B-022

ソースコード解析を利用したモデル検査に基づく欠陥抽出手法による組込みシステムの検証

Verification of Embedded System by A Method for Detecting Defects in Source Codes Using Model Checking

青木　善貴†　　松浦　佐江子†
Aoki Yoshitaka　　Matsuura Saeko

1. はじめに

プログラムは開発開始から使用停止までの使用期間，多くの担当者により修正を加えられる．担当者が変わる度に引き継ぎが行われるが，資料の不備・時間的制約等により仕様が全て正確に伝わる訳ではない．担当者は使用期間の後期になるほど，一人が複数のプログラムを受け持つことが多くなるため，一つ一つのプログラムの仕様の完全な理解は難しく，部分的な理解しかできないことになる．

そのような仕様理解の担当者は，不具合が発見された場合，全体的なプログラムの整合性を把握できないため，不具合を限定された条件においてのみ確認し，プログラムに不具合回避目的のみの修正を行う可能性が高い．そのような修正を行われたプログラムは一見正しく動作するが別の不具合を内在することが多い．

こうした「元の仕様を理解しないまま追加された新たな仕様」の部分が引き起こす不具合は，原因の想定が困難なため不具合を発見するには，しらみつぶし的な調査・テストが必要になる．これには多大な労力を要するため，システム開発作業に多大な影響を与える．また不具合の再現及び修正後の不具合解消確認も困難である．

近年，システム開発の上流工程において，システムが取りうる様々な状態の組み合わせを網羅的にチェックして，満たすべき状態に到達するかを検査するモデル検査が注目されている．

私たちは上記の内容を踏まえ，モデル検査手法を用いて稀有な条件で発生する不具合を発見する手法「モデル検査に基づく欠陥抽出手法」を提案してきた[1][2][3]．これは個々のロジックは簡単明瞭であってもそれらの種類・実行順序・実行条件が多岐にわたるシステムを想定しており，「設計者の意図しないロジックの組合せ」のために発生する不具合原因の発見及び不具合解消の確認を行うものである．

今回，この提案手法が別の特徴を持つシステムにも適用できるかを検討する．組込システムは外部環境の影響のうけるため，動作確認は実際に実機で確認するしか方法がない．しかしハードウェアの特性上，外部環境との間に誤差が生じるため動作結果が常に同一とはならず，不具合の原因を捉えるのは難しい．外部環境をモデル化する工程を付加することにより，本提案手法で組み込みシステムの不安定な動作の原因を検査できるかを検証する．本稿では，Java プログラムを対象として，本手法の手順とその支援ツールについても報告する．

†芝浦工業大学大学院　電気電子情報工学専攻
Graduate School of Engineering, Shibaura institute of technology Department of electronic engineering and computer science

2. モデル検査に基づく欠陥抽出手法の概要

2.1 手法概要

モデル検査のテクニックを使用することによりレビューやテストでの発見が困難な欠陥を抽出する手法である．また本手法のために作成したサポートツールは，モデル検査ツール UPPAAL によりシミュレートされるモデルにソースコードを翻訳するための機能を提供する．

2.1.1 モデル検査ツール

本手法では，モデルがグラフィカルに作成できる点と反例・シミュレーションがグラフィカルに確認できる点を重視してモデル検査ツール UPPAAL [4] を使用する．UPPAAL のモデルは，図 1 のように，システムの状態を表すロケーションとロケーションを結ぶ遷移で構成される．

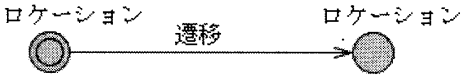


図 1 ロケーションと遷移の構成例

2.1.2 モデル化の方針

「設計者の意図しないロジックの組合せ」による不具合は”条件分岐”及び”繰り返し”処理の両方もしくはどちらかで不具合が発生していると考えられる．本手法では，ソースコードを UPPAAL モデルに変換する際には，”条件分岐”及び”繰り返し”の制御フローの構成要素に基づきモデル化する．

2.1.3 抽象化について

本手法では，既存のソースコードをリバースしてモデルを生成するため，そのままモデル化すると状態爆発を起す可能性が非常に高い．それを防ぐためにプログラムを抽象化して検査したい状態のみをモデルにする．

条件の成否，繰り返しの有無の状態を区別できるように条件分岐と繰り返しの制御構造をモデル化する．条件分岐(if 文) については図 2 のように，開始ロケーションから分岐するふたつの経路を用意することにより条件文に関して true，false の 2 状態を表すモデルに抽象化する．一方，繰り返し(for 文) については図 3 のように，反復する経路とそこから抜け出す経路を用意することにより条件文に関して反復継続，反復終了の 2 状態を持つモデルに抽象化する．

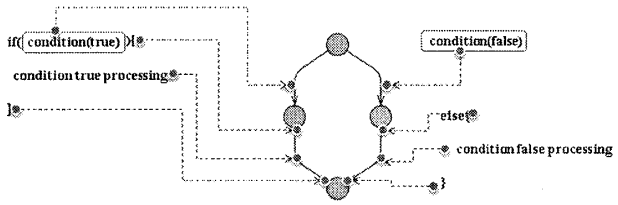


図 2 構成要素のモデル化(if)

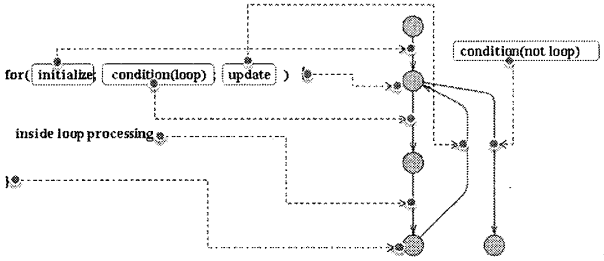


図 3 構成要素のモデル化(for)

2.1.4 検査支援ツール

支援ツールのインターフェースは Microsoft Excel を使用し，検査モデルの作成等の処理は VBA(Visual Basic for Applications) で実装する． Java ソースの JavaML 変換には JJMLT [5] を用い，自動生成される検査モデルのモデル検査ツールは UPPAAL を使用する．

2.2 モデル検査手順概要

2.2.1 モデル検査対象プログラムの登録

支援ツールを起動し，ポップアップ画面を開いて検証対象プログラムをモデル検査の対象として登録する．

2.2.2 プログラム構造図の作成

指定されたプログラムの構成要素(for 文・if 文) に基づき構造図(図 4)を生成する．構造図は，タグ構造を階層に読み換えて JavaML を木構造として捉え，その木構造の先頭タグ要素から構成要素までをつなぐ経路のみ残したものである．

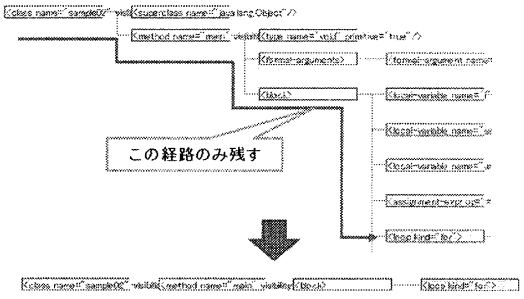


図 4　プログラム構造図

2.2.3 検査モデルの作成

モデル化範囲の先頭項目を図上でダブルクリックすることによりモデルの自動生成が開始される．支援ツールは UPPAAL のモデル図のデータファイルを作成し，UPPAAL を起動させてそのファイルを取り込ませて表示する．

2.2.4 モデル検査

検査したい状態を定義し自動生成したモデルに付加する．このモデルに対し，検査したい内容を検査式で表わして検査を行う．検査式は CTL(Computational TreeLogic) をベースとしたものである．検査結果が満たされなかった場合，UPPAAL ではどのような経緯で満たされなかったかを反例という形で表わす．不具合原因はこの反例により確認する．また動作の再現機能を用いて発生条件および問題ある個所の確認を行う．

3. 適用事例

3.1 適用対象と不具合内容

Embedded Technology Software Design Robot Contest(以下 ET ロボコン)で使用した走行プログラムをサンプルプログラムとした．これは自律型・ライントレース・ロボットの走行競技用に作成されたもので，光センサで路面の色を識別子しコースに沿って走行し，指定周回回ったのち停止する仕様となっている．ハードウェアの特性から入出力の値においてソフトウェアで想定したものと誤差が生じ，再現が困難であるため制御方式の根拠が定まらず，走行状態が安定しないという現象がでていた．ソースコードは実際に使用した C 言語のプログラムを Java に書き直したものである．

ET ロボコンのプログラムを適用事例とした理由は，組込システムであることと複数の担当者が修正しているため「元の仕様を理解しないまま追加された新たな仕様」があると思われるからである．

コンテストは毎年行われており，プログラムは引き継がれ改良を加えられてきた．新たな担当者は元のプログラムに新たな処理を追加し，実際に走行させて不具合があると，その不具合を解消のために再度修正を行い，走行させてその結果を確認することを繰り返す．走行結果の変化は，修正の影響か外部環境との間で発生する誤差の影響かの判断は難しく修正を繰り返すことにより全体的な仕様の把握は難しい状態になる．

3.2 適用事例におけるモデル化の方針

ロボットは外部環境であるコースから光センサを用いて情報(色)を得て走行するので，これをモデル化する．

実際の走行では周回コースを 2 周するが，適用事例ではコースは 1 周回分を直線にして表す．

コースを直線にしているためロボットが等速で前進すると想定すれば，その状態からの挙動の変化はないので，ロボットのステアリング制御，駆動制御，リカバリ制御等の機能については抽象化してモデル上に直接は反映しない．光センサの値取得の動きについてはモデル化したコースにおける取得できる色の非決定性により対応する．

3.3 モデル化範囲の決定

今回の不具合は，ロボットの走行状態より走行制御部分が原因であることは認識できる．また走行状態が安定しないことから外部環境も関係することもわかる．

プログラムはみっつのクラスファイルより構成される．ひとつ目は走行制御と関係ない起動用の Main クラスであり，ふたつ目は前節で抽象化した機能のドライバクラスである．これらふたつは検査対象から除外する．残りのひとつがナビゲーションクラスで走行制御に関係するためこれを検査対象とする．

検査者はプログラムの基本仕様を知っており，今回サンプル化した部分に問題があることは把握できる．サンプルプログラムを構成する java ソースコードを支援ツール上で構成要素(for 文・if 文) に基づき抽象化された構造図に変換する．図 5 が生成された構造図であり，走行制御部分に原因があることから，走行制御を司る run メソッドをモデル化の候補とする(図 5 の枠内).

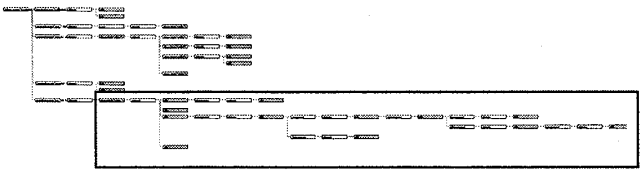


図 5　プログラム構造図

3.4 モデル検査準備

UPPAAL 上で検査するためのモデルの作成を行う．

3.4.1 モデルの自動生成(ソースコードのモデル化)

モデル化範囲の先頭項目を図上でダブルクリックすることにより構造図(図 5)の枠内の for 文・if 文を 4 節で示したモデルに置き換えモデルを自動生成する．その一部を図 6 に表わす．図 6 の点線矢印の先がソースコードの条件分岐が対応するモデル上の遷移である．このようにしてモデル化範囲のソースコードを UPPAAL の検査モデルに変換する．これを検査モデル 1 とする．

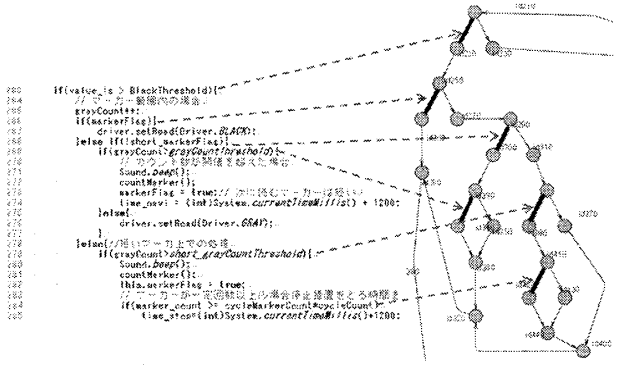


図 7　ソースコードのモデル変換(検査モデル 1)

3.4.2 ステートメントの抽象化

正常走行を想定して for 文・if 文内にあるステートメントをモデル上に反映させる．

UPPAAL の文法の大部分は C および Java のものと一致しているが完全に一致している訳ではないため，変数・ステートメントが文法の違いによりそのまま UPPAAL で使用できない場合がある．そのため，それらを適切な形に抽象化して定義する必要がある．代表的なものとして以下のふたつがある．

・コレクション

UPPAAL ではコレクションは使用できないため配列を用いてコレクションが提供するインターフェースの機能を抽象化して表現する．

・メソッド呼び出し

ステートメント内でメソッド呼び出しが行われている場合，比較先もしくは代入先の変数の型と同じ型の変数として定義する．但しこの場合，不具合の原因が当該メソッド内にあると，不具合の原因まで抽象化されてしまい不具合現象が確認できなくなる．

そのような場合，ソースコード上に定義されているメソッドならば，UPPAAL 上に新たに関数を定義しその中にメソッド内部の処理を段階的に抽象化して記述する．今回の Sensor クラスのメソッドのようにライブラリの場合は，ソースコードの段階的抽象化ができないためそのライブラリの特性を UPPAAL の検査モデルとして定義する必要がある．

3.4.3 検査目的とする状態のモデル化

モデル検査したときに検査結果を明確に確認するため検査すべき状態を定義する．ここでは指定周回終了した後の自動停止状態(AUTO_END)を定義する(図 8)．これを検査モデル 2 とする．

また規定周回で自動停止できなかった場合の状態も定義する．手動停止状態(MANUAL_END)とし，モデル 1 の終点がこれにあたる(図 9).

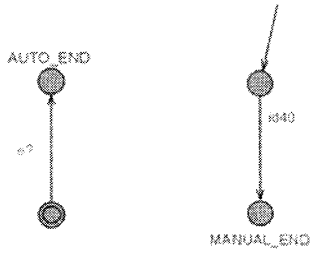


図 8 自動停止状態(検査モデル 2)　図 9 手動停止状態

3.4.1 初期値の設定

ロボットがコースを走るために走行制御部分で設定しているパラメータ値を行う．前節で定義した変数に基本的にはソースコード上で設定されている値を設定する．

モデル化するにあたり新たに値の設定が必要になるものが発生する，その場合には検査者の経験より妥当と思われる値を設定する．例えば光センサが読み取る色の値を設定が必要となる．実際は走行前にコース上で白，グレー，黒の値を読み込んで設定する．光センサは実測値の明度をある段階数で分割して白・黒の値を設定する．モデル上においては，平均的な値を閾値として灰色：41，黒：37 を設定する．この設定により 35,36,37：黒，38,39,40：灰色，41 以上：白と判定される．

3.4.2 コースの定義(外部環境のモデル化)

コースは白い下地の上に黒い線で描かれる．このコース上にマーカと呼ばれる灰色のポイントが設置される．基本的には長いマーカと短いマーカのふたつが設定される周回コースであるが，モデル検査を行うに当たり，光センサの読み取り値のシミュレーションを行うためにコースを抽象化する．コースを直線に置き換え，そこに長いマーカ・短いマーカを設定する(図 10)．ロボットは光センサでコースの色を識別しながら進むため，読み取り回数によりすすむ距離を決定するようにモデル化する．コースの始点～終点の距離を数値化して 0～1000 とし，ロボットは等速で進むものとして 1 読み取り毎に 25 進むこととした．コースの区間設定はモデル検査の非決定性を利用するため実際のコースと厳密に似せる必要はない．

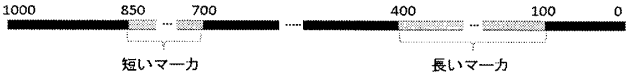


図 10　抽象化したコース

このコースより色の値を取得する光センサの機能を UPPAAL のモデルに変換すると図 11 になる．コース上での距離によりコースの特性が変わるため，区間ごとに①～⑤遷移を用意する．各遷移が表す区間は①：0～100 ②：100～400　③：400～700　④：700～850　⑤：850～1000 となる．

モデル上の式[A]は，同期処理の受信部であり，他のプロセスの送信部より呼び出されることにより遷移が移ってくる．[B]～[C]は遷移①～⑤にそれぞれ定義される．[B]は同期処理の送信部で該当遷移実行後に他のプロセスの受信部へ遷移を移す．[C]は光センサが取り得る値の範囲の設定で，この遷移内で取得できる値の連続した範囲を設定している．[D]は遷移できる条件である．距離ごとに通る遷移を分けるための距離の制限と，その遷移が表すコースの特性にあった色の制限を行う．また，コースアウトを防ぐため連続した白の取得を制限している．[E]は遷移内で取得できた色の値の受け渡しとコース上での位置を前進させるための始点からの距離のインクリメントである．

黒い色のコース(①③⑤)上ではライン内 35～37(黒)ライン外 49(白)の値を取得し，灰色のマーカ(②④)上ではライン内 38～40 ライン外 49 の値を取得する．これを検査モデル 3 とする．

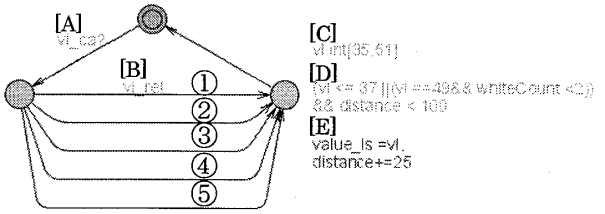


図 11　光センサモデル(検査モデル 3)

3.4.3 変数の定義

検査対象を含むメソッド内の変数定義は自動的に UPPAAL 上にコピーされる．UPPAAL で使用できない型の定義であったり，メソッド外の定義で自動設定されなかった場合は，モデル上に赤字で表示されるので UPPAAL で対応できるように修正する．

例えば図 12 の場合，変数 time_navi はメソッド外に定義された変数であるため自動設定されていない．そのため手動で定義を追加する．time_navi の比較対照は int 型であるので time_navi も int 型として定義する．

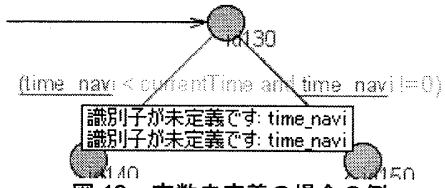
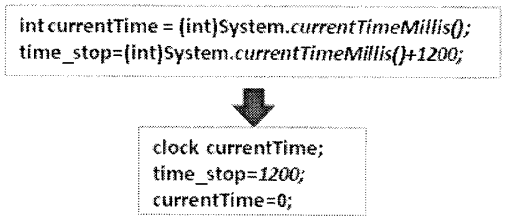


図 12　変数未定義の場合の例

3.4.4 for 文・if 文ブロック内部の定義

今回プログラム内でメソッド System.currentTimeMillis() が使用されている．これは現在時間の取得のメソッドでタイマ処理に利用している．時間関係の処理については UPPAAL のクロック変数が用意されている．但し，あくまでモデル内での相対的な時間を扱うものなので，絶対的な時間表現から相対的な時間表現にステートメントを変更する必要がある．時間計測開始時にクロック変数をリセットして時間計測を行うように対応した．(図 13).



判定式は time_stop <= currentTime

図 13　タイマ処理の変換

光センサの値を取得するメソッド lightSensor.readValue() については，3.4.2 節でモデル 3 としてモデル化しているため，モデル 1 上の該当部分より同期処理でモデル 3 を呼びだすことにより対応する．

3.4.5 検査モデル

検査モデル 1，検査モデル 2，検査モデル 3 がそろうことで検査可能なモデルとなる(図 14)．各モデルは UPPAAL の同期機能を用いて各モデルを連携させる．

モデル 2 はソースコード上の自動停止処理の部分に該当する検査モデルの遷移から同期させ，モデル 3 は光センサの値取得の部分から同期させる．

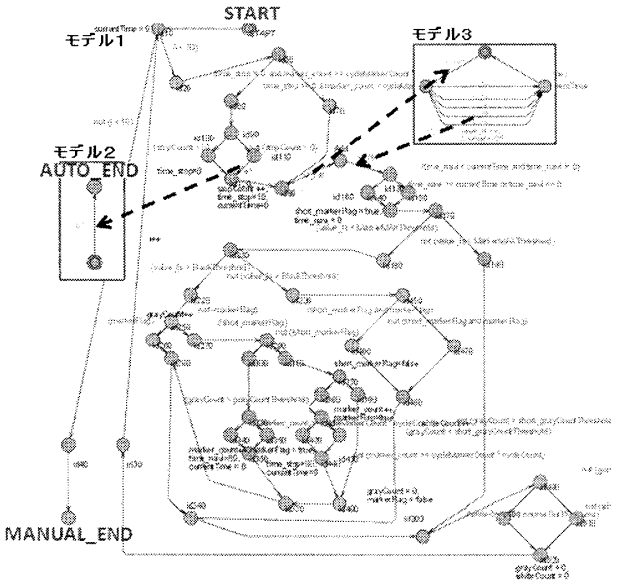


図 14　検査モデル

3.5 詳細検査

条件値に関連がある全ての変数・ステートメントを UPPAAL 上に定義することにより検査モデルは検査可能になる(図?). モデル内の緑文字は更新処理，青文字はその遷移を通過できる条件，灰色はその遷移上における値の選択可能範囲を表す.

この作成された検査モデルに対して，自動停止の終点である AUTO_END に到達できるかの検査を行う．”全ての場合においてコースの終点に到達する前に自動停止の状態に に到達することはない” との検査には” 満たされない” (到達できる) との結果がでた(図 15).

A[] not (Process3.AUTO_END && Process2.distance<1000)
属性は満たされませんでした

図 15　検査結果 1

次に手動停止の終点である MANUAL_END に到達できるかの検査を行う．”全ての場合において手動停止の状態に到達することはない”との検査には”満たされない”(到達できる)との結果がでた(図 16).

A[] not (Process.MANUAL_END)
属性は満たされませんでした

図 16　検査結果 2

上記 2 回の検査により自動停止，手動停止どちらの状態にもなることがわかる．UPPAAL のシミュレーション機能を用いて原因を探す.

自動的に止まることができない現象が発生するということはマーカの認識を誤っている可能性が非常に高い.

モデル上の分岐条件に使用されている変数を抽出する．その中から閾値として使用されているものを除く．残った変数の検査 1 ・検査 2 における変移を UPPAAL 上で比較すると，コース上の距離 300 の時点で，マーカフラグの変数とタイマ実行するためのクロック変数に差異が生じていることがわかる．このふたつの値に着目してさらに調査を行う.

走行制御のプログラムは長短マーカの認識をふたつのフラグ(markerFlag,Short_markerFlag)で行っている．マーカフラグの組合せの意味をまとめると表 1 になる.

表 1　マーカフラグの組合せの意味

markerFlag	Short_markerFlag	ロボットの状態
FALSE	FALSE	黒い色の線上，次にくるのは長いマーカ
TRUE	FALSE	灰色の線上，長いマーカ上にいる
FALSE	TRUE	黒い色の線上，次にくるのは短いマーカ
TRUE	TRUE	灰色の線上，短いマーカ上にいる

これらのマーカフラグを検査 1 と検査 2 で比較する．UPPAAL 上でそれぞれをシミュレーションしてフラグの値，タイマの時間，コース上の距離を確認すると表 2 になる．表 2 とコースの対比を図 17 に表わす.

表 2　マーカフラグ変移

		1	2	3	4	5
検査1	markerFlag	FALSE	TRUE	FALSE	FALSE	TRUE
	Short_markerFlag	FALSE	FALSE	FALSE	TRUE	TRUE
	currentTime	>=0	>=0	>=0	>1200	0>=
	distance	0	275	425	475	800
検査2	markerFlag	FALSE	TRUE	TRUE	FALSE	FALSE
	Short_markerFlag	FALSE	FALSE	TRUE	TRUE	FALSE
	curreintTime	>=0	>=0	>1200	>1200	>1200
	distance	0	275	300	425	425

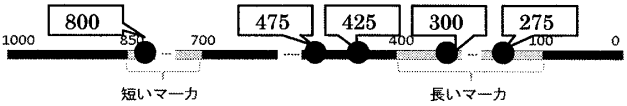


図 17　マーカフラグ変移とコースの対比

検査 2 の 5 列目に着目する．距離 425 で markerFlag=FALSE，Short_markerFlag=FLASE になっている．これはプログラムが次にくるのは長いマーカと認識していることを示す．しかし実際にくるのは短いマーカであるため長いマーカを認識するための閾値に到達しないためマーカを認識できないことが自動停止できない直接の原因と思われる.

3 列目からマーカフラグの動きが検査 1，検査 2 で異なる．この部分の動きを UPPAAL 上で比較確認すると，実行するロジックの順番が入れ替わっていることがわかる.

検査 1 では「長いマーカを認識」(2 列目)→「黒いライン走行」(3 列目)→「短いマーカを予測」(4 列目)となる．検査 2 では「長いマーカを認識」(2 列目)→「短いマーカを予測」(3 列目)→「黒いライン走行」(4,5 列目)となる.

つまりコース上の距離 300(長いマーカの中)の時点で「短いマーカの予測」処理が動いており，この処理は「長いマーカを認識」処理時にタイマ実行がかけられるものであることから，長いマーカ(灰色)から抜け出す前にタイマ時間がきて想定外の実行がされてしまう可能性があることが判明した.

3.5.1 修正案確認

タイマ実行によるフラグの制御に問題があるので時間制御は行わず，マーカフラグと色の識別カウントにより制御するロジックに変更した．このモデルで先と同じ検査を実

施したところ自動停止できるが，自動停止できない状態にはならないことが確認できた(図 18，図 19).

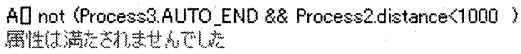


図 18　検査結果 1

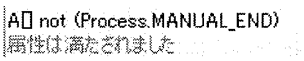


図 19　検査結果 2

3.6 適用の実機確認

前節で確認した修正を実機に適用してその効果を確認する．改修前のロジックと修正後のロジックのそれぞれ 10 回ずつテストを行い自動停止できた回数，できなかった回数を調査した．修正前ロジックが持つタイマ実行のタイミングは実際に走らせて調査し，短めの値を設定した．これは電池の電圧の低下等により想定した移動距離が得られなかった状態を再現するためである．

結果は表 3 のとおりで，修正前のロジックでは自動停止できない状態が発生したが修正後ロジックでは全て自動停止できた．

表 3　適用の実機確認

プログラムの状態	自動停止できた回数	自動停止できない回数
修正前	6	4
修正後	10	0

4. 関連研究

Achenbach[6]では，各種モデル検査ツールにおける抽象化テクニックを比較している．例題としてファイル IO のエラー処理をオープン，クローズ，エラーの 3 状態のモデルを使って抽象化しており，これを参考にして抽象化の考え方を検討した．これは各種モデル検査ツールの抽象化機能の比較が主な目的で，ソースコードより不具合を発見するという観点はなかった．

Bao[7]も Achenbach [6]と同様に確認したい状態のみ見ることができるようにプログラムを抽象化する手順の例として参考になった．これはソフトウェアコンポーネントリリース時のテストを念頭に置いたものであり，実際のプログラム開発における現場サイドでの不具合発見の運用を強く意識したものではなかった．

Thomas[5]はリアルタイムの組込みシステムのために Java ソースコードから忠実にモデル化された UPPAAL モデルを定義している．適用対象は実時間処理の組み込みシステムで色付きレンガのソートを行うが，外部環境の影響が小さいため外部環境のモデル化は行われていない．

5. まとめと課題

今回，いままで提案してきた「ソースコード解析を利用したモデル検査に基づく欠陥抽出手法」を組込システムに適用した．

適用事例として ET ロボコンのライントレース・ロボットを使用した．本来，組込システムは，実機テストでは入力値や出力がその都度若干変わることから，不具合の現象が特定しにくい．しかし外部環境も含めモデル化することにより不具合の発生原因となりうる要素を確認でき，修正結果の確認まででできた．これは不具合発見にモデル検査ツール UPPAAL を用いることにより検査結果である反例を UPPAAL のシミュレーション画面で詳細に視覚的な確認でき，検査結果を高い確度で把握できることが大きく影響している．

課題としては，検査者は基本仕様を理解していることが前提であるが，変数の定義・入力値の設定については UPPAAL の記述言語の知識が必要であるため，何らかの形での簡易設定機能の追加がある．また同様にブロック内部の処理についてもよく使用される Java のライブラリについては予めメソッド単位に定式化しておく必要性も認識した．今後はより複雑なモデルに対応できるようにしていく予定である．

参考文献

[1]青木善貴，松浦佐江子，ソースコードの解析を利用したモデル検査に基づく欠陥抽出手法の提案，第 8 回情報科学技術フォーラム,pp.387-391, 2009

[2]青木 善貴，松浦佐江子，ソースコード解析を利用したモデル検査に基づく欠陥抽出手法の提案，知能ソフトウェア工学研究会, KBSE2009-44,pp.79-84, 2009

[3]青木 善貴，松浦佐江子，モデル検査に基づくプログラム欠陥抽出作業支援ツールの開発と実践，情報処理学会創立 50 周年記念(第 72 回)全国大会，2P-3，2010

[4]UPPAAL, http://www.uppaal.com/, 2010.

[5] JJMLT, http://www.hpc.cs.ehimeu.ac.jp/ aman/project/JJmlt/, 2010.

[6]ACHENBACH, M., OSTERMANN, K,"ENGINEERING ABSTRACTIONS IN MODEL CHECKING AND TESTING",SOURCE CODE ANALYSIS AND MANIPULATION, PROC. OF .SCAM '09.,PP.

[7]BAO, T., JONES, M.D. 2009. TEST CASE GENERATION USING MODEL CHECKING FOR SOFTWARE COMPONENTS DEPLOYED INTO NEW ENVIRONMENTS. PROC.OF ICSTW '09 ,PP. 57-66,2009.

[8]THOMAS BOGHOLM, HENRIK KRAGH-HANSEN. MODEL-BASED SCHEDULABILITY ANALYSIS OF SAFETY CRITICAL HARD REAL-TIME JAVA PROGRAMS, PROC. OF JTRES 2008 , PP. 106-114.