ソフトウェア工学 77－9
（1991. 2. 7）

分散制御システムのデバッグ手法
―動作仕様を用いたイベントヒストリの検査―

平井健治 杉本明 阿部茂
三菱電機（株）中央研究所

本稿では、実行時のイベントヒストリをシステムの動作仕様を用いて検査することによって、分散制御システムをデバッグする手法について述べる。動作検証用のモデルは、システム分析時に定義した要求仕様をペトリネットに変換し、ペトリネットのトランジションに検査すべきイベントを割り当てることによって生成する。イベントの割り当てには、要求仕様とソフトウエア設計仕様とのマッピング情報を用いている。本手法では、イベントヒストリを検証用ペトリネットで検査することにより、動作の実行順序の誤りや不当なデータアクセスによるエラーを検出することができる。

Debugging of Distributed Control Systems:
Checking an Event History with Behavioral Specifications

Kenji Hirai, Akira Sugimoto, Shigeru Abe
Mitsubishi Electric Corporation.

In this paper, we present a debugging technique based on checking an event history with behavioral specifications of distributed control systems. We use a Petri net to check the event history. First we translate requirement specifications into a Petri net. Next we assign events to each transition of the Petri net. In assignment of events we use the mapping information between requirement specifications and design specifications. By our method we can detect errors such as an illegal event sequence or an illegal data access.

1 はじめに

分散制御システムでは個々のタスク内の処理は単純であるため、動作エラーの多くはタスク間の相互作用の順序の誤りが起因となって発生している。従って、デバッグ時にはタスク間の時系列処理を追跡することが重要である。

分散制御システムのデバッグ手法としてイベントヒストリを用いたアプローチが知られている [1]。この手法では、対象システムの各タスクから収集したイベントヒストリ（実行履歴）を検査し、動作の実行順序の誤りなどによるエラーを検出する。

イベントヒストリを用いたデバッグでは、膨大な量のイベントヒストリからエラーに関係する部分を同定するための支援方法が問題になる。このためにブラウジング機能やリプレイ機能を用いてデバッグを支援するシステムや、イベントヒストリを検査用のイベント系列と比較することにより自動的にエラーを検出するシステムが開発されている [1]。

エラー検出を自動化するシステムでは、図 1に示すようにプログラマがシステムの動作仕様をイベント記述言語を用いて定義する。モニタシステムは、このように定義したイベントパターンの集合から動作モデルを生成し、イベントヒストリを動作モデルを用いて検査する。実行時のイベント発生順序がプログラマの定義した仕様を満たさない場合には、動作エラーが検出される。

この手法は、特定のタスク間の相互作用によって発生するエラーパターンをイベントヒストリから抽出する場合には有効である。しかし、検査用の仕様を全てイベントパターンとして定義しなければならないため、システム全体の動作を検査する場合には、プログラマに対する負担が大きい。

筆者らはこの問題を解決するため、システム全体の動作を検査するための動作モデルをソフトウエア開発過程で定義する情報から自動生成し、イベントヒストリを検査する手法を提案する。本手法の特徴を以下に示す。

- テスト用の動作仕様の定義を行なう必要がない
- システムの動作を要求仕様のレベルで検査できる

2 動作仕様を用いたデバッグ手法

イベントヒストリ検査用の動作モデルを生成するためには、検査の対象となるイベントとそれらの発生順序についての情報が必要である。従来のイベント検査方式では、これらの情報を全てイベント記述言語によって定義していた。

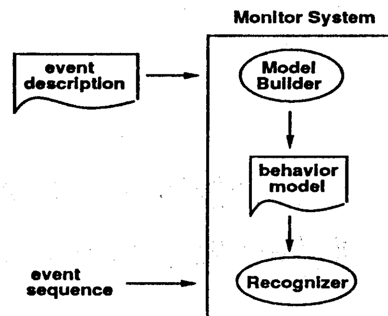

図 1: 従来のイベント検査方式

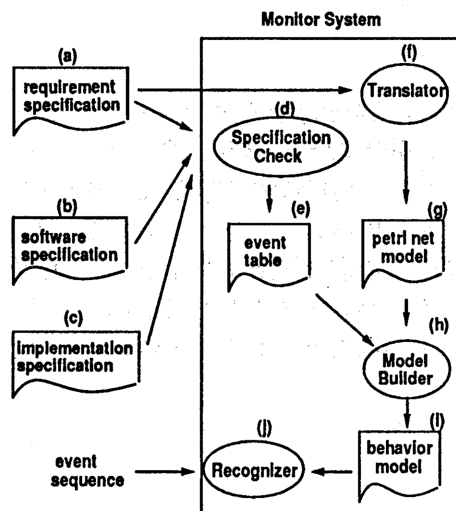

図 2: 動作仕様を用いたイベント検査方式

本研究では次のような標準的なプログラム開発過程で扱う情報を動作モデルの生成に用いる（図 2）。

(a) 要求仕様
システム内のデータや制御の流れを定義する。本研究では制御システムを対象とするため、要求仕様の定義手法として Ward[2] の変換スキーマを用いる。

(b) 設計仕様
要求仕様で定義した処理をグループ化し、タスクを割り当てる。また、これらのタスク間のインタフェースを定義する。

(c) 実装仕様
設計仕様で定義した各インタフェースの実現に用いたシステムコールの情報を記述する。

イベントの発生順序を検査するための検証用モデルとしては、並行システムの動作表現に適したペト

リネットを採用した。検証用ペトリネットは、以下の手順で生成する。

1. 要求仕様(a)をペトリネット (g) に変換する。
2. 設計仕様(b)と実装仕様(c)から検査の対象となるイベントを決定し、ペトリネットのトランジションに割り当てる。

このため、仕様検査部 (d) では、要求仕様と設計仕様で定義した情報との対応関係を検査し、イベントテーブル (e) を生成する。

動作検査部 (j) は、検証用モデル (i) を用いて実行時に各タスクから収集したイベントヒストリを検査し、実行順序の誤りによるエラーや不当なデータアクセスによるエラーを検出する。

3 検証モデルとイベント検査方法

3.1 ペトリネットモデルの生成

3.1.1 要求仕様の定義

Wardの変換スキーマでは、システムの要求仕様をデータや制御の流れを示すダイアグラムと各変換内の逐次処理を示す状態遷移図を用いて定義する。本研究では、データ、制御ダイアグラムから検証用のペトリネットを生成するためにWardの変換スキーマに以下の制限をつけている。

- 制御変換と状態遷移図を用いない
  各データ変換の処理は状態に依存しないように定義する。
- データバッファは容量1のものだけを用いる。

また、データアクセスの正当性を検査するために、データストアへのアクセス順序を指定するための表記法 (アクセス制御フロー) を導入している。

要求仕様の表記法を図 3にまとめる。

data transformation		data flow	
data store		continuous flow	
data buffer		signal flow	
external entity		access control flow	

図 3: 要求仕様の表記法

図 4に、鉄鋼プラント制御システムにおける実績データの収集及び送信機能を簡単化し、要求仕様を定義した例を示す。

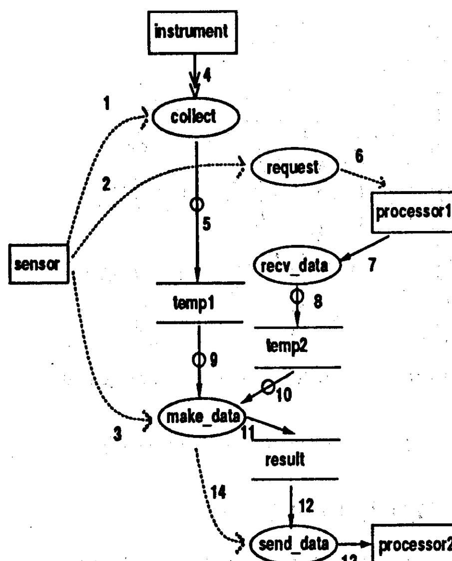

図 4: 要求仕様の記述例

3.1.2 ペトリネットへの変換

検証用ペトリネットは、各データ変換の処理をモデル化したものである。ペトリネットを生成するために、データ変換におけるフローの実行順序と各フローからペトリネットへの変換規則を次のように定義する。

[フローの実行順序]

1. アクティブな入力フロー (起動を伴う)
   シグナルフロー、他のデータ変換やエンティティからのデータフローが入力となっている場合
2. アクティブでない入力フロー
   上記以外のフロー (データストアからのフローなど) が入力として用いられている場合
3. 出力フロー

ただし、同じ種類のフローに関しては実行順序を規定しない。

[フローからペトリネットへの変換規則]

- データ変換からデータ変換へのフロー
  フローを2つのトランジションに対応させる。(出力側と入力側)
- データ変換以外のノードへのフロー
  フローを1つのトランジションに対応させる。

図 5に図 4の要求仕様から生成した検証用ペトリネットの一部を示す。要求仕様中の各フローは同じ番号のトランジションに変換されている。

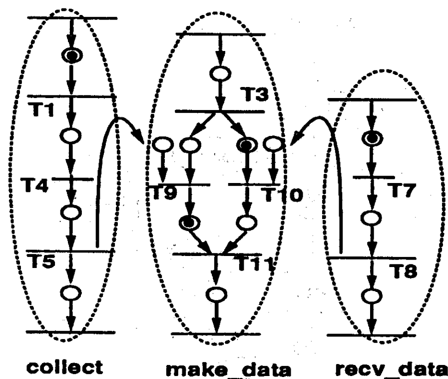

図 5: 検証用ペトリネットの生成例

3.2 検証用ペトリネット

ペトリネットは各プレース中のトークンによって実行が制御される。すなわち、発火可能（全ての入力プレースにトークンが存在し、出力プレースにトークンが存在しない）なトランジションを順に発火させ、マーキングを変化させる。

本研究では、ペトリネットを動作エラーの検出に用いるため、イベントによって実行制御をするように拡張している。すなわち、イベントヒストリ中の各イベントごとに発火すべきトランジションを決定し、イベント発生順序に従ってトランジションを発火させる。このため、検証用ペトリネットでは次の2種類のトランジションをもつ。

1. ノーマル型
   通常のトランジションでトークンによって発火の制御がされる。
2. イベント型
   トランジションに固有のイベント（設計仕様で定義されたイベント）によって発火の制御がされる。

イベント型のトランジションへのイベント割り当て方法については後述する。

3.3 イベント検査方法

検証用ペトリネットは、実行時に収集したイベントによって各トランジションを順に発火させることによって実行する。

イベント発生順序を検査するためにトランジションの発火条件と発火規則を以下のように定義する。

発火条件 全ての入力プレースにトークンが存在し、出力プレースにトークンが存在しない

発火規則 ノーマル型のトランジションが上記の発火条件を満たす場合には、そのトランジションを発火させる。イベント型のトランジションに固有のイベントが発生した時に、そのトランジションが上記の発火条件を満たすときは、そのトランジションを発火させる。

また、ペトリネットにおける動作エラーを以下のように定義する。

動作エラー イベント型のトランジションに固有のイベントが発生した時に、そのトランジションが発火条件を満たさないときは動作エラーとする。

イベント発生順序に誤りがある場合には、対応するトランジションが発火できないため動作エラーが検出される。

4 ペトリネットへのイベント割り当て

4.1 設計仕様

設計仕様では、システムを実現するためのソフトウエア構成をタスク、ファイル、外部機器の3つの要素を用いて定義する。

要求仕様の情報はグループ化されてタスクやファイルにまとめられるため、これらの要素間で新たなインタフェースの定義が必要になる。インタフェースの定義には、要求仕様と同様の表記法を用いた定義や、実現時の環境を考慮したより具体的な定義など種々のレベルの定義手法が考えられる。

本研究では、インタフェースの定義に用いるリンクを次のように分類する。

1. タスク通信リンク
   2つのタスク間でメッセージ、キューなどを用いた通信があることを表す。
2. タスク操作リンク
   タスクの起動／終了や停止／再開などの操作があることを示す。
3. デバイス操作リンク
   タスクがファイルや外部機器との間で入出力処理を行なうことを表す。

4.2 イベントの割り当て方法

前述したインタフェースの実現手法は、OSでサポートするシステムコールの種類などによって異なる。従って、ペトリネットへのイベント割り当ては、各インタフェースの実現手法と、インタフェースと

要求仕様の情報との対応関係から決定することが必要である。

4.2.1 イベント

モニタシステムでは、各インタフェースの実現に用いたシステムコールの実行履歴をイベントとして収集する。各タスクで検査の対象となるシステムコールが正常に終了した場合をイベント発生とする。モニタシステムへのイベント送信処理は、システムコールに対するスタブ関数により行なわれ、次の情報が送信される。

"イベント名：引数 in 発生タスク"

4.2.2 要求仕様と設計仕様とのマッピング

設計仕様では、要求仕様のノードから設計仕様のノードへのマッピング情報を定義する。要求仕様のフローから設計仕様のインタフェースへのマッピング情報は、各フローの両端ノードのマッピング情報から決定する。

両端ノードが同一タスクにマッピングされている場合にはインタフェースは存在しない。このように要求仕様中には、設計仕様に対応するリンクが存在するフローと存在しないフローがある。設計仕様に対応するリンクが存在しないフローは検証用ペトリネットでノーマル型のトランジションとして扱い、対応するリンクが存在するフローはイベント型のトランジションとして扱う。

各インタフェースにマッピングされる要求仕様のパターンによってペトリネットへのイベントの割り当て方法を分類する。

1. タスク → ファイル or 外部機器
   デバイス操作リンクにマッピングされる要求仕様のパターンは次の2種類である。ただし、各ケースともデータ変換はタスクにマッピングされるものとする。また、矢印はフローを表す。

   (a) データ変換 → ストア
   　　(ストアがファイルにマッピングされる場合)

   (b) データ変換 → 外部エンティティ

   各フローに対応するトランジションに対してイベントの割り当てを行なう。

2. タスク A → タスク B
   タスク通信リンクまたはタスク操作リンクにマッピングされる要求仕様のパターンは、次の3種類である。ただし、各ケースともデータ変換Aがタスク A、データ変換Bがタスク B にマッピングされるものとする。

   (a) データ変換 A → データ変換 B

   (b) データ変換 A →バッファ X →データ変換 B

   (c) データ変換 A →ストア X →データ変換 B
   　　(ストア X がタスク A にマッピングされる場合)

   (a)、(b) は、関連する2つのトランジションにイベント割り当てを行なう。
   (c) の場合は、"X → B" のフローについてだけイベント割り当てを行なう。

4.3 イベント割り当ての例

4.3.1 設計仕様と実装仕様の定義

設計仕様の定義に図 6の5種類のインタフェースを用いた場合を例として、ペトリネットへのイベント割り当て方法について述べる。これらのインタフェースは、文献 [3] を参考にしている。

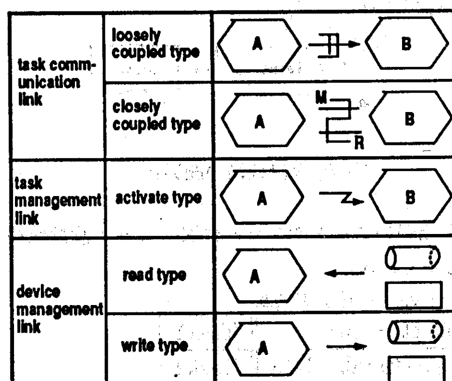

図 6: タスク間インタフェースの種類

各インタフェースは、タスク起動 (fork)、入力 (read)、出力 (write)、メッセージ送信 (send)、メッセージ受信 (recv)、応答待ち (wait) の6種類のシステムコールで実現するものとする。インタフェースとその実現に用いるシステムコールの関係を以下に示す。

1. 疎結合型通信 (メッセージキューを介した通信)
   送信側 (TaskA)　受信側 (TaskB)
   send:TaskB　recv:TaskA

2. 密結合型通信（メッセージの送信と応答の受信）

送信側（TaskA）	受信側（TaskB）
send:TaskB	recv:TaskA
wait:TaskB	send:TaskA

3. タスク起動操作
   fork:TaskB in TaskA

4. デバイス入力（ファイルや外部機器からの入力）
   read:X in TaskA

5. デバイス出力（ファイルや外部機器への出力）
   write:X in TaskA

4.3.2 イベントテーブルの生成

図 7に図 4に対する設計仕様の例を示す。

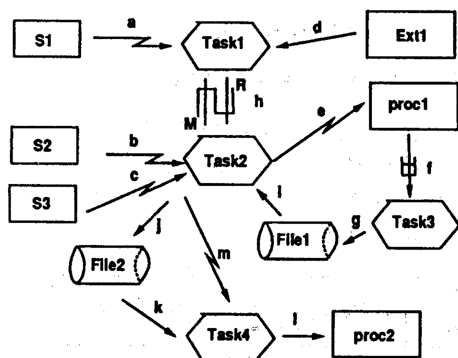

図 7: 設計仕様の記述例

要求仕様のノードやフローは次のように設計仕様に対応している。

ノードのマッピング情報

collect,temp1	Task1
request,make-data	Task2
recv-data	Task3
send-data	Task4
temp2	File1
result	File2

フローのマッピング結果

要求仕様	1	2	3	4	6	7
設計仕様	a	b	c	d	e	f

要求仕様	8	9	10	11	12	13
設計仕様	g	h	i	j	k	l

表の上段は図 4のフロー番号、下段は図 7のリンク番号を表す。フロー 5 は、両端ノードが同一タスクにマッピングされているため、対応するリンクが存在しない。

前節で定義した各インタフェースに対するイベントの情報とこのマッピング情報からイベントテーブルを生成する。密結合型のインタフェース h には 1 つしかフローが対応していないので、wait イベントだけを割り当てている。

リンク	イベント名	トランジション
a	fork:Task1 in S1	T1
c	fork:Task2 in S3	T3
d	read:Ext1 in Task1	T4
f	recv:proc1 in Task3	T7
g	write:File1 in Task3	T8
h	wait:Task1 in Task2	T9
i	read:File1 in Task2	T10

5 エラー検出例

図 8 に図 5の検証用ペトリネットを用いてイベントヒストリを検査した結果を示す。イベントの発生順序に従って対応するトランジションが発火している。図のイベントヒストリでは 7 番目のイベントの検査時に、トランジションＴ１０の発火に失敗し、エラーが検出されている。これは、要求仕様で定義したデータストア temp2 に対するアクセス順序が満たされなかったことに対応している。

```
1.fork:Task1 in S1         ->T1 fired
2.read:Ext1  in Task1      ->T4 fired
                           ->T5 fired
3.fork:Task2 in S2
4.send:proc1 in Task2
5.fork:Task2 in S3         ->T3 fired
6.wait:Task1 in Task2      ->T9 fired
7.read:File1 in Task2      ->error in T10
```

図 8 ：イベントヒストリの検査例

6 おわりに

本稿では動作仕様を用いた分散制御システムのデバッグ手法について述べた。実行時のエラーを動作仕様レベルで確認するためのビジュアルモニタも試作完了している。今後の課題として分散環境でのイベント到着順序についての問題が残されている。

参考文献

[1] Mcdowell,C.E and Helmbold,D, “Debugging Concurrent Programs”, ACM Computing Surveys,Vol.21, No.4,Dec 1989,pp593-622

[2] Paul T. Ward,The Transformation Schema: IEEE Tr.SE 12-2,1986,pp198-210

[3] H.Gomaa, “A Software Design Method for Real-Time Systems” C.ACM ,vol27,No.9,pp938-949,Sep,1984