RC-006

時間制約を考慮可能な動的検証機能を備えた自己適応システム実装フレームワーク

Towards a Framework for Self-Adaptive Systems Based on a Dynamic Verification Mechanism Considering Time Constraint

津田宏軌 † Hiroki Tsuda　中川博之 † Hiroyuki Nakagawa　土屋達弘 † Tatsuhiro Tsuchiya

概 要

システム実行時に動的に振る舞いを変更させる自己適応システムがもつ柔軟さが，設計段階において実行環境を完全に想定することの難しさから注目を集めている．無人車両や無人航空機などの技術発展に伴って，自己適応システムの活躍が期待される場は，組み込みシステムの分野へも広がりを見せている．安全面などの理由から，組み込みシステムの物理的な動作に関わる時間制約は重要視されることがあり，時間制約に基づいた適応動作は組み込みシステムの自己適応化において重要な要件となりえる．その実現には動的な時間制約の検証が必須となるが，時間を扱うことのできるフレームワークは未だ確立されていない．本研究では，時間を扱うことのできるモデル検査ツール UPPAAL の検証器を動的に扱うことで，時間制約違反に動的に適応できるような自己適応システムの設計手法と，その実装をサポートするメソッドをフレームワークとして提案する．

1. はじめに

ソフトウェアシステムの開発において，求められる要件や想定される動作環境，利用者が起こし得る操作などを設計段階で完全に定義することは難しく，実行時にシステムが想定外の環境下で動作しなければならないことも少なくない．そのため，実行時の振る舞い変更によって，ある程度の状況であれば自動的に対応できる柔軟なシステムが求められる．自己適応システムは，システムを取り巻く環境や内部状態などの情報を監視，分析した結果に応じて自らの振る舞いを刻々と変化させるメカニズム [1] を持ったシステムである．そのような柔軟さを備えた自己適応システムの研究 [2][3] が積極的に行われる中で，無人車両（UMV），無人航空機（UAV）等の技術発展に伴い，組み込みシステムの分野においてもその活躍が期待されている．組み込みシステムは物理的な動作を伴う場合も少なくなく，機能要求や安全設計においてシステム処理に厳格な時間制約が設けられることがある．組み込みシステムの自己適応化によって，システム実行時に起こってしまった時間制約違反に対して振る舞いの変更によって，再び制約が満たされるような状態が得られ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り，動的検証の枠組みの提案 [4][5][6] も行なわれている．しかし，自己適応システムの実装を目的としたフレームワーク [7][8][9] が様々な観点から提案されているが，時間を扱うことのできるフレームワークは未だ確立されていない．

そこで，本研究では時間制約つき状態遷移図をモデルとするモデル検査ツールである UPPAAL[10] に着目する．UPPAAL は時間の概念を扱うことが可能であり，リアルタイムシステムのモデル検査 [11][12] などに利用されている．しかし，UPPAAL は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，動的な検証がサポートされているわけではない．そのため，UPPAAL を動的に扱うことができるようなシステム設計手法とその実装をサポートするメソッド群をフレームワークとして提案する．このフレームワークの設計手法に基づいた実装を行うことで，UPPAAL の検証器を用いた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が実現される．

本稿では，提案する設計手法やメソッドを小型無人航空機制御システムを題材とした例題システムの設計・実装を通して紹介する．さらに，実装した例題システムのシミュレーション実験により，システムの振る舞いが変更されるのを確認する．

2. フレームワーク

本研究では，時間制約を記述することのできる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は時間制約の動的検証を用いた自己適応システムを実現させるための設計手法と，実装をサポートする Java 言語によるメソッドによって構成される．このフレームワークでは状態遷移モデルを用いた自己適応システムの設計手法を提案している．システム実行時にシステムの時間状態遷移モデルを与え，フレームワーク内でそのモデルデータを保持している．図 1 は保持している状態遷移モデルのクラス図である．時間状態遷移モデルを保持していることにより，システム実行中にシステム内部情報やシステム外部環境の変化をモデルに反映させることができる．これにより，保持している時間状態遷移モデルに対してモデル検査を実行することで，システム実行中における動的な時間制約の充足判定を可能にしている．時間制約の充足判定に時間拡張された状態遷移図をモデルとするモデル検査ツールである UPPAAL の検証器を利用している．この検証器を用いることで，システム実行中に起こる時間制約違反を検出し，適応動作の実行を実現させる．システムの時間制約違反が検出されると，それらを解決することのできるシステム構成を検索し，構成変更により解決が可能である場合には新たなシステム構成による状態遷移モデルを提示する．つまり，システムの状態変化に対する適応手段が状態遷移モデルとして示される．本研究ではコンポーネントの接続によるシステム構成を想定している．そこで，システム

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

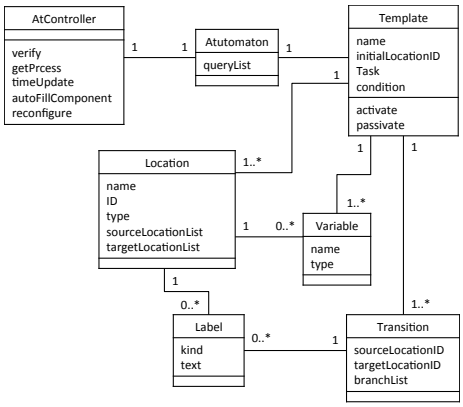

図 1: 内部保持状態遷移モデルのクラス図

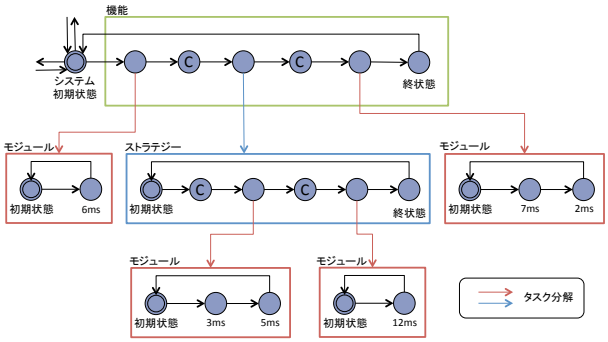

図 2: 機能のタスク分解

を構成するコンポーネントの持つ責務を状態遷移図におけるロケーションとすることで，状態遷移モデルの変更をコンポーネントの組み替えとして読み取ることができる．したがって，示された状態遷移モデル通りにコンポーネントを組み替え，システムの構成を変化させることでシステムの内部情報や外部環境の変化に適応することができる．また，フレームワークに必要な時間状態遷移モデルの構築法は 3 章で，設計手法や提供されるメソッドは 4 章にて例を挙げながら説明する．

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．

3. 状態遷移モデルの作成

本研究のフレームワークで使用する状態遷移モデルは xml 形式で与えるが，UPPAAL 上で作成し，出力したものに対応している．そのため，UPPAAL の特徴の一つであるグラフィカルエディタを用いて状態遷移図を作成することができる．状態遷移モデルは，時間制約の記述を容易にするために，システムの持つ機能単位でひとつのループとなる構成で設計する．各機能は抽象度の高い表現をした処理の連結で構成し，各処理の詳細を別テンプレートにて同様の処理連結によって表現する，この際に，段階的に具体的なコンポーネントによる処理へと詳細化していき，最終的に単一のコンポーネントで完遂できるものとなるまで分解する (図 2)．この手順によって出来上がったテンプレート群が，フレームワークに対して初期入力として与える状態遷移モデルとなる．また，分解元のロケーションと分解先のテンプレートが担う処理をタスクとして，ある処理を別テンプレートに分解することをタスク分解としてそれぞれ定義する．さらに，タスク分解によって生成されたテンプレートのうち単一のコンポーネントで完遂できるタスクであるものモジュール，そうでないものをストラテジーと定義する．このタスク分解によるテンプレート生成の際に，対象システムの仕様に従って初期構成に含まれていないコンポーネントに対応するテンプレートや，既に組み込まれているものの代替となるストラテジーに対応するテンプレートも同様に作成しておく．これによって，コンポーネントの代替による時間制約違反への適応が可能となる，

モジュール内の各処理において，その処理時間を定義することで，分解元のストラテジーや機能の実行に必要な時間が計算できる．ここで定義する処理時間は初期設定であり，システム起動直後や構成変更により該当コンポーネントが新たに利用されるようになった際に用いられる値である．システム実行中においては，動的な検証を行うために実際の計測値や他の情報を用いた類推値で適宜処理時間を更新する必要がある．この情報を基に時間制約の充足判定が行なわれ，制約が満たされていなければ，解決することができるシステム構成の検索が実行される．UPPAAL を用いて状態遷移図を作成するにあたり，テンプレートやロケーションにタスク・処理時間を定義する必要があるが，UPPAAL にはそれらを定義すべき場所が用意されていないため，本研究ではそれらの記述方法を以下の様に定めた．

- タスクの記述

  タスクは分解元のロケーションと分解先のテンプレートに定義する必要があるが，テンプレートに定義する場合は対象テンプレートの初期ロケーションに定義することにより記述方法を統一させた．図 3 のように，タスクを記述したいロケーションのコメントに

  @ task < タスク名 >

  と記述することで，xml 形式の状態遷移モデルを読み込む際に定義されたタスクとして認識される．

- 処理時間の記述

  各モジュールにおける処理時間についてもタスクと同様にロケーションのコメントに

  @ time < 処理時間 >

  のように記述して定義する．

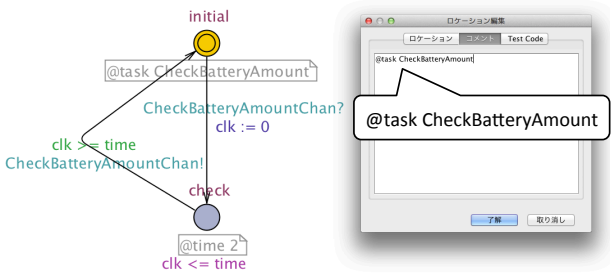

図 3: タスク・処理時間の記述

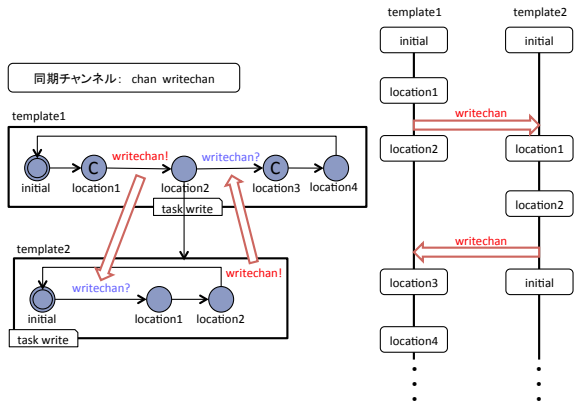

図 4: 同期チャンネルによるテンプレート連結

処理時間は実行中に更新されるため，その更新処理が容易となるように，テンプレートの持つパラメータ定数として設定しておく．定義した時間通りに遷移が起こるように設定したパラメータ変数を使ってガード・不変式の条件を記述する．各機能をタスク分解によって複数のテンプレートに分割したが，実際にその機能を実行する際には，状態遷移図を分解先も含めて深さ優先で辿ることになる．しかし，完全に別のテンプレートとして分解してしまっているため，分解元のロケーションに到達したとしても，分解先のテンプレートの状態を変化させることができない．そこで UPPAAL の同期通信機能を用いて複数のテンプレート間で同期をとり，遷移のタイミングを制御する．図 4 のように分解元のロケーションへ遷移するトランジションと分解先テンプレートの初期状態から遷移するトランジションで同期をとり，同様に分解先テンプレートの初期状態へと遷移するトランジションと分解元のロケーションから遷移するトランジションで同期をとる．これにより，それぞれのテンプレートにおける遷移のタイミングは図 4 右側のシーケンス図のようになり，分解先のテンプレートも含めて辿ることができる．

最後に，時間制約を時相論理式で記述したものをクエリとして登録する．UPPAAL では時相論理を以下のような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

述語 p と q にはロケーションへの到達の他に，クロック変数を用いた評価を記述することができる．状態遷移モデルを独立した機能単位のループによって構成したことにより，初期ロケーションにてクロック変数が初期化されるよう記述することで，各機能の終了を表す最終ロケーションへの到達とクロック変数の値を使った時間条件を論理式にて表現し，機能に課すべき時間制約を表現することができる．

4. 実験

本研究によるフレームワークを用いてシステムを設計し，シミュレーション実験を行う．この実験では小型無人航空機の制御システムを題材とし，フレームワークに沿った設計手順を紹介する．また，シミュレーションシナリオに基づき時間制約が満たされなくなった際のシステムの振る舞いを確認する．

4.1. 例題システム

この実験では小型無人航空機の制御システムを題材とする．このシステムは，用意されたコマンド群のうち指定されたものを実行できるよう内部状態やセンサー類，駆動パーツなどを制御することを目的としたシステムである．システムが実行すべきコマンドは，操縦者から無線通信にて逐次送信されると想定する．コマンドを同時に複数実行することはできず，コマンド実行中に別のコマンドが送信された場合キューに登録されずに破棄される．表 1 はシステムに用意されたコマンド一覧である．このシステムは，地上にて静止している待機状態と，地上を離れ空中を飛行している飛行状態の 2 種類の状態を持つ．また，機体には高度センサー，加速度センサーが搭載されており，個別のモーターに接続された 4 つのプロペラの回転数を制御することにより飛行する．バッテリー枯渇による墜落を防止するために，バッテリー残量が 20 %を下回ると強制着陸させる．システムに課す時間制約の条件として，全ての機能に対して時間制約を設けることとする．ただし，安全の観点から命令解読，水平移動，垂直移動，回転，姿勢制御の機能に関しては許容される遅延を短くし，制約を厳しいものとする．その他の機能についてはある程度の遅延を許容できるような制約を課すものとする．

4.2. システム設計

3 章で述べたように機能単位でのループを作るため，システムに持たせる機能から設計する．本実験ではシステムに持たせるべき機能として以下の 10 の機能を用意した．

表 1: 各コマンドの実行機能系列

コマンド	待機状態	飛行状態	**実行機能系列**
離陸	○	×	1 → 2 → 4 → 5
着陸	×	○	1 → 3 → 5
充電	○	×	1 → 6
前進	×	○	1 → 7 → 4 → 5
後退	×	○	1 → 7 → 4 → 5
左移動	×	○	1 → 7 → 4 → 5
右移動	×	○	1 → 7 → 4 → 5
上昇	×	○	1 → 8 → 4 → 5
下降	×	○	1 → 8 → 4 → 5
左旋回	×	○	1 → 9 → 4 → 5
右旋回	×	○	1 → 9 → 4 → 5
宙返り	×	○	1 → 10 → 4 → 5

1. 命令解読（CommandDecode）：受信したコマンドを解読し，機能の実行系列を決定する．
2. 離陸（TakeOff）：待機状態ならば一定の高さまで浮上し，飛行状態へと移行させる．
3. 着陸（Landing）：飛行状態ならばその場で降下し，プロペラを停止させる．停止後に待機状態へと移行させる．
4. 姿勢制御（BalanceControle）：加速度センサーから情報を取得し，分析する．分析結果に従ってその場で静止できるような各モーターの回転数算出し，その値に従ってモーターを回転させることにより機体を安定させる．
5. バッテリー確認（CheckBattery）：バッテリー残量を確認し，強制着陸実行のための設定値と比較する．規定値を下回り，強制着陸が必要と判断された場合は即座に実行する．
6. 充電（ChargeBattery）：待機状態であればバッテリーへの充電を行う．
7. 水平移動（MoveHorizontal）：指定された方向へと水平に機体を移動させるために必要な各モーターの回転数を算出し，その値に従ってモーターを回転させる．
8. 垂直移動（MoveVertical）：指定された方向へと垂直に機体を移動させるために必要な各モーターの回転数を算出し，その値に従ってモーターを回転させる．
9. 回転（Turn）：指定された方向へと機体を回転させるために必要な各モーターの回転数を算出し，その値に従ってモーターを回転させる．
10. 宙返り（LoopTheLoop）：宙返り実行に十分な高度があるかを高度センサーにより確認し，十分な高度があると判断されればモーターを回転させ，宙返りを実行する．

各コマンドについてこれらの機能を用いて実現させるための実行機能系列を定義する．表 1 内に各コマンドの実行機能系列を機能に添えられた番号を用いて示している．表中の命令解読以降の系列が各コマンドのデコード論理となる．

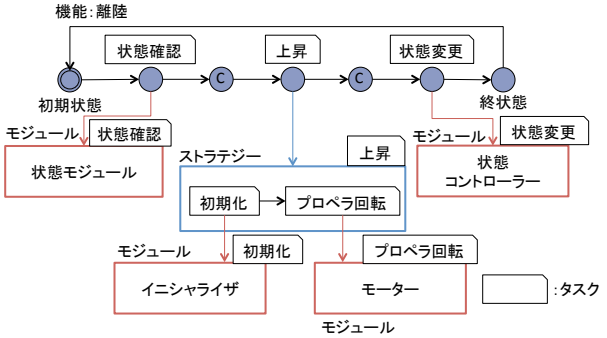

図 5: 離陸機能のタスク分解

次に, 定義した各機能について処理系列を作り，状態遷移図を作成する．作成した状態遷移図内の各処理について 3 章で述べたタスク分解を行ない，単一コンポーネントによる処理まで詳細化していく．図 5 は離陸機能のタスク分解を分解先の状態遷移図を省略して表記したものである．離陸機能は状態確認，上昇，状態変更の 3 つの処理からなり，このうち状態確認，状態変更は共に単一コンポーネントによる処理が可能であるため，それぞれ状態モジュール，状態コントローラーモジュールへとタスク分解を実行する．残る上昇処理には初期化，プロペラ回転の 2 工程が必要だが，これらは単一コンポーネントでは処理できないため，この 2 つの処理を持つ上昇ストラテジーとして分解する．上昇ストラテジーが持つ 2 つの処理である初期化とプロペラ回転は共に単一コンポーネントで処理できるため，それぞれイニシャライザモジュール，モーターモジュールへとタスク分解を実行する．この行程を繰り返し，完成した全ての機能の状態遷移図を統合したものが図 6 であり，図中の赤い数字は機能番号と対応している．さらに，初期構成に含まれていないいくつかのコンポーネントについても対応するモジュールを作成しておく．最終的に，このシステムでは図 6 に示した統合された機能状態遷移図に加え，4 つのストラテジー，17 のモジュールの合計 22 の状態遷移図が作成された．また，タスク分解により作成された各モジュール内の処理について，処理時間の初期値を定義すしておく．ここまでの行程でシステムが持つ全ての状態遷移図が完成したため，定義した処理時間の初期値を添えて初期構成に必要な各テンプレートをインスタンス化し，システム構成として登録する（図 7）．

最後に，システムが満たすべき時間制約を定義する．本シミュレーション実験ではシステムの持つ各機能に対して時間制約を設ける．表 2 は各機能に設けた時間制約一覧である．各機能について，処理系列に含まれるタスクに定義した処理時間から求めた理論値と設定された時間制約を比較し，許容される遅延を計算している．許容遅延が赤文字で記されている機能については，安全面などの理由から大きな遅延が許されず，時間制約が厳しくなっている機能である．表内の各項目について，クロック変数と対応するロケーションを用いた時相論理式で記述し，クエリとして登録する．

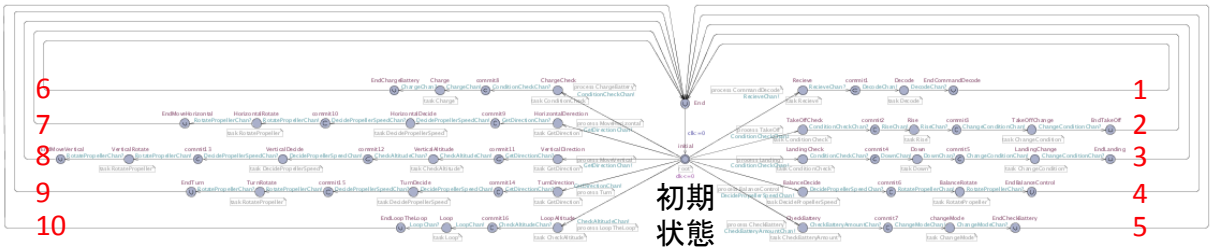

図 6: 小型無人航空機制御システムの最上層状態遷移図

```
root = Drone(true);
Receive = Reciever(true,3);
Decode = Decoder(true,5);
Conditioncheck = CondChecker(true);
Rise = RiseStrategy(true);
Initialize = Initializer(true,6);
ChangeCondition = ConditionController(true,1);
Down = DownStrategy(true);
StopPropller = Stop(true,4);
DecidePropellerSpeed = DecideStrategy(true);
GetGyrodate = GyroSensor(true,2);
AnalyzeGyroData = GyroAnalyzer(true,3);
DecideEachPropellerSpeed = SpeedCalculater(true,8);
RotatePropeller = Motor(true,20);
CheckBatteryAmount = Battery(true,2);
ChangeMode = ModeController(true,1);
Charge = Charger(true,100);
GetDirection = Direction(true,4);
CheckAltitude = AltitudeSensor(true,3);
Loop = LoopStrategy(true);
system root,Receive,Decode,conditionCheck,Rise,Initialize,
       ChangeCondition,Down,StopPropeller,DecidePropellerSpeed,
       GetGyroData,AnalyzeGyroData,DecideEachPropellerSpeed,
       RotatePropeller,CheckBatteryAmount,ChangeMode,Charge,
       GetDirection,CheckAltitude,Loop;
```

図 7: 小型無人航空機制御システムのシステム宣言

4.3. 実装

前節までに設計したシステムを実際にコーディングする．システムがもつ各機能の実現に必要な処理が単一コンポーネントで処理できる単位まで分解されているため，コンポーネント別にコーディングを行ない，状態遷移図を辿るような順に実行されるようにすれば良い．コーディングに際して，フレームワークが提供するメソッドとして図 1 の AtController クラスが持つ以下のものが利用できる．

- void verify()
  システムにおける現在の状態遷移モデルを xml ファイルに出力し，検証器にかけることで時間制約の充足判定を行う．検証結果はテキスト形式でファイルに出力される．

- ArrayList<String> getProcess(String processName)
  実行したい機能名を引数に与えることで，状態遷移図を辿り，実行すべきモジュール名のリストを返す．

- int timeUpdate(String tempName, Sting locName, int newTime)
  実行時に各コンポーネントの処理時間に変化があった場合，このメソッドを用いて状態遷移モデルに登録されている処理時間を更新する．引数として更新対象のコンポーネントに対応するテンプレート名，ロケーション名と新しい処理時間を与える．正常に更新が完了すれば 0 が返されるが，引数で与えた名前のテンプレートが見つからなかった場合は-1 が返され，同様にロケーションが見つからなければ-2 が返される．

表 2: 時間制約一覧

機能	時間制約	理論値	許容遅延
1. 命令解読	10ms	8ms	2ms
2. 離陸	40ms	29ms	11ms
3. 着陸	40ms	27ms	13ms
4. 姿勢制御	35ms	33ms	2ms
5. バッテリー確認	10ms	3ms	7ms
6. 充電	115ms	103ms	12ms
7. 水平移動	40ms	37ms	3ms
8. 垂直移動	43ms	40ms	3ms
9. 回転	40ms	37ms	3ms
10. 宙返り	30ms	23ms	7ms

- int autoFillComponent(String TempName)
  引数で与えられた名前のテンプレートを差し替えることによって時間制約を満足させることを試みるメソッド．図 8 に疑似コードを示す．差し替えによって時間制約を満足できたならば 0 を返し，満足できなかった場合は 1 を返す．また，引数で与えた名前のテンプレートが見つからなかった場合は-1，システム構成の書き換えに失敗した場合は-2 を返す．

- int reconfigure()
  検証器による出力ファイルを確認し，満たされていない制約があれば，タスクの代替によって時間制約を満たすことができるようなシステムの構成を検索する．図 9 に疑似コードを示す．クエリに含まれるロケーションから状態遷移図を辿りながらタスクの差し替えを試していく，時間制約を満たすようなシステム構成が発見できれば 0 を返し，見つからなければ-1 を返す．

よって，図 1 から得られるデコード論理に基づいて実行すべき機能を決定し，機能を実現するための実行モ

```
for each template{
     if(template.name == targetName){
          targetTemplate <- template;
          targetTask <- template.task;
     }
}
if(not find) return -1;
for each template{
     if(otherTemplate.task == targetTask){
          newTemplate <- otherTemplate;
          passivate(targetTemplate);
          activate(newTemplate);
          rewrite system configuration;
          if(fail rewrite) return -2;
          verify;
          if(all query satisfied){
               return 0;
          }else{
               activate(targetTemplate);
               passivate(newTemplate);
               return 1;
          }
     }
}
```

図 8: autoFillComponent の疑似コード

```
get unsatisfied queries;
for(each query){
     location <- get target location from query;
     while(location == initial location){
          if(location has processing time){
               autoFillComponent(template(include location).name);
               if(success){
                    return 0;
               }
          else if(location has task){
               recursive with location in template having same task;
          }
          location <- one of source location, all paths executed by recursive;
     }
     return -1;
}
```

図 9: reconfigure の疑似コード

ジュールリストを getProcess メソッドから得ることで，モジュール名を使って各コンポーネントへと処理を分岐させるだけで機能が実現される．また各機能が実行されるたびに verify メソッドを呼ぶことで，実行中の時間制約違反に対して適応動作が自動的に実行される．

4.4. シミュレーションシナリオ

前節にて設計したシステムを用いてシミュレーション実験を行った．実験のために下記の 2 つのシナリオを用意した．これらのシナリオを用いたシミュレーションにより，本研究によるフレームワークに基づいた設計によって実装されたシステムに時間制約の違反が発生した場合における，システムの振る舞いを確認した．本研究では時間制約違反が発生する要因としてコンポーネント故障を想定したシミュレーションを行っている．故障としては，ハードウェアにおける物理的なものとバグなどによるソフトウェアに起因するものを扱った．シミュレーションに置けるシステムの振る舞いが，時間制約の動的検証によって制約違反を検出し，適応動作が実行されたものであることを示し，提案フレームワークが時間制約の動的検証により振る舞いの変更が可能な自己適応システムの実現に有効であることを明らかにする．

シナリオ 1．状態確認コンポーネントの故障：機体には地上にて待機中，あるいは飛行中であるかを表す現在の状態が記憶されており，それを確認するためのコンポーネントに対応するモジュール“ CondChecker ”が存在する．このシナリオではこのコンポーネントが故障してしまったことにより現在の状態を取得できなくなってしまい，このコンポーネントを用いている実行機能系列を最後まで実行することができず，結果的に処理時間が大幅に増加したと見なされ時間制約を満たすことができなくなってしまった場合を想定する．

実行系列の中断が制約違反として検出されると，モジュール“ CondChecker と同じ状態確認タスクを持つような別のモジュール，あるいはストラテジーが存在しないか検索される．システムは状態確認のタスクを持つストラテジー“ AlterCondChecker ”を持っており，このコンポーネントが検索により発見され，構成の変更によるコンポーネント故障への適応が期待される．ストラテジー“ AlterChecker ”は記憶されている現在の状態を直接取得するのではなく，機体に搭載された高度センサーと加速度センサーの情報から機体が飛行中であるかを推定し，現在の状態を判断するストラテジーである．現在の状態を直接取得しないため，コンポーネントの故障に関わらず動作するが，処理工程が増えるため状態確認タスクに必要な時間が増加してしまう．しかし，システム設計段階においてストラテジー“ AlterChecker ”による状態確認タスクの代替による処理時間増加に影響される離陸や着陸，充電機能には，このような処理時間増加をある程度許容できるような時間制約の設定がなされているため，タスクの代替によって故障への適応がなされるはずである．

シナリオ 2．モーターの劣化：機体には 4 つのプロペラが搭載されており，それぞれ独立したモーターによって回転駆動されている．モーターの劣化により，モーターの起動時間が増加することによるモーター回転タスクの処理時間増加によって，表 2 にて許容遅延が赤文字で記された時間制約の厳しい機能において制約が満たされなくなる．このような，実行系列を完遂したが時間制約が満たされなかった場合を想定する．

処理時間の更新などにより時間制約が満たされなくなると，構成の変更により制約を満たすことができるような構成を作ることができるかどうかの検索が実行される．検索時には，時間制約が満たされていない機能のループを深さ優先で辿り，タスクを他のストラテジーやモジュールによって代替することにより時間制約を充足できるかどうかが試行される．水平移動や回転，姿勢制御などプロペラの回転数を変化させることによって機体を移動させる場合，それぞれの移動に合わせて各モーターの回転数を算出し，実際にモーターの回転数を変化させるという 2 段階の処理が行なわれる．2 つの処理はそれぞれ初期構成においてモジュール“ SpeedCalculater ”とモジュール“ Motor ”が担当している．このシステムでは，初期構成のモーターに代わることのできる駆動部品は搭載されておらず，プロペラ回転のタスクを代替することができない．しかし，このシステムは“ SpeedCalculater ”と同じタスクをもつモジュール“ RoughCalculater ”を持っている．このモジュールは異なる算出アルゴリズムを用いることにより，算出数値の精密さは劣るが算出にかかる時間を大きく低減させたコンポーネントに対応しているものである．よって，モーター回転数算出のタスクをモジュール“ RoughCalculater ”が代替することにより，モーター劣化による処理時間増加への適応が期待される．

4.5. 実験結果

図 10 はシナリオ 1 実行時のログである．実行ログの 7 行目までは，直前のコマンド実行処理終了後の時間制約

```
1  Battery 85%..
2  ModeController:
3       normalmode
4 //////// xml file generating ... ///////
5 //////// generation completed ///////
6 verify completed
7 queries are sutisfied
8 sim1
9 execute simulation 1: CondiCheck module trouble
10 landing
11 execute processCommandDecode
12 Reciever:
13  command receive
14 Decoder:
15 execute processLanding
16 CondChecker:
17  CondCkeck Module is brokendown
18 //////// xml file generating ... ///////
19 //////// generation completed ///////
20 verify completed
21 trouble is solved
22 retry process Landing
23 execute processLanding
24 AltitudeSensor:
25  Altitude 50
26  condition: flying
27 GyroSensor:
28  collecting data..
29 Moter:
30  rotate motors
31 Stop:
32  stop motors
33 ConditionController:
34 change condition: staying
35 execute processCheckBattery
36 //////// xml file generating ... ///////
37 //////// generation completed ///////
38 verify completed
39 queries are sutisfied
40 exit
```

図 10: シナリオ 1 の実行ログ

```
root = Drone(true);
Receive = Reciever(true,3);
Decode = Decoder(true,5);
Conditioncheck = AlterCondChecker(true);
Rise = RiseStrategy(true);
Initialize = Initializer(true,6);
ChangeCondition = ConditionController(true,1);
Down = DownStrategy(true);
StopPropller = Stop(true,4);
DecidePropellerSpeed = DecideStrategy(true);
GetGyrodate = GyroSensor(true,2);
AnalyzeGyroData = GyroAnalyzer(true,3);
DecideEachPropellerSpeed = SpeedCalculater(true,8);
RotatePropeller = Motor(true,20);
CheckBatteryAmount = Battery(true,2);
ChangeMode = ModeController(true,1);
Charge = Charger(true,100);
GetDirection = Direction(true,4);
CheckAltitude = AltitudeSensor(true,3);
Loop = LoopStrategy(true);
system root,Receive,Decode,conditionCheck,Rise,Initialize,
        ChangeCondition,Down,StopPropeller,DecidePropellerSpeed,
        GetGyroData,AnalyzeGyroData,DecideEachPropellerSpeed,
        RotatePropeller,CheckBatteryAmount,ChangeMode,Charge,
        GetDirection,CheckAltitude,Loop;
```

図 11: シナリオ 1 の適応動作後のシステム構成

充足判定が行なわれている．8 行目にてシナリオ 1 を実行し，意図的に故障を注入している，続いて 10 行目にて故障検出時の動作を確認するため，状態確認タスクが含まれる着陸機能が実行されるよう着陸コマンドを送信している．そして 17 行目にてコンポーネントの故障によって実行系列が中断されたことが検出されている．それに対して，18〜21 行目にて適応動作が起き，問題が解決されたことが読み取れる．結果，22 行目以降にて実行できなかった処理が再度実行され，送信した着陸コマンドによって期待される動作が得られている，また，図 11 は適応動作後の xml ファイルを UPPAAL で開き，システム構成を確認したものである，赤線部分で示されたように，状態確認タスクを担っていたモジュール“ CondCheck ”がストラテジー“ AlterCondCheck ”に差し代わっていることが分かる．これらのことから，シナリオ 1 のシミュレーションについては期待する動作が見られていると言える．

図 12 はシナリオ 2 実行時のログの一部を省略したものである．いくつかのコマンドを送信，実行を確認した

```
1  Battry:
2       Battery 70%..
3  ModeController:
4       normalmode
5  //////// xml file generating ... ///////
6  //////// generation completed ///////
7  verify completed
8  queries are sutisfied
9  sim2
10 execute simulation 2: Motor deterioration
11 processing time of Motor is updated
12 moveup
13 execute processCommandDecode
14 Reciever:
15       command recieve
16 Decoder:
17 execute processMoveVertical
18 Direction:
19  Direction up
20  AltitudeSensor:
21      Altitude 50

        --- 中略 ---

22      rotate motors
23      hovering..
24 execute processCheckBattery
25 Battry:
26      Battery 55%..
27 ModeController:
28      normalmode
29 //////// xml file generating ... ///////
30 //////// generation completed ///////
31 verify completed
32 queries are not sutisfied
33 try reconfiguration
34 //////// xml file generating ... ///////
35 //////// generation completed ///////
36 verify completed
37 reconfigure completed
38 exit
```

図 12: シナリオ 2 の実行ログ

```
root = Drone(true);
Receive = Reciever(true,3);
Decode = Decoder(true,5);
Conditioncheck = CondChecker(true);
Rise = RiseStrategy(true);
Initialize = Initializer(true,6);
ChangeCondition = ConditionController(true,1);
Down = DownStrategy(true);
StopPropller = Stop(true,4);
DecidePropellerSpeed = DecideStrategy(true);
GetGyrodate = GyroSensor(true,2);
AnalyzeGyroData = GyroAnalyzer(true,3);
DecideEachPropellerSpeed = RoughCalculater(true,8);
RotatePropeller = Motor(true,25);
CheckBatteryAmount = Battery(true,2);
ChangeMode = ModeController(true,1);
Charge = Charger(true,100);
GetDirection = Direction(true,4);
CheckAltitude = AltitudeSensor(true,3);
Loop = LoopStrategy(true);
system root,Receive,Decode,conditionCheck,Rise,Initialize,
        ChangeCondition,Down,StopPropeller,DecidePropellerSpeed,
        GetGyroData,AnalyzeGyroData,DecideEachPropellerSpeed,
        RotatePropeller,CheckBatteryAmount,ChangeMode,Charge,
        GetDirection,CheckAltitude,Loop;
```

図 13: シナリオ 2 の適応動作後のシステム構成

後に 9 行目にてシナリオ 2 を実行し，意図的にモーター回転タスクの処理時間を増加させた．その後，9 行目にて上昇コマンドを送信したことにより，28 行目まで上昇のための処理が実行されていることが分かる．そして 29 行目のコマンド実行処理終了後の時間制約充足判定にて，満足されていない制約が存在していることが 32 行目から分かる．そのため，33 行目からシステムの再構成の検索が実行されている．その結果 37 行目までの適応動作で時間制約を満足するシステムの構成が発見され，問題が解決されている，また，図 13 は適応動作後のシステム構成である．赤線部分から分かるように，回転数算出タスクを担っていたモジュール“ SpeedCalculater ”が“ RoughCalculater ”に差し代わっていることが分かる．これらのことから，シナリオ 2 のシミュレーションについても期待する動作が見られているといえる．

シナリオ 1 では制約違反の原因となったコンポーネントを使用しないシステム構成をとることにより時間制約を満足し，シナリオ 2 では制約違反の原因となったコンポーネントがシステム構成に欠かせない場合であっても，

関連コンポーネントの再構成によって時間制約を満足している．2 つのシナリオを用いたシミュレーション実験を通して，フレームワークによって提案される設計手法に基づいて実装されたシステムが時間制約の動的検証により時間制約違反を検出し，期待された適応動作をみせることが確認された．このことから，フレームワークによる設計手法が動的検証によって振る舞いの変更が可能な自己適応システムの実現に有効であるといえる．

5. まとめ

本研究では，自己適応化の期待が高まる組み込みシステムに多く見られるような時間制約の検証を実行時に行うことで，システムの振る舞いを変更するという柔軟な対応ができる自己適応システムの設計手法や，その実装をサポートするようなメソッド群を提供するフレームワークを提案した．提案される設計手法では，時間拡張された状態遷移モデルを用いた設計とモデル検査ツール UPPAAL の検証器を利用することによりシステム実行時の動的な時間制約検証を可能にしている．このフレームワークによる設計手法やメソッド群を小型無人航空機制御システムの設計・実装を通して紹介した．さらに，実装したシステムのシミュレーション実験により，システムの振る舞いが変更されるのを確認し，提案フレームワークによる設計法が有効であることを示した．

提案フレームワークはシステムの持つ時間制約に焦点を絞り，制約の充足状況に応じて振る舞いの変更が行われるような自己適応システムの設計手法を提案するものである．そのため，時間制約以外の要因評価による振る舞いの変更をサポートしておらず，手法の拡張や既存のフレームワークとの融合が課題となっている．また，提案される設計手法では各機能やコンポーネントによる処理時間を独立に設計・定義しているが，それらが相互作用の影響から独立した定義では正確なシステムモデルを構築できない場合が考えられ，機能やコンポーネント間の作用をシステムモデルに組み込むための手法の確立といった課題が残る．

参考文献

[1] M. Shaw, “Beyond objects: A software design paradigm based on process control,” SIGSOFT software Engineering Notes, pp.20:27–38, Jan. 1995.

[2] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” ACM Transaction on Autonomous and Adaptive Systems, vol.4, no.2, pp.14:1–14:42, may 2009. http://doi.acm.org/10.1145/1516533.1516538

[3] M.C. Huebscher and J.A. McCann, “A survey of autonomic computing—degrees, models, and applications,” ACM Comput. Surv., vol.40, no.3, pp.7:1–7:28, aug 2008. http://doi.acm.org/10.1145/1380584.1380585

[4] M.U. Iftikhar and D. Weyns, “Activforms: Active formal models for self-adaptation,” Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp.125–134, SEAMS 2014, ACM, New York, NY, USA, 2014. http://doi.acm.org/10.1145/2593929.2593944

[5] A. Filieri, G. Tamburrelli, and C. Ghezzi, “Supporting self-adaptation via quantitative verification and sensitivity analysis at run time,” IEEE Transactions on Software Engineering, vol.42, pp.75–99, 2015.

[6] 小川賢人, 中川博之, 土屋達弘, “大きな振る舞い変更にも適用可能な自己適応システムの動的モデル検査法,” 研究報告ソフトウェア工学 (SE), pp.1–8, 2016.

[7] 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. http://dx.doi.org/10.1007/s00450-012-0232-2

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

[9] 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.

[10] K.G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a nutshell,” International Journal on Software Tools for Technology Transfer, vol.1, no.1-2, pp.134–152, 1997.

[11] 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. http://dl.acm.org/citation.cfm?id=1806209.1806212

[12] T.K. Iversen, K.J. Kristoffersen, K.G. Larsen, M. Laursen, R.G. Madsen, S.K. Mortensen, P. Pettersson, and C.B. Thomasen, “Model-checking real-time control programs - verifying lego mindstorms systems using uppaal,” In Proc. of 12th Euromicro Conference on Real-Time Systems, pp.147–155, IEEE Computer Society Press, 2000.