時間制約検証機能 API

本研究では，フロー図の作成と時間制約の設定，検証にモデル検査ツールである UPPAAL[10] を使用する．UPPAAL はタイミング付きオートマトンを用いたモデル検査ツールであり，システムの動作を形式的に検証することができる．搭載されているグラフィカルエディタを用いて作成された状態遷移モデルにクロック変数を追加することで，状態遷移にかかる時間を記述でき，システムの処理時間をモデリングすることができる．また検証式を記述し検証器に与えることでモデル検査を行うことができる．UPPAAL では状態遷移図をテンプレート，状態をロケーションと呼称しているため，本研究でもこの呼称を用いる．本手法では自己適応システムは UPPAAL で作成したフロー図（xml ファイル）をセットアップ時に読み込み，時間制約の検証時には処理時間を埋め込んだフロー図を書き出し，UPPAAL の検証器を使って検証を行い，検証結果を得る．

図 8 は本手法で扱うフロー図の略図である．フロー図は全体を表すルートテンプレート（図中緑枠）と，ひとつのタスクを表すタスクテンプレート（図中赤枠）から構成されている．ルートテンプレートの各ロケーションはタスクを表しており，機能の実現に必要な一連のタスクが羅列される．タスクロケーションは int 型の time

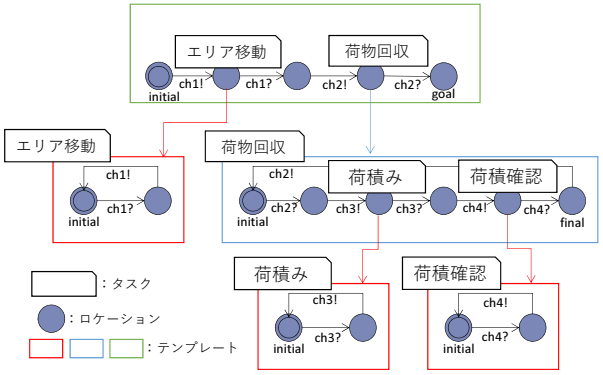

図 8: UPPAAL で作成したフロー図

というパラメータを持ち，書き出し時には測定された処理時間が代入される．また clk というクロック変数を持ち，ロケーションの不変式 $clk <= time$ と，遷移条件 $clk >= time$ の組合わせにより time クロック後に遷移が発生する．タスクは処理時間を測定できる箇所に対して設定されるため，タスクの粒度が不揃いになることがある．またタスク数が多くなるとフロー図が長くなり可読性が低下する．そこでフロー図の可読性を高めるため，複数のタスクを 1 つにまとめ，1 つのテンプレートとして作成できるようにしている（図中青枠）．このテンプレートにも同じように初期状態のロケーションにタスク名を付けるが，このタスクは実際のタスクではないため注意が必要である．

本手法では以下の検証パターンを用いて時間制約の設定を行う．

- E<>p: あるパスにおいて，最終的に状態 p が到達可能である．

これを用いて，1 クロック 1 秒として機能 ApproachAndPickUp が 30 秒以内に完了するという時間制約は以下のような記述となる．

- E<>root.EndPickUp and root.clk <= 30

先述のフレームワーク上にこのフロ－図を読み込み操作する機能を追加することで，コンポーネントとフロー図を扱え，時間制約検証が可能な実装フレームワークとなる．読み込んだフロー図情報を格納している Automaton クラスのインスタンスと，ComponentManager クラスのインスタンスを渡された AtCompController クラスが持つメソッドを利用することで，コンポーネントとフロー図を結びつけた一元管理が可能となる．図 9 は実装のために利用可能な主なメソッドである．タスクの処理時間の更新は timeUpdate メソッドを用いて行われ，コンポーネントマネージャーはこれらの API を用いて処理差倍率と寄与率の設定，時間制約の充足検証，制約違反を解消するコンポーネント構成の探索を行うことができるため，この API を追加したフレームワークによって，時間制約検証が可能なアーキテクチャベースの自己適応システムが実装が可能となる．

AtCompController
at: Automaton
cm: ComponentManager
addMagnification(Component comp1, Component comp2, float magnification): void
attachComponentTypeToTask(String task, String type, float contributionRate): void
getComponentTypeList(String task): ArrayList<String>
timeUpdate(String task, int newTime): void
verify(): void
isSatisfiedQ(int num): boolean
predictSwapComponent(Component comp1, Component comp2): void

図 9: AtCompController クラスの主要メソッド

5. 実験

提案するフレームワーク有用性を検証するため，提案手法に従った運搬システムの実装実験を行った．本章では実装手順とシステムの振る舞いを示し，提案手法の有効性を評価する．

5.1. システム実装

本実験では運搬システムを構築した．このシステムはロボットが基地から指定されたエリアに移動して荷物を乗せ，基地にその荷物を届けるという一連の動作を繰り返すシステムである．基地からエリアへの道のりとその帰り道は異なる道であるが距離は等しい．図 10 は実験で使用したロボットである．ロボットはカラーセンサー，ジャイロセンサー，超音波センサー，タッチセンサーと左右のモータとタイヤから構成されている．基地から指定されたエリアまでは道が敷いてあり，ロボットはカラーセンサーを用いて道を識別し左右のモーターを操作して移動する．超音波センサーは前方の物体との距離を測定し，前方に障害物があればそれを避けて移動する．ジャイロセンサーを用いて指定されたエリアと基地への到着判定を行う．そしてタッチセンサーを用いて荷物が積み込まれたか，または下ろされたかを判定する．このシステムのシステム構成図および初期状態のコンポーネント接続図には図 5 の接続図を用いた．コンポーネントタイプ backtobase には代替コンポーネントが用意されている．コンポーネント BackToBase2 は BackToBase の 2 倍の速度で移動するよう RotateWheels に指令を出す．この 2 つのコンポーネント間では移動速度が 2 倍になる

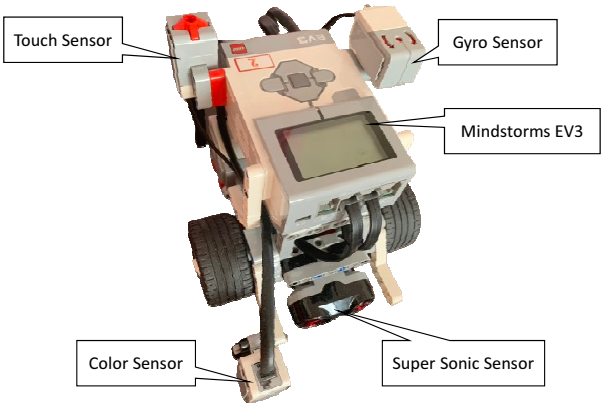

図 10: 実験で使用したロボット

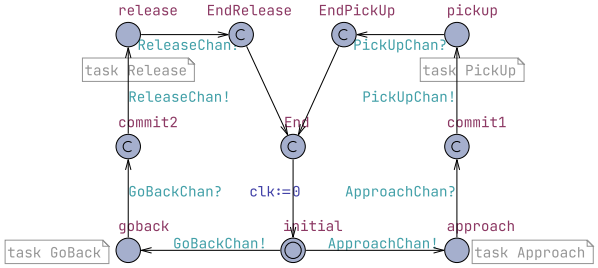

図 11: 運搬システムのフロー図

Task	Component Type	Contribution Rate
Approach	approachobject	1.0
	colorsensor	0.0
	gyrosensor	0.0
	supersonicsensor	0.0
	rotatewheels	0.0
GoBack	backtobase	1.0
	colorsensor	0.0
	gyrosensor	0.0
	supersonicsensor	0.0
	rotatewheels	0.0

表 1: タスクに対するコンポーネントタイプの寄与率

ため，差し替えた際には処理時間を 1/2 倍にすると予想される．そのためコンポーネント BackToBase に対する BackToBase2 の処理差倍率は 0.5 とした．

図 11 はこのシステムのフロー図である．このシステムでは基地から指定されたエリアに移動して荷物を乗せるまでにかかる時間と，基地に戻りその荷物を下ろすまでにかかる時間に対して，それぞれ 30 秒以内に完了するという時間制約を設定した．表 1 はタスク Approach と GoBack に紐付けられたコンポーネントタイプと寄与率である．運搬システムは実際の動作を伴い，また各コンポーネントは並列に動作するため，走行速度の目的地までの移動時間に対する寄与率は 1.0 とし，他のタイプの寄与率は 0 と設定した．

このロボットの実装には LEGO MindStorms EV3 を使用する．MindStorm に jar ファイルを転送しシステムを実行するが，MindStorm 上では UPPAAL の検証器を動作させることができないため，検証は PC で行い，検証結果をロボットへ送信するシステムとした．

実験では，エリアから基地の道中に障害物を検知し，これを避けるように移動したために基地への到着が遅れてしまった場合のシステムの振る舞いを観測した．

5.2. 実験結果

図 12 は運搬システムを実行したときの実行ログである．ログからは，荷物の収集を完了して（44 ～ 46 行目）基地に帰還中（47 ～ 58 行目），障害物を検知して退避行動を取っていることが分かる（53 行目）．その後基地に到着して荷物を下ろし（68 ～ 71 行目），PC 側に依頼した検証の結果，基地に戻りその荷物を下ろすまでにかかる時間に対する時間制約への違反が検出されていることが確認できる（72 ～ 74 行目）．これを受けシステムはタスク GoBack の処理時間を早くするため，未接続

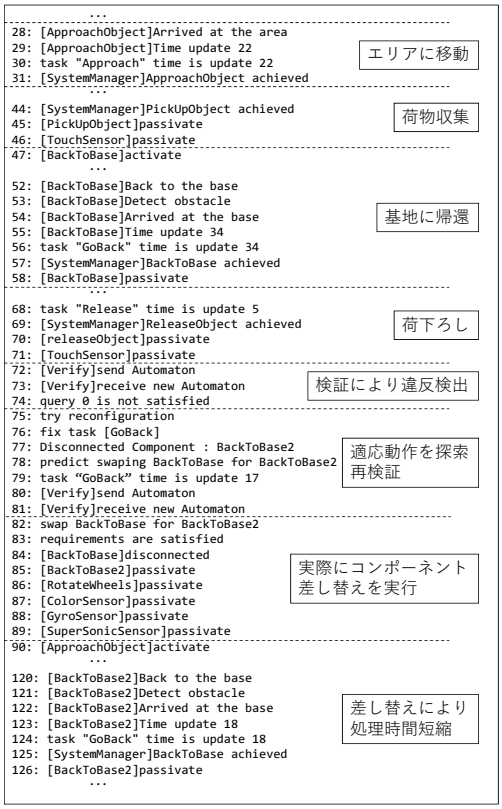

図 12: 運搬システムの実行ログ

状態であった BackToBase2 を BackToBase と差し替えるという適応候補を出して再度検証にかけ（75 ～ 81 行目），その結果制約違反が解消される予想ができたため，実際にコンポーネントを差し替えていることも確認できる（82 ～ 89 行目）．差し替え後のタスク GoBack の処理時間は 18 秒になっており，差し替え前よりも処理時間を短くできていることが分かる（120 ～ 126 行目）．またこの処理時間は，予測される処理時間との差が 1 秒となっており，予測が正しく行われていることも確認できる．このように，運搬システムは荷物の収集と荷下ろしを繰り返し，制約違反があればコンポーネントを差し替えることで解消を試みることが確認できた．

```
 1: private void perform() {
 2:     switch (flag) {
         ...
46:     case 2:
47:         if (f) {
48:             f = false;
49:             cm.activate("backtobase");
50:         }
51:         Mode backToBaseMode = null;
52:         if (!backToBasePort.isEmpty()) {
53:             backToBaseMode = backToBasePort.get();
54:         }
55:         if (backToBaseMode == Mode.ACHIEVED) {
56:             System.out.println("[SystemManager]BackToBase
achieved");
57:             cm.passivate("backtobase");
58:             flag = 3;
59:             f = true;
60:         }
61:         break;
           ...
```

図 13: SystemManager クラスの perform メソッド

```
 1: public void controllSystem() {
        ...
11:     System.out.println("fix task [GoBack]");
12:     for (String type : atc.getComponentTypeList("GoBack")) {
13:         boolean finish = false;
14:         Component connectedComp = this.getConnectedComponent(type);
15:         for (Component disconnectedComp :
this.getDisconnectedComponent(type)) {
16:             atc.predictSwapComponent(connectedComp, disconnectedComp);
17:             verify.sendAndReceive();
18:             if (atc.isSatisfiedQ(0)) {
19:                 this.swapComponent(connectedComp, disconnectedComp);
20:                 finish = true;
21:                 break;
22:             } else {
23:                 atc.predictSwapComponent(disconnectedComp, connectedComp);
24:             }
25:         }
26:         if (finish)
27:             break;
28:         }
29:     }
```

図 14: CarryRobotManager クラスの適応動作記述

図 13 は SystemManager コンポーネントの perform メソッドの一部を示したものである．PickUpObject コンポーネントが責務を達成した後，SystemManager は基地に帰還するために，コンポーネントタイプが backtobase であるコンポーネントを活性状態にする．backtobase タイプのコンポーネントには BackToBase コンポーネントと BackToBase2 コンポーネントの 2 つがあり，SystemManager はこのうち接続されているコンポーネントを活性化させる必要があるが，SystemManager はコンポーネントマネージャーの activate メソッドを用いることで，タイプ名のみから接続されているコンポーネントの活性化が可能となっている．図の実行ログからも差し替え後に BackToBase2 を活性化していることが確認でき，コンポーネントの差し替えを意識した実装が可能であることが確認できる．

また図 14 は CarryRobotManager クラスの一部を示したものである．タスク GoBack と紐付けられているコンポーネントタイプにおいて，現在接続状態にあるコンポーネントを取得し，未接続状態のコンポーネントを 1 つずつ仮差し替えを行っている．このプロセスにより，接続中の BackToBase コンポーネントと未接続状態の BackToBase2 コンポーネントの仮差し替えが試行されたことが確認できる．また，predictSwapComponet メソッドを用いて処理差倍率と寄与率から予測される処理時間を計算し，再度検証を依頼した結果，isSatisfied メソッドから違反が解消されることが予想されることを知り，swapComponent メソッドを用いて実際にコンポーネントを差し替えるという記述により，実行ログに記されている処理を実現していることが確認できる．

これらのことから，ComponentManager クラスと AtCompController クラスに用意されたメソッドを用いてコンポーネントの仮差し替えによる処理時間の予測と時間制約検証，実際のコンポーネントの差し替えを行うシステムの実装が可能であることが分かった．

6. 考察

実験では，コンポーネントの実装パターンを参考にして，システム設計時に作成したコンポーネント接続図から各コンポーネントの実装と接続を記述することができている．またコンポーネントマネージャーの実装パターンを参考にし，コンポーネントで表された管理対象シ

ステムを管理し，提供されているメソッドを用いてコンポーネントの差し替えが実現されていることから，提案するフレームワークは，コンポーネントの差し替えを適応動作とする自己適応システムの実装に有用なサポートをしているといえる．

また実験では，各タスクの処理時間を計測し，システム実行中に時間制約の検証ができている．そして提供されているメソッドを用いることで，コンポーネントを差し替えた場合のタスクの処理時間の予測ができており，「コンポーネント構成を変更することでタスクの処理時間を短縮する」という認識を持った適応動作記述ができていることから，提案するフレームワークは，システム実行中の時間制約の検証と，コンポーネント差し替えを主軸とした時間制約違反の解消を提供するものであるといえる．

一方で，提案するフレームワークでは，コンポーネント接続図と UPPAAL で作成するフロー図のテンプレートは示しているが，管理システムのコンポーネントへの分割手法や，タスクの分解手法は提示しておらず，フレームワークの使用者に委ねられている設計箇所である．これらのより詳細な手法を検討する必要がある．

また，処理差倍率と寄与率の設定についても，本研究ではその枠組みだけを提示し，具体的な値の推定手法に関する議論はしていない．処理差倍率と寄与率の推定手法とその推定値の妥当性に関する検証手法も検討する必要がある．

7. まとめ

本研究では，コンポーネントの差し替えを適応動作とする自己適応システムにおいて，時間制約検証が可能な実装フレームワークを提案した．コンポーネントの差し替え候補を示すため，各コンポーネントにコンポーネントタイプを追加し，またコンポーネントタイプとタスクを紐付けることで，タスクの処理時間の短縮のために差し替えるべきコンポーネントの探索を可能とした．そしてコンポーネントを差し替えた場合の予測処理時間を算出するため，同タイプの 2 つコンポーネント間の性能の違いの程度を表す「処理差倍率」と，タスクに対するコンポーネントタイプの関与の程度を表す「寄与率」を導入した．実験ではこのフレームワークを用いて運搬システムを実装し，このフレームワークがコンポーネントの差し替えを適応動作とした自己適応システムの実装に有用であることと，コンポーネントの差し替えによる処理時間の変化の予測が正しく行えることが確認できた．

今後は，コンポーネント接続図とフロー図のより詳細な設計方法と，処理差倍率と寄与率の体系的な算出方法を検討し，現在使用者に委ねられている設計箇所を自動化したフレームワークの実現を目指したい．

謝辞

本研究は，JSPS 科研費 JP23K28060 の助成を受けた．

参考文献

[1] D. Lemos, et al., “Software engineering for self-adaptive systems: A second research roadmap,” Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited PapersSpringer, pp.1–32 2013.

[2] D. Weyns, “Software engineering of self-adaptive systems,” Handbook of software engineering, pp.399–443, 2019.

[3] J. Kramer and J. Magee, “Self-managed systems: an architectural challenge,” Future of Software Engineering (FOSE’07)IEEE, pp.259–268 2007.

[4] H. Nakagawa, A. Ohsuga, and S. Honiden, “Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops,” 2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems, pp.59–68, 2012.

[5] O. Gheibi, D. Weyns, and F. Quin, “Applying machine learning in self-adaptive systems: A systematic literature review,” ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol.15, no.3, pp.1–37, 2021.

[6] J. Cámara, P. Correia, R. De Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura, “Evolving an adaptive industrial software system to use architecture-based self-adaptation,” 2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)IEEE, pp.13–22 2013.

[7] J. Cámara, P. Correia, R. deLemos, and M. Vieira, “Empirical resilience evaluation of an architecture-based self-adaptive software system,” Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures, pp.63–72, 2014.

[8] H. Tsuda, H. Nakagawa, and T. Tsuchiya, “Towards self-adaptation on real-world hardware: A preliminary lightweight programming framework,” 2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, pp.176–177, IEEE, 2015.

[9] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel, “Modes for software architectures,” Proceedings of the Third European Conference on Software Architecture, pp.113–126, EWSA’06, Springer-Verlag, Berlin, Heidelberg, 2006.

[10] K.G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a nutshell,” International journal on software tools for technology transfer, vol.1, pp.134–152, 1997.