実世界指向自己適応フレームワークにおける動的検証メカニズム関する調査研究（継続）

代表研究者　　中　川　博　之　　大阪大学　大学院情報科学研究科　准教授

1 はじめに

近年，CPS（Cyber-Physical System）やIoT（Internet of Things）に代表されるように，ITと物理的機器を接続する高度システムの実現が求められている．CPSやIoTに基づいた高度システムにおいては，膨大な通信・ネットワーク機器，センサデバイスなどの多様な実世界デバイスの存在を前提としており，これらの機器の人手による保守はますます難しくなっている．また，近年のソフトウェアシステムでは，設計時に実行時の環境を完全に想定することが難しい．これらの背景から，システム実行時に自発的に振る舞いを変更することで環境変化に対する適応機能を実現する自己適応システムが，次世代ソフトウェアシステムのあるべき設計スタイルとして注目を集めている．

自己適応システムに関する研究はモデル化や適応のメカニズム[1]をはじめ，実装のためのフレームワーク[2][3][4]など様々な研究が進められている．その一方で，組み込みシステムのようにハードウェアを構成要素として持つシステムにおいては，災害時などの未知の環境下での救助ロボットといった自己適応システムの活躍が期待されている[5]ものの，このような自己適応システムを実装するのためのフレームワークは未だ確立されていない．実装例もサーバーシステム[6][7]やWEBアプリケーション[8]などの高性能サーバ環境分野と比べて非常に少ない．

本調査研究ではハードウェアを構成要素に持つ自己適応システムに焦点を当て，実世界において効果的に利用可能な自己適応フレームワークについて検討する．また，自己適応システムには振る舞いの妥当性を検査するための動的検証メカニズムが必要になるが，特に安全面などの理由から物理的な動作を伴う組み込みシステムでは欠かせない時間制約に着目する．実行時に起こるであろう時間制約違反に対して適応動作を駆動させ，振る舞いを変更させることにより，制約が満たされるような状態に遷移させることが期待される．このためにはシステム実行中における動的な時間制約の検証が必要である．そこで，本調査研究では時間制約つき状態遷移図を用いた検査が可能なUPPAAL[9]を動的に利用する．UPPAALはすでにリアルタイムシステムのモデル検査[10][11]などに利用されているが，システム設計時の設計者とのインタラクションによる検証を前提としたツールであり，動的な検証を目的としたものではない．UPPAALを動的に扱うことができるフレームワークを実現することで，時間制約の動的検証や制約違反への適応動作が可能な自己適応システムの実現が可能となる．本調査研究では，組み込みシステムに適した自己適応システム実装フレームワークを実装し，実際のハードウェアを用いた簡便な自己適応システムを実装した．本報告書ではその実験結果についても述べる．

2 動的検証メカニズムの概要

本調査研究では，検証対象として，組み込みのシステムを使う上で欠かすことのできない要素である時間制約に特に着目した．時間制約を扱うことができるモデル検査ツールUPPAALを利用することにより，時間制約に関する動的検証機能を備えたフレームワークの実現を目指す．フレームワークの実装にはJava言語を用い，時間制約の検証にモデル検査ツールUPPAALを利用する．本フレームワークを用いて実際にハードウェアを構成要素とする自己適応システムを実装し，時間制約を考慮可能な自己適応システムが実現できることを示す．

2-1 UPPAAL

UPPAAL は時間制約を扱うことのできるモデル検査ツールであり，時間制約を記述できるよう拡張された状態遷移図をモデルとして扱うものである．UPPAAL にはクロック型の変数が提供されており，全てのクロック変数は同じタイミングでインクリメントされ，個別にリセットすることができる．この変数の存在により，状態遷移のタイミング制限や時間制約の条件を記述することが可能になっている．UPPAAL は Java で記述されたグラフィカルエディタと C++によって記述された検証器を組み合わせた構造となっており，グラフィカルエディタを用いて作成した時間状態遷移モデルと時相論理を用いた検証式を XML 形式で検証器へ入力として与えることでモデル検査を実行している．また，UPPAAL では各状態遷移図をテンプレート，状態遷移図を構成する状態をロケーションと称している．

2-2 フレームワーク構成

図 1 にフレームワークの概要を示す．自己適応システムの分野ではコンポーネントベースのシステムを扱うことが多く[12] [13]，本研究でもコンポーネントベースのシステムを対象システムとして想定した．このフレームワークでは時間制約を満たしているかどうかの検証に,図中赤枠の UPPAAL が持つ検証器を用いる．状態遷移モデルのデータをシステム動作中にシステムの実際の処理時間を用いて更新し，UPPAAL の中間ファイル形式で出力，検証することで時間制約の動的な検証を可能にしている．また，フレームワークを動作させるためには，システムの初期状態を表す状態遷移モデルの情報をフレームワークへ初期入力として与える必要がある．この初期入力は UPPAAL グラフィックエディタを用いて状態遷移図群を作成し，生成された中間ファイルをそのまま用いることができる．

システム実行中に行った検証にて時間制約の違反が検出された場合，フレームワーク内に保持している状態遷移モデルに対して変更を加え，新しい状態遷移モデルの候補を生成する．生成した候補を中間ファイルとして出力し，引き続き時間制約違反が検出されたならば，候補の生成・検証を繰り返す．時間制約を満たすような候補が見つかれば，そのモデルに従って実際のシステムに変更を加えることで時間制約違反に適応する．システムに変更を加えるにあたり，このフレームワークはコンポーネントベースのシステムを扱うため，システムの変更とはシステムを構成するコンポーネントの組み合わせを変えるということを意味する．つまりは，コンポーネントの有効化/無効化や入れ替えといった操作がシステムの変更に相当する．これらの操作をシステムのモデルに対して行うためには,モデルがコンポーネント単位で扱える必要がある．しかし，検証に利用する UPPAAL の検証器では状態遷移モデルを使用しているため，状態遷移モデルであって，さらにコンポーネント単位で扱うことのできるようモデルの作成を工夫する必要がある．

3 モデルの作成

本フレームワークの適用にあたり，システムをコンポーネント単位で扱うことができ，かつ，UPPAAL で検証可能なモデルの作成方法を説明する．説明にあたってドローンの制御システムを取り上げる．離陸，着陸，移動の 3 つの機能を持った小型無人航空機制御システムを構築するという仮定で，そのモデル作成の流れを紹介する．モデルの作成には UPPAAL の GUI エディタを利用する．

3-1 タスクの分解

まず始めに，コンポーネント単位で扱うことができるように，機能を各処理単位で分解していく作業を行う．これを本研究ではタスク分解と称する．図 2 の上部は小型無人航空機制御システムに持たせる 3 つの機能のループを用意した状態遷移図である．どの機能についても同様の作業を行うため，ここでは離陸機能を例にとって以降の流れを紹介する．離陸機能とは文字通り地上で静止している小型無人航空機のプロペラを回転させ，浮上，飛行状態へと移行させる機能である．この機能を実現させるのに必要な一連の処理をそれぞれ個別のタスクとして定義する．そして，図 2 下部のように，各状態に実行されるタスクを割り当てることによって，機能を表すループを状態遷移図で表現する．また，用いるタスクは厳密なものとせず，ある程度抽象さを持たせたものとする．この例では，現在静止状態であるかの確認，機体の上昇，飛行状態へと変更の 3 つのタスクで表現した．

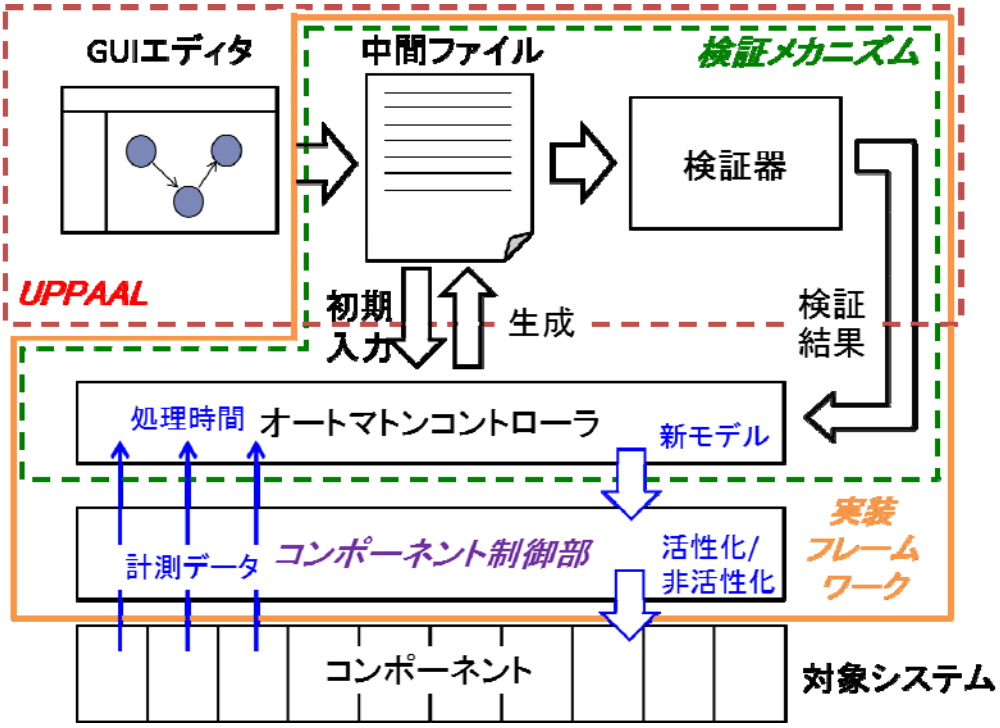

図 1：動的検証メカニズムを包含するフレームワーク構成

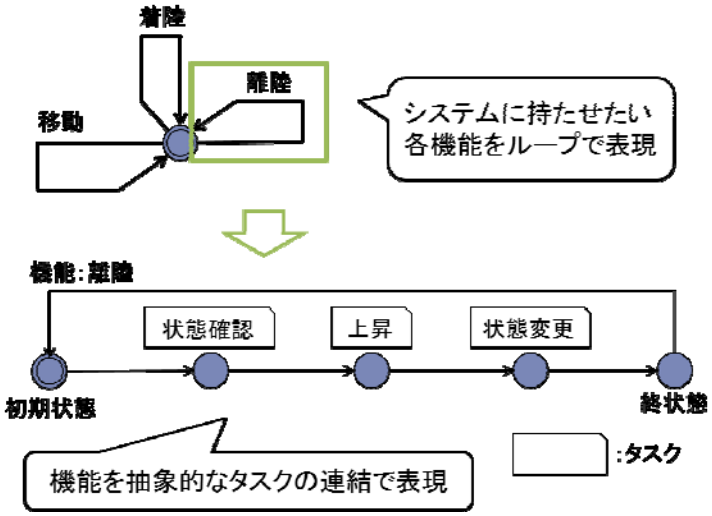

図 2：タスク表現

次に，先ほど定義した抽象的なタスクをより具体的なタスクへと分解する．図 3 は離陸機能に対してタスク分解を行った様子を示したものである．タスク分解では，あるロケーションが持つタスクを，より具体的なタスクの連結に分解することであり，分解後のタスク連結を新たな状態遷移図として作成する．つまり，UPPAAL 上では分解後のタスク連結を新たなテンプレートとして作成する．作成した新たな状態遷移図内のタスクがまだ十分に具体化されていない場合，さらにその状態遷移図についてタスク分解を行う．このようにしてタスク分解を繰り返し，徐々に具体的なタスクへと分解していく．そして，最終的に状態遷移図内の全てのタスクがシステムの持つ 1 つのコンポーネントで処理できる内容となるまで分解を行う．本調査研究では，そのように 1 つのコンポーネントだけで全てのタスクを実行できる状態遷移図をモジュール，2 つ以上

のコンポーネントによる処理が必要となるものをストラテジーと称している．よって，タスク分解によって生成される状態遷移図がストラテジーである場合，さらにタスク分解を行う必要がある．図 3 では，一連のタスク分解作業によって 1 つのストラテジーと 4 つのモジュールへと分解されている．この方法では，分解後の末端状態遷移図が必ずモジュールとなり，モジュールが 1 つのコンポーネントによって処理が完結しているおかげで，状態遷移図とコンポーネントを 1 対 1 に結びつけて捉えることができる．つまり，この例の場合，離陸機能は 4 つのコンポーネントによる処理で実現されていると見ることができる．これによって，モジュールと称される状態遷移図単位の操作を，実際のコンポーネント単位の操作として扱うことができる．

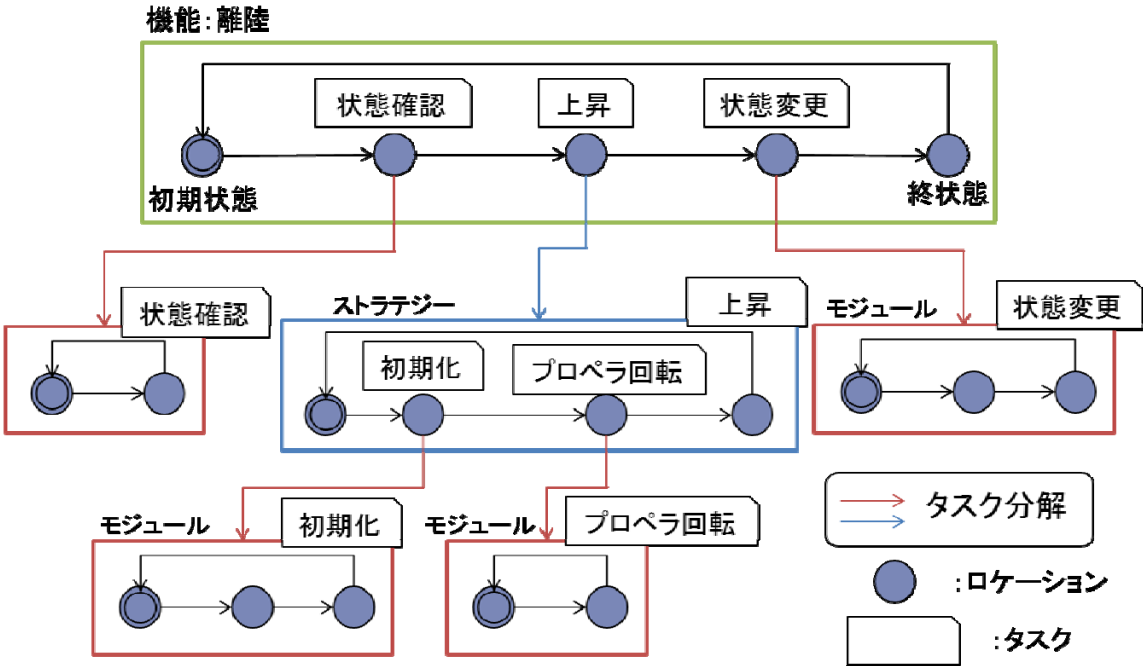

図 3：タスク分解

3-2 状態遷移図間の同期

前節では各機能におけるタスクをコンポーネントレベルまで分解する手法を示した．これにより，1 つの機能は複数の状態遷移図によって表される．しかし，このままでは各状態遷移図内での処理時間しか検証できないため，機能単位での時間制約などの検証が行えない．そこでタスク分解元，分解先の状態遷移図間で同期を取る事によって，複数の状態遷移図に跨って処理時間を検証できるように工夫する必要がある．本フレームワークでは UPPAAL の同期チャネル変数を用いてタスク分解元のロケーション前後の遷移と，分解先の状態遷移図の終始で同期を取ることによって実現している．図 4 に状態遷移図の同期の例を示す．図 4 左部は，図 3 における上昇ストラテジーとそのタスク分解によって生成された初期化とプロペラ回転のモジュールについて，状態遷移図間でチャネル変数を用いた同期を行っているものである．図のように，分解元のロケーションへと遷移する際に，分解先の状態遷移図での遷移が発生するように同期信号を送信する．同様に分解先の状態遷移図が全てのタスクを終え，初期状態へと戻る際に，分解元のロケーションから遷移が発生するように同期信号を送信する．このように同期信号の送受信を記述する事により，3 つの状態遷移図における遷移は図 4 右部分のシーケンス図のようになる．これにより，3 つの状態遷移図にかかる合計の処理時間を取得し，検証に用いることができる．また，図 4 の状態遷移図に’C’ と記述されたロケーションが含まれているが，これはコミットロケーションと呼ばれるものであり，UPPAAL の仕様上，連続して同期信号の送受信を行いたい場合に，該当ロケーション間に配置する必要がある．本フレームワークではタスクを持たないロケーションとして位置付けられ，次節の初期処理時間設定も行う必要はない．

3-3 時間制約と初期処理時間の設定

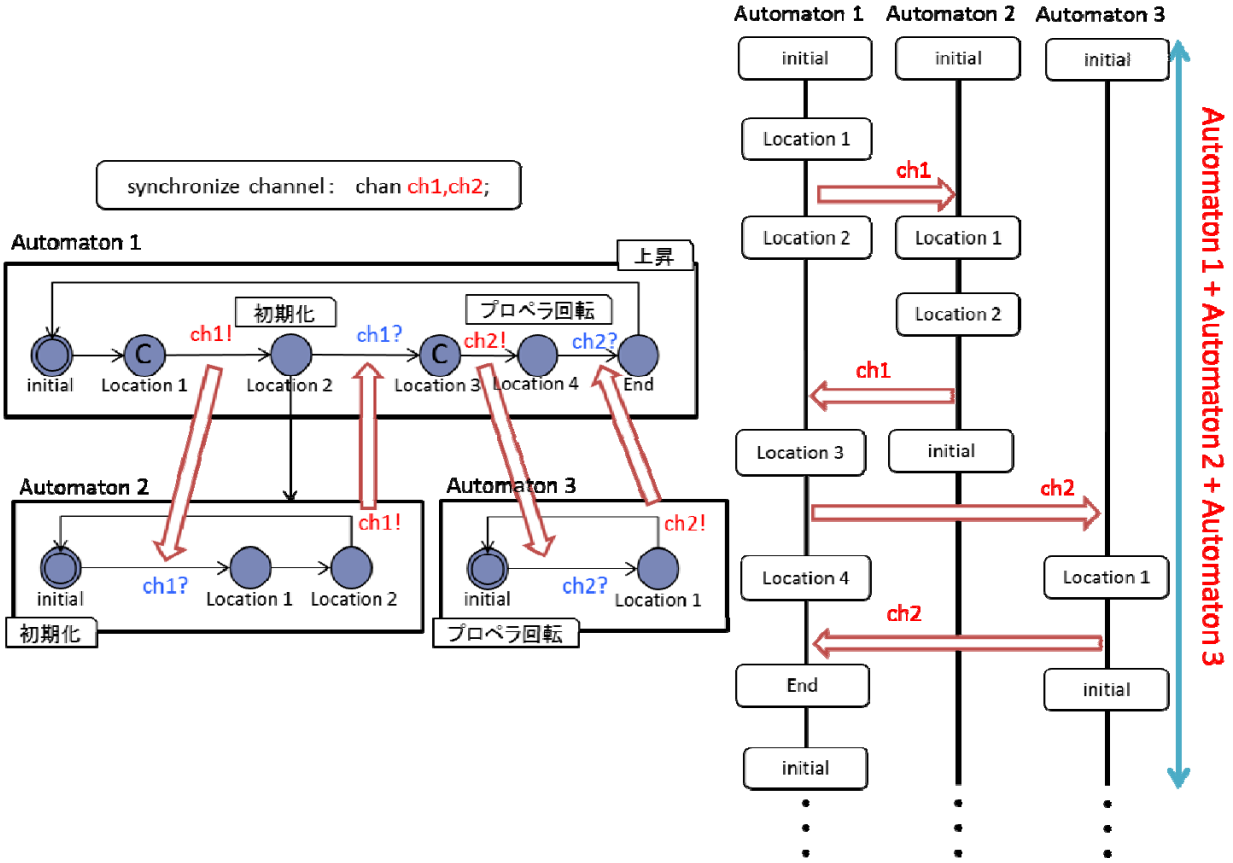

図 4：状態遷移図の同期とシーケンス図

モデル作成の最後に，状態遷移図群のタスクに対して初期処理時間の設定と，システムに対する時間制約の設定を行う．図 5 は，図 3 のプロペラ回転モジュールに初期処理時間を設定したものである．本フレームワークでは，コンポーネントの組み替えと時間に関する検証を考慮するために，全ての状態遷移図が boolean 型変数を 1 つと int 型変数を分解されていないタスクの数だけ持つ必要がある．特に後者の int 型変数はシステムの処理時間を状態遷移図に反映させるために欠かせないものとなる．図 5 には initial と rotate の 2 つのロケーションがあり，initial から rotate へと遷移する際にクロック変数が初期化されている．そして，rotate の不変式 clk <= time と rotate から initial への遷移条件 clk >= time の双方を考慮すると，rotate ロケーションでクロックが消費され，クロックの値がパラメータ変数と等しくなったタイミングで遷移が発生することとなる．

併せて，時間制約の設定を行う．UPPAAL で時間制約を記述するには時相論理を以下の記号を用いて記述する．

- A[] p: 全ての実行パスで常に p が成り立つ
- A<> p: 全ての実行パスでいずれ p が成り立つ
- E[] p: p が常に成り立つパスがある
- E<> p: いずれ p が成り立つパスがある
- p --> q: p が成り立てばそのうち q が成り立つ

例えば，1 クロック 1 ミリ秒として上昇ストラテジー（テンプレート名：Rise）に 50 ミリ秒以内に処理が終わるという時間制約を設定する場合は，

E<> Rise.End and Rise.clk <= 50

のように記述する．

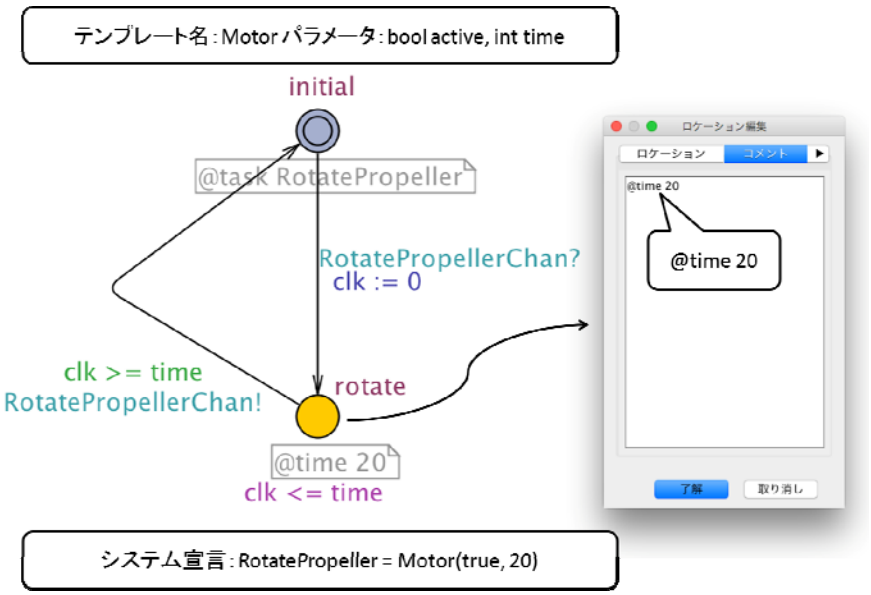

図 5：時間制約と初期処理時間の設定

4 実装フレームワークとしての統合

ここまでで示した検証メカニズムにより，時間制約に関する動的な検証を行うことができる．このフレームワークを用いて実際にハードウェアによる自己適応システムを実装するには，フレームワークによって得られた新しいモデルに従ってコンポーネントを組み替える機構が必要となる．この機構は，図 1 のコンポーネント制御部に該当する．

このコンポーネント制御には，文献[14] のフレームワークを用いた．このフレームワークは，既存のフレームワーク[15]（以下，参考フレームワーク）を参考に実装したものである．図 6 は本研究のコンポーネント制御部に用いる実装フレームワークのクラス関係と主要メソッドを示したものである．参考フレームワークからの最も大きな変化は，JADE の Behaviour クラスの継承していたものを Java の Thread クラスの継承へと変更している点である．従来，参考フレームワークでは Behaviour として記述したプロセスを並行に動作させることができるエージェントプラットフォーム（JADE）[16] を拡張利用することで，複数のコンポーネントの並行動作を実現しているが，コンポーネントの並行動作の実現が JADE に依存していた．JADE は Behaviour の並行動作以外にも GUI による操作やプラットフォーム間の通信がサポートされている．組み込みシステムにおいては軽量なシステムが望ましく，また，GUI や通信といった機能がサポートされているとは限らないため，本研究では，参考フレームワークを軽量化したフレームワークを用いる．

なお，参考フレームワークはコンポーネントベースの設計をもつシステムを対象としており，各コンポーネントに制御ループにおける責務を割り当てることで実装をパターン化し，プログラミングのコスト軽減を図っている．提案する新たなフレームワークでもこれらの特徴を引き継いでいる．

自己適応システムの実現には，MAPE-K ループ[17] などの制御ループ[18] を用いることが有効であると考えられている．これは監視（Monitor），分析（Analyze），意思決定（Plan），実行（Act）の 4 つを繰り返すことで自己適応が実現されるというメカニズムであり，しばしば拡張的に知識領域（Knowledge）を用いたフィードバックを用いることがある．コンポーネント制御部は，対象システムの内部情報やセンサーから得られる情報を通してその周辺環境を監視する．得られた情報から，現在の状態でサービスの継続が可能であるか，システムの要件は満たせているかなどを分析し，満たされていない項目があれば現在のコンポーネント構成を変更し，新しいモデルとして生成する．そして，この際に加えた変更をオートマトンコントローラへ

と伝えるとともに，新しいモデルに従って実際のコンポーネント構成を変化させる．一方でオートマトンコントローラは，対象システムのコンポーネント単位の処理時間を監視する．得られた処理時間の情報を用いて，課せられた時間制約が守られているかを検証し，違反があれば現在のコンポーネント構成を変更し時間制約を満たすような新しいモデルを作成する．そして，コンポーネント制御部へと変更を伝え，実際のコンポーネント構成を変化させる．

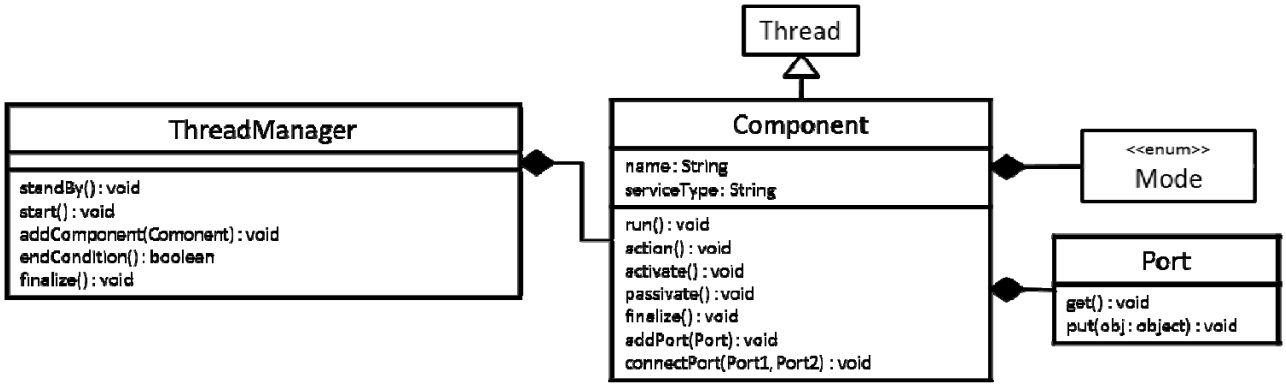

図 6：フレームワークのクラス関係図と主要メソッド

5 動作実験

ここまでで示した実装フレームワークを用いて，LEGO Mind-Storms EV3 上に自己適応システムを構築した．この実装を通してフレームワークを用いた自己適応システム実装手順や，その有効性を示す．

5-1 例題システム

本論文では，例題システムとして物体収集システム（図 7）を扱う．このシステムは，ロボットが点在している物体を運搬し，所定の箇所へと集めるというシステムである．ロボットにはカメラ，超音波センサー，カラーセンサーが備えられており，それぞれ前方の画像，正面物体との距離，ロボット直下の床色が取得できる．ロボットは，カメラから得られる情報を元に，赤く塗られた収集対象物を，青いマーカーで示された収集箇所へと運搬する．収集箇所手前に描かれた黒い線の内側まで運搬すれば，その物体は収集完了となる．システムは探索，接近，運搬の 3 つの大きな機能を持つ．探索ではカメラから画像を取得し，画像内の赤い物体もしくは青い物体の位置を判定する．対象の色が検出されなければ，その方向には目的物がないと判断し，その場で少し旋回，もう一度探索を実行する．探索によって対象物が発見されると，システムは接近を実行する．接近ではタイヤを回転させて目的物へ向けて移動する．この時，探索時と同様にカメラから画像を取得し，対象物の位置情報から進行方向を微調整する．さらに，超音波センサーを用いて対象物への到達の判定を行う．対象物への到達が確認されると，システムは対象物をアーム部分に保持したまま探索によって収取箇所を探す．収集箇所を発見すると，システムは運搬を実行する．運搬では，対象物を保持したまま収取箇所へ向けて移動する．この際，接近時と同様に画像判定による微調整が行われる．さらに，カラーセンサーを用いて到達ラインの検出が行われる．到達ラインまで運搬されたのが確認されると，システムは再び収集対象物の探索を実行し，システムは継続して動作する．システムに関する要件として速やかな物体の収集を実現させるため，システムの各機能には時間制約を設定し，制約に違反した場合は何らかの方法で時間あたりの収集効率の向上を図ることとする．

ロボットは LEGO MindStorms EV3 を使用し実装した．なお，動的検証のために利用している UPPAAL の検証器はバイナリ形式で配布されているため，MindStorms 上で動作させることができない．従って，MindStorms 内で生成した中間ファイルを PC へ送信し，PC 上で受信したファイルを検証，その結果を返信するよう設計した．

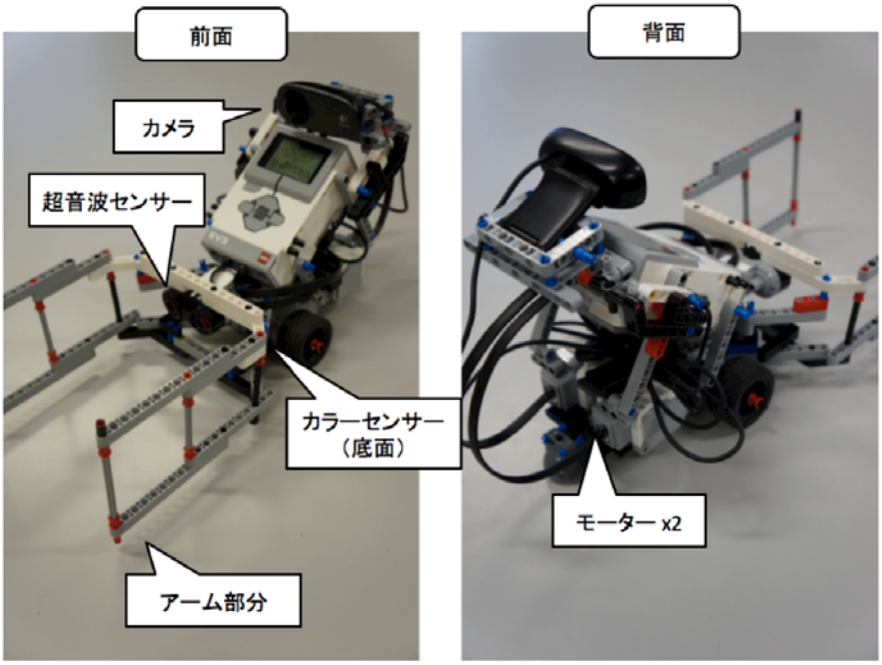

図 7：物体収集システム

5-2 各モデルの作成

時間制約の動的な検証のために，物体収集システムの機能を表す状態遷移図群を設計した．本実験では，探索機能のループ，接近機能のループ，運搬機能のループを作成した．探索機能は 2 つの抽象的なタスク RotateWheels と ImageAnalyze を持っており，他の 2 つの機能についてもそれぞれ異なる 2 つの抽象的なタスクを持っている．これらの抽象的なタスクはそれぞれタスク分解し，状態遷移図を作成した．

続いて，コンポーネント構成を設計した．本フレームワークでは，各コンポーネントに制御ループに関する責務を割り当てる．ここでの責務とは制御ループにおける情報収集（Collect），情報分析（Analyze），意思決定（Decide），実行（Act）の事である．このフレームワークでは C（Collect），A（Analyze）に加えて情報分析と意思決定を統合した AD（Analyze and Decide）の 3 種類のいずれかの責務を割り当てる．あらかじめコンポーネントの責務を明別しておく事で，コンポーネントの責務ごとにあるプログラム記述における共通点や類似性を利用できるため，プログラミングのコスト軽減に繋がる．

5-3 時間制約の設定

モデル生成後にシステムに関する要件から時間制約を設定した．本実験で扱うシステム対しては，探索，接近，運搬の各機能に対して時間制約を設定した．例えば探索機能に関する時間制約として以下を設定した．

E<> root.EndSearch and root.clk <= 30

本実験では，全ての機能について 1 クロックあたり 1 秒として時間を取得し，状態遷移図に埋め込む．従ってこの記述により，探索機能は 30 秒以内に処理が終了する，つまり，30 秒以内に対象を発見するという制約を課したこととなる．

5-4 シナリオ

動作実験のために用意したシナリオは次のとおりである．システムが正常に動作し運搬作業を行なっている途中，ある物体の運搬を終えた後に次の運搬対象物を探索すると，今までに運搬した物体と比べて収集箇所から遠い位置にある物体が検知される．ロボットは検知された物体を収集するため接近を試みるが，物体

までの距離が遠いと，接近機能の完了までに長い時間がかかり，その後の時間制約に関する検証で時間制約違反が検出されてしまう．この時，時間制約違反に対する適応としてコンポーネント構成の変更が発生し，次回の接近機能の実行時の振る舞いが変化することが期待される．接近機能において，システムの初期状態では物体を 1 つずつ運搬し，単純に収集対象物と収集箇所を往復するという戦略が用いられている．

このシナリオにおいては，接近機能の戦略が 2 つ同時に収集するというものに変わることが期待される．この戦略では 1 つ目の収集対象物へと到達後,収集箇所へ向かうのではなく,別の収集対象物へと接近し 2 つの取集対象物をアーム部分に保持する．そして，その状態のまま収集箇所へ運搬することで 2 つの対象物を同時に収集する．この戦略では 1 つ目の対象物へ到達後に運搬を行わないため，時間あたりの収集効率が上がる．しかし，1 つ目の対象物をアーム部分に保持したまま次の対象物へと向かうため，2 つ目の対象物への接近時に超音波センサーによる検知ができない.そのため 2 つ目の対象物の取得に失敗する可能性がある．そのような理由から，システムの初期状態ではこの戦略は採用されていない．時間制約違反が発生し，どうしても時間あたりの収集効率を上げる必要がある際に採用できるように，対応するコンポーネントや状態遷移図を作成している．

5-5 実験結果

実験の結果，収集対象物を収集箇所から遠く離して配置すると，期待通りに振る舞いの変更が見られた．収集箇所周辺のみに物体を配置した場合には振る舞いの変更は見られなかったが，これは時間制約違反が発生しなかったためである．動的検証による振る舞い変更のタイミングは，期待された通りであり，安定した動作が確認できた．

物体収集システム実行時のログを確認したところ，比較的近くにある収集対象物への到達時においては，検証結果が"Yes"，つまり時間制約を満たした動作であることが確認された．一方，比較的遠くにある収集対象物への到達時については，検証結果が"No"となり，コンポーネントの構成変更が促されることが確認できた．コンポーネントのコンフィギュレーションを確認したところ，1 つの対象物に接近する戦略である ApproachObject コンポーネントが 2 つの対象物を同時に運搬する戦略である CatchTwoObject コンポーネントに変わっていた．つまり，システムが自身で時間制約違反の可能性を判断し，コンポーネントの構成を変化させたことが確認できた．

6 まとめ

本調査研究では，実世界での利用を考慮した自己適応システム実装フレームワークについて検討した．また，自己適応システムに必要となる動的検証メカニズムとして，特に実世界での動作時に強い制約となり得る時間制約を対象とした動的検証メカニズムについて検討した．このフレームワークは，時間制約を動的に検証するためにモデル検査ツール UPPAAL を用い，実行中のシステムを構成するコンポーネントからそれぞれの処理時間を取得し，その値を用いてリアルタイムなモデルを生成する．本研究では，MindStorms を用いた簡易な自己適応システムを実装することで，同フレームワークの有効性を評価した．今後は実装した機能を拡張することで，動的検証の対象を拡充していく予定である．

【参考文献】

[1] B. Chen, X. Peng, Y. Yu, B. Nuseibeh, andW. Zhao, “Self-adaptation through incremental generative model transformations at runtime,” Proc. of the 36th International Conference on Software Engineering (ICSE’14), pp.676–687, ICSE 2014, ACM, 2014.

[2] V.E. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos, “Requirements-driven software evolution,” Comput. Sci., vol.28, no.4, pp.311–329, Nov. 2013.

[3] N. Esfahani, A. Elkhodary, and S. Malek, “A learning-based framework for engineering feature-oriented self-adaptive software systems,” IEEE Transactions on Software Engineering, vol.39, no.11, pp.1467–1493, Nov. 2013.
[4] H. Nakagawa, A. Ohsuga, and S. Honiden, “Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops,” Proc. of IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems (SASO’12), pp.59–68, IEEE, sept. 2012.
[5] S. Niemczyk and K. Geihs, “Adaptive run-time models for groups of autonomous robots,” Proc. of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS’15), pp.127–133, IEEE Press, 2015.
[6] R. Asadollahi, M. Salehie, and L. Tahvildari, “Starmx: A framework for developing self-managing java-based systems,” Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS’09. ICSE Workshop on, pp.58–67, May 2009.
[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, “ Rainbow: architecture-based self-adaptation with reusable infrastructure,” Computer, vol.37, no.10, pp.46–54, Oct. 2004.
[8] 中島震, “自己適応 web アプリケーションシステム: 概念アーキテクチャと実現フレームワーク, ” コンピュータソフトウェア, vol.29, no.3, pp.54-69, 2012.
[9] K.G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a Nutshell,” 1997.
[10] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, “Testing real-time systems using uppaal,” Formal Methods and Testing, eds. by R.M. Hierons, J.P. Bowen, and M. Harman, pp.77–117, Springer-Verlag, Berlin, Heidelberg, 2008.
[11] T.K. Iversen, K.J. Kristoffersen, K.G. Larsen, M. Laursen, R.G. Madsen, S.K. Mortensen, P. Pettersson, and C.B. Thomasen, “Modelchecking real-time control programs–verifying LEGO mindstorms systems using UPPAAL,” Proc. of 12th Euromicro Conference on Real-Time Systems (ECRTS’00), pp.147–155, 2000.
[12] D. Sykes, W. Heaven, J. Magee, and J. Kramer, “From goals to components: a combined approach to self-management,” In Proc. of the International Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS ’08), pp.1–8, ACM, Leipzig, Germany, 2008.
[13] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic, “PLASMA: a plan-based layered architecture for software model-driven adaptation,” Proc. of the IEEE/ACMinternational conference on Automated software engineering (ASE ’10), pp.467–476, ACM, 2010.
[14] H. Tsuda, H. Nakagawa, and T. Tsuchiya, “Towards self-adaptation on real-world hardware: A preliminary lightweight programming framework,” Proc. of the IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO’15), pp.176–177, Sept. 2015.
[15] 中川博之, 大須賀昭彦, 本位田真一, “ビヘイビア記述に基づく自己適応システム実装フレームワークの提案,” 人工知能学会論文誌, vol.26, no.1, pp.1-12, 2011.
[16] Telecom Italia, “JADE: Java agent development framework,” http://jade.tilab.com/. http://jade.tilab.com/
[17] J.O. Kephart and D.M. Chess, “The vision of autonomic computing,” IEEE Computer, vol.36, no.1, pp.41–50, 2003.
[18] M. Shaw, “Beyond objects: a software design paradigm based on process control,” SIGSOFT Software Engineering Notes, vol.20, pp.27–38, Jan. 1995.

〈発　表　資　料〉

題　　名	掲載誌・学会名等	発表年月
A Framework for Updating Functionalities Based on the MAPE Loop Mechanism	The 42nd IEEE Computer Software and Applications Conference (COMPSAC 2018)	2018年7月 (発表予定, 採録決定)
時間制約を考慮可能な自己適応システム実装フレームワークの検討	電子情報通信学会 知能ソフトウェア工学研究会(SIG-KBSE), 信学技報 KBSE2017-40, pp.121-126	2018年3月
MAPEループ構造に基づいた機能更新フレームワークに関する考察	電子情報通信学会 知能ソフトウェア工学研究会(SIG-KBSE), 信学技報 KBSE2017-32, pp.1-6	2018年1月
MAPEループを用いたIoTデバイスの効率的な再利用法の検討	ウィンターワークショップ 2018・イン・宮島 (WWS2018), pp.74-75	2018年1月
STAMP／STPAを用いたCyber-Physical Systemsの検証	IPA 第2回STAMPワークショップ	2017年11月
IoT システムのアーキテクチャモデルを用いた安全性の検証	ソフトウェア工学の基礎ワークショップ(FOSE2017)	2017年11月
A Tool to Edit and Verify IoT System Architecture Model	The ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017)	2017年9月
ソフトウェア工学の最前線 ～ソフトウェアが社会のすべてを定義する時代～：[未来に向かって] IoT時代の環境適応型ソフトウェア	情報処理学会学会誌 情報処理 58(8), pp.702-704	2017年8月
IoT システムアーキテクチャのモデリング記法によるモデル検査支援手法の試作と評価	電子情報通信学会 知能ソフトウェア工学研究会(SIG-KBSE), 電子情報通信学会 ソフトウェアサイエンス研究会 (SIG-SS),情報処理学会 ソフトウェア工学研究会(SIGSE)合同研究発表会,信学技報 KBSE2017-5, pp.1-6	2017年7月