Vol.2015-EMB-38 No.6
2015/8/28

動的評価による分散組込みシステムの振舞観測および解析

畑 尚志[1,a)]

概要：分散組込みシステムは多数のサブシステムを組み合わせた構造になるため単一システムに比べて開発規模や費用が増加する問題がある. 開発規模, 費用の増加には外部導入による水平分業的なアプローチが有効だと考えられるが, 分散システムにおける既存の開発手法は設計や実装などの自主開発を前提とした提案が多く, 汎用サブシステムを導入した場合の性能評価手法や問題が発生した場合の問題特定方法など既存の資産を利用した場合に関する提案はほとんど無い. そこで, 本研究では内部がブラックボックスのサブシステムを組み合わせて構築したシステムの振舞を観測する手法を提案する. 具体的には分散システム上で動作するアプリケーションの動作状況を観測し問題個所の特定や, 処理時間等の振舞観測を可能にする. 提案手法はサブシステム自らの振舞いを観測する仕組みを実装し, 動的評価によってイベントログを取得する. 次に動的観測によって取得したイベントログを振舞のモデルと比較することでイベントの分類, 関連づけを行い振舞の分析を可能にする. 本稿では振舞観測の実機評価, モデル比較の検討により提案手法の実現性や有効性の評価を行った.

1. はじめに

M2M, IoT などの組込み機器がネットワークを介して協調動作を行う分散組込みシステムが注目されている. 分散システムは多数のサブシステムを組み合わせた構造であるため, 単一システムに比べて開発規模や費用が増加する問題がある. 開発規模の増加に対しては汎用サブシステムの外部導入による水平分業的なアプローチが有効であると考えられるが, 分散システム開発における提案は設計や実装など自主開発を前提とするものに偏っており, 既存のサブシステムを組み合わせて評価するような提案はほとんどない. たとえば, 図 1 に示す Sys_SW のスイッチが押されると Sys_Disp にメッセージが送信されて LED が点灯するだけの単純なシステムであっても, スイッチ押下から LED 点灯までの処理時間を評価する汎用的な手法は無い. そこで, 本研究では分散システムの振舞いをシステムとして評価する手法を検討する.

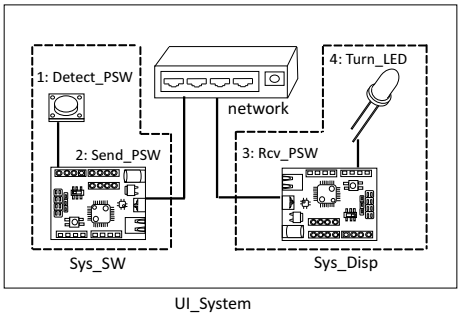

図 1 LED 点灯システム

2. 先行研究

分散システムの振舞いを評価する手法として, さまざまな方法が提案されている.

AADL[1] はタスクのスケジューリング, リソースマネージメントなどの非機能的側面の検証に適した言語である. ソフトウェアだけでなくハードウェアを含めたアーキテクチャを記述可能で, タスクの処理時間やスケジューリングポリシーを定義することでシステムのリアルタイム性能シミュレーションが行える.

論理的な側面のシミュレーション手法として時間オートマトンを使用した手法が提案されている. 時間オートマトンは時間制約を含む有限状態オートマトンを表現することが可能なモデルの記述方法である. UPPAAL[2] などのシミュレーションツールで利用でき, 複数の有限状態オートマトンの並列動作が評価ができる.

Runtime Verification はシステムの振舞いを線形時間時相論理 (LTL:Linear-time Temporal Logic) 式で記述した制約にしたがって検証する手法である. Bartocci[3] らは分散システムのモデルを UPPAAL でシミュレーションし, その結果に対して LTL 式による検証を行う手法を提案した.

AADL は非機能的側面の評価が出来るが, 論理面の評価

[1] ルネサスエレクトロニクス
Renesas Electronics Corporation
[a)] hisashi.hata.wh@renesas.com

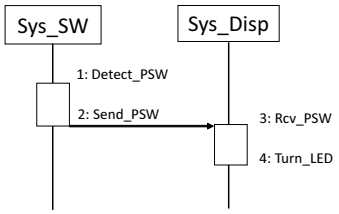

図 2 LED 点灯シーケンス

が出来ない. 時間オートマトンと Runtime Verification は論理の時間的側面の評価が可能だが, バックグラウンドで動作するタスクからの干渉などの評価はできない. また, 先行研究の手法はシミュレーションを基盤にしているため, モデルと実動作が異なる恐れがある. そのため, 設計値と実機動作の比較検証が望まれるが, 実機の振舞いを観測する方法が無いため設計と実機の一致評価が難しいという課題がある.

3. 提案手法の概要

本研究では分散システムの振舞を観測することでシステムの性能評価やサブシステム間の不整合個所の特定を自動的に行う仕組みを構築することを目標とする. ここで振舞とは分散システム内部で並列して発生するイベントを時系列にそって, 機能ごとに分類した状態を指す. 図 1 のシステムで例では, LED を点灯するアプリケーションを構成する 4 つのイベント (Detect_PSW, Send_PSW, Rcv_PSW, Turn_LED) が図 2 に示すように一つの機能, 処理単位ごとに分類され, 時間的整合性を持った状態で評価できるならば振舞が観測できているとする.

提案手法を解説するにあたり解析対象となる分散システム開発の前提条件を定義する. 分散システムの開発体制はシステム全体を構築するセットメーカとサブシステムを開発するサブシステムベンダ (SS ベンダ) で構成されるとする. セットメーカは複数の SS ベンダからサブシステムを購入し組み合わせることで分散システムを構築する. この時, セットメーカは分散システムのネットワーク構成を定義出来るが, サブシステムのソフトウェアへはアクセスできないとする. 同様に SS ベンダはサブシステムのソフトウェアへアクセス出来るが, 分散システムシステムのネットワーク構成や他者のサブシステムの情報を知ることは出来ない.

これらの制限は水平分業による開発体制を想定して策定している. 想定する体制ではセットメーカはシステム統合テストで発生する問題に対し, 提案手法を利用して問題個所の特定や発生状況の記録を行い, SS ベンダがその情報をもとに原因の分析や対策を行う.

本研究で提案する振舞観測は動的評価工程と分類工程に分けられる. 動的観測工程では各サブシステムに自分自身を観測する仕組みを組込み, システムを実際に動作させてイベントログを取得する. 分類工程ではサブシステムの動作フローを記述したモデルとイベントログを比較して振舞を取得する.

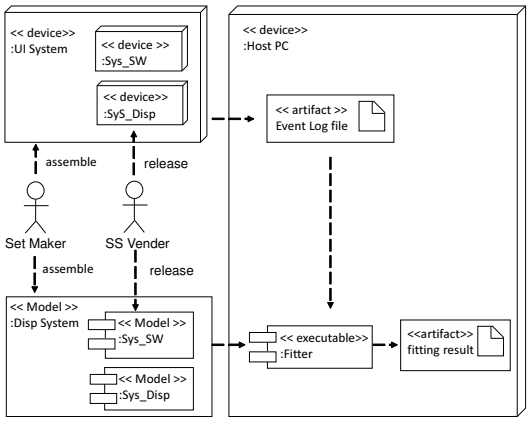

図 3 目標とするワークフローの配置図

提案手法が目標とするワークフローを図 3 に示す. SS ベンダはサブシステムとサブシステムのモデルを作成する. セットメーカはサブシステムを組み合わせて分散システムを作成し, 動的観測によってイベントログを取得する. 次にセットメーカはサブシステムモデルを組み合わせてシステムモデルを作成し, 分類を行うプログラム (Fitter) にイベントログとシステムログを入力することで解析結果を取得する.

4. 動的観測工程

振舞を観測するためにサブシステムに自分自身の振舞を観測するための仕組み (BISM: Build In Self Moniter) を提案する. BISM は時間同期機構と実機評価を組み合わせることで分散システムの振舞い観測を行う.

4.1 時間同期機構

分散システム上で動作するアプリケーションの振舞いを観測することは難しい. これは分散システムが疎結合 (メモリを共有せず, 通信インターフェースで接続) であることに起因する. 例として図 2 のアプリケーションの処理時間を計測する場合を考える. 処理時間の計測には処理の開始と終了時間を計測する必要があるが, 図 2 では処理の開始と終了が物理的に離れたサブシステムで発生するため, 処理の開始を記録する時計 (タイマなど) と終了を記録する時計が異なることになる. そのため, 単純に 2 つのイベントの発生時間を比較することが出来ない. これは 2 つの時計の時間を正確にあわせることが難しい事や, 時計を駆動するクロックの周波数がジッタなどの物理的要因によって変動するためである.

一般に疎結合なシステムでは各サブシステムは異なるクロックソースによって動作するため, それぞれに所属する時計も異なる時間軸の時刻を刻むことになる. 分散システ

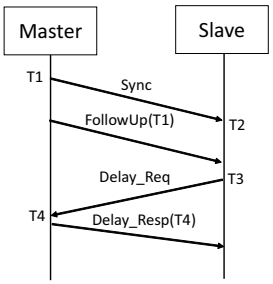

図 4　IEEE1588 時間配信プロトコル

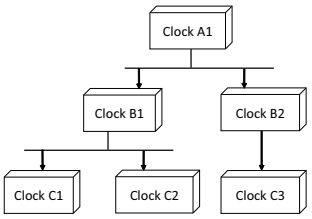

図 5　IEEE1588 同期化ツリー

ム上に定義されるアプリケーションの振舞いを観測するにはイベントの発生時刻をイベントが発生したサブシステムの時計で計測する必要があるが, それぞれの時計は時間軸が異なるためシステム全体で時間的な整合性を保ったまま観測をすることが出来ない.

そこで BISM ではシステム全体でアクセス可能な共通時計を導入することにした. 共通時計は各サブシステムに BISM 用の時計を実装し, すべての時計が同じ時間になるように周期的に時刻補正を行うことで実現する.

時間合わせには IEEE1588 を使用する. IEEE1588 は LAN 経由の高精度な時間配信プロトコルを定義しており, 時計を駆動するクロックの周波数誤差やインターフェースの通信遅延などによる時刻のずれを補正する. 主に産業分野で利用されており, モータを駆動するパルスの位相同期, 複数スピーカによる音響合成などのアプリケーションで実績がある. EtherCAT, Ethernet AVB, Ethernet TCN などの主要な産業, 車載 LAN 規格が対応済みで, すでに多くの IEEE1588 対応マイコンが量産中である.

時間合わせはマスターとスレーブの間で時間配信プロトコルを行うことで実現する (図 4). まずマスターがスレーブに Sync を送信して送信時間 T_1 と受信時間 T_2 を記録する. つぎに Flollow_up(T_1) を送信してスレーブに送信時間 T_1 を伝える. 同様にスレーブが delay_req をマスターに送信し, 送信時間 T_3 と受信時間 T_4 を記録する. 最後にマスターが Delay_resp(T_4) を送信することでスレーブは T_1 から T_4 の時間を取得する. スレーブは時間配信プロトコルで得た送受信時間を利用して遅延等の補正値を算出する. 例えばインターフェースの遅延時間は $T_{delay} = \frac{(T_2-T_1)+(T_4-T_3)}{2}$ で求められる (ここで, 遅延時間は送信と受信で変わらないとする). 時間配信プロトコルを実施するとマスターは周期的に時刻情報を送信し, スレーブは受信した時刻情報を遅延時間で補正して自身の時計を更新する. これによりマスターはスレーブの時計を μsec の精度で同期させることが可能になる.

IEEE1588 はネットワークに参加するノードをツリー状に時間同期する (図 5). 時間同期の元となるノード A1 がグランドマスタークロック (IEEE1588 の基準となる時計) となり B1, B2 に時刻を配信する. つぎに B1, B2 がマスターとなりスレーブ (C1, C2, C3) に対して時間配信をおこなう. これにより IEEE1588 はネットワーク全体のノードの時間を同期させる.

```
1  void sys_a_sw_func(){
2      if(check_sw()){
3          que_send_command(SW_STATE);
4      }
5  }
```

図 6　挿入前のコード

```
1  void sys_a_sw_func(){
2      if(check_sw()){
3          bism_log(Detect_PSW);
4          send_command(PSW);
5          bism_log(Send_PSW);
6      }
7  }
```

図 7　挿入後のコード

4.2　動的評価

時間同期が終了すると分散システムを実際に動作させてイベントログを取得する. イベントログの出力は SS ベンダがソースコードの任意の場所にログ出力コードを挿入して実現する.

図 2 の Sys_SW を例とすると, 通常のソースコードが図 6 の時, 図 7 に示すようにイベントの対応する箇所にログ出力コードを挿入する. ここで図 7 の 2 行目が Detect_PSW イベント, 4 行目が Send_PSW の処理コードにあたり, 3 行目と 5 行目がログ出力コードである. ログ出力処理は実行されるとイベントの識別番号, 共通時計のタイムスタンプ, ユーザ定義情報をサブシステムがアクセス可能な記憶領域に保存する. 記憶領域は観測対象のアプリケーション動作を大きく妨げなければ任意のデバイスを選択可能である. 状況に応じてネットワーク出力やローカルリソースの RAM, ROM, メモリカード等に保存する. BISM は共通時計をサブシステムごとに用意してタイムスタンプの付加をイベント発生時に行うことを特徴とする. イベント発生と同時にタイムスタンプの付加を行うためログをローカルストレージへの蓄積することが出来る.

動的評価が終了するとセットメーカはサブシステムに保存されているイベントログを収集してマージする. これにより, システム全体で発生したイベントを時系列に並べたイベントログの取得が完了になる.

4.3　実機評価

BISM の有効性を評価するため試作評価を行った. 試作評価では GR-SAKURA ボード 3 台を対応するソフトウェ

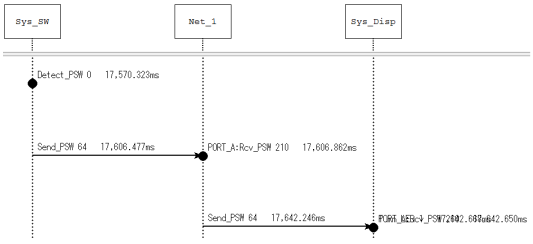

図 8 LED 点灯処理の観測結果

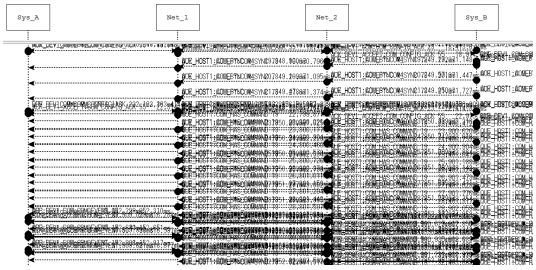

図 9 複数タスクの観測結果

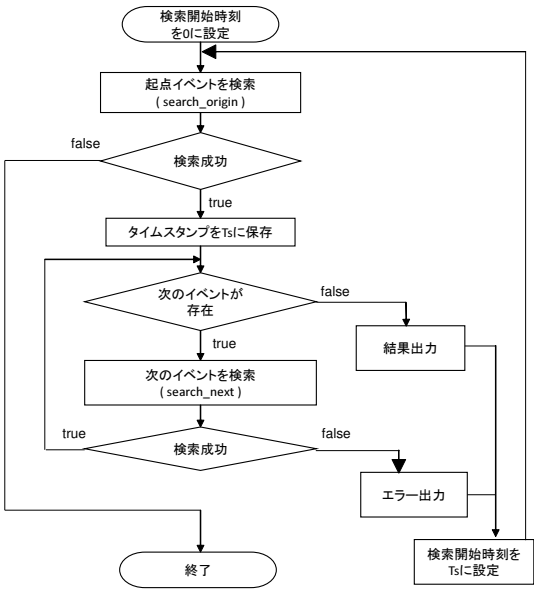

図 10 Fitter の基本動作フローチャート

アを書き込むことで図 1 の UI_System を構築した．GR-SAKURA は Renesas Electronics の RX63N マイコンが搭載された評価ボードで, CPU のクロックを 98MHz, ログの記憶領域に内臓 RAM を 2KB 割り当てた．RX63N には IEEE1588 対応の Ethernet が搭載されていないため, ボード間通信に UART, 同期時計に 32bit タイマ, IEEE1588 プロトコルをソフトウェアエミュレートして時間同期を実装した．ネットワーク構成を 3 台の直列接続とし, IEEE1588 におけるグランドマスタークロックを Sys_Disp に割り当て Sys_Disp から Net, Net から Sys_SW の順に同期処理を行った．

4.4 評価結果

実機評価では時計の同期精度が数百 μsec 程度しか得られなかった．これは同期処理をソフトウェアエミュレートしたためであり, IEEE1588 対応インターフェースを使用すればより高精度の同期時計が使用できる．同期精度の低下は複数のサブシステムが関与する振舞観測で問題を引き起こす可能性があるが, 今回はサブシステム間通信に通信遅延が数 $msec$ かかるプロトコル (ハンドシェイク有の UART 通信) を使用したため時計のずれが通信遅延に隠匿されて大きな問題にならなかった．

実機評価の結果を UML のシーケンス図に習って, 時系列に表示した (図 8)．システム間通信を矢印, システム内部で発生したイベントを円で表示している．左から Sys_A, Net, Sys_B のライフラインを表す．Sys_A のライフラインでは Detect_PSW イベントが発生した後に Send_PSW によって, 発生したイベント情報が Net のライフラインに送信されている事がわかる．Net のライフラインでは Send_PSW の矢印を受けて Net から Sys_Disp に向けてコマンドの中継している様子が観測できている．最後に Sys_Disp のライフラインではコマンド受信を受けて Turn_LED イベントが発生している．

実機評価によって分散システムの同期観測の実現性について示すことができた．しかし, 観測したイベントログはアプリケーションごとに分類されていないため利用面での課題があることも分かった．図 9 にサブシステムや観測するイベントが増加した例を示す．図から明らかなように観測するイベントが増えると人手で評価することが困難になる．そこで, 次章ではイベントログの解析手法について提案する．

5. 分類工程

本章では時系列に記録されたイベントのログをアプリケーションごとに分類する手法を提案する．提案手法ではソフトウェアの動作順序のモデル (フローモデル) を定義し, イベントログとフローモデルを分類プログラム (Fitter) に入力することで分類を行う．モデルの記述は既存言語に適したものが無かったため独自開発した言語を使用する．

5.1 Fitter アルゴリズムの概要

フローモデルは振舞いを時系列に発生するイベント列 (フロー) として表現する．特にアプリケーションの最初に発生するイベントを起点イベントと呼称し, すべてのアプリケーションは任意の起点イベントを先頭とするイベント駆動な処理として取り扱う．Fitter の基本動作フローチャートを図 10, Fitter の動作例を図 11 に示す．ここでフローモデルのイベントは (所属するサブシステム)::(イベント名) で記述している．図 11 左のフローモデルの例では Sys_A で Event_1 が発生した後に Sys_B で Event_2 が発生することを表す．

動作例において Fitter は図 11 のフローモデルとイベ

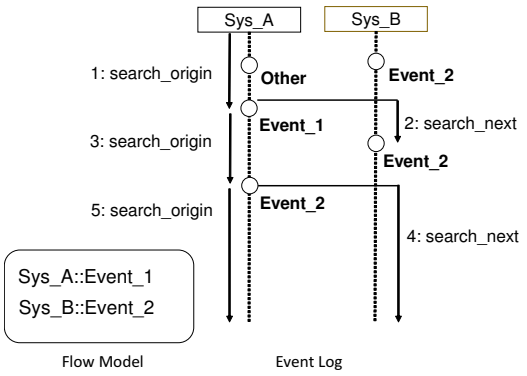

図 11　基本動作の例

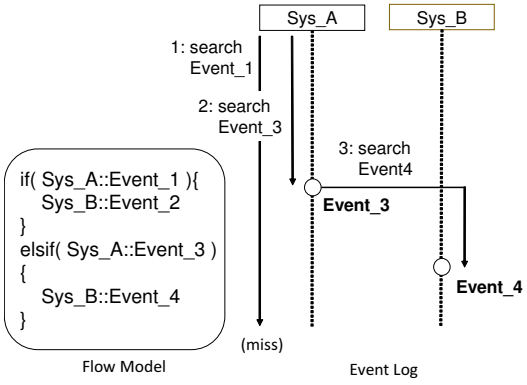

図 12　分岐動作の例

ントログが与えられると, フローモデルを読み込んで検索開始時刻をログの先頭に設定する. 次にモデルから起点イベント (Sys_A::Event_1) を読み込んで検索を行う (1:Search_origin). 起点イベントが見つかるとタイムスタンプを Ts に保存して次のイベント (Sys_B::Event_2) の検索を行う (2:Search_Next). 図 11 のフローモデルでは Sys_B::Event_2 が終端イベントなので Fitter は検索結果を出力し, 検索開始時刻を Ts に戻しフローモデルの検索を最初からやり直す. これは 3:Search_origin, 4:Search_Next が該当する. ただし 4:Search_Next では Event_2 が見つからないため Fitter はエラーを出力して次のフロー検索に移る. Fitter は一連の処理を起点イベントが見つからなくなるまで繰り返す. 図 11 では 5:Search_Origin で起点イベントが見つからないため終了となる.

フローモデルでは単純なイベント列の検索に加えて条件分岐や繰り返し等の制御構文も実装する. 例として条件分岐の記述と動作例を図 12 示す. 例では最初の if 文の検索式 Sys_A::Event_1 を実効するが Event_1 が見つからず失敗する. Fitter は if 文が失敗したため検索開始時刻を巻き戻し次の elsif 文に移り Sys_A::Event_3 を検索する. Event_3 は存在するため elsif 文の中の Sys_B::Event_4 を検索する.

5.2　フローモデルの記述

フローモデルの記述手法について検討を行う. フローモデルはアプリケーションに注目した場合と, システム構成に注目した場合で異なる記法が必要になる. 本研究ではアプリケーション視点のモデルをシナリオフロー記述, システム構成視点のモデルをシステムフロー記述としてフローモデルの記述手法の提案を行う.

```
1   flow switch_to_led(){
2         Sys_SW::Detect_PSW
3         Sys_SW::Send_PSW
4         Sys_Disp::Rcv_PSW
5         Sys_Disp::Turn_LED
6    }
```

図 13　シナリオフロー記述

5.2.1　シナリオフロー記述

シナリオフロー記述ではアプリケーションが起動して終了するまでに発生するイベントを記述する. 図 1 の UI_System におけるシナリオフローの記述例を図 13 に示す. 記述例ではスイッチを押下して LED 点灯までの四つのイベントを発生順に記載している. これは期待値と実動作を比較することになるため, テストシナリオなどの検証に近いアプローチである. このような記述手法の利点としては期待するイベントが発生しなかった場合にエラー出力ができる, 評価したいイベントを任意に指定することができるなどが有る.

一方, 難点としては想定外のイベントが発生していた場合でも正常判定とする可能性があげられる. これはシナリオフローでは期待するイベントを選択的に評価する方式であり, 想定外のイベントの発生は考慮しないためである.

また, アプリケーションが複数のサブシステムに渡って動作するような場合に, 一つのフローに異なるサブシステムに所属するイベントを記述する必要がある. これは前提条件の “SS ベンダは他のサブシステムや分散システムのネットワーク構成を知ることができない” と合わせて考えると問題がある. SS ベンダは自身が開発したサブシステムの情報しか知ることが出来ないため, 異なるサブシステムが関与するフローを記述することができない. そのため, シナリオフローを記述可能な人物がセットメーカーに限定されてしまう. シナリオフローを記述するにはサブシステムの動作を理解する必要があるが, システム規模が増加するにつれて理解すべき分量も増加することになるため, モデルのスケーラビリティの面で問題があるといえる.

5.2.2　システムフロー記述

システムフロー記述ではサブシステムごとにフローモデルを記述することでファイルの分割と再利用を可能にする. システムフロー記述でのフローモデル構成例を図 14, フローモデル記述例を図 15-17 に示す. これまでは LED が点灯するシステムを例として説明を行ってきたが, 今回は押下したスイッチに応じて異なる振舞いをするシステムを

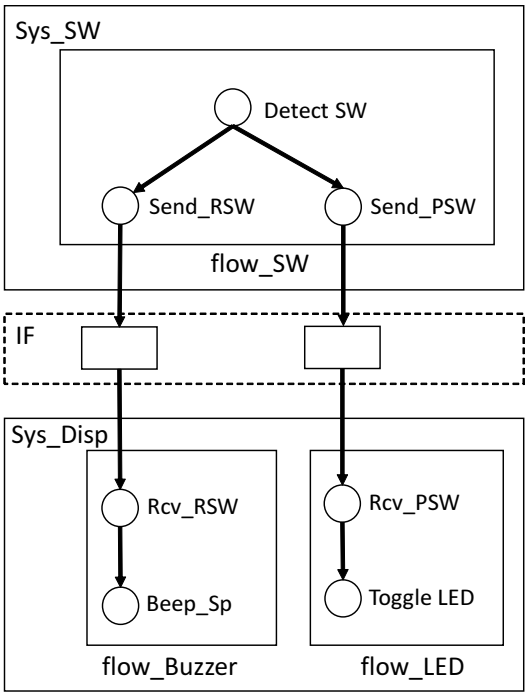

図 14　システムフローのアーキテクチャ

```
1   system Sys_SW{
2     flow SW(){
3       Sys_SW::Detect_SW
4       if(Sys_SW::Send_RSW){
5             load(IF, bind_r)
6       }else(Sys_SW::Send_PSW){
7             load(IF, bind_p)
8       }
9     }
10  }
```

図 15　Sys_SW のシステムフロー記述

```
1   system IF{
2     flow bind_r(){
3       load(Sys_Disp, Beep)
4     }
5
6     flow bind_p(){
7       load(Sys_Disp, LED)
8     }
9   }
```

図 16　IF のシステムフロー記述

```
1   system IF{
2     flow Beep(){
3       Sys_Disp::Rcv_RSW
4       Sys_Disp::Beep_Sp
5     }
6
7     flow LED(){
8       Sys_Disp::Rcv_PSW
9       Sys_Disp::Turn_LED
10    }
11  }
```

図 17　Sys_Disp のシステムフロー記述

考える.

図 14 のフローモデル構成例は上のブロックから Sys_SW, IF, Sys_Disp のモデルを表している. モデルの構築において Sys_SW, Sys_Disp のモデルを各サブシステムの開発元の SS ベンダが提供し, セットメーカがインターフェースのモデル (IF) を解して接続することを意図している. 図 15 が Sys_SW のフロー記述に該当し, 4 行目の Send_RSW 検索が成功すると IF モデルの bind_r を呼び出し, 6 行目の Send_PSW 検索が成功すると IF モデルの bind_P を呼び出す. 今回は説明のため 5 行目, 7 行目のフロー呼び出しでシンボル名を直に記述しているが実際の記述では抽象化を行い柔軟に定義できるようにする. Sys_SW のモデルから呼び出された IF のフローはそれぞれ対応する Sys_Disp のフローを呼び出す (図 16 3 行目, 7 行目). 最後に IF から呼び出された Sys_Disp のフローが所望のイベント検索を行って処理を終了する.

シナリオフロー記述では SS ベンダが提供するモデルを組み合わせて分散システムのモデルを構築することを想定する. この記述方法ではセットメーカーはモデルを構築した後に起点イベントが所属するフローを指定するだけで, Fitter が自動でログを探索してイベントの分類を行ってくれる. シナリオフローでは期待動作と振舞いの比較を行うため予想外のイベントが発生すると取りこぼす恐れがあったが, システムフローでは各サブシステムが取りうる振舞いとの比較を行うため, 予想外の動作が発生してもサブシステムとして定義された振舞いならばトレースを行うことができる. また, 開発元が作成したフローモデルを使用するため SS ベンダとのコミュニケーションが容易になるなどの利点がある. 難点としては動作トレースを行うだけなので後から期待動作と比較する必要がある, 不要なイベントも検索してしまうなどがある.

5.2.3　モデル記述方法の考察

提案手法ではモデルを使用した分類手法を提案し, 注目する対象によってシナリオフローモデルとシステムフローモデルで書き分けが必要なことを示した.

シナリオフローモデルは期待動作と実動作の比較が可能なため検証段階で利用できる. また, 任意のイベントを選択して評価できるため, Runtime Verification などの他の検証手法と組み合わせや SysML などのシステムエンジニアリングで用いるモデルと実測したイベントを比較するなどの応用利用が期待される

システムフローモデルはイベントログとモデルを比較することで, 各サブシステムが取る可能性のあるフローの中からもっとも実動作に近いフローを抽出する. 実動作に

添ったフローを抽出するためエラー解析などのデバッグ用途やシステム動作の調査など実機動作が不明の場合での利用が期待される. また, 実機動作からフローの自動抽出が可能なため, システムフローで抽出したフローに対して動作確認を行い, 確認済みのフローをシナリオフローとして連続稼働検証に用いる等の応用が考えられる.

両者のモデルは一長一短があるため用途によって使い分ける必要がある.

6. まとめ

分散システムの振舞観測の観測手法と分類手法の検討を行った. 時間同期と観測プログラムを利用した分散システムの観測手法を提案し, 試作評価によりシステム上で動作するアプリケーションの振舞いを物理的な区切りを超えて評価できることを示した. また, モデルを使用してしてイベントログから振舞いを抽出する手法を提案し, 性能評価やデバッグの自動化の可能性を示した.

今回の評価で行った試作評価は同期精度, ログバッファ容量等に制約があり実使用環境で想定される複雑な評価を行うことが出来なかった. 今後はより実機に近い評価環境を構築してより複雑な振舞いに対する観測を行い, 提案手法の有用性の実証を行っていく.

参考文献

[1] Peter H. Feiler, D. P. G.: *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, Addison-Wesley Professional (2012).

[2] Behrmann, G., David, A., Larsen, K. G., Hakansson, J., Petterson, P., Yi, W. and Hendriks, M.: UPPAAL 4.0, *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, QEST '06, Washington, DC, USA, IEEE Computer Society, pp. 125–126 (online), DOI: 10.1109/QEST.2006.59 (2006).

[3] Bartocci, E.: Sampling-based Decentralized Monitoring for Networked Embedded Systems, *Proc. of HAS 2013: the Third International Workshop on Hybrid Autonomous Systems*, vol. 124, Electronic Proceedings in Theoretical Computer Science, pp. 85–99 (online), DOI: 10.4204/EPTCS.124.9 (2013). talk: HAS 2013, Rome, Italy; 2013-03-17.