ソフトウェア工学 122−19
（1999．3．19）

組込み制御用ソフトウェア次世代テスト・デバッグシステム

岡本 渉　田村 文隆　川田 秀司　内平 直志
株式会社 東芝　研究開発センター

組み込み制御用ソフトウェアは通常、並行プログラムとして記述される。並行プログラムに対するテスト・デバッグの難しさは、ある操作に対して一度テストを行って、不具合が発生しなかったからといって、その操作の正当性が保証されないという点にある。また不具合の原因究明においても不具合が再現されにくいという点が大きな問題である。本稿ではこのような「再現性のない不具合」と呼ばれる並行プログラム特有の問題点を解決するために開発された、Java を対象としたテスト・デバッグ支援システムの概要、及びその背景にある技術について述べる。

The testing and debugging system for embedded systems using concurrent programs

Wataru Okamoto, Fumitaka Tamura, Hideji Kawata and Naoshi Uchihira
TOSHIBA Corporation　Research and Development Center

Control software for embedded systems is usually written as a concurrent program. Concurrent programs are non-deterministic, and testing and debugging for them is generally more difficult than for sequential programs. This non-deterministic behavior causes a big problem, in that the defects don't always appear in testing or debugging even if the concurrent programs contain them. Therefore it is very difficult to reproduce such defects in debugging process and it takes a long time to detect the defects. To resolve this problem, we have developed a system for testing and debugging Java concurrent programs. In this paper, we present the outline and technical aspects of this system.

1　はじめに

組み込み制御用のプログラムはその多くが並行プログラムとして記述される。並行プログラムでは、複数のタスクあるいはスレッドが絡み合って、ひとつのプログラムを構成しており、それらが１つのオブジェクトを共有する場合なども多い。そのため、スレッドがオブジェクトにどのようなタイミングでアクセスするかなどによって、プログラムの処理が変化する可能性を含んでいる。

このように並行プログラムでは逐次プログラムとは異なり、入力される値だけでなく、そのタイミングによっても動作が異なるという特徴がある。そのため、ある不具合の発生が報告されたとしても、各スレッドがどのような状況で不具合が発生したのかなど、まずその不具合を再現するだけでも苦労しているのが実状である。このような不具合は「再現性のない不具合」（厳密には再現しにくい不具合）と呼ばれ、並行プログラムの特徴のひとつとなっている。これらの不具合を発見し、原因を究明するためにはスレッドが実行されるタイミングも含めたテストが必要である。本稿で紹介する次世代 Java テスト・デバッグ支援システムは、これらの並行プログラム特有の不具合を発見し、原因究明することを目的として開発されたシステムである。このシステムの特徴は、まず第一に並行プログラムに存在する複数のスレッドをユーザ側で自在にコントロールできる点である。この機能により、再現が難しかった不具合も容易に再現することが可能で

ある。また、第二の特徴として自動網羅テスト機能がある。再現性のない不具合は、タイミングの違いによってプログラムが取り得る状態すべてをチェックすることで発見できるが、プログラムが取り得る状態数はあまりにも多く、計算機の能力を超えてしまっている。したがって、この手法を実際に適用するときには状態数を減らすための、言い換えれば、既にテストした状態と同じとみなせる状態はテストしないための機構が必要である。自動網羅テスト機能には、この機構が実現されており、効率的な不具合発見を行うことができる。

本稿ではまず最初に並行プログラムに対するテストの現状および問題点を述べ、その後、本システムの特徴である、マルチスレッド対応デバッグ機能と自動網羅テスト機能について例を交えながら述べる。なお、本システムは Java で記述されたプログラムを対象としているが、その理由として、1) Java は embedded Java や現在仕様が検討されている Realtime Java など、将来組み込み分野での使用が拡大していく可能性が高い、2) 言語仕様そのものが並行プログラムの作成を容易にしている、といった点が挙げられる。

2 並行プログラムに対するテストの現状および問題点

並行プログラムのテスト・デバッグの問題点は、逐次プログラムとは異なり、不具合に再現性のない場合がある点である。これは並行プログラムでは、存在する各スレッドが実行されるタイミングによって動作が変化する可能性があるためである。従って、そのテストではタイミングも含めた検証が必要となるが、テストケースとして考えられる数は膨大であり、まともに扱うことは現実的に不可能である。このような背景から、制御用プログラムでは、同期命令を多用するなどタイミングによる不具合を避けるための工夫がなされていることもあるが、逆にデッドロックといった問題点が発生する危険も多く、テストの問題はますます深刻なものとなっている。

ここで並行プログラムのテストがいかに困難なものであるかを示すため、簡単な例を示す。図 1. のよう

*P*1 (優先度 高)	*P*2 (優先度 低)
`1: b = 2;`	`1: wait (2s)`
`2: a = 2;`	`2: c = a + b;`
`3: input b;`	`3: output c;`

図 1: 並行プログラムの例。各スレッド *P*1、*P*2 で用いられている変数は共通とする。

な 2 つのスレッドからなる並行プログラムに対し、例えば 1 を入力するケースを想定してみよう。この並行プログラムは *P*1 のほうが *P*2 よりも優先度が高いため、*P*1 が入力待ちになるまで実行される。入力待ちの間、優先度の低い *P*2 が実行されるわけだが、問題は *P*2 がどこまで実行された段階で *P*1 への入力が行われるか、である。例えば *P*2 の 2 行目が実行される前に *P*1 への入力が行われれば、*P*2 からの出力値は 3 となる。逆に *P*2 の 2 行目が実行された後に入力が行われると、*P*2 は 4 を出力することになる。

このようにどのタイミングで入力するかによって、出力が異なるというのは並行プログラムのテストにおける本質的な問題点である。つまり、ある入力値をもったテストケースに対して一度テストを行い、正常な動作を確認できたとしても、それは再び同様のテストケースで正常に動作することを保証するものとはなり得ない。また、このような不具合はタイミングによるものであり、仮に不具合が存在することがわかったとしても、なかなかそれを再現できないでいるのが現状である。

こういった状況に対して、あらゆるタイミングによるプログラムの振舞いすべてをテストすればよいと考えるかもしれない。しかし、存在し得るプログラムの振舞いとは膨大な数である。例えば並行に実行され得る $P_1 \sim P_{10}$ の10個のスレッドを持ったプログラムがあるとする。10個のスレッドすべてが実行可能だとすると、プログラムの動作として考えられるパターンの数は、10の順列であるから、$10! = 3628800$ 通りとなる。すなわち、このような極めて単純なプログラムに対してでさえ、3628800通りのテストを行わなければ、完全なテストを行ったことにはならないのである。

上に示したように並行プログラムには様々な実行パターンがあり、そのことが原因で不具合が生じる可能性がある。さらにテストをするにも実行パターンの数が多すぎるため、どのようにテストを行ってよいのかすら模索状態となっている。

3 テスト・デバッグシステムの概要

ここで紹介する次世代 Java テスト・デバッグ支援システムは、Java で記述された組み込み用並行プログラムをターゲットとしたテスト・デバッグ支援システムである。本システムの最も大きな特徴としては以下の二点が挙げられる。

1. 並行プログラム対応デバッグ機能
2. タイミング自動網羅テスト機能

本システム開発の主な動機は、並行プログラムに含まれる不具合を発見し、その原因を究明することである。この点から見ると、1. は不具合の原因究明、2. は不具合の発見に対応していると言えよう。

本システムの第一の特徴である並行プログラム対応デバッグ機能は、従来のデバッガの機能に特に並行プログラムをデバッグするための特殊機能を付け加えたものである。この機能には、スレッド単位のプログラム制御、スレッド間の相互作用を視覚的に認知できるタイムチャート機能などが含まれている。第二の特徴であるタイミング自動網羅テスト機能は、登録されたプログラムの振舞いをもとに、スレッド実行のタイミングの違いが原因で生じる、プログラムの異なる振舞いを自動的に検出する機能である。次節以降ではこれらの特徴について詳しく述べる。

4 並行プログラムに対するデバッグ手法

従来のデバッガは逐次プログラムに対するデバッガの延長線上で作成されているため、想定する状況へプログラムを誘導することが難しく、並行プログラムの作成者にとって決して使いやすいと言える環境ではなかった。本システムでは、このような問題点に対し、「スレッドの細かなコントロール」、「実行状況のビジュアル表示」を基本方針としてデバッグ環境を提供している。

4.1 スレッド単位の制御

並行プログラムをデバッグするには、各スレッドを完全にユーザ側でコントロールできる環境が望ましい。この理由は、前述したような再現性のない不具合は、まず各スレッドをいかにして不具合が発生する状況に導くかが困難だからである。従来のデバッガでは、そのようなユーザが望む状況にプログラムを導くのが難しく、またその状況に導けたとしても多くの操作が必要であった。その問題に対し、本システムでは「スレッド間の排他実行機能」を実現し提供している。スレッド間の排他実行機能とは、ユーザプログラム実行中の任意の時点において存在する複数のスレッドのうち、ユーザの選んだ特定のスレッドのステップ

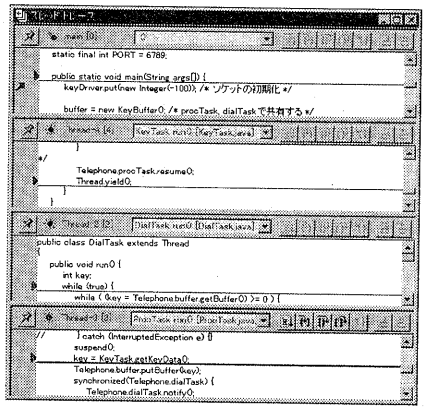

図 2: スレッド トレース ウィンドウ。ユーザがスレッドを自由に選択して排他的にそのスレッドを実行させることができる。

実行のみを行い、その他のスレッドの実行を再開させないという機能である。GUI においては、個々のスレッドに対し各ウィンドウが割り当てられ、それぞれ独自にステップ実行することができるというものである (図 2. 参照)。この機能により、プログラムの状態をユーザの希望通りに導くことが可能となり、再現性のない不具合も再現することができる。

4.2 タイムチャート機能

スレッド単位のプログラム制御に加え、各時点で各スレッドがどのように他のオブジェクトにアクセスし、どのような影響を他のスレッドに及ぼしているかを知ることができると便利である。そのような視点から我々は本システムに「タイムチャート機能」と呼ばれる機能を導入した。タイムチャート機能とは横軸に時間をとり、プログラム中の各スレッド間の関係を示した図である。このチャートはデバッガ実行とともに動的に作成され、各スレッドがどのオブジェクトへアクセスしているのかといった情報や、スレッドがその時どのような状態 (run, sleep, suspend など) にあったかなどを知ることができる。

図 3. はタイムチャートを模した図である。この図において太い実線で描かれた鍵型の線はスレッドを表しており、点線はその他のオブジェクトを表している。スレッドの鍵型はオブジェクトへのアクセスを意味しており、またスレッド間に垂直に引かれた矢印はスレッド間の相互作用を表している。これらにより、「resume をかけた」あるいは「このオブジェクトへアクセスした」などの情報を視覚的に得ることができる。

この図で扱っているサンプルは電話機のプログラムを例にとったもので、*keyTask*、*dialTask*、*procTask* という 3 つのスレッドが存在している。*keyTask* はキー入力を受け付けるスレッド、*dialTask* は外部にパルス信号を出力するスレッド、*procTask* は *keyTask* と *dialTask* の橋渡し的な役割を持ち、入力値を読みに行き、それを出力値として *dialTask* に与えるスレッドである。このサンプルには不具合が含まれており、それは “1, 2” という入力があるにも関わらず、出力は 1 を落として 2 しかない、というものである。この原因は、2 つのスレッド *keyTask* 及び *procTask* が共有しているオブジェクト *key* に同時アクセスして共有変数を上書きしてしまったことにある。このような共有変数、共有オブジェクトへの同時アクセス、またはアクセスの順序がおかしいなどの状況は、タイムチャートの機能を用いれば一目瞭然で知ることが

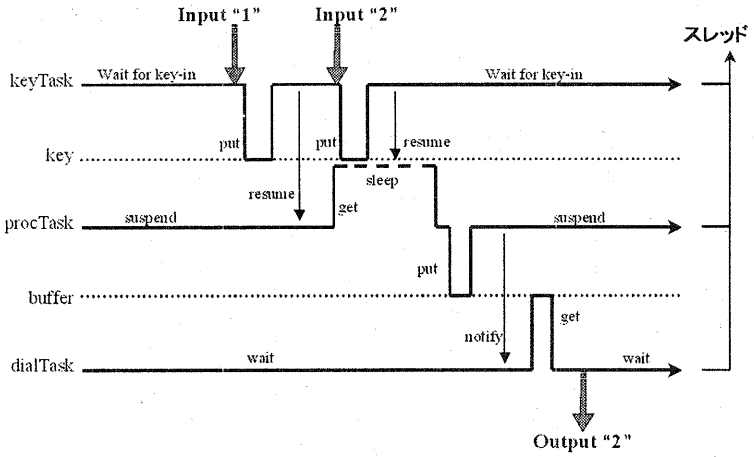

図 3: タイムチャートを示した模式図

でき、不具合の原因究明に大きく役立つと期待される。

5 並行プログラムの自動網羅テスト機能

並行プログラムに不具合が存在することがわかっているが、その原因が突き止められないといった場合には、上に記した並行プログラム対応デバッグ機能を用いて、調査することができる。では、タイミングによる不具合があるかどうかわからない場合にはどのようにテストを行えばよいだろうか。それに対する本システムの解答として「タイミング自動網羅テスト機能」がある。この機能は、タイミングの違いによって発生する再現性のない不具合を発見するために導入された機能で、グローバル状態遷移解析技術 ([1],[2]) を基礎として実現している。詳しくは後述するが、この機能はある登録されたプログラムの振舞い (「種シナリオ」と呼ぶ) をもとにして、タイミングの違いが原因でプログラムにどのような異なる振舞いが存在するかをユーザに提示してくれる機能である。

グローバル状態遷移解析の基礎となる概念は、すべての動作を状態遷移図の形式で生成し、それをチェックするというものである。これはタイミングの組み合わせによるプログラムの振舞いをすべて列挙することを意味し、確かにこの手法を用いれば、実際に発生し得る処理の流れすべてを把握することが可能である。しかしながら、現実問題として、タイミングの組み合わせは無限に近い数が存在し、到底すべてをチェックすることは不可能である。これに対し、近年ではすべての状態遷移を調査したものと同等の結果を導けることを保証しつつ、いかに小さな状態遷移図を生成するかといった研究が盛んで半順序法 ([3]) やスリープセット法 ([5]) といった手法が提案されている。

5.1 枝刈りアルゴリズム

本システムではタイミングによる不具合を発見するための手法としてグローバル状態遷移解析技術を基本にしているが、その状態数の爆発を回避する方法として、「プログラムの振舞いとして同等とみなせるものはできる限り１つの動作パターンに帰着させる」枝刈り手法を実装している (１つの動作パターンを “枝”

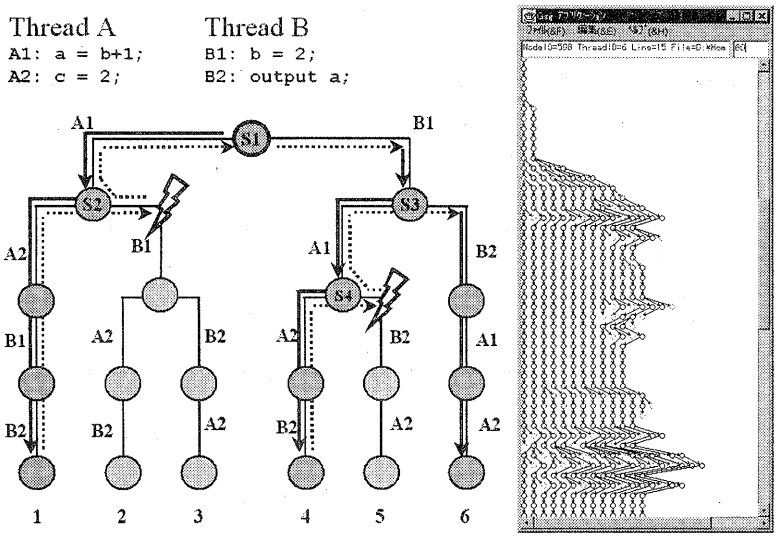

図 4: 状態遷移図

と呼び、複数の動作パターンを１つのパターンに帰着させる様を “枝を刈る” と表現する)。我々のシステムで用いられている枝刈り手法は、枝刈りのための情報となる変数間の依存関係を、あらかじめ解析してから遷移図を生成するというのではなく、依存関係を集めながら動的に遷移図を生成するという点に特徴がある。この手法は生成にオーバーヘッドがかかるが、より正確な依存関係を知ることができ、そのため枝刈りの効率がよい。Java など参照型変数が多く使用される言語では依存関係を特定するのが困難となっており、しかもその上にデバッガを動作させるため１つの枝を作るのに時間がかかる。そのような背景もあり、我々のシステムで用いられている動的な枝刈り手法は大きな効果が期待される。

本システムで実装されている動的枝刈り機構を単純な例を用いて説明する。図 4. は A、B ２つのスレッドから構成されるプログラムの状態遷移図である。それぞれのスレッドのプログラムは図に併せて示している通りである。また、その中で用いられている変数はすべて共通で、優先度は同じとする。状態遷移図を見ると明らかなように、プログラムの動作として考えられるパターンは６通り存在する。結論から先に述べると、我々の枝刈り方式を適用すれば、６通り中３通りだけを調べることによって、６通りすべてをチェックしたのと同等の結論が得られる。

では、動的枝刈り機構はどのような手順で、６通りの実行パターンを３通りに帰着させたのだろうか。枝刈り手法の本質的な概念は「依存関係のない命令は実行順序に依らないので、どちらか一方の順序に従ってのみテストすれば十分」というものである。図 4. のサンプルにしたがって見てみよう。このサンプルでは優先度は A、B のスレッドともに同じと仮定しており、どちらも実行可能な場合はスレッド A をとりあえず実行するとしている。まず初期状態 $S1$ からスレッド A の最初の命令 $A1$ を実行する。さらに、$A2$、$B1$、$B2$ と実行していき、最初の実行パターンが完了し、枝 1 が作られる。次に枝刈り機構は選択肢がある位置まで戻る。この例では $A2$ か $B1$ かを選べる状態 $S2$ まで戻る。さてここで、枝 1 において $A2$ はそれ以降の命令 $B1$、$B2$ とも依存関係を持っていないことが判明している。したがって、$A2$ は $B1$、$B2$ の後から実行しても結果に影響しない。逆に言えば、$B1$、$B2$ を $A2$ より先に実行したとしても結果は今、実

行したパターンと変わらないと言うことができる。これらの理由から、$B1$ を実行するパターン (枝 2、3) は枝 1 と同等とみなすことができ、枝 2、3 を切り落とすことができるのである。

状態 $S2$ から $B1$ を実行するという枝は刈ることができたので、さらに上の状態、すなわち初期状態 $S1$ へ戻る。今調査してきたパターンは $A1$ から実行されるものだったが、その命令 $A1$ は $B1$ との依存関係を持つことがわかっている。したがって、状態 $S2$ のように枝刈りを行うことは不可能であり、$B1$ から始まるパターンも実際に調査していかなくてはならない。今までと同様にとりあえずスレッド A を優先的に、プログラムが停止するまで実行する。停止したら選択可能な状態にまで溯り、依存関係を判定し、別の選択肢へ進むかどうかを決定していく。この手順により $\{B1, A1, B2, A2\}$ という実行パターン (枝 5) も切り落とすことができ、調査すべきなのは3つのシナリオだけだという結果が得られるのである。

このように我々のシステムではプログラムを実行しながら動的に、その依存関係を調べ、調査すべき必要のない枝を刈っていく。この方法を用いることにより遷移図の作成に時間はかかるが、効率的に枝を刈ることができ、結果的に小さな状態遷移図に抑えることができるのである。

5.2 自動網羅テスト機能を用いたテストサイクル

この節では、自動網羅テスト機能の実際の使用法も含めて、本システムを用いた並行プログラムの開発サイクルについて述べる。図 5. はその概念図である。まずユーザは作成された並行プログラムに対して種シナリオを作成する。種シナリオの登録とは、プログラムに存在する複数のスレッドを逐一、排他的に実行させ、その実行順序や入力値などをシステムに記憶させる作業である。この種シナリオに対して自動網羅テスト機能を実行させると、種シナリオと異なる振舞いをするシナリオをユーザに報告する。ただし、システムが提示するのは「タイミングによって異なる振舞いをするものが何通りあります」という情報だけなので、それらの振舞いが仕様にあっているか、あるいは不具合なのかはユーザが判定するところである。この判定には、自動テスト機能の一部である「シナリオ再生・検証」機能を動作させる。この機能は登録されたシナリオや自動生成されたシナリオをユーザに示す形で実行するもので、どのようなシナリオが生成されたのか、すなわちどのような順序でスレッドが実行されたのかという点が確認できるようになっている。

この段階で何らかの不具合が発見された場合には、それに対処するようプログラムを修正しなければならない。不具合の原因究明の際には、タイムチャート機能などが非常に役に立つだろう。プログラムの不具合を修正した後は再び種シナリオを登録し、同じ手順を繰り返すことになる。もし自動生成されるシナリオが存在しない場合には、種シナリオで与えた入力パターンに関しては、タイミングによる不具合は存在しないことが保証される。その後は逐次プログラムに対するテスト・デバッグ工程と同様に次の入力パターンに対するテストへと移行する。

逐次プログラムのテストではテストケースとしてのいくつかの入力パターンを与えるが、本システムについてもそれは同様である。本システムが保証するのは、与えられた入力パターンに対してタイミングによる不具合がないことであり、それ以外の入力パターンに対して不具合がないことを保証するものではない点には注意すべきである。

6 まとめ及び今後の展望

本システムは並行プログラムの再現性のない不具合の撲滅を目的として開発されたシステムであり、不具合の発見、さらにその原因究明を支援する機能を持つ。不具合の発見に関しては動的枝刈り機構を持つ自動網羅テスト機能、不具合の原因究明に関しては並行プログラムに特化した操作性を持つデバッグ機能が

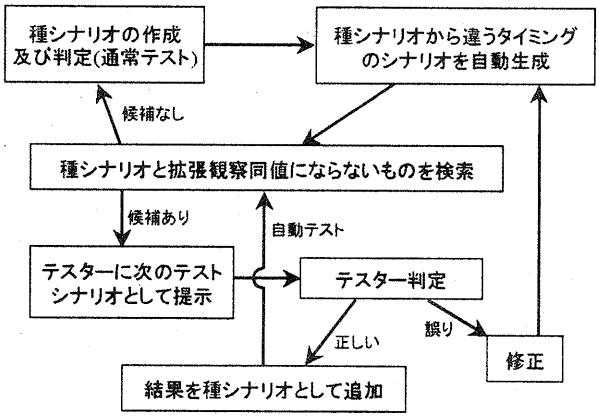

図 5: 自動網羅テスト機能を用いた並行プログラムの開発サイクル

大きな役割を果たす。これらの機能により、今まで困難を極めていた並行プログラムのテスト・デバッグ作業は軽減されると思われる。

次に今後の展望について述べる。まず、このシステムには実時間に関する仕様をプログラムが満たすかどうかといった評価を行う機能は備わっていない。組み込み制御用プログラムなどでは、例えば、あるスレッドが正しく一定時間おきにある仕事を行っているかといったチェックがしばしば重要となる。我々のシステムを組み込み分野で使用することを想定した場合、このような実時間に関する仕様のチェック機能を追加することが重要と思われる。

また、本システムの特徴である動的枝刈り機構を備えた効率的なグローバル状態遷移解析を用いたとしても、対象となるユーザプログラムが大きくなると、それが取り得る状態数は指数関数的に増加し、現実的にテスト不可能となってしまう。このような問題に対する根本的な解決策は未だ得られていないが、少なくとも、より効率的な枝刈りの方式を導入することは必要となると思われる。

参考文献

[1] J.R. Burch, E.M. Clarke, *et al.* , Symbolic model checking, 10^{20} states and beyond, In Proc. LICS (1990).

[2] E.M. Clarke *et al.* , Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications, ACM TOPLAS, Vol.8, No.2 (1986).

[3] P. Godefroid & P. Wolper , Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties, *Proc. CAV'91*, LNCS Vol.575, Springer-Verlag (1991).

[4] N. Uchihira and H. Kawata , 'Scenario Based Hypersequential Programming', 2nd International Workshop on Software Engineering for Parallel and Distributed Systems, IEEE Computer Society Press, (1997).

[5] P. Wolper & P. Godefroid , Lecture Notes in Computer Science, Springer-Verlag, Hildesheim, August, (1993).