修 士 論 文

モデル検査のための
環境モデリング手法に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

西端 浩和

2009 年 3 月

修 士 論 文

モデル検査のための
環境モデリング手法に関する研究

指導教官　青木 利晃 特任准教授

審査委員主査　青木 利晃 特任准教授
審査委員　小川 瑞史 教授
審査委員　岸 知二 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

710054 西端 浩和

提出年月: 2009 年 2 月

概 要

本研究では、モデル検査に適した環境モデリング手法を提案した。提案手法により外部環境の振舞いを記述した検査モデル生成の効率化を行った。提案手法を RTOS の仕様である OSEK/VDX に適用実験し、評価を行った。

目 次

第1章　はじめに

1.1　背景

近年、社会システムにおいてさまざまなものにソフトウェアが組み込まれている。その為、ソフトウェアには高い信頼性が求められる。ソフトウェアの信頼性を保証する手段の一つとしてモデル検査がある。モデル検査を行うためには設計書や仕様書、ソースコードなどから作成される検査対象モデル、要求仕様書や機能要求書から作成される検査項目が必要である。検査対象モデルには検査する対象の振舞いを記述する。検査項目は検査したい性質は論理式により記述する。モデル検査を行う際、この2つ以外に適切なイベントを発生させる外部環境もモデル化する必要がある。外部環境の振舞いを記述したのものを検査モデルと呼ぶこととする。本研究では外部環境のモデリング手法に焦点を当てる。

モデル検査の問題点の一つとして、検査モデルを生成する事が困難である事が挙げられる。検査モデルを生成する上で障害となる点は2点ある。モデル検査による信頼性向上のためには、検査したい機能ごとに検査モデルを組み合わせて検出したい誤りを発見できる事が望ましい。しかしながら、検査モデルを組み合わせるには複雑な制約があるため整理して記述する必要がある点が一つめである。また、組み合わせる機能によっては検査モデルの組み合わせのパターン数が膨大になると想定される。そのすべてにおいて検査モデルを生成し、モデル検査を実施するのは困難である点がもう一つである。

高い信頼性が求められるソフトウェアの中でも、特に組み込み機器はその傾向が顕著である。その中でも組み込み器機上のアプリケーションだけでなく、それを動かす Real Time Operating System (以下 RTOS) にも信頼性の確保が求められている。

1.2　研究の目的

本研究の目的は、モデル検査に適するように外部環境をモデル化する手法を提案する事である。提案する環境モデリング手法により検査モデルの効率的生成を目指す。検査モデル生成の問題点の解決方法として以下の2つの手法を提案する。モデリングの際に環境の組み合わせに関する制約を整理して記述する事により1つめの問題点を解決する。同じ振舞いをすると考えられる組み合わせを同意義パターンを定義して削除する事により2つめの問題点を解決する。

環境モデリングには一般的に広く使われるモデリング言語 UML を用いる。提案する環境モデリング手法の記述に十分なように UML の記述能力を拡張する。UML で記述した

環境モデルから Promela 記述の検査モデルを生成できる手法を提案する。

モデル検査はレビューや試験では確認が難しいシステムの検査に適している。それらのシステムとしてマルチスレッドやマルチタスクなどの並行動作するシステムであるや、非同期イベント処理や通信プロトコルなどのタイミングによって動作が変化するシステムが挙げられる。提案手法の適用対象をこれらの特徴をを持つ RTOS である OSEK/VDX 仕様とする。OSEK/VDX 仕様とは自動車におけるエンジンコントロールユニットで用いられる RTOS の業界標準仕様である。適用した実験を元に提案手法の考察、評価を行う。

第2章　モデル検査と問題点

2.1　モデル検査

2.1.1　モデル検査概要

モデル検査とは形式的手法のひとつである。形式的手法では、数学的・論理的基盤に基づいて検査したい性質の正しさを証明する。モデル検査では、ソフトウェアやハードウェアなどを検査対象とした状態遷移モデルを有限オートマトンに対応付け、有向グラフで表現する。特徴は有効グラフの全ての遷移系列を網羅的に全自動探索する事である。網羅探索を行うため、通常の試験などでは発見しにくいデッドロックや無限ループの検出に適している。モデル検査の代表的な問題点として状態爆発問題がある。状態爆発問題とは、状態数の指数関数的に増加により、有限であるコンピュータのメモリ量の不足により検査不能になる問題である。この問題を回避する方法は、検査したい性質に関する情報以外を抽象化するなどである。

2.1.2　モデル検査ツール SPIN

モデル検査ツール SPIN とは、AT&T Bll 研が開発したモデル検査ツールである [1][2]。Spin では並行プロセス、および個々のプロセス内で非決定的な振舞いを専用記述言語 Promela で記述する。記述をもとに、可能な動作を網羅的に探索して、指定した性質が成立するかどうか自動的にチェックする。検査する性質は、ラベルや表明、性質オートマトンで指定可能である。並行動作や非決定動作を乱数により選択して実行するシミュレーション実行機能がある。また、LTL(Linear Temporl Logic) を性質オートマトンに自動変換する機能も組み込まれている。検査する性質に違反した場合は、反例として違反にいたる経路を示すことができる。Promela 中では状態をラベル、遷移を goto 文で記述するのが一般的である。遷移のガード条件を非決定的に記述可能という特徴がある。

2.1.3　モデル検査実施に必要なもの

モデル検査を実施するためには、以下の 3 つが必要である。

- 検査対象モデル

検査したいシステムの振舞いを記述する。設計書や仕様書、ソースコードなどから検査する内容に関連する部分を切り出し、検査に必要な最低限の情報に抽象化して作成される。

- 検査モデル

  検査対象のシステムがどのような環境で実行されるかを表現したモデルである。検査対象への入出力イベントや通信方式などを記述する。記述する振舞いはユーザ、外部システム、ネットワーク環境、インフラ、ミドルウェアなどが挙げられる。

- 検査項目

  検査対象モデルに対して、検査する性質を論理式や表明により記述する。進行性 (Progress)、安全性 (Safety)、到達可能性 (Reachability) などを検査するために用いる。

  本研究では 2 つめの検査モデルのモデリング手法に着目する。

2.2 モデル検査の問題点

2.2.1 小規模事例：キッチンタイマー

本研究ではモデル検査の問題点を把握するために小規模事例としてキッチンタイマーを用いた。キッチンタイマーとは料理などに用いられるシンプルなタイマーである。本研究で用いるキッチンタイマーの仕様は「組み込みアカデミー 2」[3] で使用したものである。このキッチンタイマーは設定した時間が経過したらアラーム発信する。キッチンタイマーは図 2.1 に示すように二つの入力 (スイッチ) と二つの出力 (LED) を持つ。

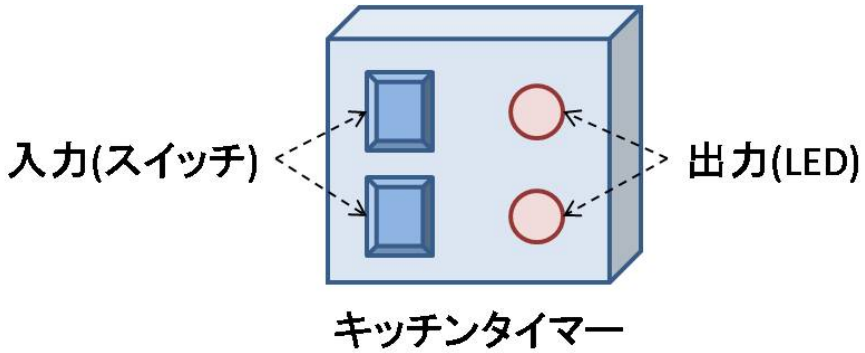


図 2.1: キッチンタイマーのイメージ

小規模事例の実験で用いるキッチンタイマーの仕様の以下に簡単に示す。

- 時間設定 (スイッチ 1) で計時時間を設定
- 計時開始 (スイッチ 2) で計時を開始
- 出力が LED2 個なので、設定時間の表示は 2 進数として、設定数は 0〜3 とする
- 計時中は、LED を交互に点滅 (500ms 周期) する
- 設定時間経過後、アラームとして LED を 3 秒間同時点滅 (100ms 周期) する
- 計時中に時間設定を押すと計時終了し、計時開始を押すと一時停止する
- アラーム中に時間設定、計時開始を押すとアラームは停止する

キッチンタイマーの構成の図 2.2 に示す。キッチンタイマーはキー入力監視タスク、計時タスク、表示タスクの３つから成り立つ。キー入力監視タスクは、キードライバを経由してスイッチが押下されるのを監視する。キーが押下されるとキー情報を計時タスクへ通知する。計時タスクはキー入力監視タスクからのキー情報通知を待つ。キー情報が通知されると、状態に応じて時間の設定、計時の開始、計時の停止、時間経過の監視の機能を実行する。また、ユーザーに結果を知らせるために表示タスクに表示制御情報を通知する。表示タスクは、計時タスクからの制御情報の通知を待っている。通知を受けると、LED の点灯/消灯、交互点滅や全点滅といった制御を実行する。処理のタイミングは OS の周期ハンドラからの通知によって決定する。

本研究では計時タスクと表示タスクを検査対象とする。キー入力監視タスクとユーザをまとめたもの、周期ハンドラを外部環境とする。

2.2.2 大規模事例：OSEK/VDX 仕様

OSEK/VDX 仕様とはドイツ・フランスの自動車産業が業界標準作成を目標としたプロジェクトで提示された OS 仕様である [4]。OSEK/VDX 仕様書は以下のサイトより、フリーでダウンロード可能である。2009 年 2 月現在バージョン 2.2.3 が発行されており、本研究ではこのバージョンを対象にしている。

http://portal.osek-vdx.org

OSEK/VDX の主な機能としてタスク管理機能、応用状態 (アプリケーションモード)、割り込み処理機能、イベント制御機能、資源 (リソース)、警告 (アラーム)、伝言 (メッセージ)、フックルーチンなどがある。OSEK/VDX の仕様書は自然言語で記述されているため意味に曖昧性を含む。機能間の制約は仕様書のいたるところに書かれているため、機能のモデル化においては他の機能を記述した部分を参照して考慮する必要がある。

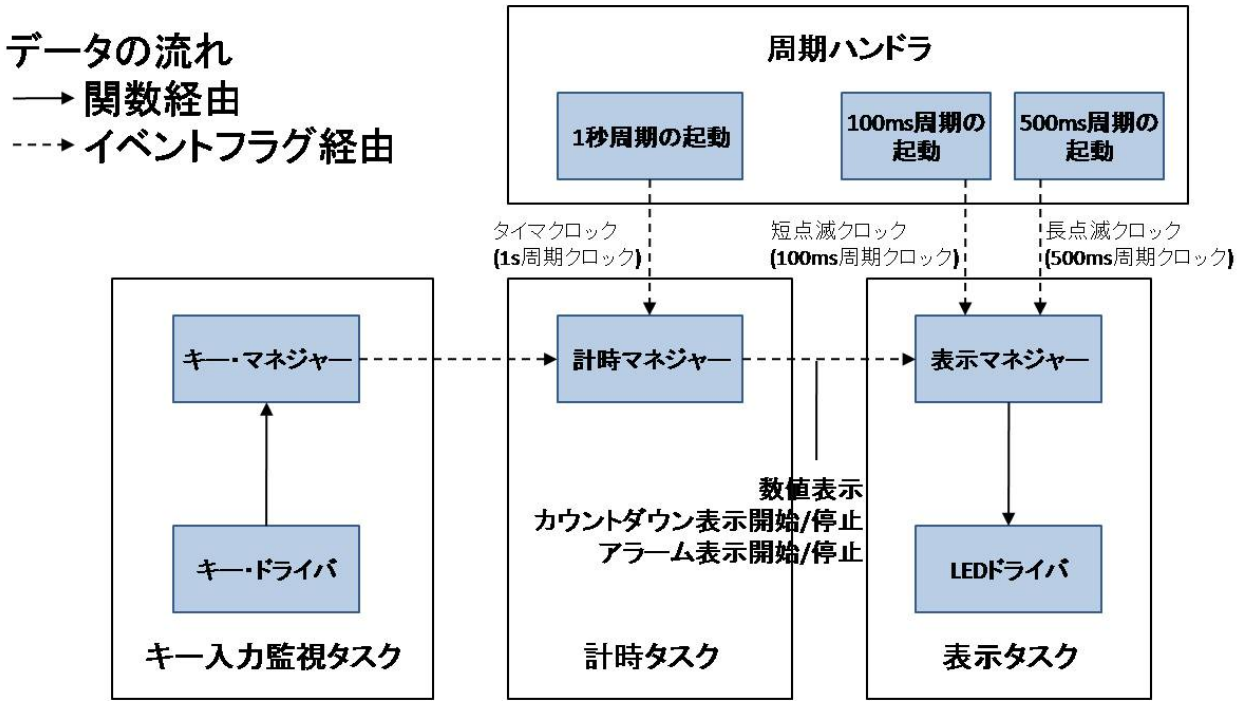


図 2.2: キッチンタイマー 機能ユニット関連図

本研究では OSEK/VDX 仕様の機能の内、タスク管理機能と資源に関する機能をモデル検査の対象とした。その理由は、タスクの振舞いはリアルタイム OS の主機能であり、高信頼性が求められるためである。OSEK ではタスクの状態モデルが拡張タスクと基本タスクの 2 つ用意されている。拡張タスクと基本タスクの違いは、拡張タスクには WAITING 状態が認められている点である。拡張タスクはイベントをセットしそのイベントを待つ間 WAITING 状態へ遷移し CPU を開放する。WAITING 状態にあるタスクはイベントが終了すると自分で READY 状態へ遷移し優先度に基づいてレディキューにつながれる。基本タスクにはこのようなイベントに関するセットが認められていないため、WAITING 状態へ遷移することはない。本研究ではイベント制御機能を対象としていないため、この基本タスク (以下タスクと呼ぶ) を対象とする。資源機能について説明する。タスクは GetResource というサービスコールを発行する事により資源を占有する。この時、プライオリティシーケンスが起こる。プライオリティシーケンスとはタスクが資源を占有時、資源に設定されている優先度が一時的にタスクの優先度になるという仕様である。これにより、資源を占有しているタスクの優先度が高くなるため、他のタスクが RUNNING 状態にする事はなくなる。また資源占有中は SUSPENDED 状態へ遷移する事が認められておらず、SUSPENDED 状態へ遷移するようなサービスコールの呼び出しは禁止されている。資源が解放されるとタスクの優先度は元に戻る。

本研究ではサービスコールにより、タスクが SUSPENDED 状態、RUNNING 状態、READY 状態、資源が解放状態 (FREE 状態)、占有状態 (OCCUPIED 状態) に正しく遷移

するかを検査する。対象とするサービスコールを内容を簡単に以下に示す。

- $ActivateTask(task\_ID)$

  呼ばれた ID のタスクを READY 状態へ遷移させる。

- $TeriminateTask()$

  RUNNING 状態のタスクを SUSPENDED 状態へ遷移させる。

- $ChainTask(task\_ID)$

  RUNNING 状態のタスクを SUSPENDED 状態へ遷移させ、呼ばれた ID のタスクを READY 状態へ遷移させる。

- $GetResource(resource\_ID)$

  呼ばれた ID の資源を OCCUPIED 状態へ遷移させる。

- $ReleaseResource(resource\_ID)$

  呼ばれた ID の資源を FREE 状態へ遷移させる。

2.2.3 モデル検査を行う上での問題点

キッチンタイマーの事例と OSEK/VDX 仕様の事例をもとにモデル検査を行う際、以下の点が問題点して判明した。

1. 複数の環境を想定して検査モデルを生成する際、それらの振舞いを統合する必要がある。しかしながら、検査対象と外部環境の入出力関係には複雑な制約がある。よって、それらを整理して記述する必要がある。

   (キッチンタイマーの事例)

   検査対象に対して各周期ハンドラは呼び出さる順番に制約がある。100ms 周期クロックが 5 回、500ms 周期クロックが 1 回、100ms 周期クロックが 5 回、500ms 周期クロックが 1 回、1s クロックが一回の順番で呼び出されるべきである。100ms 周期クロックと 1s クロックが一定時間内に同回数呼びだされようような検査モデルでは周期タイミングに関する正しい検査を行うことはできない。

   (OSEK/VDX 仕様の事例)

   タスク、資源の振舞いの制約には以下のような例がある。

   (a) 優先度が高い他のタスクが Running 状態のタスクがある場合、Ready 状態に遷移する。

   (b) 複数のタスクの中で Running 状態になるのは高々1 つである。

(c) Running 状態以外のタスクは資源を占有できない。

また、ある遷移により、他の遷移が誘導されてしまう場合がある。例えば、RUNNING 状態のタスクと READY 状態にタスクがあるとする。この時、TerminateTask が発行されると RUNNING 状態のタスクが SUSPENDED 状態に遷移するとすぐに、READY 状態にタスクが RUNNING 状態へ遷移される。他のタスクの遷移を誘導するかは他のタスクや資源の状態、優先度に影響されるためそれらを考慮して検査モデルを作成しなければならない。

2. 状態爆発問題により検査できる振舞いには限界があり、すべてを探索する事が困難である。

   (キッチンタイマーの事例)

   外部環境としてのユーザの人数は制限する必要がある。

   (OSEK/VDX 仕様の事例)

   検査対象に対してタスク、資源の実体化数を制限する必要がある。

3. 検査モデルの種類は、外部環境の組み合わせパターンにより膨大になる。

   (OSEK/VDX 仕様の事例)

   実体化される多重度、属性の値の直積分だけの組み合わせパターンが想定される。そのすべての検査モデルを生成、モデル検査を実施するのは困難である。

2.2.4 問題点の解決方法

2.2.3 節で挙げた各問題点を解決する方法を環境モデリングの視点から提案する。以下の特性を満たすモデリングが実現できればモデル検査のリスクを軽減できると考えられる。

1. 外部環境を分離して記述することにより、任意に環境を組み合わせてモデル検査が行える。任意に組み合わせることにより、一つ一つの環境の振舞いに考慮して検査モデルを作成を容易にする。

2. 検査したい振舞いと検査できる振舞いが明確になっている。モデル検査により正しいと証明したい性質に着目してモデリングされている。

3. 組み合わせパターンの内、同じ振舞いをすると考えられるものは無視し、モデル検査の実施を効率的にする。

第3章　環境モデリング手法

本章では提案する環境モデリング手法について説明を行う。まず、モデル化に用いるモデル記述言語である UML について簡単に述べる。次に、提案手法の全体の流れを図を用いて説明する。そして、各ステップの詳細な説明を各節で行う。

3.1　モデル記述言語 UML

提案手法のモデリングにおいてモデル記述言語である UML を用いる [5]。UML とは、オブジェクトモデリングのために標準化した仕様記述言語である。UML はオブジェクト指向分析・設計の結果をグラフィカルに表現できる。UML には以下に示す 9 種類のダイアグラムが用意されている。

- ユースケース図
- クラス図
- オブジェクト図
- シーケンス図
- コラボレーション図
- ステートマシン図 (ステートチャート図)
- アクティビティ図
- コンポーネント図
- デプロイメント図

本研究では、上記の中から 2 つのダイアグラムを用いてモデリングを行う。1 つ目は、検査対象システムとの静的な関係を記述するためのクラス図である。2 つ目は、各外部環境の振舞いを記述するためのステートマシン図である。なお、本研究で取り扱う情報の中には既存の UML では表記できないものがある。そこで、UML の表記法の拡張を行う。

3.2　提案手法の概要

本研究では 2.2.4 で提案したも問題点の解決法の内、外部環境を分離して記述し、任意に統合しモデル検査できる点と、同じ振舞いをする組み合わせパターンを除去する点に重点を置く。1 つめの環境の統合法はインスタンスレベルの遷移条件を記述し、遷移可能性の可否により行う。よって、クラスレベルの遷移条件をモデリングする手法を提案する。また、2 つめの組み合わせパターン除去はクラス図をオブジェクト化する際に同意義になるパターンを定義することにより行う。

提案手法の流れを以下と図 3.1 に示す。

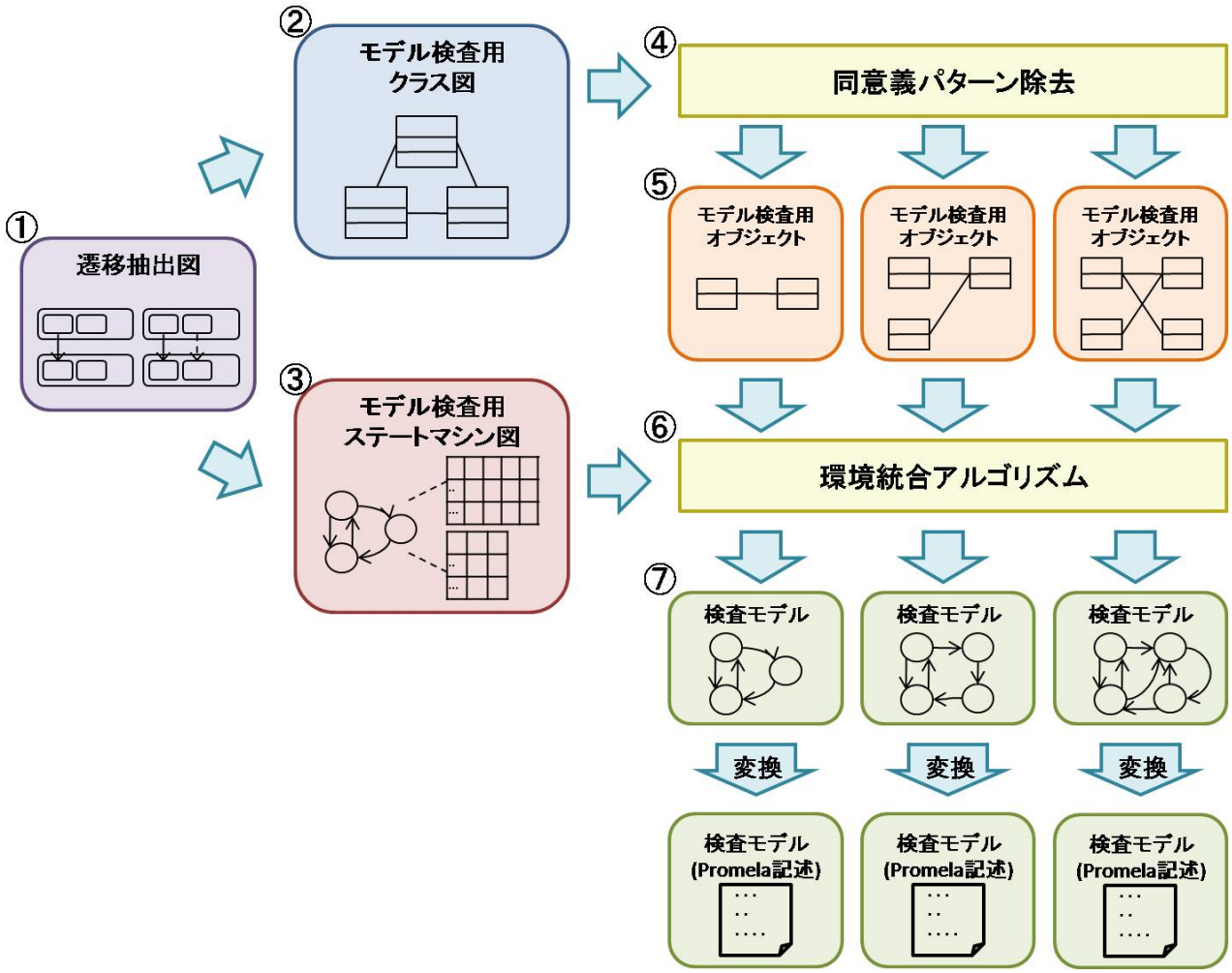


図 3.1: 提案手法の流れ

① 検査する内容をもとに遷移抽出図を作成する。遷移抽出図とは本研究独自のダイアグラムである。遷移抽出図作成の目的は、各状態からの遷移にどのようなものが

あるかを確認することである。遷移抽出図にはクラスの状態に着目し、状態から発生する可能性のあるすべての遷移を記述する。

② 検査対象と外部環境の静的な関係をモデル検査用クラス図として記述する。各クラス間の関連、多重度、属性の情報を記述する。

③ 個々の外部環境の振舞いをモデル検査用ステートマシン図として記述する。各クラスのモデル検査用ステートマシン図には遷移条件表と誘導遷移表を付加する。遷移条件表にはクラスレベルの遷移条件を表記する。誘導遷移表には誘導されるクラスと遷移後の状態を表記する。

④ モデル検査用クラス図の多重度、属性を実体化してモデル検査用オブジェクトを生成するが、組み合わせパターン数が多くなり過ぎるため、同意義パターンを定義して組み合わせパターンを削減する。

⑤ モデル検査用クラス図をもとにモデル検査用オブジェクトを生成する。モデル検査用オブジェクトは実体化された多重度や属性の情報を持つため、個々に異なる。

⑥ 各モデル検査用オブジェクトをもとにモデル検査用ステートマシン図のクラスレベルの遷移条件表と誘導遷移表を実体化する。モデル検査用ステートマシン図とインスタンスレベルの遷移条件表、誘導遷移表を環境統合アルゴリズムに入力し検査モデルを生成する。

⑦ 環境統合アルゴリズムにより得られた検査モデル (ステートマシン図で表記) を Promela に変換する。ステートマシン図と Promela の対応関係を示す。

3.3 遷移抽出図

本手法ではクラスレベルの状態からの遷移に着目する。遷移によっては他の遷移を誘導する遷移が考えられる。このような遷移を**誘導遷移**と呼ぶこととする。遷移元の状態と遷移後の状態が同じ遷移であっても誘導される遷移が異なる場合、遷移自体が異なるものとして考える。状態毎に遷移できる可能性のある遷移をすべて洗い出す。洗い出した遷移を図化したものを遷移抽出図と呼ぶ。遷移抽出図に表記する内容は以下である。

- C：クラス c の集合 $c \in C$
- $c.S$：クラス c の状態集合
- $c.O$：クラス c の操作集合
- Y：誘導遷移の集合 $Y \subseteq c.S \times c.S$
- T：遷移集合 $T \subseteq c.S \times O \times Y \times c.S$

遷移抽出図の例を図 3.2 に示す。例図においてクラス X が状態 X_1 で発生する可能性のある遷移は 3 パターンある。1 つ目はクラス X が操作 X_O_1 を行う、S_1 から S_2 への遷移である。2 つ目はクラス X が操作 X_O_2 を行う、S_1 から S_3 への遷移である。3 つ目はクラス X が操作 X_O_1 を行い S_1 から S_2 への遷移し、クラス Y の状態を S_1 から S_2 へ誘導する遷移である。1 つ目と 3 つ目は操作も遷移先も同じであるが、誘導する場合と誘導しない場合がある。このような場合、異なる遷移であるとし別々に記述する。クラス X が状態 S_2 で発生しうる遷移は、クラス X が操作 X_O_1 を行う、S_2 から S_1 への遷移のみである。クラス X が状態 S_3 で発生しうる遷移は、クラス X が操作 X_O_2 を行い S_3 から S_2 への遷移し、クラス Y の状態を S_1 から S_2 へ誘導する遷移のみである。クラス Y が状態 S_1 で発生しうる遷移はない。クラス Y の状態 S_1 が遷移する場合は、他のクラスによる遷移による誘導遷移だけである。クラス X が状態 S_2 で発生しうる遷移は、クラス X が操作 Y_O_1 を行う、S_2 から S_1 への遷移のみである。

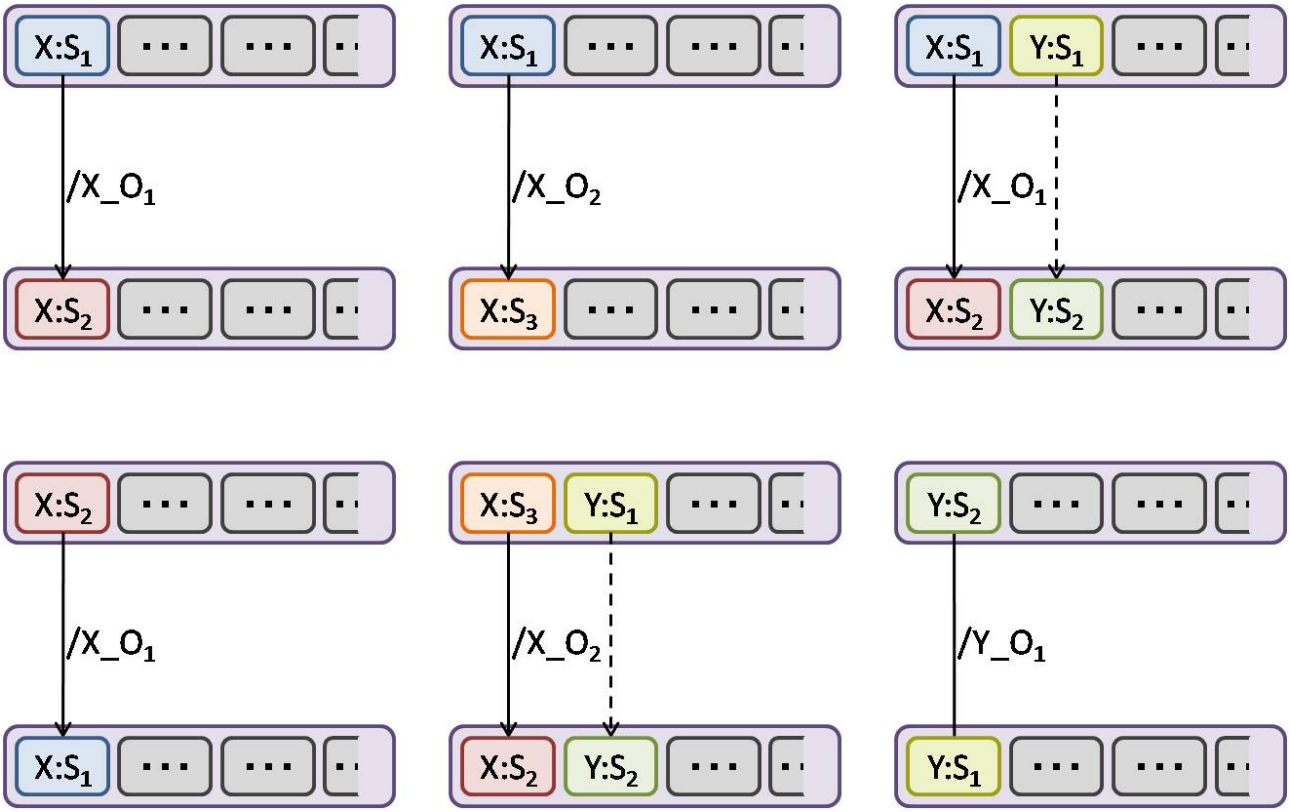


図 3.2: 遷移抽出図の例

3.4　モデル検査用クラス図

検査対象と外部環境の静的な関係を記述したクラス図を**モデル検査用クラス図**と呼ぶこととする。

3.4.1　モデル検査用クラス図の構成

検査対象と外部環境の関係を考慮してモデル検査用クラスを記述する。外部環境は検査する性質に考慮し、分離して記述する事が望ましい。モデル検査用クラス図に記述する情報を以下に示す。

- C：クラス c の集合 $c \in C$
- $c.A$：クラス c の属性集合
- $c.O$：クラス c の操作集合
- M：多重度 $M \in N \cup \{*\}$
- R：クラス間の関連集合 $R \subseteq M_X \times C_X \times C_Y \times M_Y$ (M_X は C_X の多重度、M_Y は C_Y の多重度である。)

操作集合 $c.O$ は遷移抽出図の各状態からの遷移の操作の集合と一致するように記述する。

モデル検査用クラス図の例を図 3.3 に示す。例では検査対象クラス $TARGET$ に対して、クラス X とクラス Y が外部環境としている。外部環境のクラスには、そのクラスが持つ属性と操作を記述する。また、クラス間の関連には多重度を記述する。

3.5　モデル検査用ステートマシン図

クラスレベルの状態の遷移を表現するため、ステートマシン図を記述する。本研究では、既存の UML では表記できない情報を取り扱ため、UML の拡張を行った。拡張したステートマシン図を**モデル検査用ステートマシン図**を呼ぶこととする。追加して持たせる情報はクラスレベルの遷移条件と誘導遷移に関する情報である。各モデル検査用ステートマシン図は遷移条件図と誘導遷移図を持つ。遷移条件図にはクラスレベルのガード条件を記述する。誘導遷移図には誘導するクラスと、クラスの遷移先の集合を記述する。

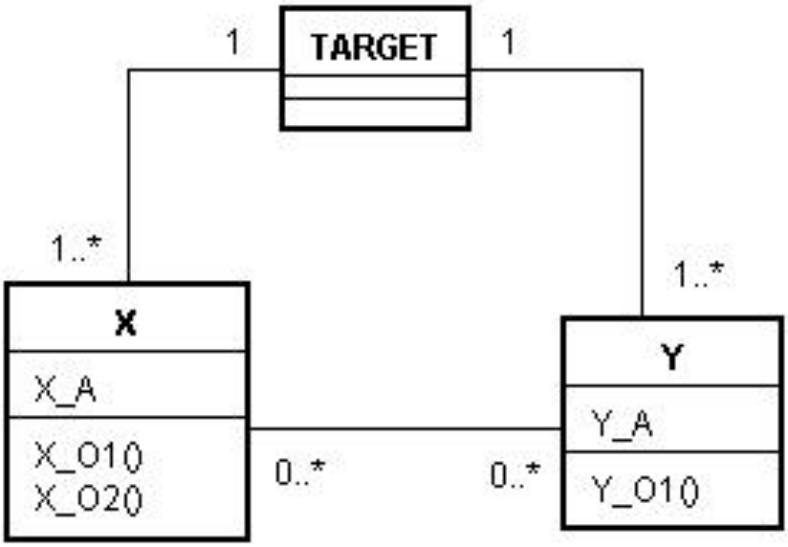


図 3.3: モデル検査用クラス図の例

3.5.1 モデル検査用ステートマシン図の構成

モデル検査用ステートマシン図は遷移抽出図の遷移の情報をもとに記述する。遷移抽出図の遷移集合 T に、クラスレベルの遷移条件 CC と誘導遷移 CY の情報を追加したクラスレベルの遷移集合 CT を記述する。モデル検査用ステートマシン図に記述する情報を以下に示す。

- S：状態 s の集合 $s \in S$
- s_0：初期状態 $s_0 \in S$
- $s.O$：状態 s の操作の集合
- CC：クラスレベルの遷移条件の集合 $cc \in CC$
- CY：クラスレベルの誘導遷移の集合 $cy \in CY$
- CT：クラスレベルの遷移集合 $CT \subseteq S \times O \times CC \times CY \times S \quad ct \in CT$

クラスレベルの遷移条件と誘導遷移の情報を一般的な UML のステートマシン図では表記できない。よって、本研究独自の表記法を定義する必要がある。各遷移条件 CC は通常の UML でガード条件を表記する箇所に記述する。つまり、”[”と”]”の間である。誘導遷移 CY は操作 O の後に”{”と”}”の間に表記する。また、誘導条件がない場合は Null と記述する。表記法をまとめると以下のようになる。

[遷移条件 cc] / 操作 o { 誘導遷移 cy }

クラス X とクラス Y のモデル検査用ステートマシン図の例を図 3.4 と図 3.5 に示す。

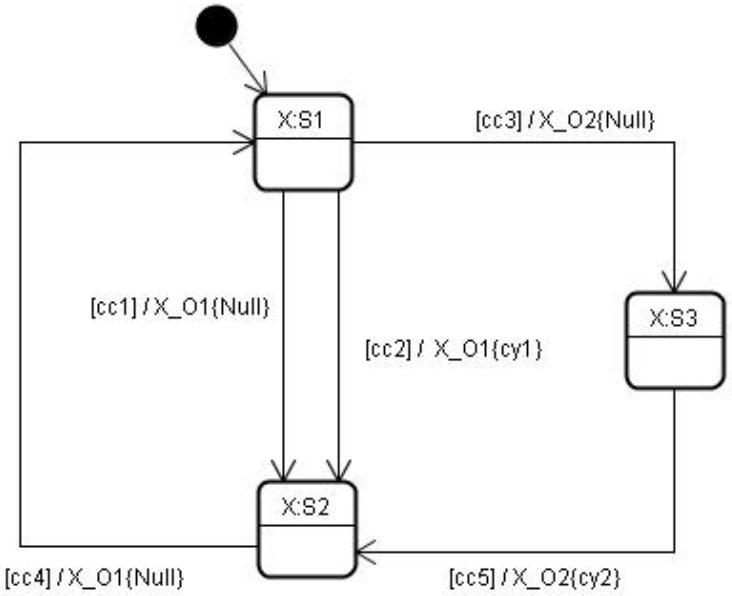


図 3.4: モデル検査用ステートマシン図の例 (クラス X)

クラスレベルの遷移条件集合 CC と誘導遷移集合 CY は機械的な処理が行いやすいように表としてまとめる。また、表として記述する事により遷移の制約を明確に記述できる。

3.5.2 遷移条件表

クラスレベルの遷移条件集合 CC を表として表したものを**遷移条件表**と呼ぶ事とする。遷移条件表にはクラスレベルで取り扱える遷移条件のみを記述する。つまり、インスタンス化してはじめて特定できる遷移条件については記述しない。遷移条件表は各遷移条件 cc を他クラスの状態、属性の値、関連の有無など様々な要因から考え整理して記述する。CC は真理値型の論理式 cf の組み合わせにより成り立つ。遷移条件表には cc を cf の任意の組み合わせで表現できるように記述する。行に cc、列に論理式 cf を記述する。CF はクラスレベルで表現できる真理値型の式である。各項には TRUE、FALSE、”-”(ハイフン) を記述する。TRUE、FALSE はそれぞれ cf が真の場合と偽の場合を意味する。”-”は cf が cc に関係しない事を表現する。cc を各行の論理積で記述する。また、論理和の表現は破線による複数の行として記述する。遷移条件表の例を表 3.1 に示す。

例図の各行の意味を説明する。cc_1 は cf_1、cf_2 が真で cf_4 が偽、cf_3、cf_5 は関係しない論理式である事を意味する。つまり、

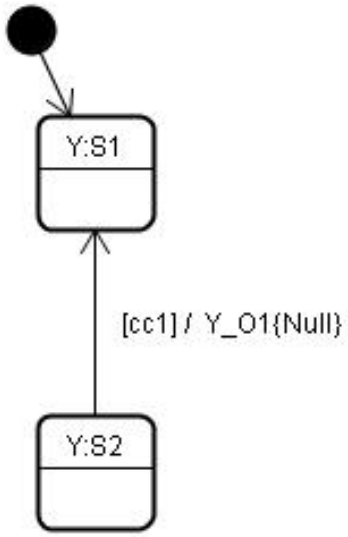


図 3.5: モデル検査用ステートマシン図の例 (クラス Y)

表 3.1: クラス X のクラスレベルの状態遷移表の例

	cf_1	cf_2	cf_3	cf_4	cf_5
cc_1	TRUE	TRUE	−	FALSE	−
cc_2	FALSE	TRUE	−	−	−
cc_3	−	TRUE	TRUE	−	TRUE
	−	FALSE	TRUE	−	−
cc_4	FALSE	FALSE	TRUE	TRUE	−
cc_5	−	−	−	−	TRUE

$cc_1 = cf_1 \wedge cf_2 \wedge \neg(cf_4)$

を意味する。同様に、各 cc は

$cc_2 = \neg(cf_1) \wedge cf_2$

$cc_3 = (cf_2 \wedge cf_3 \wedge cf_5) \vee (\neg(cf_2) \wedge cf_3)$

$cc_4 = \neg(CF_1) \wedge \neg(cf_2) \wedge cf_3 \wedge cf_4$

$cc_5 = cf_5$

を意味する。状態遷移表の情報をモデル検査用ステートマシン図にそのまま記述することもできると考えられるが、本研究では表として分離して記述した。なぜなら、遷移条件を各側面ごとに着目した論理式に分離して記述する事により遷移条件の実体化が行いやすくなるためである。これにより、機械的にインスタンスレベルの遷移条件を生成することが容易になる。

3.5.3 誘導遷移表

クラスレベルの誘導遷移集合 CY を表として表記したものを**誘導遷移表**と呼ぶこととする。表記する内容は誘導されるクラス YC と、その遷移先の状態 YS である。誘導しない遷移については無視し、誘導する遷移の情報だけをまとめる。表記法は各行に cy を記述し、それに対する YC と YS を列とする。複数の誘導遷移がある場合は破線による複数の行として記述する。誘導遷移図の例を表 3.2 に示す。

表 3.2: クラス X のクラスレベルの誘導遷移表の例

	YC	YS
cy_1	yc_1	ys_2
cy_2	yc_2	ys_1
	yc_3	ys_2

例図の各行の意味を説明する。cy_1 は論理式 yf_1 が成り立つクラスを ys_2 に遷移させるという意味である。cy_2 は二つの遷移を誘導を意味する。yf_2 が成り立つクラスを ys_1、yf_3 が成り立つクラスを ys_2 に遷移させるという意味である。

各 YF はクラスレベルの内容で記述する必要があり、インスタンスレベルでしか表記できない内容は記述しない。

3.6　同意義パターン定義

モデル検査用クラス図の多重度、属性に対して任意の値を実体化しモデル検査用オブジェクトを生成する。しかしながら、このモデル検査用オブジェクトの組み合わせのパターン数は、各クラスの実体化した値の直積をとるため膨大になる。例として、２つの外部環境のクラスを持つモデル検査用クラス図の組み合わせパターン数を計算する。それぞれのクラスの多重度を 3 として実体化する。また、属性の値を 2 値の変数として実体化する。このとき、多重度実体化の段階で組み合わせパターン数は $2^{3\times3} = 512$ になる。さらに、属性の値を実体化すると $512 \times 2^3 \times 2^3 = 32768$ の組み合わせパターンになる。このすべてのモデル検査用オブジェクトに対して検査モデルを生成する事は困難である。そのため、本研究では同じ振舞いをすると考えられる組み合わせパターンを同一と見なすことでパターン数の削減を行う。多重度、属性を実体化する場合に同じ振舞いをするかは検査対象による異なる。よって、検査対象に対する同意義パターンを定義し組み合わせパターン数を削減する。定義する同意義パターンは多重度同意義パターンと属性同意義パターンの 2 つである。つまり、本研究では図 3.6 に示すように多重度同意義パターンと属性同意義パターンの 2 段階で組み合わせパターン除去を行う。

3.6.1　多重度同意義パターン

多重度を実体化する事で得られるオブジェクトのパターン数は直積をとるため膨大になる。そのすべてに対して検査モデルを生成し、モデル検査を行う事が信頼性の高いソフトウェアの設計につながると考えられる。しかしながら、オブジェクトの組み合わせパターン数は多重度に対して指数関数的に増加するため、すべてのオブジェクトに対してモデル検査する事は困難である。よって、多重度を実体化した集合の中で同じ振舞いを行うと考えられる組を同一とする**多重度同意義パターン**を定義しパターン数を制限する。モデル検査用オブジェクトは無向グラフで表現できる。ノード、エッジに対して多重度同意義パターンを与え、組み合わせパターン数を削減する。多重度同意義パターンは検査対象によって異なるため、検査対象毎に定義する。

3.4 節で記述したモデル検査用クラス図をもとに多重度を実体化した例を図 3.7 に示す。実体化する多重度はクラス X に対して 2、クラス Y に対して 1 とする。この時、クラス X とクラス Y の関連は 2 つである。よって、組み合わせパターンは 2 つの関連がある場合とない場合の $2^2 = 4$ パターンとなる。

図 3.7 において右上の図と左下の図は X の一方が Y と関連し、もう一方は Y と関連しない。よって、全体の振舞いは同じになると推定できる。このような組み合わせパターンを多重度同意義パターンとし組み合わせパターン数を削減する。

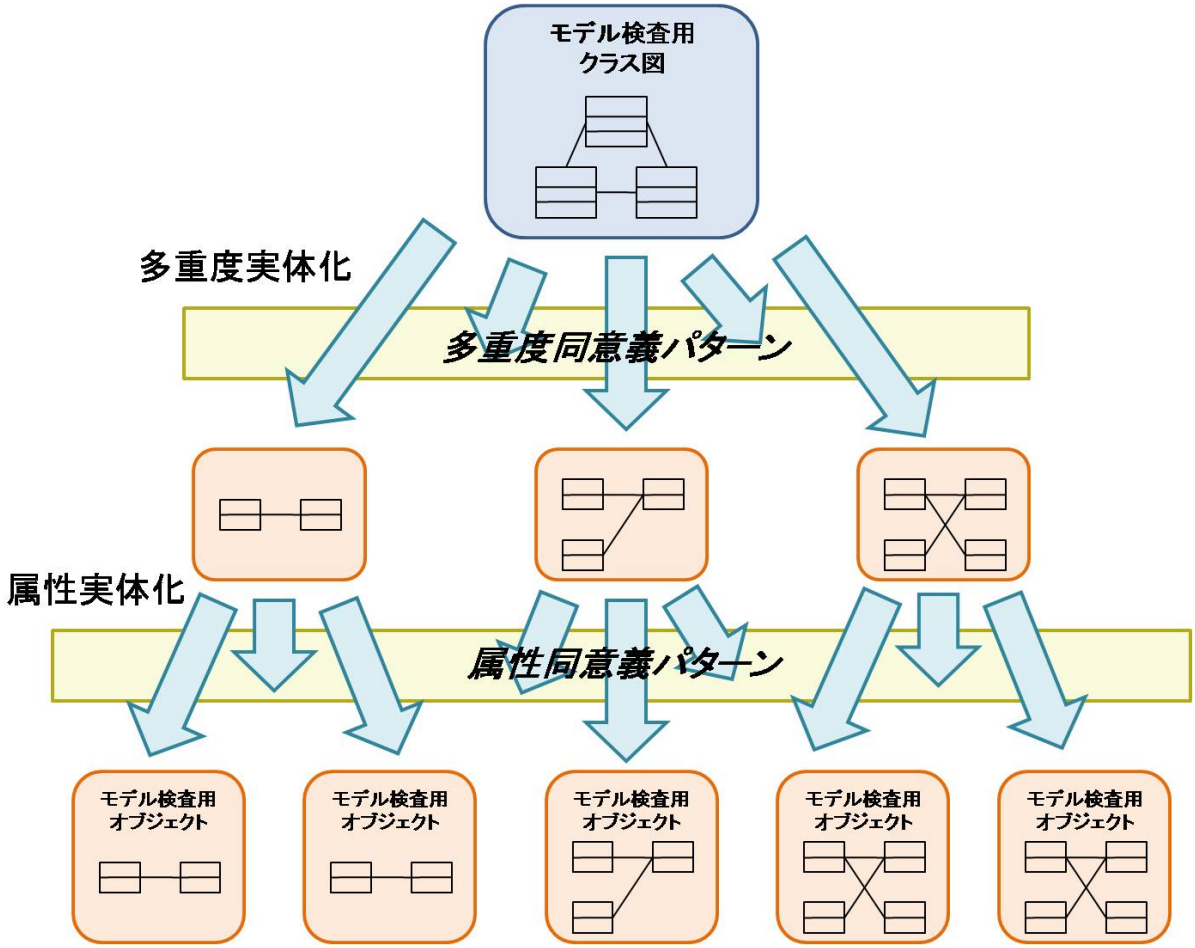


図 3.6: 同意義パターンによる組み合わせパーン除去の流れ

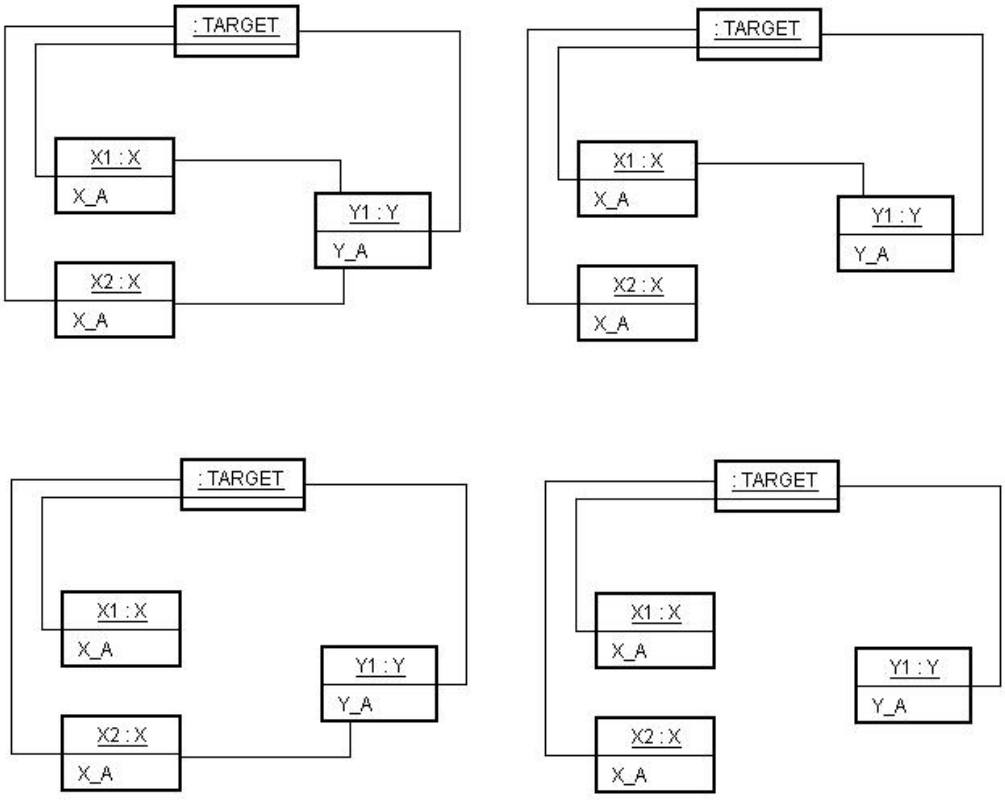


図 3.7: 多重度の実体化の例

3.6.2 属性同意義パターン

多重度実体化同様、属性の実体化においてもパターン数は指数関数的に増加する。そのため、組み合わせパターン数を現実的な数に制限する事はモデル検査の効率化につながると考えられる。よって、属性の実体化した集合の中で同じ振舞いを行うと考えられる組を同意義とする**属性同意義パターン**を定義し組み合わせパターン数を削減する。しかしながら属性を実体化するため、同意義であると定義しても実際には振舞いが異なる場合も考えられる。そのため属性同意義パターンを定義する際は、検査対象の検査したい性質に十分に考慮する必要がある。定義法は多重度同意義パターンと同様、オブジェクトを無向グラフとして表現する。ノードに属性の情報を追加し、エッジとの関係性をもとに、属性同意義パターンを定義する。属性同意義パターンは検査対象によって異なるため、検査対象毎に定義する。

図 3.7 の左上の組み合わせパターンに対して属性を実体化した例を図 3.8 に示す。実体化する属性は X_ A を { 1,2 } の 2 値、Y_ A を { 1 } の 1 値とする。この時、$2^2 \times 1^1 = 4$ の組み合わせパターン数となる。

図 3.8 の右上と左下の組み合わせパターンに着目する。左下の図は X_ A=1 である X_1 と X_ A=2 である X_2 が Y_1 とリンクしている。右下の図は X_ A=2 である X_1 と X_ A=1 である X_2 が Y_1 とリンクしている。これらはオブジェクトとしては全く別のものである。しかしながら、検査する性質が Y_1 の Y_ A に対して一方のオブジェクトの X_ A が大きく、もう一方の X_ A が等しい場合のみ検査すれば十分である場合が考えられる。そのような場合、両方とも検査モデルを生成する手間は冗長である。よって、属性同意義パターンを定義し組み合わせパターン数を削減する。

3.7 モデル検査用オブジェクト

モデル検査用クラス図をもとに多重度、属性をインスタンス化してオブジェクトを生成する。生成されたオブジェクトの中で同意義パターンより除去されたなかったものを**モデル検査用オブジェクト**と呼ぶこととする。

3.7.1 モデル検査用オブジェクトの構成

モデル検査用オブジェクトを以下に示す。

- OB：オブジェクトの集合 $ob \in OB$
- $ob.A$：インスタンスレベルの属性集合
- OBR：インスタンスレベルの関連集合 $ObR \subseteq Ob \times Ob$

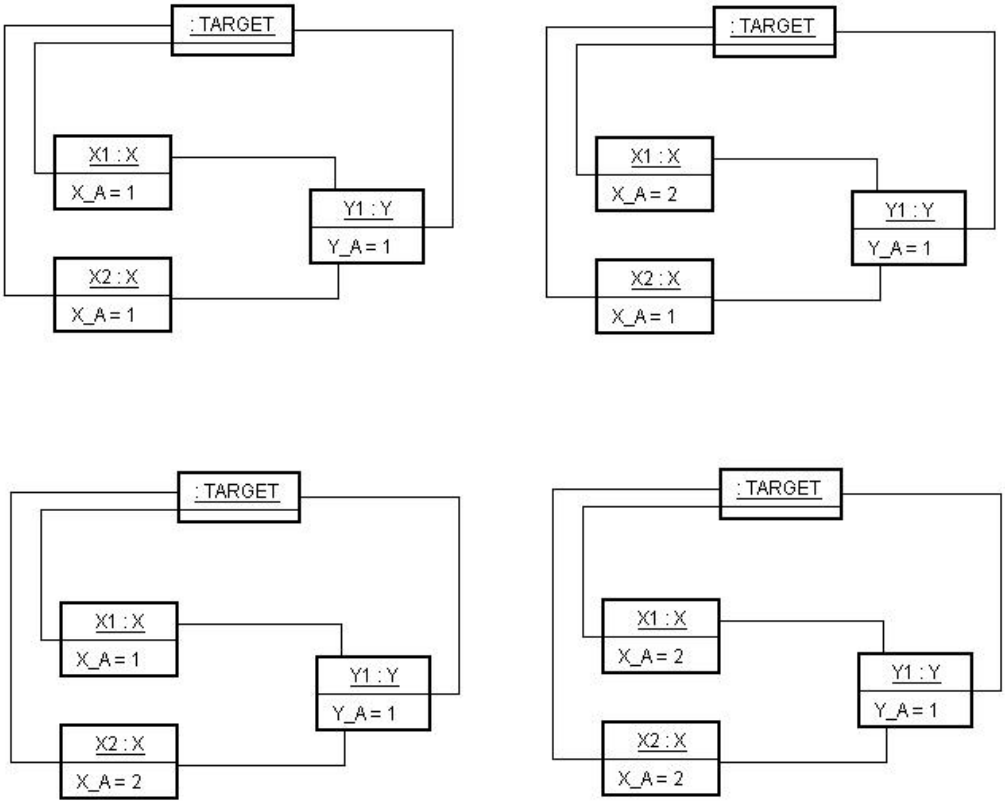


図 3.8: 属性の実体化の例

3.8 環境統合アルゴリズム

各モデル検査用オブジェクトに対して環境統合を行う。本研究では、この環境統合の処理手順である**環境統合アルゴリズム**を作成した。この環境統合アルゴリズムを適用するために、モデル検査用ステートマシン図のクラスレベルの遷移条件表と誘導遷移表を実体化する。

3.8.1 遷移条件表の実体化

モデル検査用オブジェクトをもとにクラスレベルの遷移条件表を実体化する。具体的にはクラスレベルの論理式 CF をインスタンスレベルの論理式 OBF として実体化する。OBF はインスタンスレベルならばすべて記述可能である。すべての OBF が特定されれば、インスタンスレベルの遷移条件集合 OBC が得られる。

クラスレベルの遷移条件表の表 3.1 の実体化の例を表 3.3 に示す。各 cbc は obf の組み合わせで記述できる。

$$obc_1 = obf_1 \wedge obf_2 \wedge \neg(obf_4)$$

$$obc_2 = \neg(obf_1) \wedge obf_2$$

$$obc_3 = (obf_2 \wedge obf_3 \wedge obf_5) \vee (\neg(obf_2) \wedge obf_3)$$

$$obc_4 = \neg(obf_1) \wedge \neg(obf_2) \wedge obf_3 \wedge obf_4$$

$$obc_5 = obf_5$$

表 3.3: 遷移条件表の実体化の例

	obf_1	obf_2	obf_3	obf_4	obf_5
obc_1	TRUE	TRUE	−	FALSE	−
obc_2	FALSE	TRUE	−	−	−
obc_3	−	TRUE	TRUE	−	TRUE
	−	FALSE	TRUE	−	−
obc_4	FALSE	FALSE	TRUE	TRUE	−
obc_5	−	−	−	−	TRUE

3.8.2 誘導遷移表の実体化

モデル検査用オブジェクトをもとにクラスレベルの遷移条件表を実体化する。インスタンスレベルの誘導遷移集合OBYの各要素であるobyをYOBとYSで記述する。クラスレベルで記述した式YFをもとに誘導されるオブジェクトYOBを実体化する。遷移先の状態YSはインスタンス化しても変化しない。

クラスレベルの誘導遷移表の例3.2の実体化の例を表3.4に示す。oby_1はオブジェクトY_1が状態S_2に誘導遷移されるという情報である。oby_2はオブジェクトY_1が状態S_2、オブジェクトY_2が状態S_2に誘導遷移されるという情報である。

表 3.4: 誘導遷移表の実体化の例

	YOB	YS
oby_1	Y_1	S_2
oby_2	Y_1	S_2
	Y_2	S_2

3.8.3 環境統合アルゴリズムへの入力と出力

環境統合アルゴリズムに対する入力情報を以下に示す。

- OB：オブジェクトの集合 $ob \in OB$
- $ob.A$：オブジェクトの属性集合
- $ob.R$：オブジェクトの関連集合 $ob.R \subseteq Ob \times Ob$
- $ob.S$：オブジェクトの状態集合 $ob.s \in ob.S$
- $ob.s_0$：オブジェクトの初期状態 $ob.s_0 \in ob.S$
- $ob.O$：オブジェクトの操作集合
- OBC：インスタンスレベルの遷移条件の集合 $obc \in OBC$
- OBY：インスタンスレベルの誘導遷移の集合 $oby \in OBY$
- OBT：インスタンスレベルの遷移集合$OBT \subseteq OB.S \times OB.O \times OBC \times OBY \times OB.S$
  $obt \in OBT$

OB、$ob.A$、$ob.R$はモデル検査用オブジェクトに記述されている。$ob.S$、$ob.S_0$、$ob.O$についてはモデル検査用ステートマシン図に記述されている内容と同じである。OBC、OBYは遷移条件表の実体化、誘導遷移表の実体化により得られる。$OB.S$、$OB.S_0$、$OB.O$、OBC、OBY よりインスタンスレベルの遷移集合OBTが得られる。

環境統合アルゴリズムによる出力情報を以下に示す。出力は検査モデルであり、ステートマシン図で表記する。

- US：統合状態の集合 $US \subseteq ob.s_1 \times ob.s_2 \times, ..., \times ob.s_x$ (xは統合するオブジェクトの数) $us \in US$
- us_0：初期統合状態 $us_0 \in US$
- $us.O$：統合状態の操作集合
- UT：統合状態の遷移集合$UT \subseteq US \times US.O \times US \quad ut \in UT$

3.8.4 木構造による環境統合

環境統合は木探索アルゴリズムにより行う。環境統合アルゴリズムはキュー構造、リスト構造、関数を用いる。各関数を以下のように定義した。

•$Enqueue(Q, a)$
要素 a をキュー Q の最後尾に追加する。

•$Dequeue(Q)$
先頭の要素 a をキュー Q から除く。除いた要素 a を返す。

•$CheakCondition(obt)$
obt の obc が成立するならば true、成立しないならば false を返す。

•$NextUS(us, obt)$
統合状態 us から遷移 obt を行った遷移後の統合状態 us' を返す。

•$AddUT(D, us, obt)$
統合状態 us、統合遷移 obt.O、遷移 obt による遷移後の統合状態 us' の情報を UT としてリスト D に追加する。

環境統合アルゴリズム$EnmironmentConbine$を以下に示す。引数としてインスタンスレベルの遷移集合 OBT を与える。統合状態の遷移 UT はリスト構造に保存するものとす

る。本研究では木構造の探索アルゴリズムに幅優先探索を用いているが、深さ優先探索でも結果は同じとなる。

```
• EnvironmentCombine(OBT){
    空のキュー Q を作成
    空のリスト D を作成
    各 ob の s_0 の組み合わせを us_0 とする
    Enqueue(Q,us_0)
    while(Q!=EMPTY)
      x=Dequeue(Q)
        for(i=0;x のすべての obt_i;i++){
        if(CheakCondition(obt_i)==true){
          AddUT(D,x,obt_i)
          if(NextUS(x,obt_i) がマークされていない){
            x をマークする
            Enqueue(x,obt_i)
          }
        }
      }
    }
}
```

図 3.9 のモデル検査用オブジェクトの例に対して、環境統合アルゴリズムを適用した例を図 3.10 に示す。

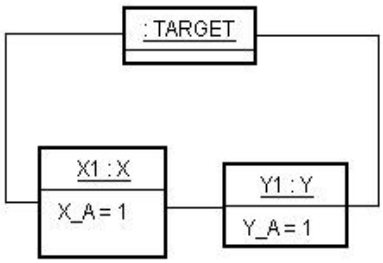


図 3.9: モデル検査用オブジェクトの例

各オブジェクトの初期状態である $X : S_1$ と $Y : S_1$ の統合状態を根とする木構造を作成する。根である統合状態 $(X : S_1, Y : S_1)$ について探索する。まず、$X : S_1$ の遷移を見る。$X : s_1$ が持つ遷移は 3 種類ある。遷移の遷移条件が真ならば遷移し、偽ならば無視する。1 つめの遷移は遷移条件が偽のため、無視する。2 つめの遷移は遷移条件が真のため、X オブジェクトが S_3 に遷移できる。よって、$(X : S_1, Y : S_1)$ の子に $(X : S_3, Y : S_1)$ に追

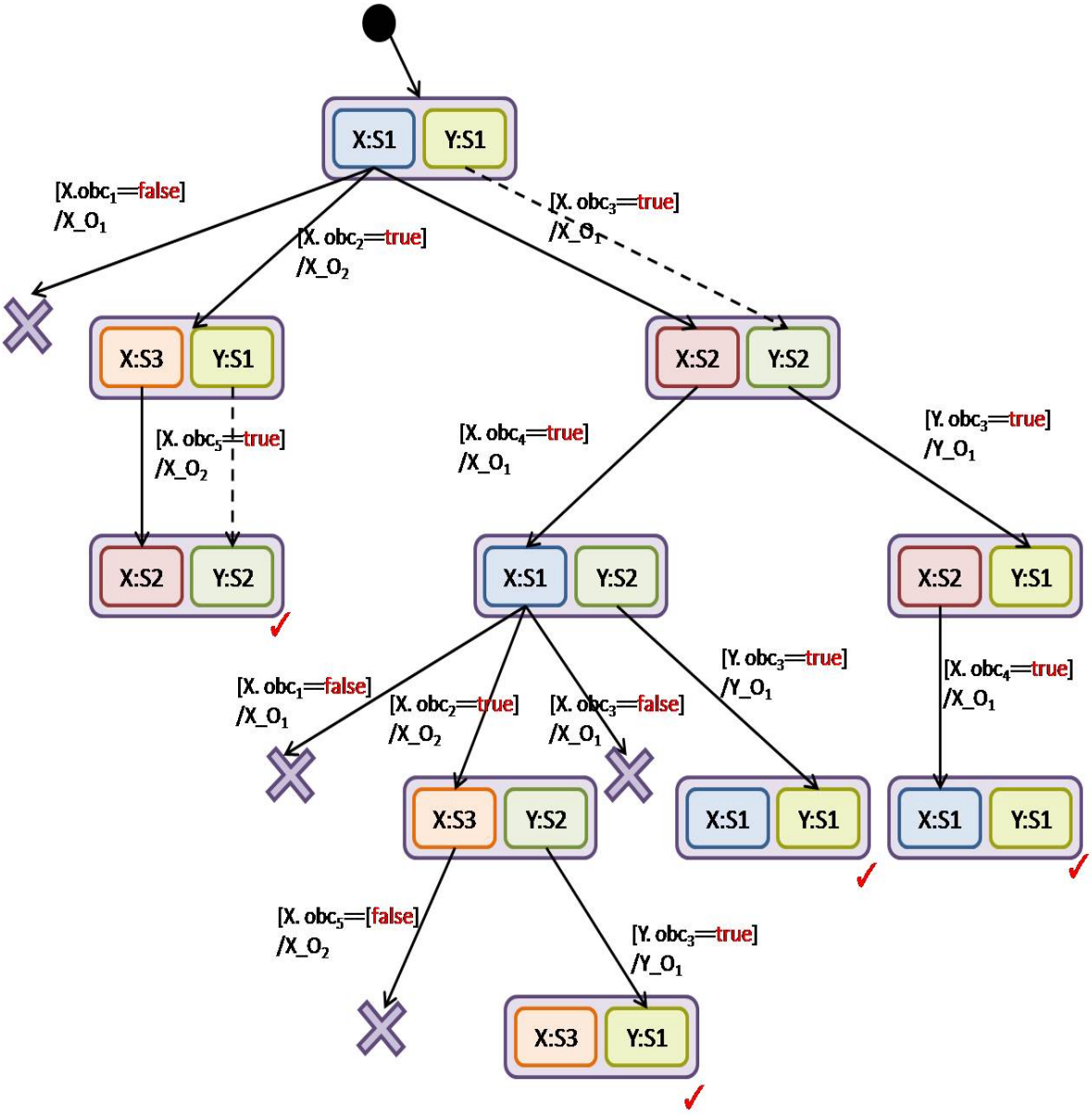


図 3.10: 環境アルゴリズムの適用例

加する。3 つめの遷移は遷移条件が真のため、X オブジェクトが S_2 に遷移できる。この遷移は Y オブジェクトを S_1 に誘導させる誘導遷移を持つ。よって、$(X : S_1, Y : S_1)$ の子に $(X : S_2, Y : S_2)$ に追加する。次に、$Y : S_1$ の遷移を見る。オブジェクト Y は遷移を持たない。統合状態 $(X : S_1, Y : S_1)$ のすべてのオブジェクトについてすべての遷移を洗い出した。よって、統合状態 $(X : S_1, Y : S_1)$ の探索は終える。幅優先探索により統合状態 $(X : S_3, Y : S_1)$ について探索する。$X : S_3$ の遷移は遷移条件が真のため、X オブジェクトが S_2 に遷移できる。この遷移は Y オブジェクトを S_1 に誘導させる誘導遷移を持つ。よって、$(X : S_1, Y : S_1)$ の遷移先は $(X : S_2, Y : S_2)$ となる。$(X : S_2, Y : S_2)$ は既に木構造にあるため無視する。同様に、$Y : S_1$ の遷移を見るが、オブジェクト Y は遷移を持たないため、$(X : S_3, Y : S_1)$ の探索は終える。次に $(X : S_2, Y : S_2)$ の探索を行う。$X : S_2$ の遷移は遷移条件が真のため、X オブジェクトが S_1 に遷移できる。よって、$(X : S_2, Y : S_2)$ の子に $(X : S_1, Y : S_2)$ に追加する。$Y : S_1$ の遷移は遷移条件が真のため、Y オブジェクトが S_1 に遷移できる。よって、$(X : S_2, Y : S_2)$ の子に $(X : S_2, Y : S_1)$ に追加する。$(X : S_2, Y : S_2)$ のすべての遷移を洗い出したため探索は終える。このように木構造のすべての葉の探索が完了したら、アルゴリズムは終了する。

3.9 検査モデル

環境統合アルゴリズムにより検査モデルが得られる。検査モデルはステートマシン図で記述されている。だが、モデル検査器 SPIN により検査を行うためには専用言語 Promela に変換する必要がある。よって、本研究では Promela への変換法を提案する。

3.9.1 検査モデルの構成

検査モデルはステートマシン図で表現する。図 3.10 の例で得られた検査モデルを図 3.11 に示す。

3.9.2 Promela への変換

ステートマシン図記述の検査モデルの各振舞いは Promela 記述に対して 1 対 1 に対応させることができる。各対応を表 4.10 に示す。なお、初期状態は *proctype* 内のラベルの中で最も上位に記述する事とする。

表 4.10 の対応により、図 3.11 を Promela に変換した。Promela 記述に変換した検査モデルを図 3.12 に示す。

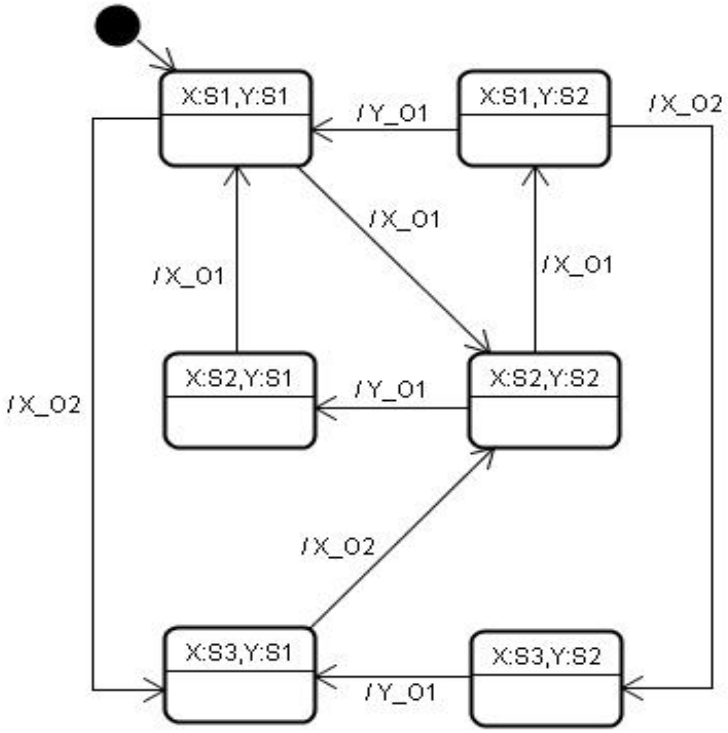


図 3.11: 検査モデルの例

表 3.5: ステートマシン図記述の検査モデルと Promela 記述の検査モデルの対応

ステートマシン図記述	Promela 記述
状態	ラベルで表現
操作	inline 関数で表現
遷移	if 文と goto 文で表現

```
active proctype X_Y(){
XS1_YS1:
  if
  ::X_O1() ->
    goto XS2_YS2;
  ::X_O2() ->
    goto XS3_YS1;
  fi;

XS1_YS2:
  if
  ::X_O2() ->
    goto XS3_YS2;
  ::Y_O1() ->
    goto XS1_YS1;
  fi;

XS2_YS1:
  if
  ::X_O1() ->
    goto XS1_YS1;
  fi;

XS2_YS2:
  if
  ::X_O1() ->
    goto XS1_YS2;
  ::Y_O1() ->
    goto XS2_YS1;
  fi;

XS3_YS1:
  if
  ::X_O2() ->
    goto XS2_YS2;
  fi;

XS3_YS2:
  if
  ::Y_O1() ->
    goto XS3_YS1;
  fi;
}
```

図 3.12: Promela 記述の検査モデルの例

第4章　例題を用いての適用実験

本章では、提案手法の OSEK/VDX 仕様に対して適用実験の各ステップの結果を説明する。

4.1　実験:遷移抽出図

Task クラスと Resource クラスの各状態に対する遷移抽出図を記述する。各クラスの各状態の遷移抽出図を図 4.1 から図 4.4 に示す。遷移の分析は遷移元となるタスクや資源 (以下自タスク、自資源と呼ぶ) の状態からの視点で行う。また、自タスク、自資源以外のタスクを他タスク、他資源と呼ぶこととする。

- Task クラス:Suspended 状態

  Suspended 状態からの遷移は 3 種類ある。いずれも操作 *ActivateTask* を呼び出しての遷移である。1 つめは、自タスクのみが Running 状態になる遷移である。2 つめは、自タスクが Running 状態になり、Running 状態のタスクを Ready 状態に誘導する遷移である。3 つめは、自タスクのみが Ready 状態になる遷移である。

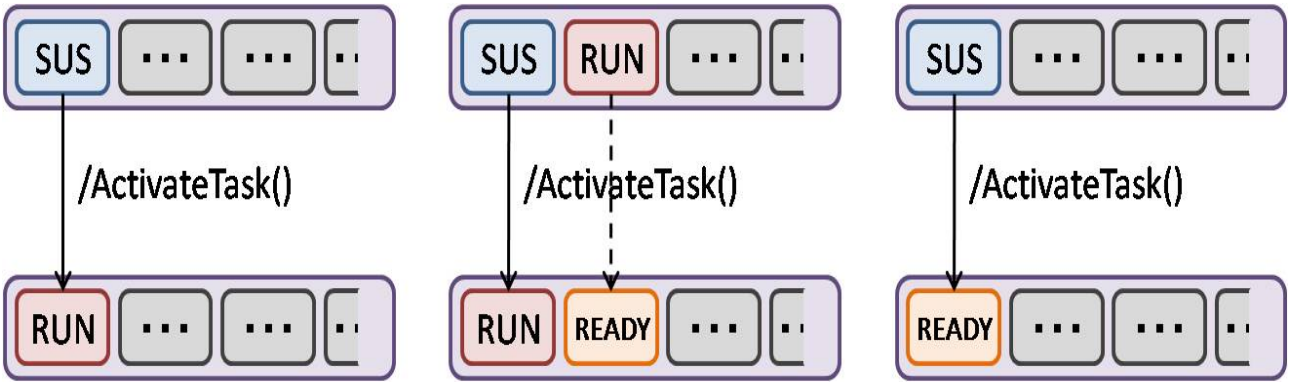


図 4.1: Task クラス:Suspended 状態の遷移抽出図

- Task クラス:Running 状態

  Running 状態からの遷移は 6 種類ある。*TerminateTask* を呼び出しての遷移は 2 種類ある。1 つめは、自タスクのみが Suspended 状態になる遷移である。2 つめは、自

タスクが Suspended 状態になり、Ready 状態のタスクを Running 状態に誘導する遷移である。自タスク自身に対しての $ChainTask$ を呼び出し場合の遷移は 2 種類ある。1 つめは、自タスクのみが再び Running 状態になる遷移である。2 つめは、自タスクが Ready 状態になり、Ready 状態のタスクを Running 状態に誘導する遷移である。他タスクに対して $ChainTask$ を呼び出す場合の遷移は 2 種類ある。1 つめは、自タスクが Suspended 状態になり、Suspended 状態のタスクを Running 状態に誘導する遷移である。2 つめは、自タスクが Suspended 状態になり、Suspended 状態のタスクを Ready 状態に誘導し、かつ Ready 状態のタスクを Running 状態に誘導する遷移である。

- Task クラス:Ready 状態

  Ready 状態からの遷移はすべて他クラスからの遷移に誘導される遷移である。よって、Task クラス:Ready 状態に対する遷移抽出図はない。

- Reasource クラス:Free 状態の遷移抽出図

  Free 状態からの遷移は 1 種類のみである。$GetResource$ を呼び出し、Occupied 状態になる遷移である。

- Reasource クラス:Occupied 状態

  Occupied 状態からの 2 種類である。いずれも $ReleaseResource$ を呼び出しての遷移である。1 つめは、自資源のみが Free 状態になる遷移である。2 つめは、自資源が Free 状態になり、Running 状態のタスクを Ready 状態に誘導し、かつ Ready 状態のタスクを Running 状態に誘導する遷移である。

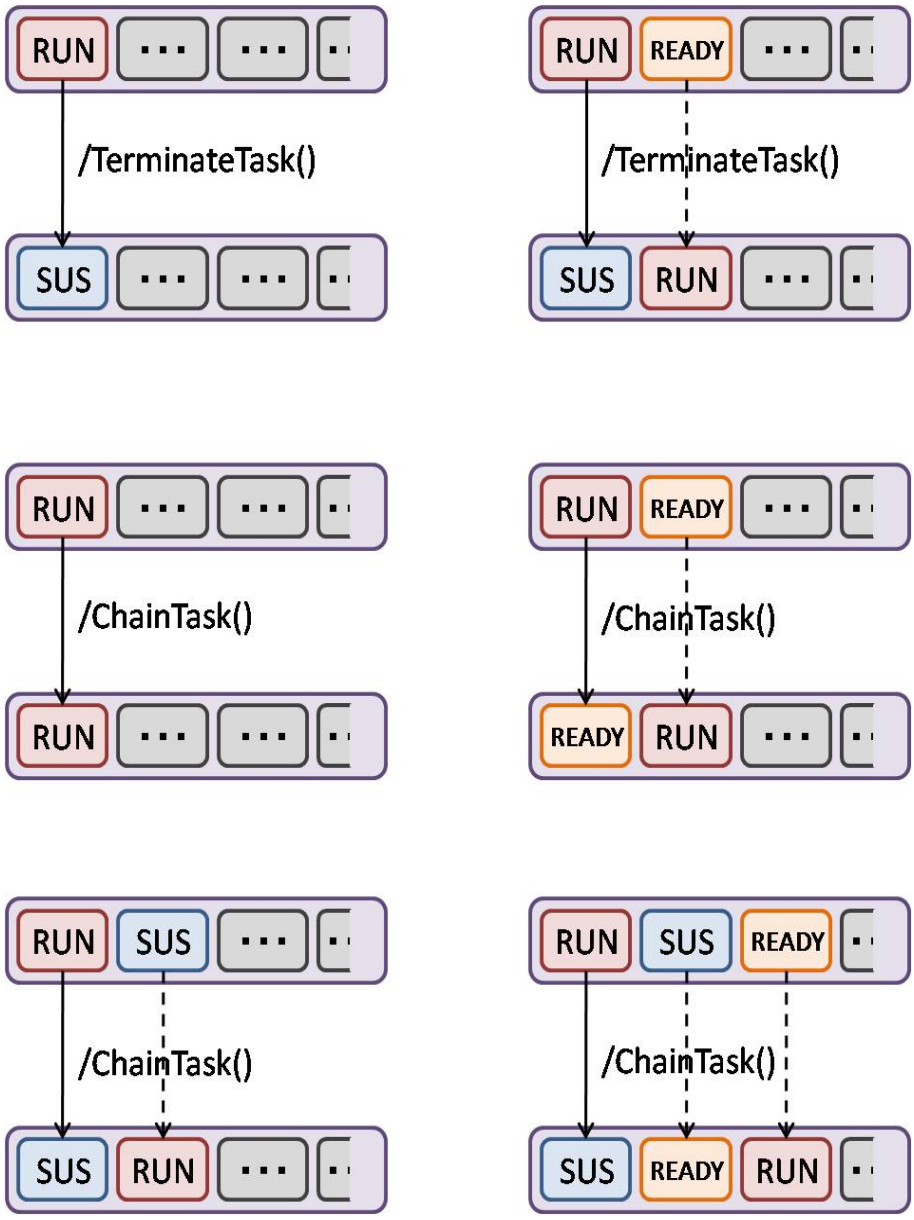


図 4.2: Task クラス:Running 状態の遷移抽出図

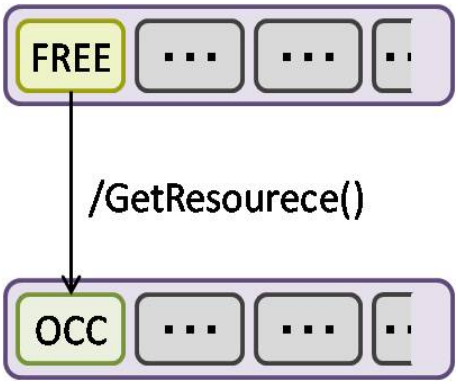


図 4.3: Resource クラス:Free 状態の遷移抽出図

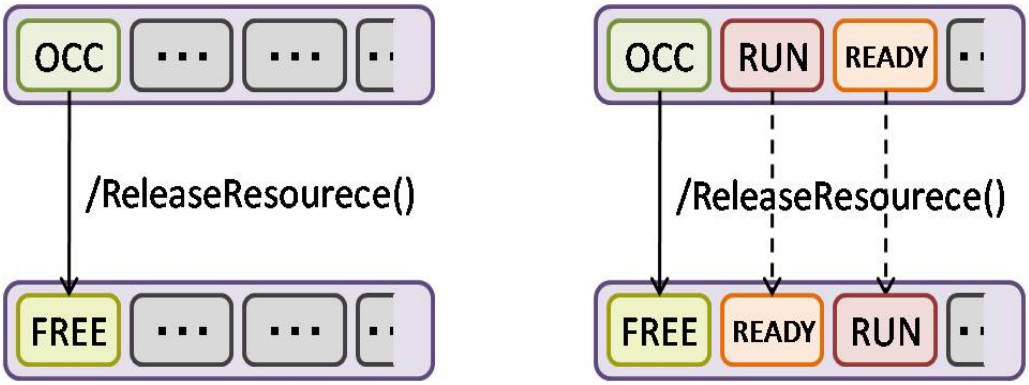


図 4.4: Resource クラス:Occupied 状態の遷移抽出図

4.2 実験:モデル検査用クラス図

検査対象に対してのモデル検査用クラス図を図 4.5 に示す。検査対象 OSEK に対しての外部環境クラスはタスククラスと資源クラスの二つである。RTOS クラスとタスククラスには関連があり、多重度は 1 つの RTOS に対して 0..∗ である。同様に RTOS クラスの資源クラスにも関連があり、多重度は 1 つの RTOS に対して 0..∗ である。タスク、資源間にも関連がある。タスクに対しての資源の多重度は 0..8 であり、資源に対してのタスクの多重度は 1..∗ である。タスク間同士にも関連があり、その多重度は 1..∗ である。タスククラスは属性 tpriority、操作 $ActivateTask$、$TerminateTask$、$ChainTask$ を持つ。資源クラスは属性 rpriority、操作 $GetResource$、$ReleaseResource$ を持つ

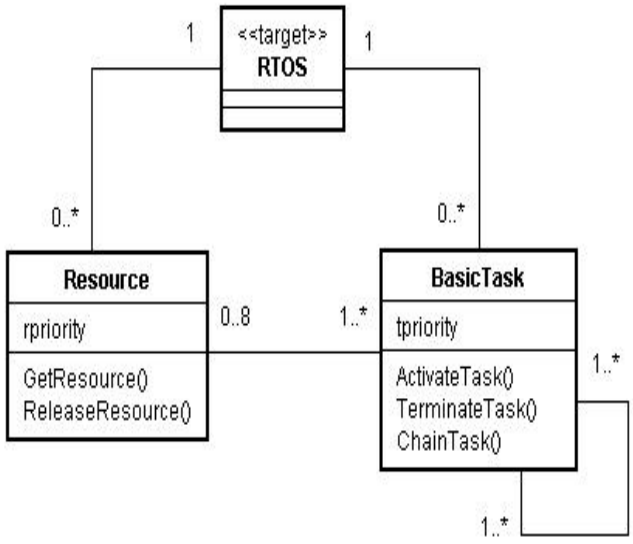


図 4.5: OSEK/VDX 仕様のモデル検査用クラス図

4.3 実験:モデル検査用ステートマシン図

タスククラスのモデル検査用ステートマシン図を図 4.6 に示す。各状態の遷移は遷移抽出図をもとに記述する。各遷移に対して遷移条件 cc の情報を付加する。また、誘導遷移がある場合は cy の情報を付加する。cc はクラスレベルの遷移条件を記述する。cy はクラスレベルの誘導遷移を記述する。

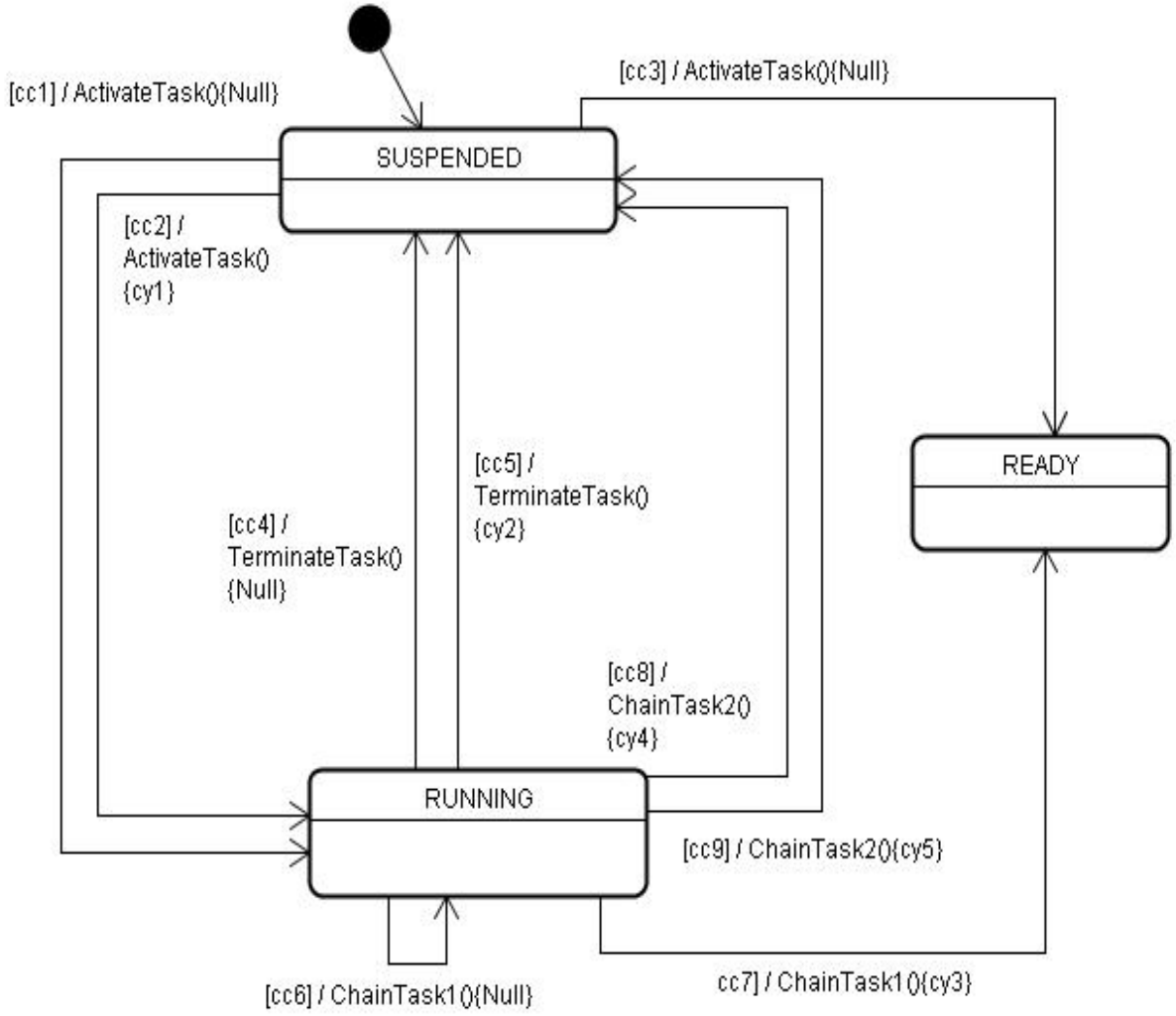


図 4.6: OSEK/VDX 仕様のモデル検査ステートマシン図 (タスククラス)

図 4.6 の例について各 cc と cy を分析する。
分析は自タスクや他タスク、資源の状態や属性の比較などを中心に行う。

- Suspended 状態からの遷移

  Suspended 状態からの遷移は $ActivateTask$ が起点となる場合だけである。まず、他タスクの状態と優先度に着目する。Running 状態の他タスクが存在しない場合を想定

する。Running 状態の他タスクが存在しないならば性質上 Ready 状態のタスクは存在しない。よって、すべてのタスクが Suspended 状態ということになる。このとき、*ActivateTask* が発行されたならば自タスクは Running 状態に遷移する。Running 状態の他タスクが存在する場合を想定する。自タスクの優先度が Running 状態の他タスクより高い場合は、自タスクは Running 状態へ遷移する。Running 状態の他タスクは Ready 状態へ誘導遷移される。自タスクの優先度が Running 状態の他タスクより低い場合は、自タスクは Ready 状態へ遷移する。次に、資源の状態と優先度に着目する。Running 状態へ遷移するためには、OSEK/VDX 仕様の性質上すべての資源が Free 状態である必要がある。Running 状態の他クラスが資源を占有している場合、*ActivateTask* が発行されても優先度に関係なく自タスクは Ready 状態へ遷移する。

- Running 状態からの遷移

  Running 状態からの遷移は *TerminateTask* と *ChainTask* が起点となる場合である。Running 状態のタスクは 1 つしか存在しないはずなので他タスクの状態は Suspended 状態か Ready 状態のいずれかである。

  まず、*TerminateTask* が発行された場合を分析する。Ready 状態の他タスクが存在しない場合を想定する。Ready 状態の他タスクが存在しない場合はすべての他タスクが Suspended 状態である。よって、*TerminateTask* が発行されたならば自タスクは Suspended 状態に遷移する。Ready 状態の他タスクが存在する場合を想定する。この場合、自タスクが Running 状態から Suspended 状態へ遷移すると同時に、最も優先度の高い Ready 状態の他タスクが Running 状態へ誘導遷移される。また Running 状態から他状態へ遷移するためには、すべての資源が Free 状態である必要がある。

  次に、*ChainTask* が発行された場合を分析する。*ChainTask* は自タスクに対して発行される場合と、他タスクに対して発行される場合がある。これらは振舞いが異なるため、別々に分析する。

  まず、*ChainTask* が自タスクに対して発行される場合を分析する。Ready 状態の他タスクが存在しない場合を想定する。他タスクのすべてのが Suspended 状態であるため、自タスクは Running 状態から再び Running 状態に遷移する。Ready 状態の他タスクが存在する場合を想定する。すべての Ready 状態の他タスクの優先度より自タスクの優先度が高い場合、自タスクは Running 状態から再び Running 状態に遷移する。いずれかの Ready 状態の他タスクの優先度より自タスクの優先度が高い場合、自タスクは Running 状態から Ready 状態に遷移し、最も優先度の高い Ready 状態の他タスクが Running 状態に誘導遷移される。

  次に、*ChainTask* が他タスクに対して発行される場合を分析する。*ChainTask* は Ready 状態もしくは Running 状態の他タスクに対して発行されない。Ready 状態の他タスクが存在しない場合を想定する。Ready 状態の他タスクが存在しない場合は

すべての他タスクがSuspended状態である。よって、自タスクはRunning状態からSuspended状態へ遷移し、呼び出されたタスク(以下呼タスクと呼ぶ)はSuspended状態からRunning状態に誘導遷移される。Ready状態の他タスクが存在する場合を想定する。このとき、呼タスクの優先度がすべてのReady状態の他タスクの優先度より高い場合、自タスクはRunning状態からSuspended状態へ遷移し、呼タスクはSuspended状態からRunning状態に誘導遷移される。Ready状態の他タスクが存在する場合を想定する。呼タスクの優先度がReady状態のいずれかの他タスクの優先度より低い場合、自タスクはRunning状態からSuspended状態へ遷移し、呼タスクはSuspended状態からReady状態、最も優先度の高いReady状態の他タスクはRunning状態に誘導遷移される。またRunning状態から他状態へ遷移するためには、すべての資源がFree状態である必要がある。

分析したタスククラスの各ccとcyを、遷移条件表と誘導遷移表としてまとめて表記した表を表4.1と表4.2に示す。

ここで$MAXptask(Task)$をいう関数を便宜的に用意した。関数$MAXptask(Task)$の仕様を以下に示す。

•$MAXptask(Task)$
タスク集合$Task$の中で最も優先度が高い$task$を返す

資源クラスのモデル検査用ステートマシン図を図4.7に示す。
図4.7の例について各ccとcyを分析する。
分析は自資源や他資源、タスクの状態や優先度を中心に行う。

- Free状態からの遷移 Free状態からの遷移は$GetResource$が起点となる場合だけである。資源がFree状態からOccpuied状態に遷移するとき、自資源と関連するタスク(以下関連タスク)がRunning状態である必要がある。他の資源の状態には影響されないと考えられる。

- Occupied状態からの遷移 Occupied状態からの遷移は$ReleaseResource$が起点となる場合だけである。自資源がOccupied状態ということは自資源に関連しているタスクのいずれかがRunning状態ということである。よって、Ready状態のタスクの有無について分析する。まず、Ready状態のタスクがない場合を想定する。Ready状態のタスクがない場合は1つのタスクがRunning状態で、その他のタスクはSuspended状態である。よって、自資源はOccupied状態からFree状態へ遷移する。Ready状態のタスクがある場合を想定する。Running状態のタスクの優先度が、すべてのReady状態のタスクの優先度より高い場合、自資源はOccupied状態からFree状態へ遷移する。Running状態のタスクの優先度が、Ready状態のいずれかのタスクの優先度より低い場合、自資源はOccupied状態からFree状態へ遷移、Running状態のタス

表 4.1: タスククラスの遷移条件表

	cf_1	cf_2	cf_3	cf_4	cf_5	cf_6	cf_7
	YTask∋$^{\exists}$yt yt.s==RUN	xt.p > rt.p	YTask∋$^{\exists}$yt yt.s==READY	ReadyTask∋$^{\forall}$ret xt.p > ret.p	calledt.s==SUS	calledt.p > ret.p	FreeResource∋$^{\forall}$fr fr.s==FREE
cc_1	FALSE	−	−	−	−	−	TRUE
cc_2	TRUE	TRUE	−	−	−	−	TRUE
cc3	TRUE	FALSE	−	−	−	−	−
	TRUE	−	−	−	−	−	FALSE
cc_4	−	−	FALSE	−	−	−	TRUE
cc_5	−	−	TRUE	−	−	−	TRUE
cc_6	−	−	FALSE	−	−	−	TRUE
	−	−	TRUE	TRUE	−	−	TRUE
cc_7	−	−	TRUE	FALSE	−	−	TRUE
cc_8	−	−	FALSE	−	TRUE	−	TRUE
	−	−	TRUE	−	TRUE	TRUE	TRUE
cc_9	−	−	TRUE	−	TRUE	FALSE	TRUE

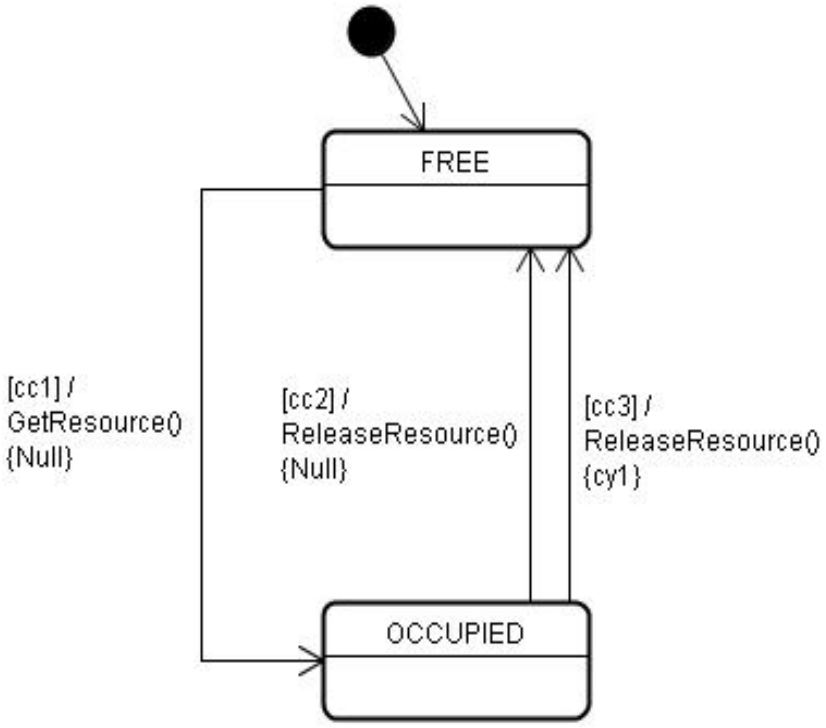
xt	自タスク
Ytask	他タスク集合
runt	RUN状態のタスク
ReadyTask	READY状態のタスク集合
calledt	呼び出されたタスク
FreeResource	FREE状態の資源集合

表 4.2: タスククラスの誘導遷移表

	YC	YS
cy_1	runt	READY
cy_2	MAXptask(ReadyTask)	RUN
cy_3	MAXptask(ReadyTask)	RUN
cy_4	calltask	RUN
cy_5	calltask	READY
	MAXptask(ReadyTask)	RUN

task	誘導されるタスク
runt	RUN状態のタスク
ReadyTask	READY状態のタスク集合
calledt	呼び出されたタスク

図 4.7: OSEK/VDX 仕様のモデル検査ステートマシン図 (資源クラス)



クは Ready 状態、最も優先度の高い Ready 状態のタスクは Running 状態に誘導遷移される。また Occupied 状態の他資源がある場合は、タスクの状態、優先度に関係なく、自資源は Occupied 状態から Free 状態へ遷移する。

分析した資源クラスの各 cc と cy を、遷移条件表と誘導遷移表としてまとめて表記した表を表 4.3 と表 4.4 に示す。

表 4.3: 資源クラスの遷移条件表

	cf_1	cf_2	cf_3	cf_4
	RelationTask∋$^{\exists}$rt rt.s==RUN	Task∋$^{\exists}$t t.s==READY	ReadyTask∋$^{\forall}$ret runt.p >= ret.p	YResource∋$^{\exists}$yr yr.s==OCC
cc_1	TRUE	-	-	-
cc_2	TRUE	-	-	TRUE
	TRUE	TRUE	TRUE	FALSE
	TRUE	FALSE	-	FALSE
cc_3	TRUE	TRUE	FALSE	FALSE

RelationTask	関連のあるタスク集合
Task	タスク集合
ReadyTask	Ready状態のタスク集合
YResource	他資源集合
runt	RUN状態のタスク

表 4.4: 資源クラスの誘導遷移表

	YC	YS
cy_1	runt	READY
	MAXptask(ReadyTask)	RUN

runt	RUN状態のタスク

モデル検査用ステートマシン図は各状態からの遷移に対する制約だけに着目して記述する。すべての制約に関してモデリングするという事は、検査したいスケジューリング機

能を設計するの事になる。それでは、検査するためのスケジューリング機能自体の正しさを保障する必要が出てきて本末転倒である。よって、本研究のモデリングの粒度はクラスレベルの遷移について分析できる所までとした。

4.4　実験:実体化によるモデル検査用オブジェクト生成

4.4.1　実験:多重度同意義パターン除去

タスク、資源の RTOS に対する多重度を実体化する。実体化する多重度をタスクを 1..2、資源を 0..2 とする。タスク、資源間の関連の有無により組み合わせパターン数が決まる。各多重度に実体化時の組み合わせパターン数を以下に示す。また、それぞれのパターンを図 4.8 から図 4.13 に示す。

- Task1 つ、Resource なしの場合 $2^0 = 1$
- Task1 つ、Resource1 つの場合 $2^1 = 2$
- Task1 つ、Resource2 つの場合 $2^2 = 4$
- Task2 つ、Resource なしの場合 $2^0 = 1$
- Task2 つ、Resource1 つの場合 $2^2 = 4$
- Task2 つ、Resource2 つの場合 $2^4 = 16$

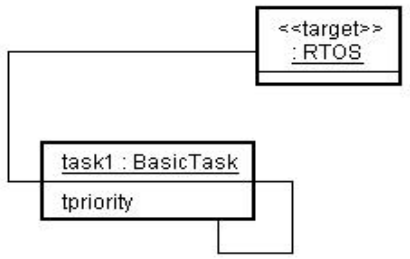


図 4.8: 多重度実体化後の組み合わせパターン (Task1 つ、Resource なし)

図 4.12 に着目して多重度同意義パターンを考える。左上の図は資源オブジェクトはタスクオブジェクトと関連がない。資源オブジェクトは単独ではサービスコールを発行しない。よって、左上の図ではタスクと資源を組み合わせた動作をしない。つまり、実質的に検査する動作は図 4.11 と同じであると考えられる。よって、タスクオブジェクトと関連がない資源オブジェクトがあるパターンを多重度同意義パターン①とする。

右上の図は $Task_1$ と $Resource_1$ に関連があり、$Task_2$ と $Resource_1$ に関連がない。左下の図は $Task_2$ と $Resource_1$ に関連があり、$Task_1$ と $Resource_1$ に関連がない。これらは両方とも 2 つのタスクオブジェクトの内、一方のタスクオブジェクトと資源オブジェクトに関連があり、もう一方のタスクオブジェクトは資源オブジェクトと関連がない。よって、これらのオブジェクトの組み合わせの振舞いは同じであると考えられるため、多重度同意義パターン②とする。

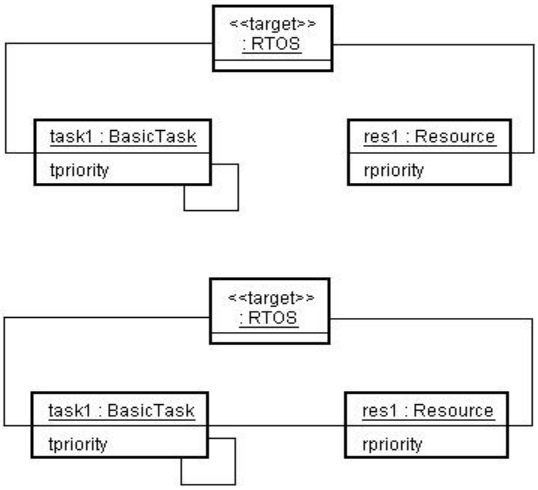


図 4.9: 多重度実体化後の組み合わせパターン (Task1 つ、Resource1 つ)

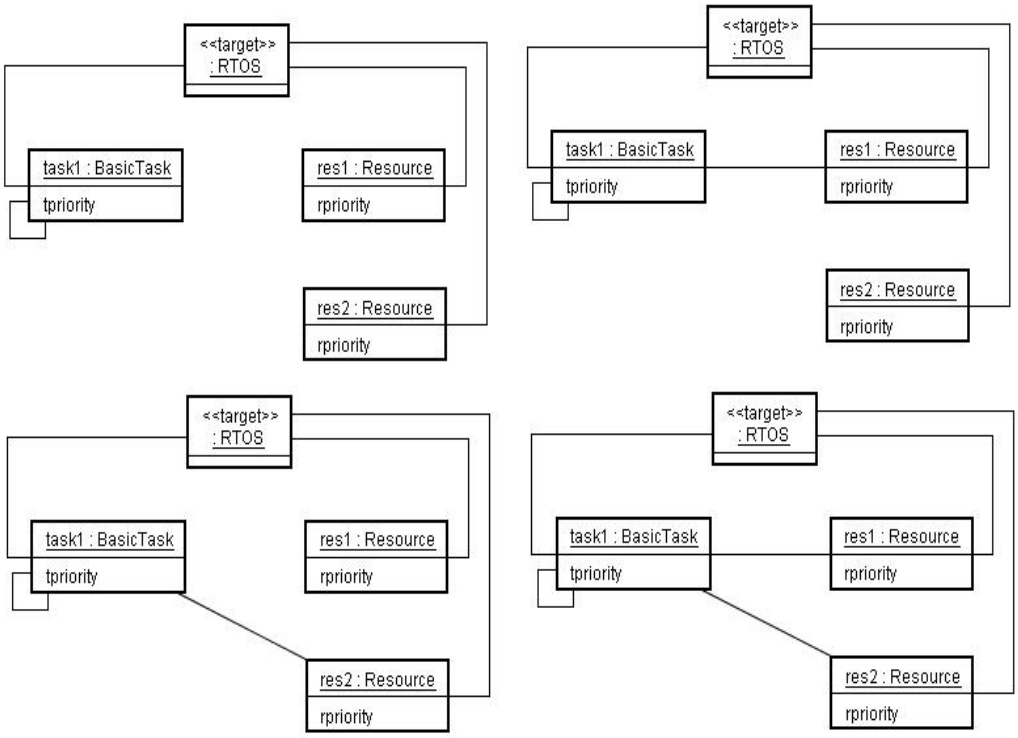


図 4.10: 多重度実体化後の組み合わせパターン (Task1 つ、Resource2 つ)

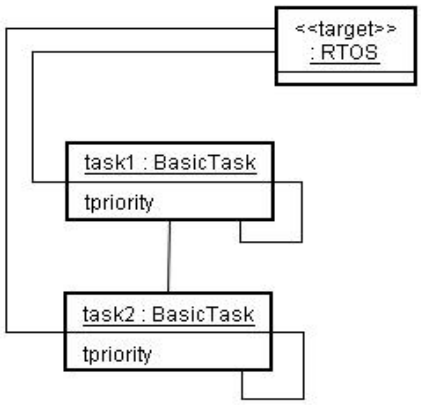


図 4.11: 多重度実体化後の組み合わせパターン (Task2 つ、Resource なし)

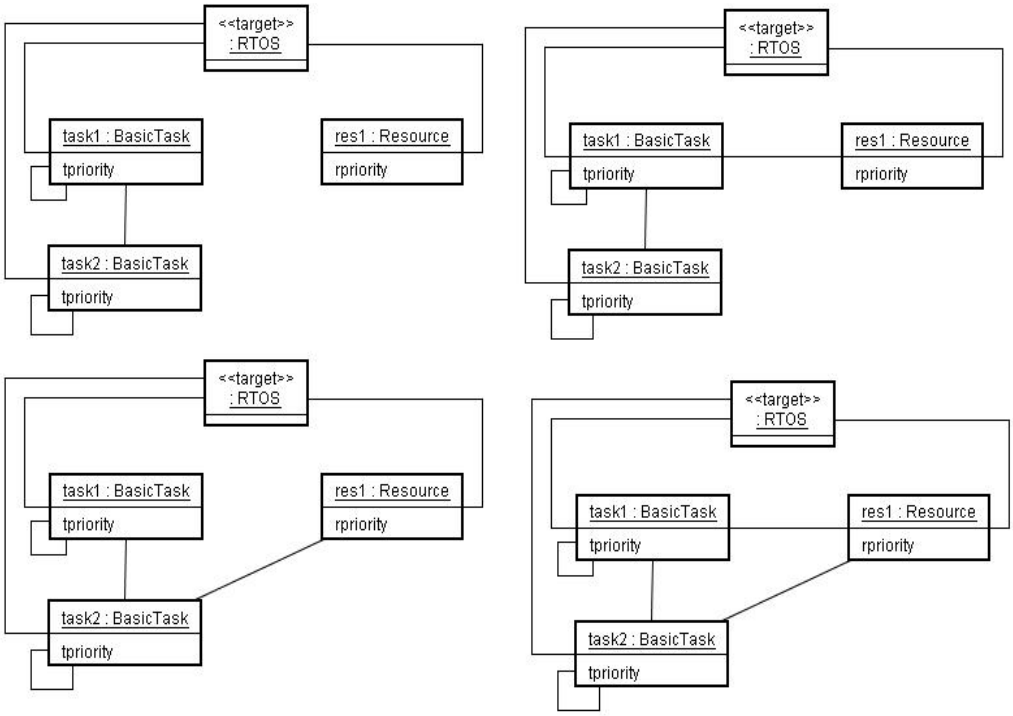


図 4.12: 多重度実体化後の組み合わせパターン (Task2 つ、Resource1 つ)

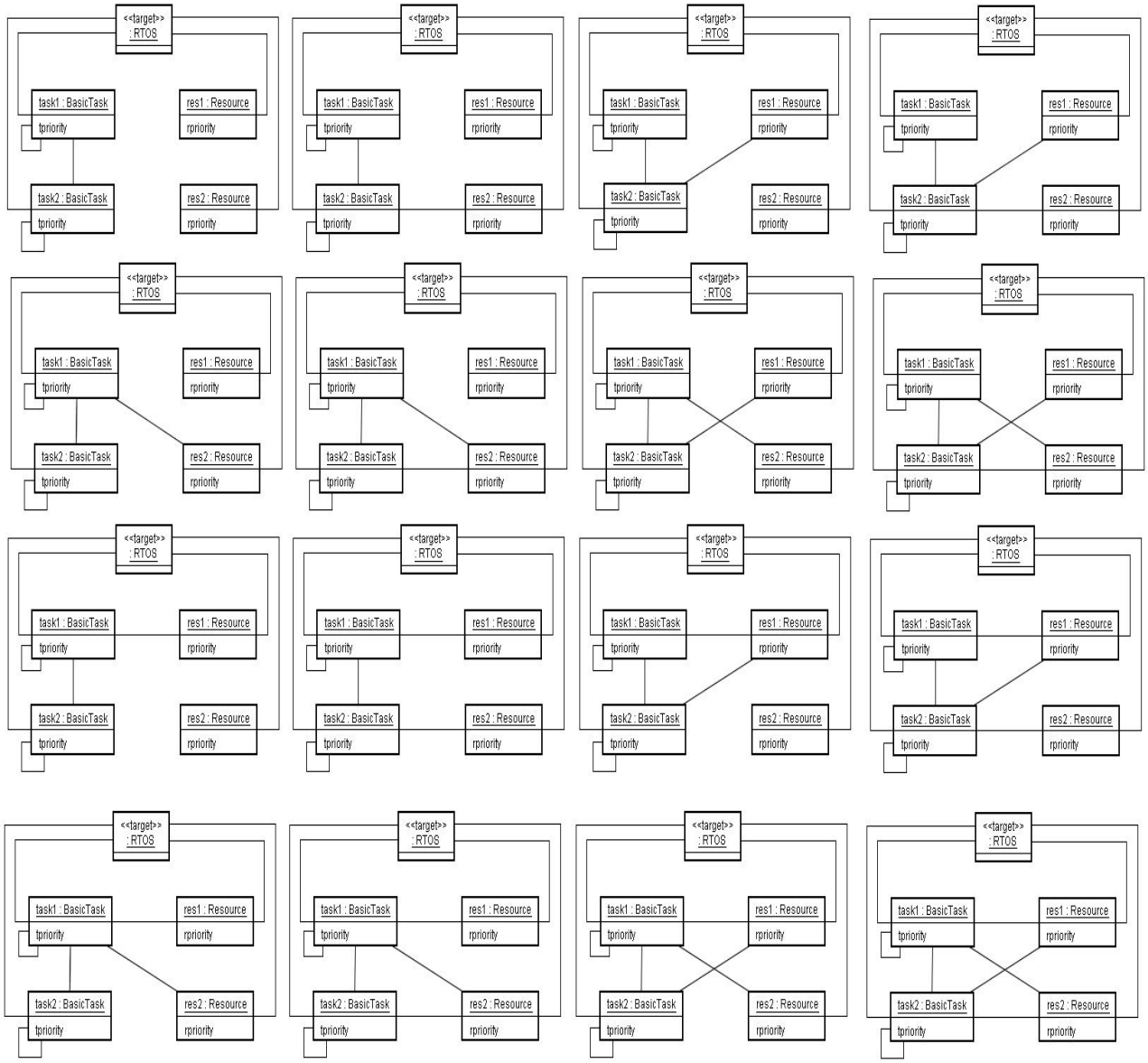


図 4.13: 多重度実体化後の組み合わせパターン (Task2 つ、Resource2 つ)

それぞれの多重度同意義パターンを定義する。多重度同意義パターン定義では関連するオブジェクトの数に着目する。ここで$Count(ob)$をいう関数を便宜的に用意する。関数$Count(ob)$の仕様を以下に示す。

•$Count(ob)$
オブジェクトobにリンクする関連の数を返す

- 多重度同意義パターン①

  資源オブジェクト$Resource_1, Resource_2, ..., Resource_N$に対して、

  $(Count(Resource_1) > 0)$ && $(Count(Resource_2) > 0)$ &&, ..., && $(Count(Resource_N) > 0)$

  を満たさない組み合わせを除去する。

- 多重度同意義パターン②

  タスクオブジェクト$Task_1, Task_2, ..., Task_N$と資源オブジェクト$Resource_1, Resource_2, ..., Resourc$に対して、

  集合$\{Count(Task_1), Count(Task_2), ..., Count(Task_N)\}$と

  集合$\{Count(Resource_1), Count(Resource_2), ..., Count(Resource_M)\}$

  が一致する組み合わせを同意義とし、一方の組み合わせを除去する。

図4.13の例について多重度同意義パターン①と多重度同意義パターン② を用いて組み合わせパターンを除去したものを図4.14と図4.15に示す。$\times$印はパターン除去、$=$印は同意義である事を表している。

定義した多重度同意義パターンの除去により組み合わせパターン数は以下のように減少した。各組み合わせの多重度同意義パターン除去後のパターンを図4.16から図4.21に示す。

- Task1つ、Resourceなしの場合 $2^0 = 1 \Rightarrow 1$
- Task1つ、Resource1つの場合 $2^1 = 2 \Rightarrow 1$
- Task1つ、Resource2つの場合 $2^2 = 4 \Rightarrow 1$
- Task2つ、Resourceなしの場合 $2^0 = 1 \Rightarrow 1$
- Task2つ、Resource1つの場合 $2^2 = 4 \Rightarrow 2$
- Task2つ、Resource2つの場合 $2^4 = 16 \Rightarrow 4$

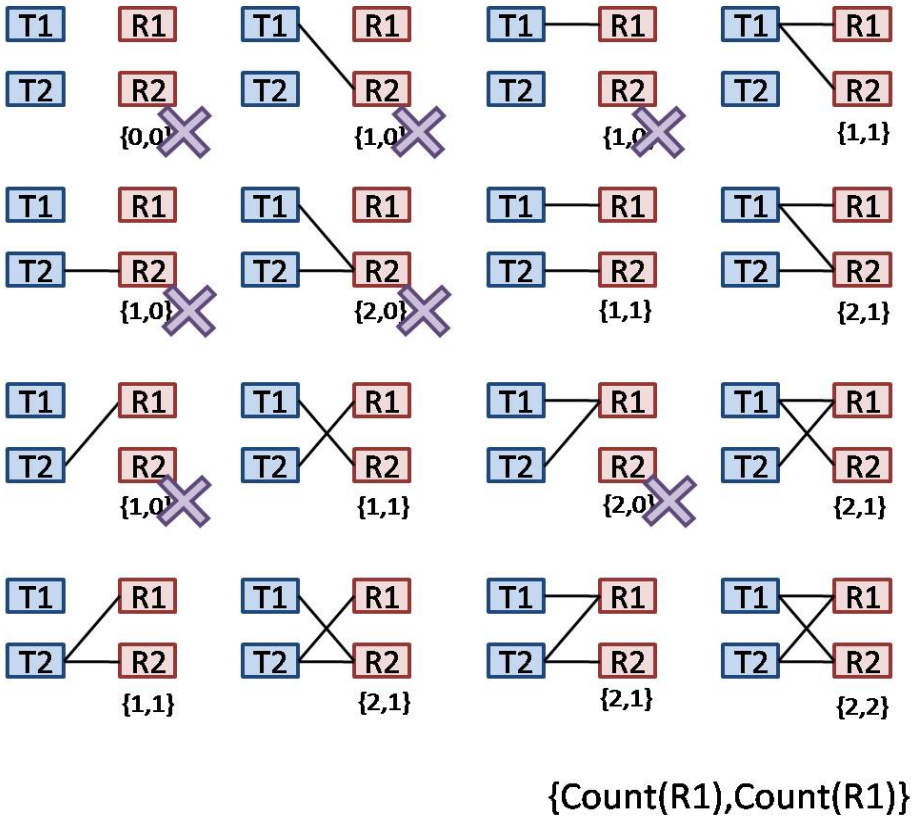


図 4.14: 多重度同意義パターン①による組み合わせパターン除去の例 (Task2 つ、Resource2 つ)

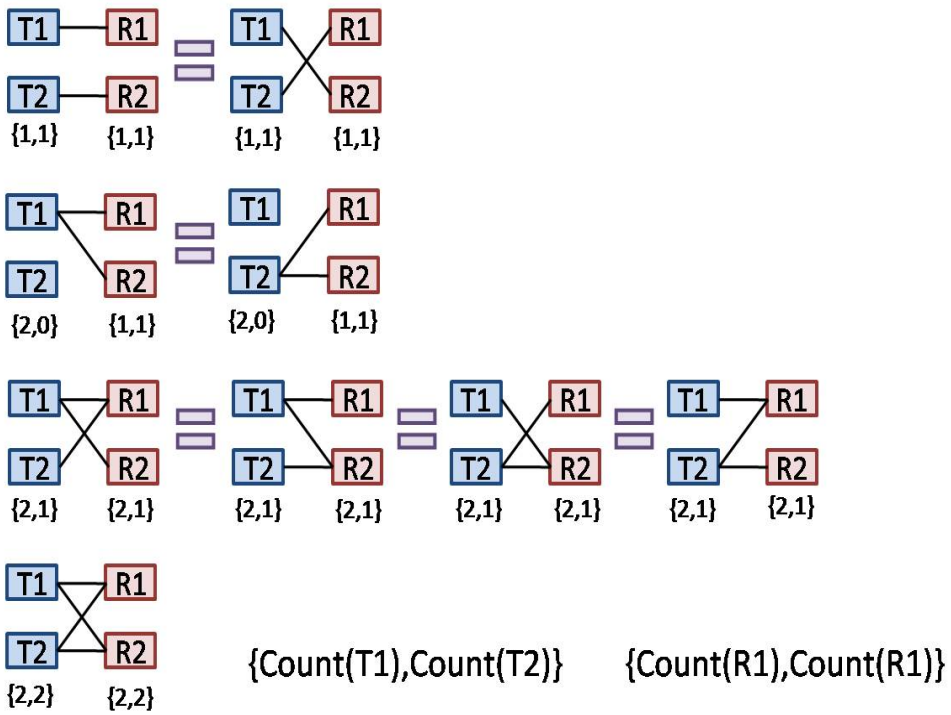


図 4.15: 多重度同意義パターン②による組み合わせパターン除去の例 (Task2 つ、Resource2 つ)

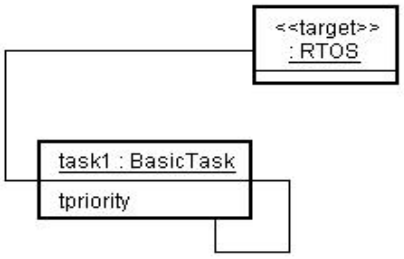


図 4.16: 多重度同意義パターン除去後の組み合わせパターン (Task1 つ、Resource なし)

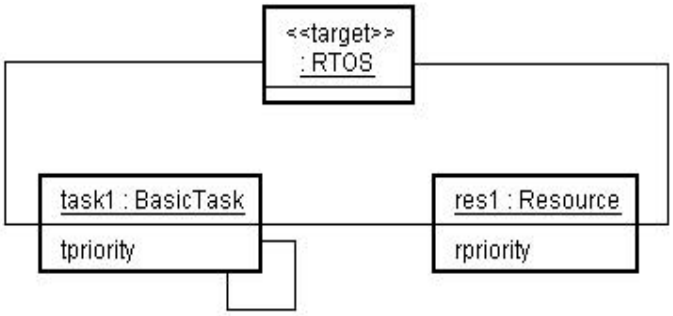


図 4.17: 多重度同意義パターン除去後の組み合わせパターン (Task1 つ、Resource1 つ)

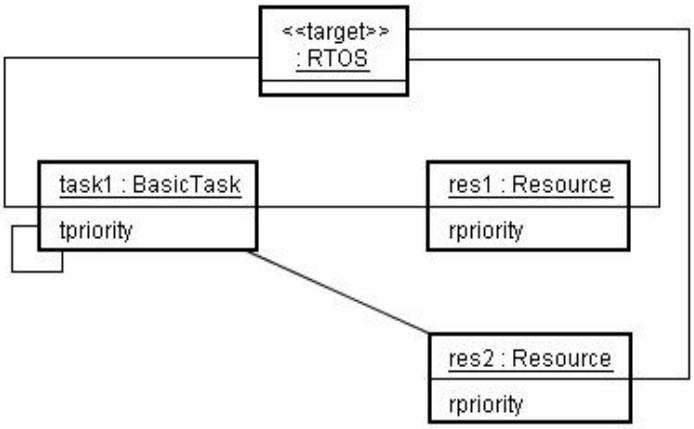


図 4.18: 多重度同意義パターン除去後の組み合わせパターン (Task1 つ、Resource2 つ)

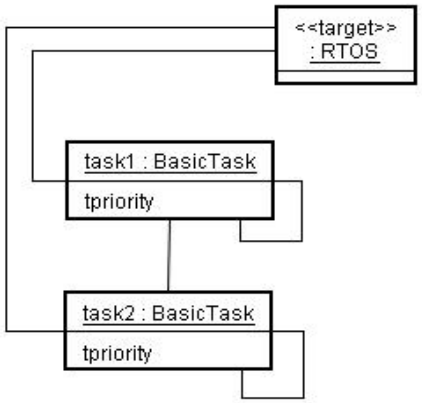


図 4.19: 多重度同意義パターン除去後の組み合わせパターン (Task2 つ、Resource なし)

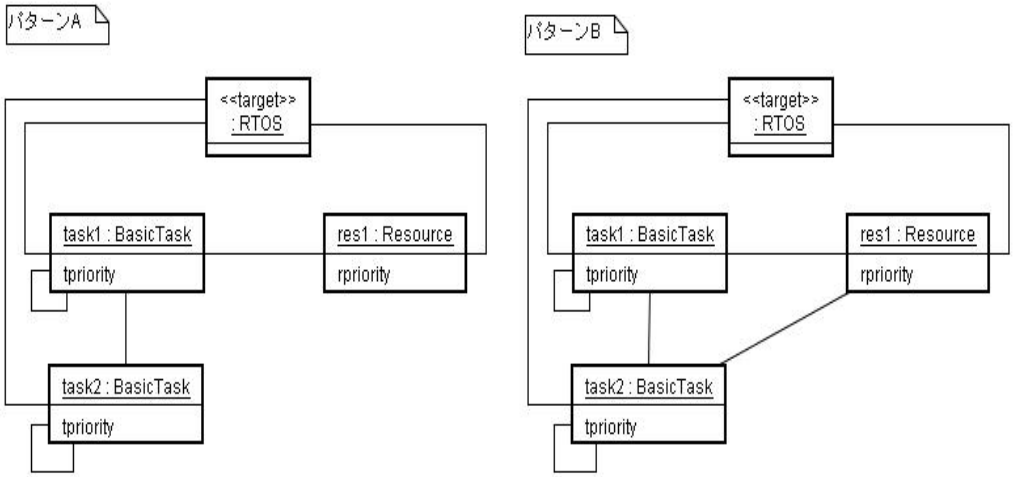


図 4.20: 多重度同意義パターン除去後の組み合わせパターン (Task2 つ、Resource1 つ)

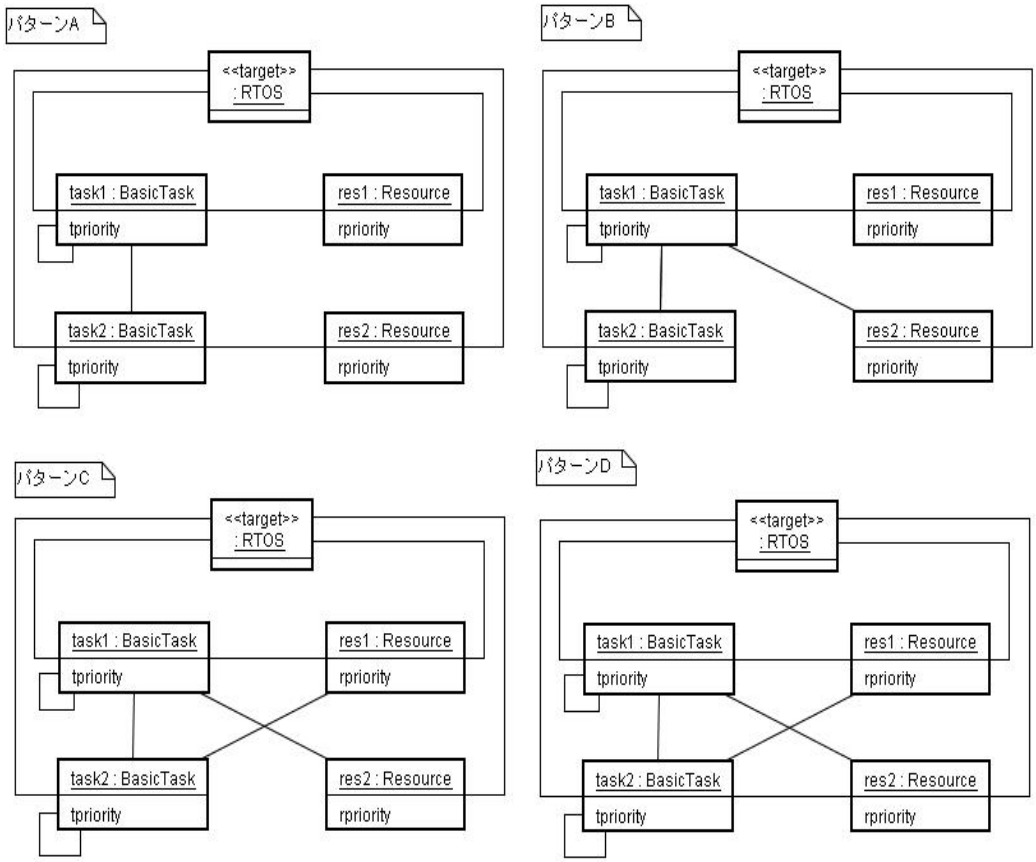


図 4.21: 多重度同意義パターン除去後の組み合わせパターン (Task2 つ、Resource2 つ)

4.4.2 実験:属性同意義パターン除去

タスクの属性 tpriority を { Thigh,Tlow } の 2 値で実体化する。同様に資源の属性 rpriority を { Rhigh,Rlow } の 2 値で実体化する。OSEK/VDX 仕様のプライオリティーシーケンスの機能より、$Tlow <= Thigh << Rlow <= Rhigh$ となるように実体化する。

この時、組み合わせパターン数は以下のようになる。各組み合わせパターンを図 4.22 から図 4.43 に示す。なお、多重度の実体化において複数の組み合わせパターンが得られた組み合わせはパターン別に図化している。

- Task1 つ、Resource なしの場合 $1 \times 2^1 \times 2^0 = 2$
- Task1 つ、Resource1 つの場合 $1 \times 2^1 \times 2^1 = 4$
- Task1 つ、Resource2 つの場合 $1 \times 2^1 \times 2^2 = 8$
- Task2 つ、Resource なしの場合 $1 \times 2^2 \times 2^0 = 4$
- Task2 つ、Resource1 つの場合 $2 \times 2^2 \times 2^1 = 16$
- Task2 つ、Resource2 つの場合 $4 \times 2^2 \times 2^2 = 64$

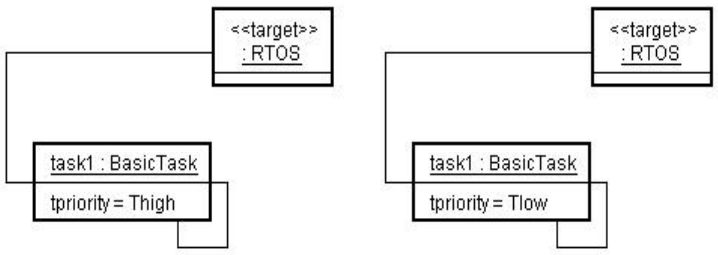


図 4.22: 属性実体化後の組み合わせパターン (Task1 つ、Resource なし)

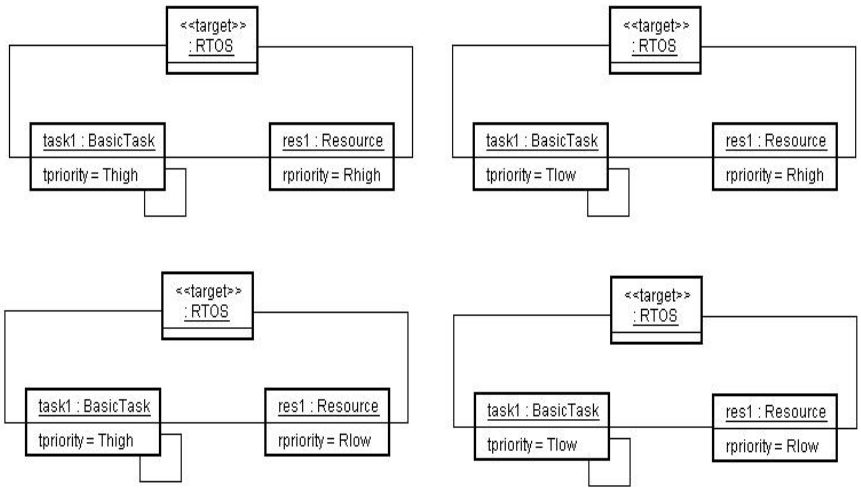


図 4.23: 属性実体化後の組み合わせパターン (Task1 つ、Resource1 つ)

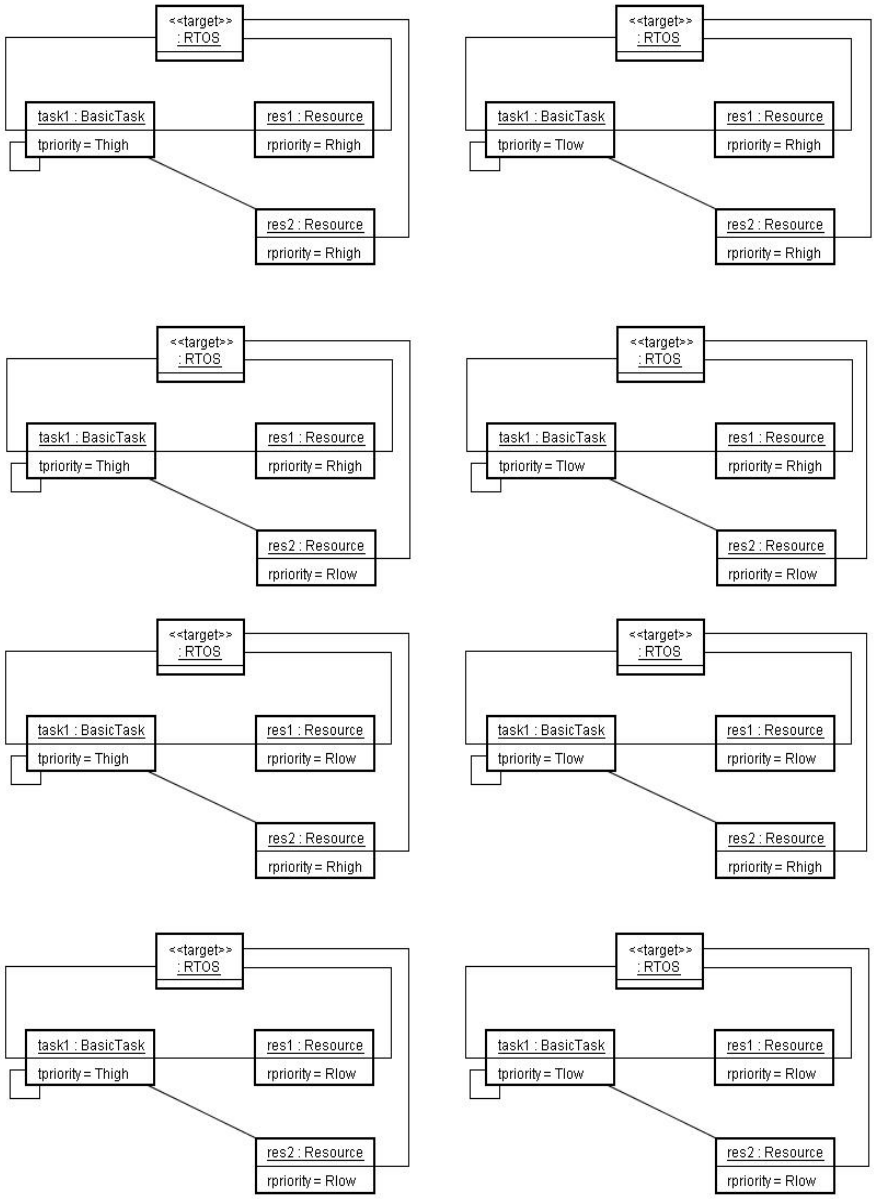


図 4.24: 属性実体化後の組み合わせパターン (Task1 つ、Resource2 つ)

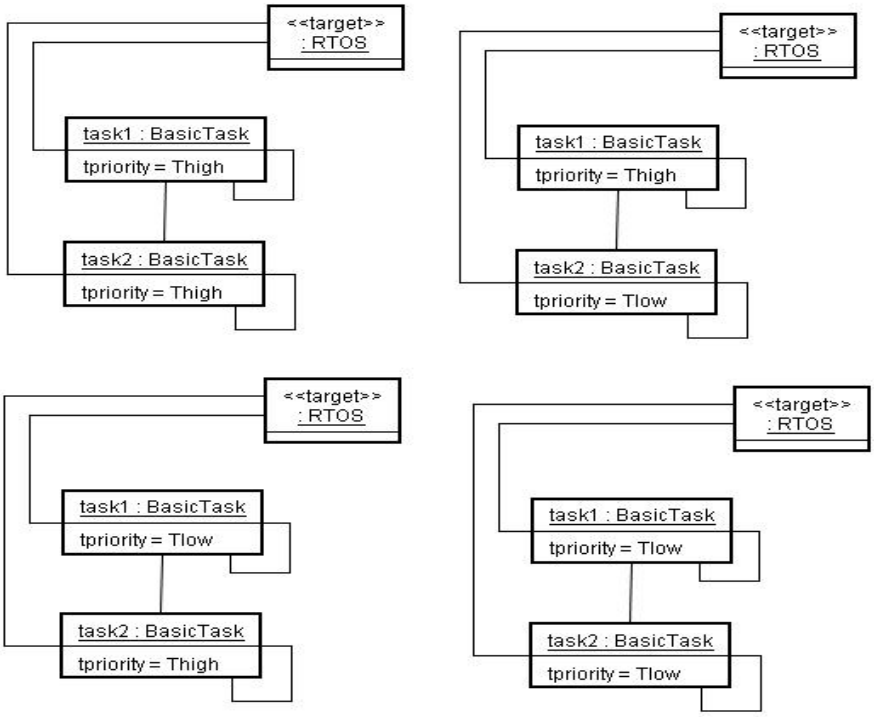


図 4.25: 属性実体化後の組み合わせパターン (Task2 つ、Resource なし)

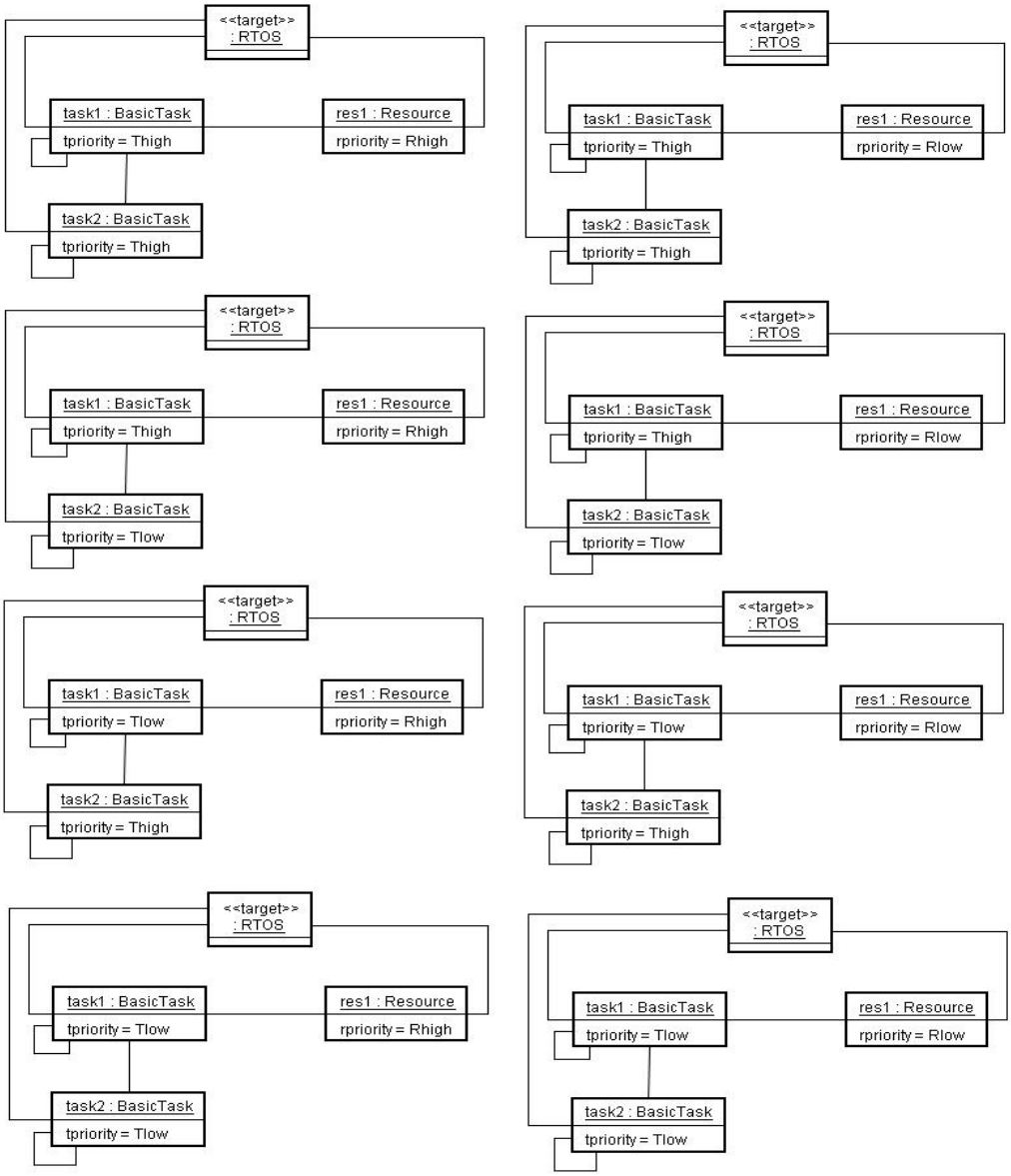


図 4.26: 属性実体化後の組み合わせパターン (Task2 つ、Resource1 つ、パターン A)

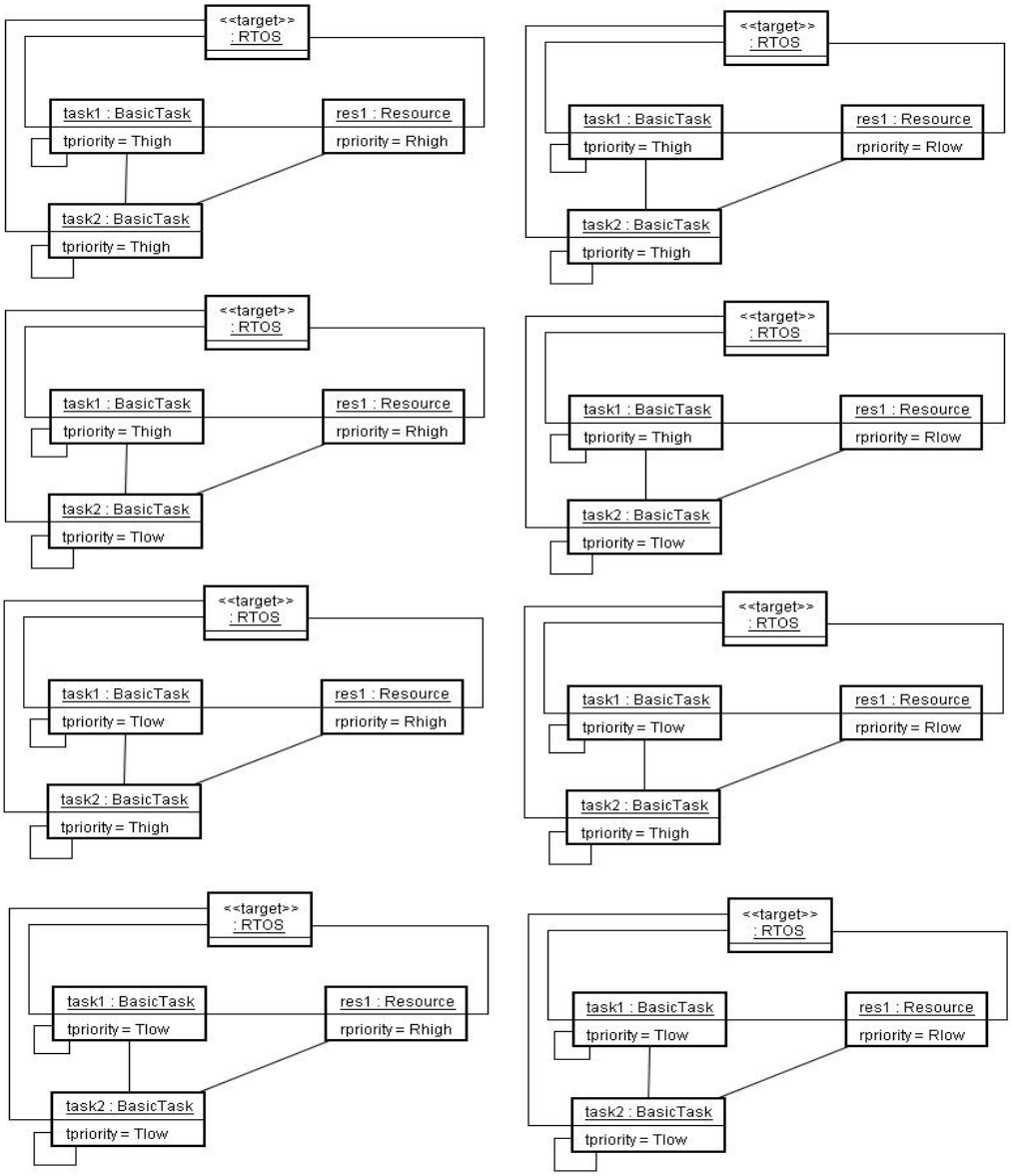


図 4.27: 属性実体化後の組み合わせパターン (Task2 つ、Resource1 つ、パターン B)

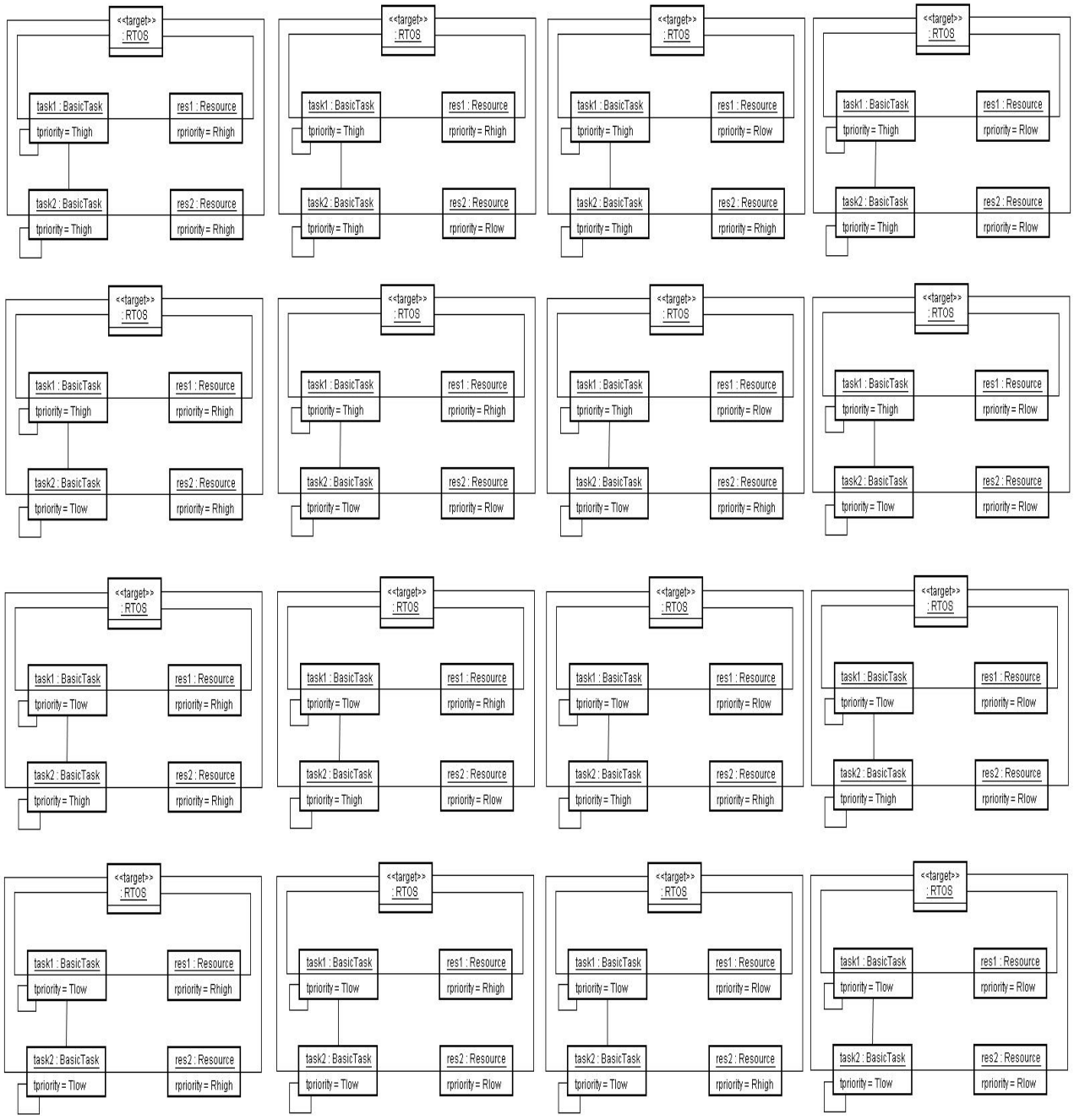


図 4.28: 属性実体化後の組み合わせパターン (Task2 つ、Resource2 つ、パターン A)

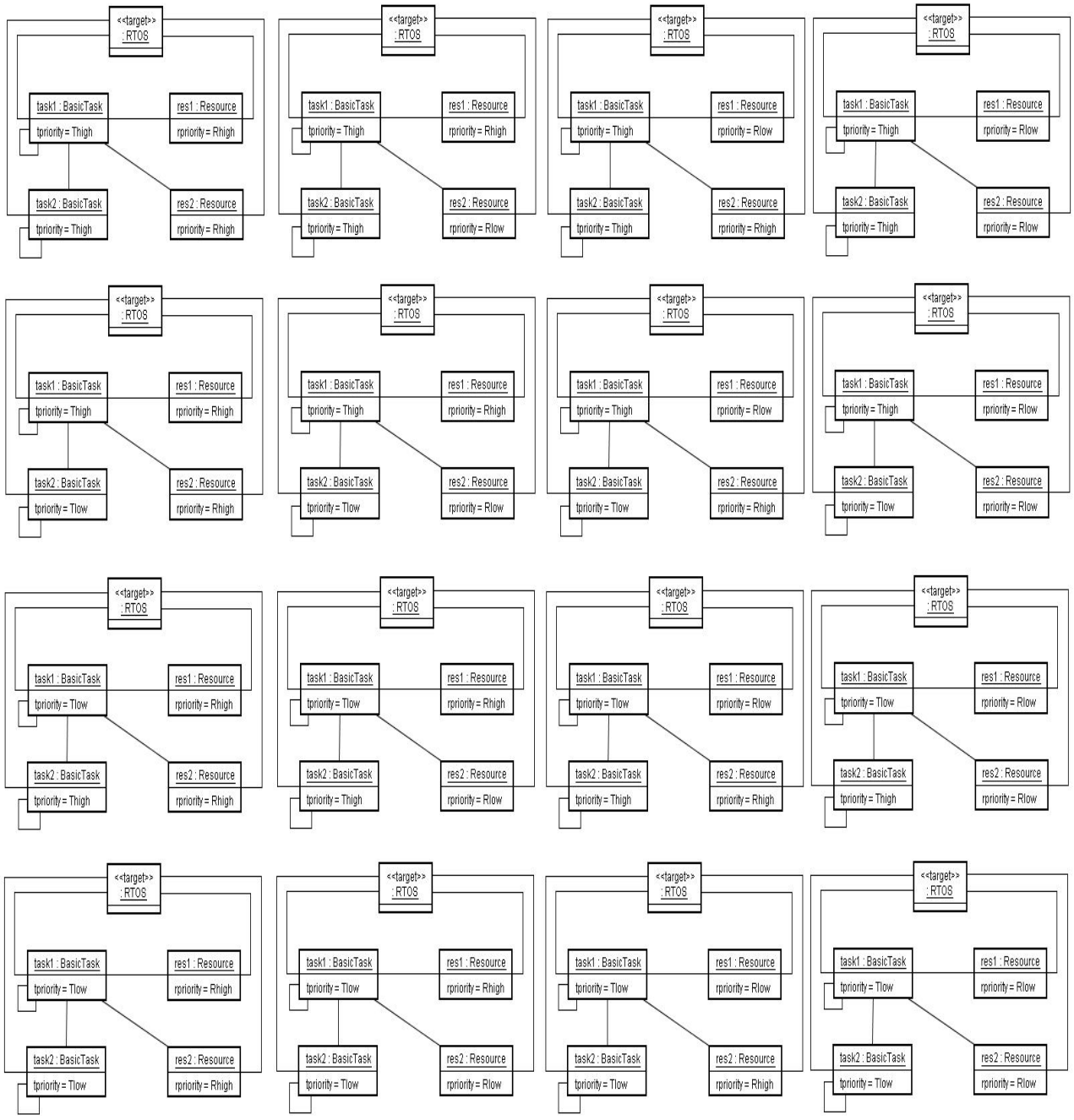


図 4.29: 属性実体化後の組み合わせパターン (Task2 つ、Resource2 つ、パターン B)

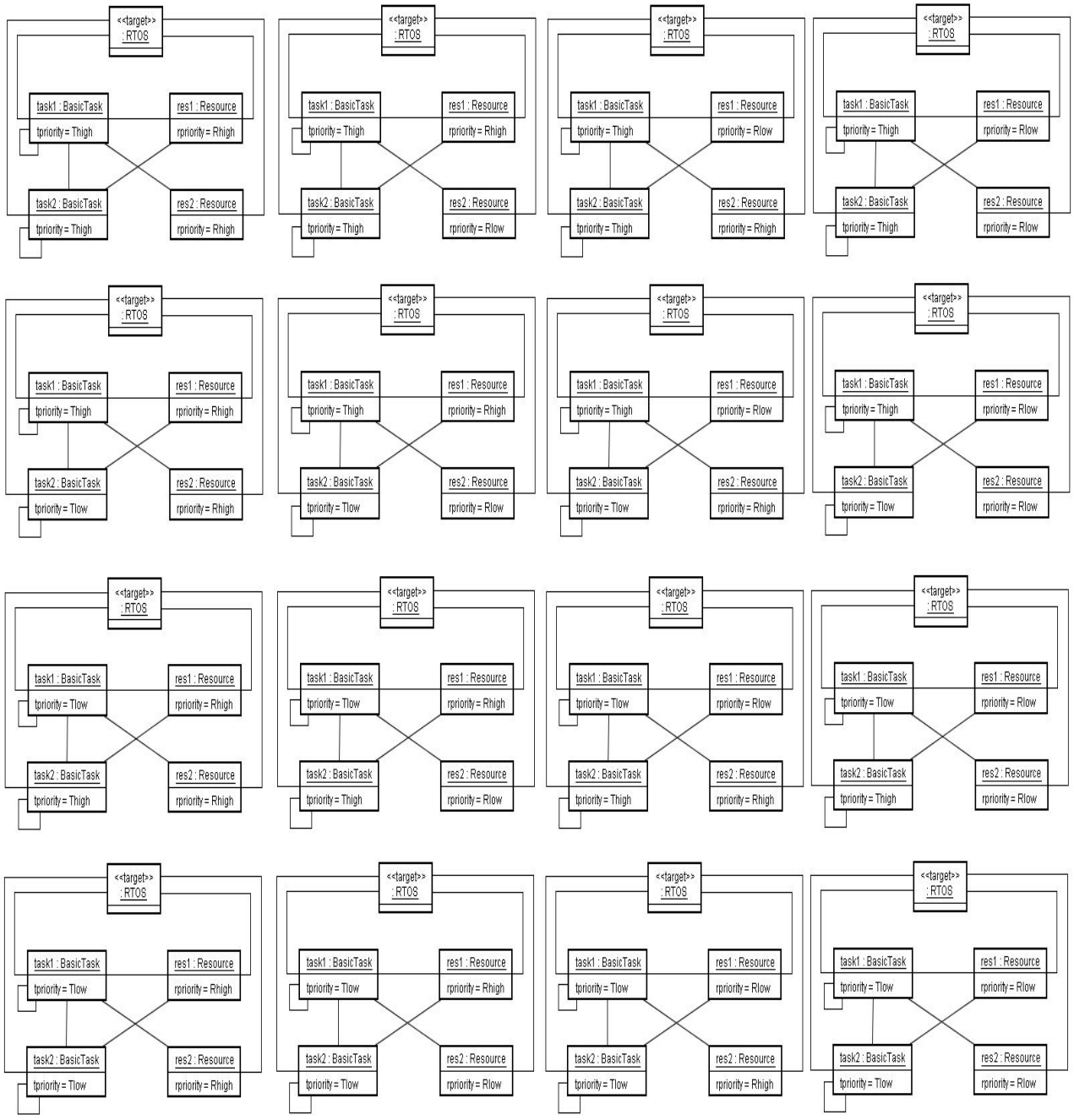


図 4.30: 属性実体化後の組み合わせパターン (Task2 つ、Resource2 つ、パターン C)

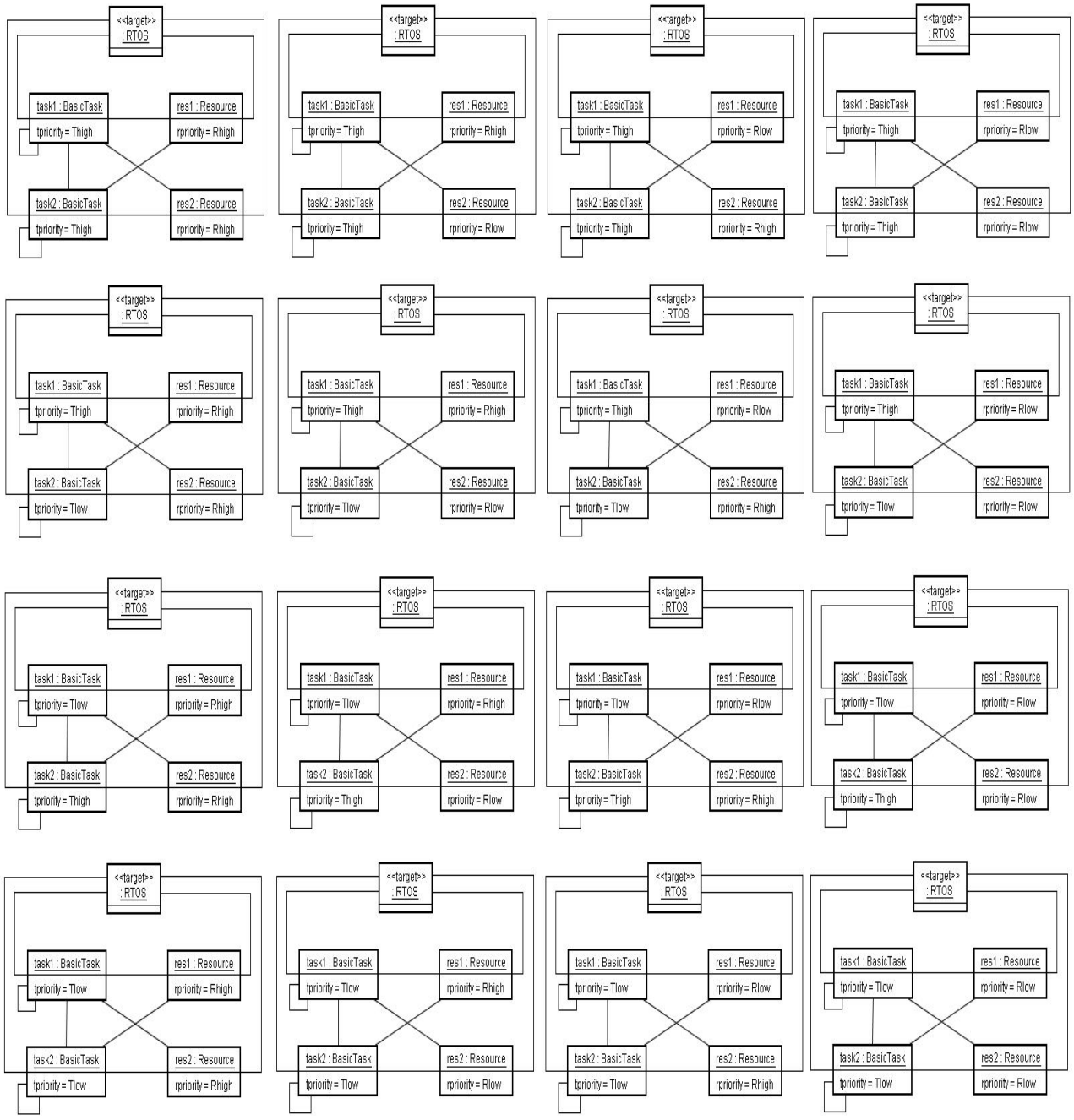


図 4.31: 属性実体化後の組み合わせパターン (Task2 つ、Resource2 つ、パターン D)

図4.28に着目して属性同意義パターンを考える。$Task_1.tpriority = Thigh$、$Task_2.tpriority = Thigh$の場合と$Task_1.tpriority = Tlow$、$Task_2.tpriority = Tlow$の場合があるが、$Tlow < Thigh << Rlow < Rhigh$ の関係性より、$Task_1.tpriority = Thigh$、$Task_2.tpriority = Thigh$の場合を検査すれば、タスクオブジェクトの優先度が等しい場合の検査として十分であると考えらえる。同様に$Resource_1.rpriority = Rhigh$、$Resource_2.rpriority = Rhigh$ の場合と $Resource_1.rpriority = Rlow$、$Resource_2.rpriority = Rlow$ の場合があるが、$Resource_1.rpriority = Rhigh$、$Resource_2.rpriority = Rhigh$ を検査すれば、資源オブジェクトの優先度が等しい場合の検査としては十分であると考えられる。よって、タスクオブジェクト間の優先度の関係性が等しく、かつ資源オブジェクト間の優先度の関係性が等しい場合を属性同意義パターン①とする。

次に、各タスクオブジェクトと資源オブジェクト間の関連に着目する。$Task_1.priority = Thigh$、$Task_2.priority = Tlow$、$Resource_1.rpriority = Rhigh$、$Resource_2.rpriority = Rlow$のパターンの場合と$Task_1.priority = Tlow$、$Task_2.priority = Thigh$、$Resource_1.rpriority = Rlow$、$Resource_2.rpriority = Rhigh$ のパターンの場合を例に考える。例では優先度の高いタスクオブジェクトは優先度の高い資源オブジェクトと関連し、優先度の低いタスクオブジェクトと優先度の低い資源オブジェクトと関連する。これらの組み合わせパターンの動作は等しいと考える事ができる。よって、関連の存在するタスクオブジェクトと資源オブジェクトの各優先度の関係性が等しい場合を属性同意義パターン②とする。

それぞれの属性同意義パターンを定義する。属性同意義パターン定義では各オブジェクト間のの優先度の関係性に着目する。ここで便宜的に3つの関数$TTCompare(tpriority, tpriority)$、$RRCompare(rpriority, rpriority)$、$TRCompare(tpriority, rpriority)$ を用意する。

関数 $TTCompare(tpriority, tpriority)$ の仕様を以下に示す。

- $TTCompare(tpriority_1, tpriority_2)$
  $tpriority_1 == tpriority_2$ならば 0 を返す
  $tpriority_1 < tpriority_2$ならば 1 を返す
  $tpriority_1 > tpriority_2$ならば 2 を返す

関数 $RRCompare(rpriority, rpriority)$ の仕様を以下に示す。

- $RRCompare(rpriority_1, rpriority_2)$
  $rpriority_1 == rpriority_2$ならば 0 を返す
  $rpriority_1 < rpriority_2$ならば 1 を返す
  $rpriority_1 > rpriority_2$ならば 2 を返す

関数 $TRCompare(tpriority, rpriority)$ の仕様を以下に示す。

•$TRCompare(tpriority_1, rpriority_2)$
$tpriority_1 == Thigh$、$rpriority_2 == Rhigh$ ならば 0 を返す
$tpriority_1 == Thigh$、$rpriority_2 == Rlow$ ならば 1 を返す
$tpriority_1 == Tlow$、$tpriority_2 == Rhigh$ ならば 2 を返す
$tpriority_1 == Tlow$、$tpriority_2 == Rlow$ ならば 3 を返す

- 属性同意義パターン①

  タスクオブジェクトの優先度 $Task_1.tpriority$、$Task_2.tpriority$ と資源オブジェクトの優先度 $Resource_1.rpriority$、$Resource_2.rpriority$ に対して、

  $TTCompare(Task_1.tpriority, Tast_2.tpriority)$ と

  $RRCompare(Resource_1.rpriority, Resource_2.rpriority)$ が一致するならば同意義とし、一方の組み合わせを除去する。

- 属性同意義パターン②

  タスクオブジェクトの優先度は $Task_1.tpriority$、$Task_2.tpriority$、資源オブジェクトの優先度は $Resource_1.rpriority$、$Resource_2.rpriority$ である。

  組み合わせパターンの各関連に対して $TRCompare(Task_N.tpriority, Resource_M.rpriority)$ の集合が一致するならば同意義とし、一方の組み合わせを除去する。

図 4.28 の例について属性同意義パターン①と属性同意義パターン② を用いて組み合わせパターンを除去したものを図 4.32 と図 4.33 に示す。=印は同意義である事を表している。

属性同意義パターンに組み合わせパターン数は以下のように減少する。各組み合わせの属性組み合わせパターン除去後のパターンを図 4.34 から図 4.43 に示す。属性同意義パターンにより、削減されなかったオブジェクトがモデル検査用オブジェクトである。

- Task1 つ、Resource なしの場合 $2^1 \times 2^0 = 2 \Rightarrow 1$
- Task1 つ、Resource1 つの場合 $2^1 \times 2^1 = 4 \Rightarrow 1$
- Task1 つ、Resource2 つの場合 $2^1 \times 2^2 = 8 \Rightarrow 2$
- Task2 つ、Resource なしの場合 $2^2 \times 2^0 = 4 \Rightarrow 2$
- Task2 つ、Resource1 つの場合 (パターン A) $2^2 \times 2^1 = 8 \Rightarrow 3$
- Task2 つ、Resource1 つの場合 (パターン B) $2^2 \times 2^1 = 8 \Rightarrow 2$
- Task2 つ、Resource2 つの場合 (パターン A) $2^2 \times 2^2 = 16 \Rightarrow 5$

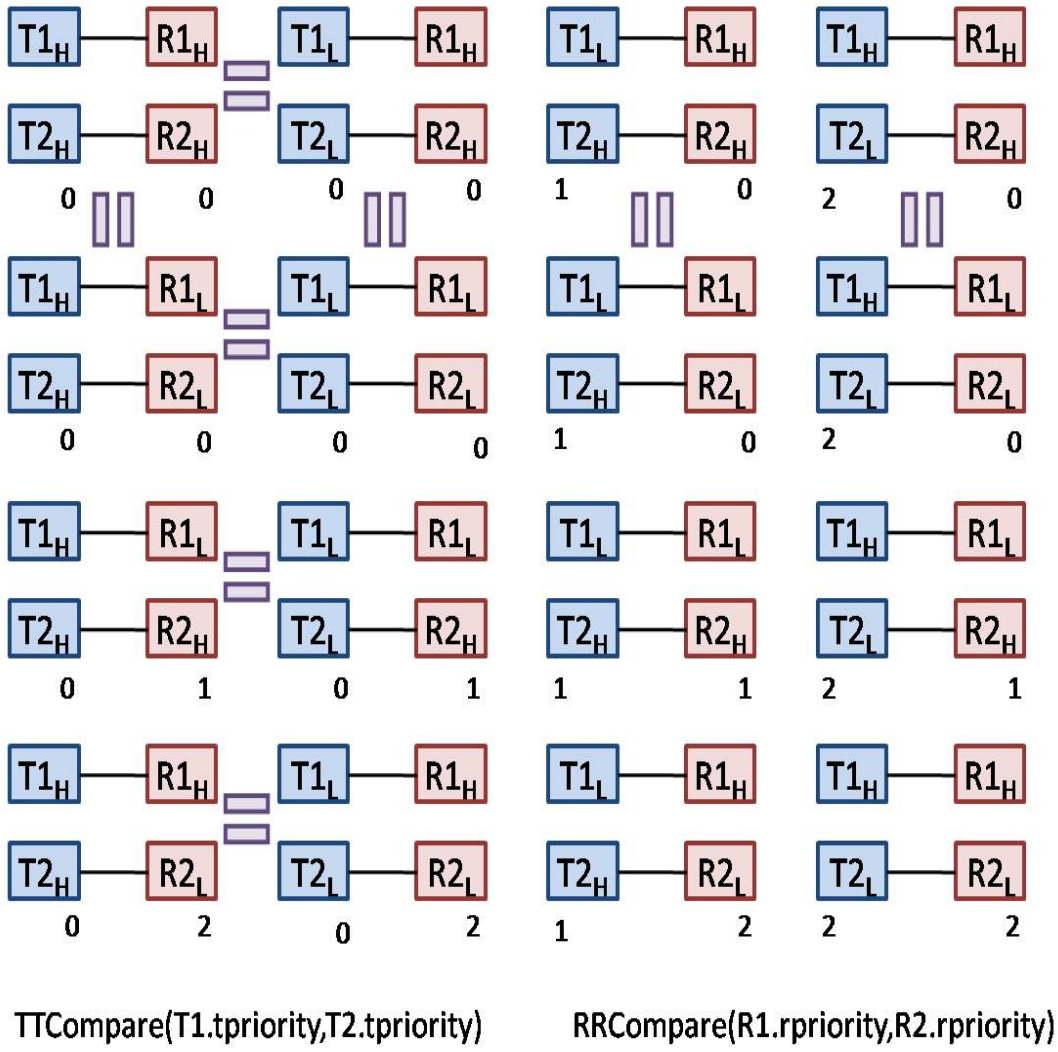


図 4.32: 属性同意義パターン①による組み合わせパターン除去の例 (Task2 つ、Resource2 つ、パターン A)

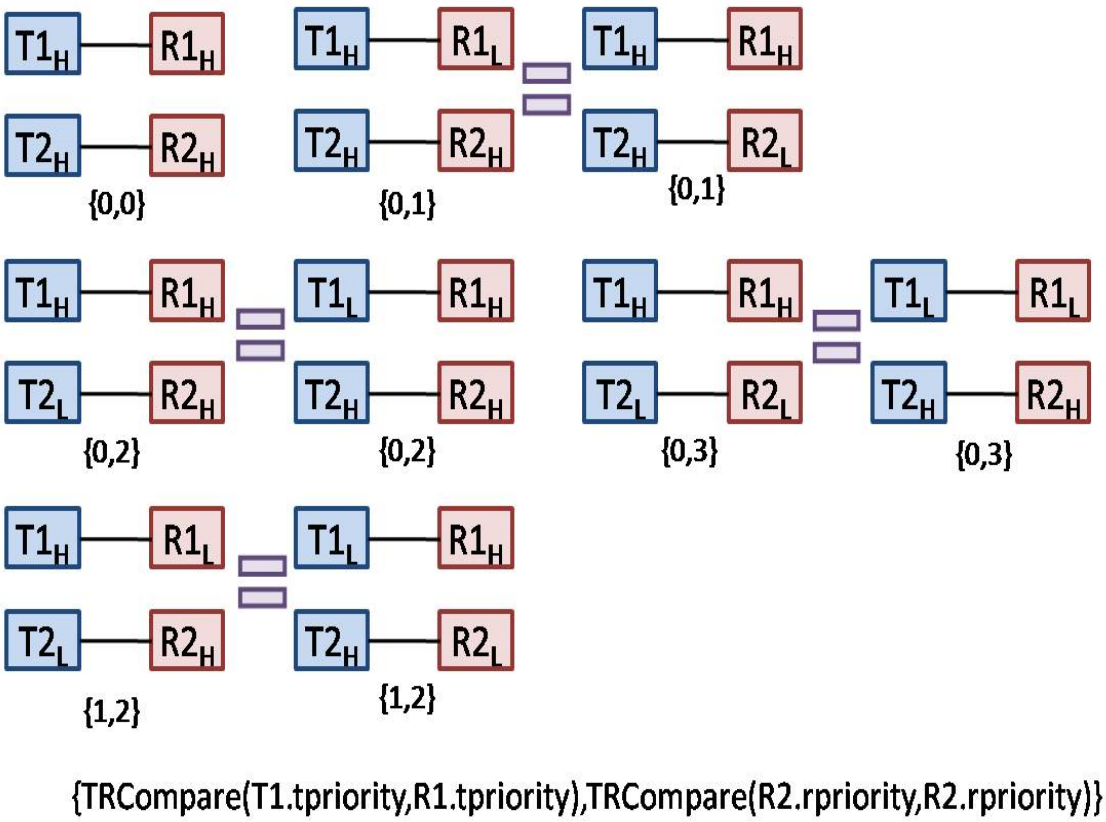


図 4.33: 属性同意義パターン②による組み合わせパターン除去の例 (Task2 つ、Resource2 つ、パターン A)

- Task2 つ、Resource2 つの場合 (パターン B) $2^2 \times 2^2 = 16 \Rightarrow 6$
- Task2 つ、Resource2 つの場合 (パターン C) $2^2 \times 2^2 = 16 \Rightarrow 9$
- Task2 つ、Resource2 つの場合 (パターン D) $2^2 \times 2^2 = 16 \Rightarrow 4$

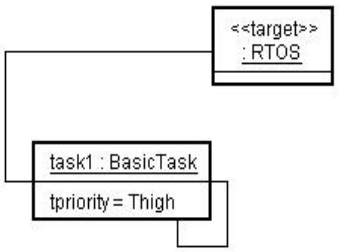


図 4.34: 属性同意義パターン除去後の組み合わせパターン (Task1 つ、Resource なし)

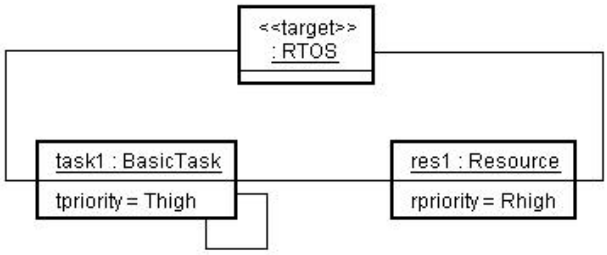


図 4.35: 属性同意義パターン除去後の組み合わせパターン (Task1 つ、Resource1 つ)

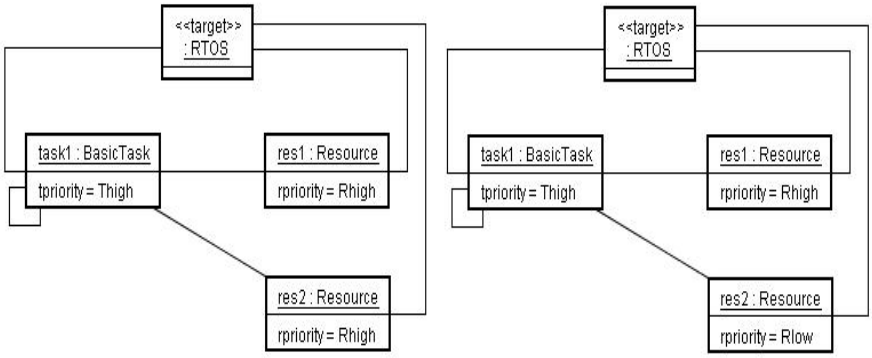


図 4.36: 属性同意義パターン除去後の組み合わせパターン (Task1 つ、Resource2 つ)

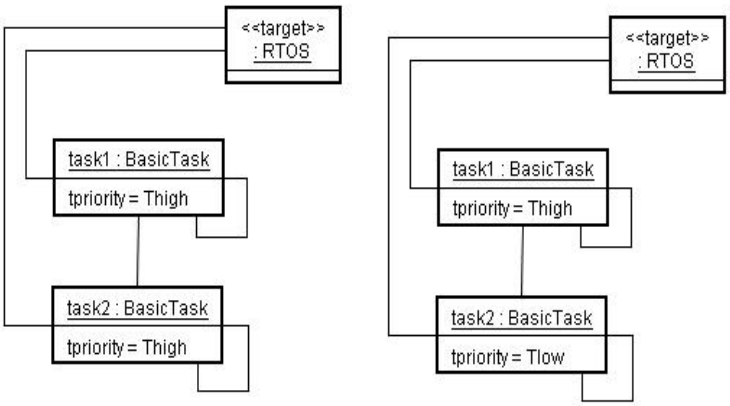


図 4.37: 属性同意義パターン除去後の組み合わせパターン (Task2 つ、Resource なし)

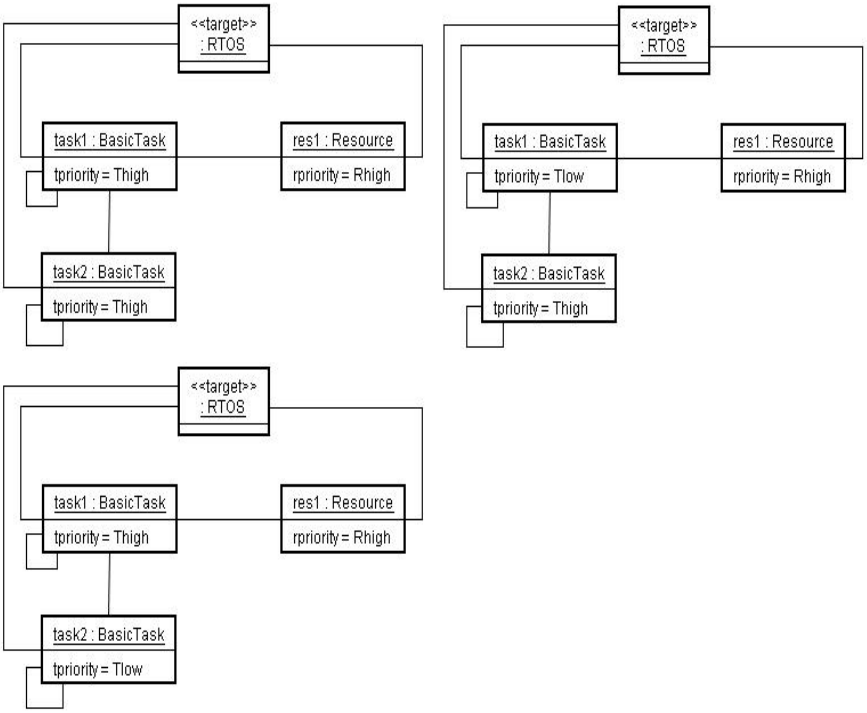


図 4.38: 属性同意義パターン除去後の組み合わせパターン (Task2 つ、Resource1 つ、パターン A)

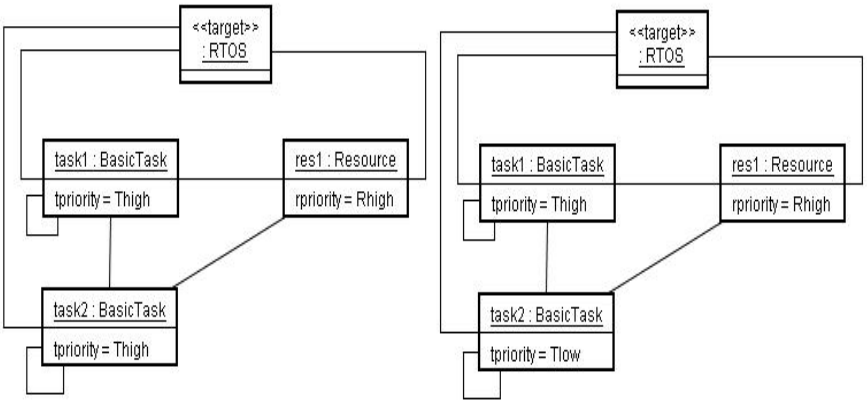


図 4.39: 属性同意義パターン除去後の組み合わせパターン (Task2 つ、Resource1 つ、パターン B)

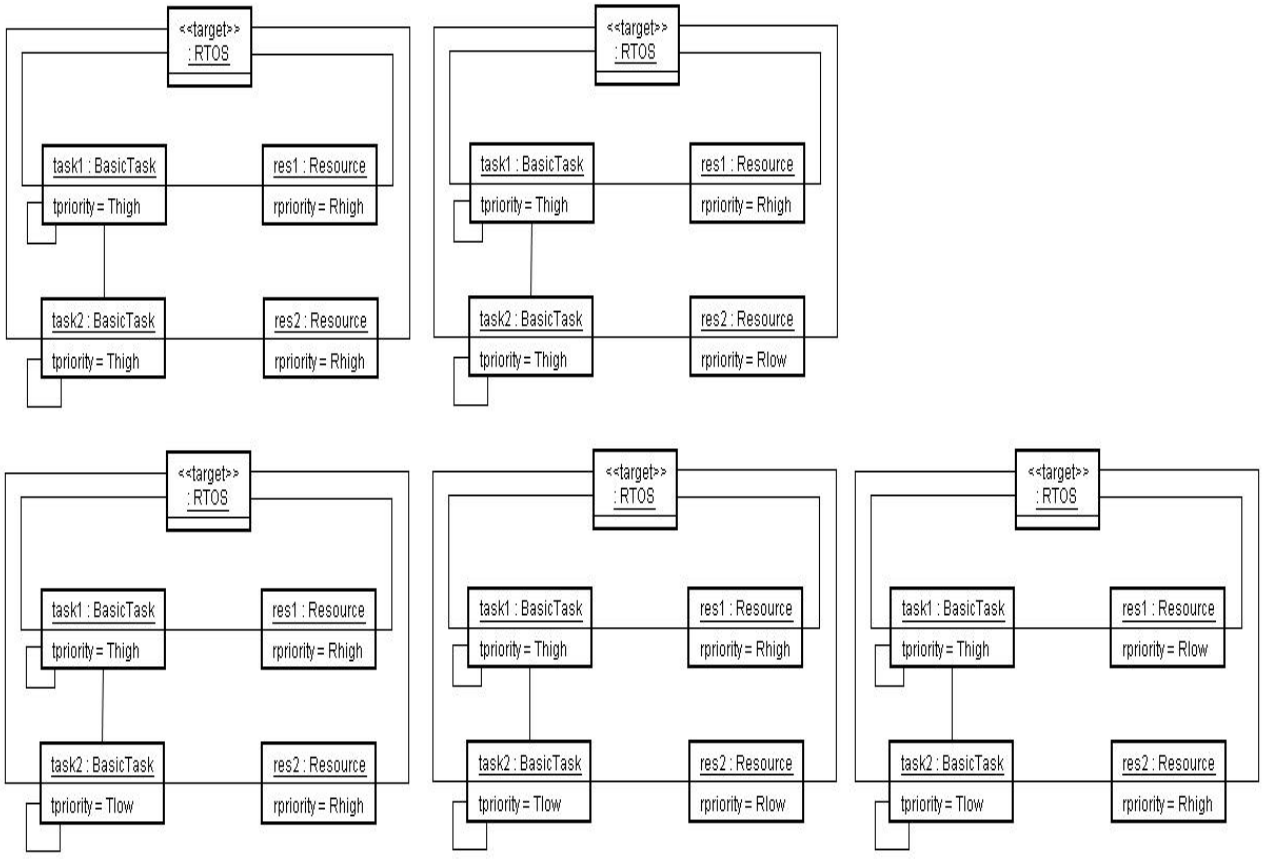


図 4.40: 属性同意義パターン除去後の組み合わせパターン (Task2 つ、Resource2 つ、パターン A)

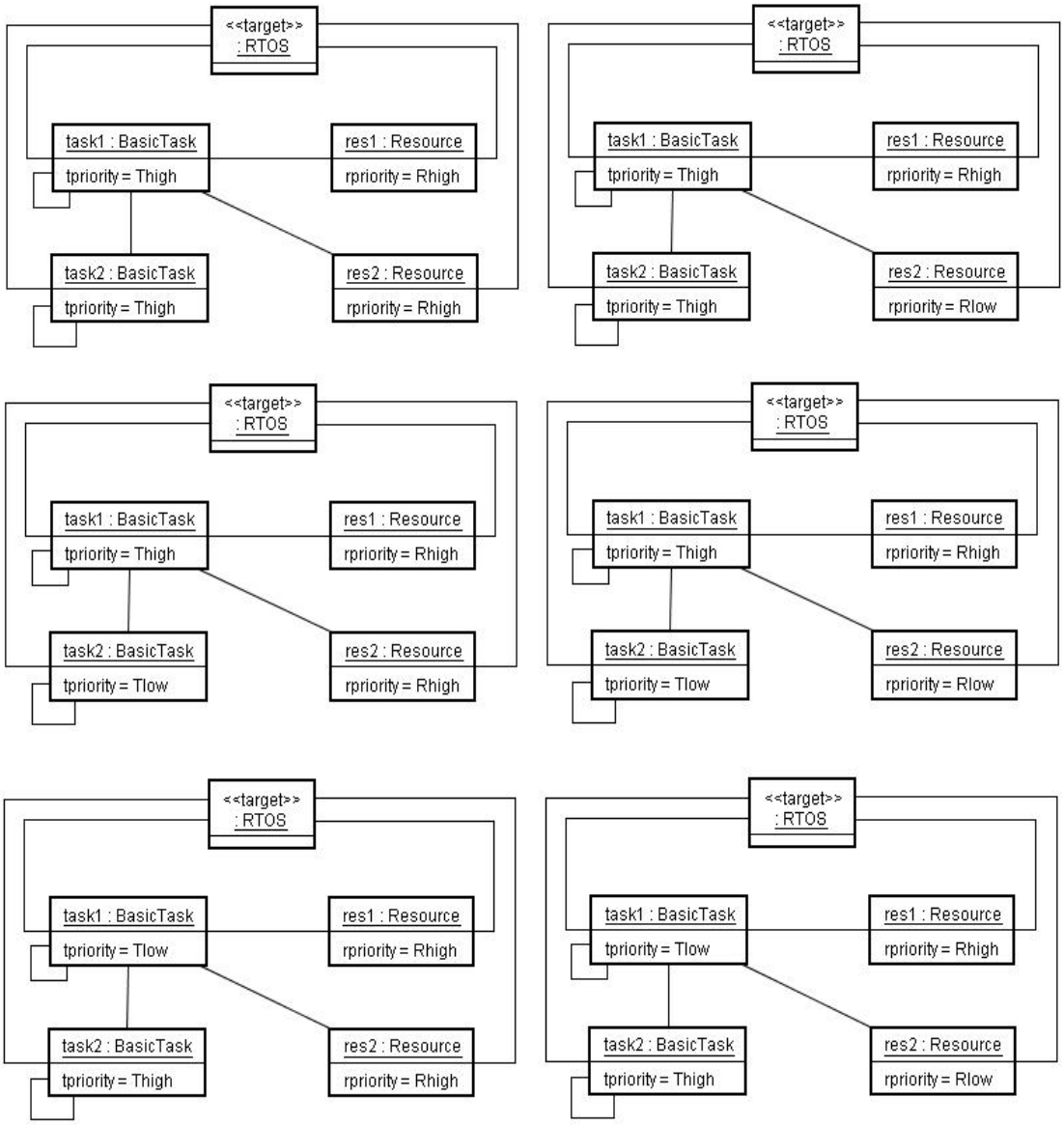


図 4.41: 属性同意義パターン除去後の組み合わせパターン (Task2 つ、Resource2 つ、パターン B)

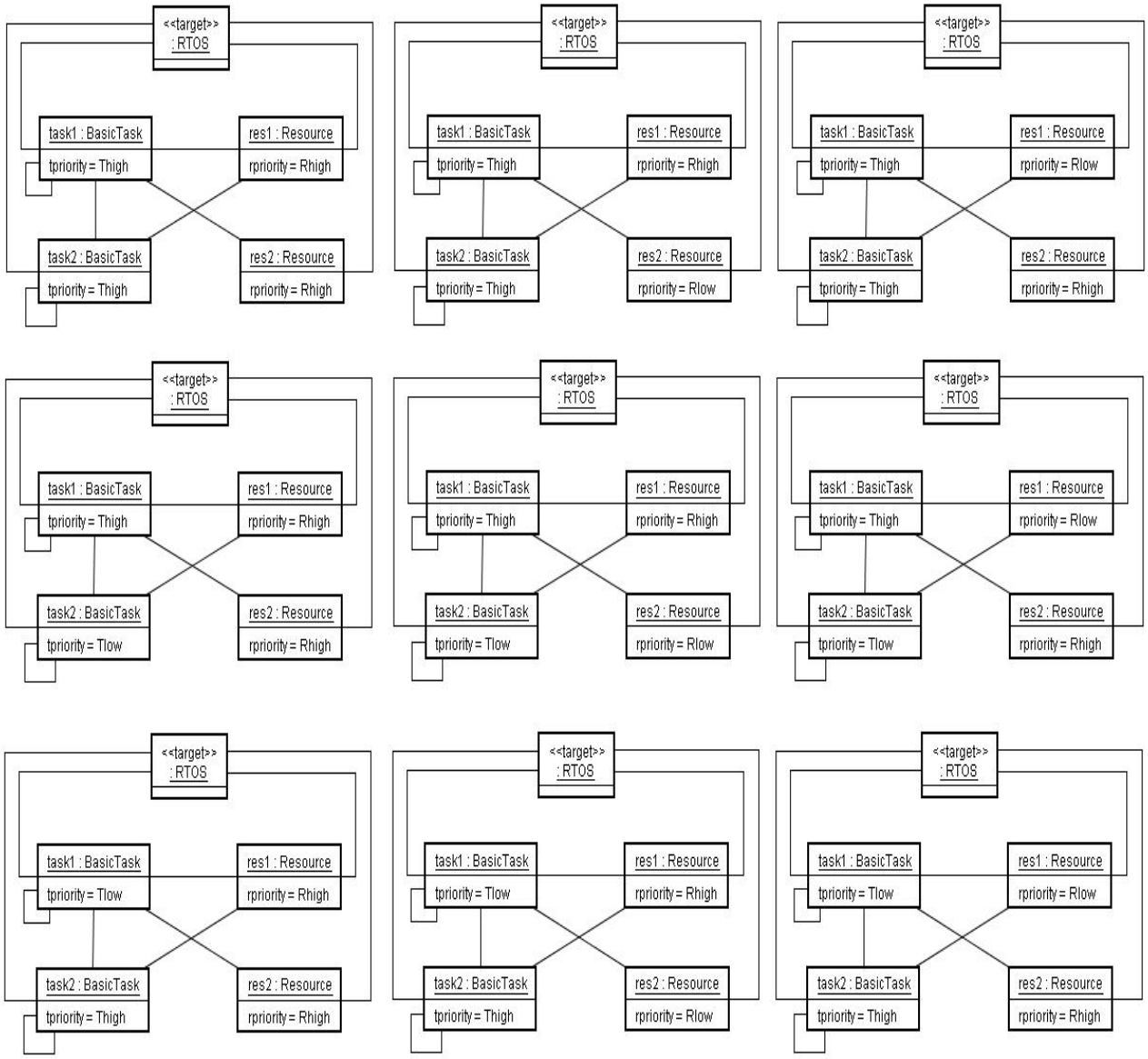


図 4.42: 属性同意義パターン除去後の組み合わせパターン (Task2 つ、Resource2 つ、パターン C)

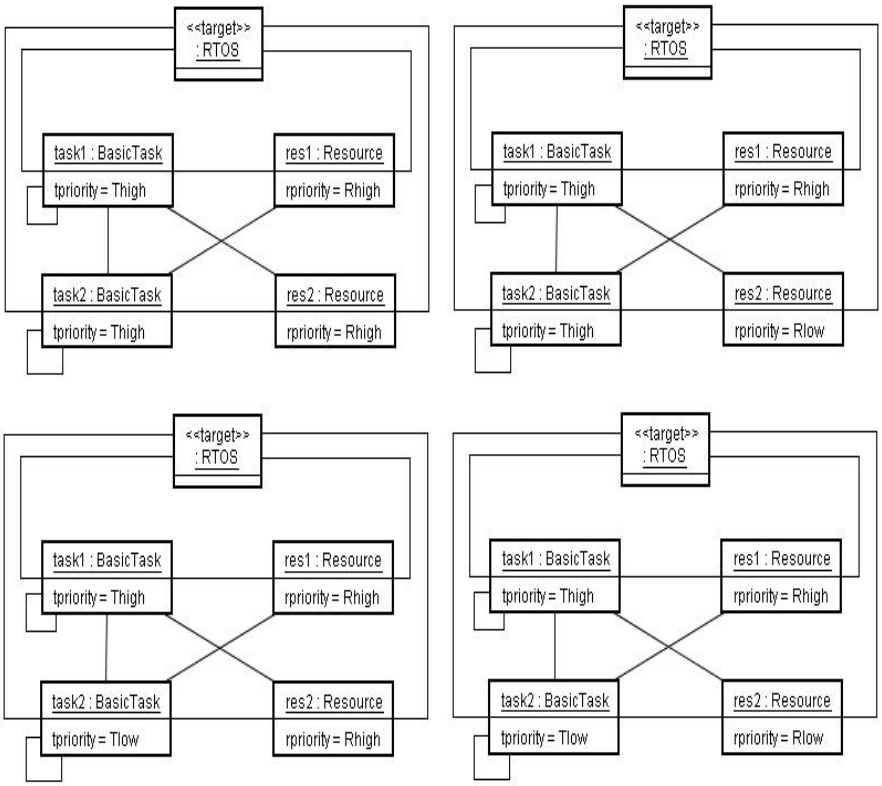


図 4.43: 属性同意義パターン除去後の組み合わせパターン (Task2 つ、Resource2 つ、パターン D)

4.5　実験:検査モデル生成

検査モデルを生成するためにクラスレベルの遷移条件表と誘導遷移表をインスタンスレベルに実体化する。遷移条件表と誘導遷移表が実体化されれば遷移も実体化する。インスタンスレベルの遷移集合と環境統合アルゴリズムにより検査モデルを生成する。

4.5.1　実験:遷移条件表の実体化

図 4.38 の Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$ のモデル検査用オブジェクトの例とクラスレベルの遷移条件表 4.1 と 4.3 を元に遷移条件表を実体化する。クラスオブジェクトについて実体化した遷移条件表を表 4.5、資源オブジェクトについて実体化した遷移条件表を表 4.6 に示す。

4.5.2　実験:誘導遷移表の実体化

4.5.1 同様、図 4.38 の Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$ のモデル検査用オブジェクトの例とクラスレベルの誘導遷移表 4.2 と 4.4 を元に誘導遷移表を実体化する。実体化した誘導遷移表を表 4.7 に示す。

4.5.3　実験:環境統合アルゴリズム

遷移条件表と誘導遷移表が実体化されれば、モデル検査用ステートマシン図の情報よりインスタンスレベルの遷移集合が得られる。

実体化した遷移条件表 4.5、4.6 と、実体化した誘導遷移表 4.7 の情報を元にインスタンスレベルの遷移集合が得られる。インスタンスレベルの遷移集合を 4.8 に示す。

インスタンスレベルの遷移集合を元に環境統合アルゴリズムを行う。環境統合アルゴリズムの実行例を表 4.9 に示す。表では各オブジェクト $Task_1$、$Task_2$、$Resource_1$ の状態の統合状態 us を左側の 3 列で表現している。統合状態 us の obc が真ならば誘導遷移を含む遷移後の統合状態を記述する。遷移後の統合状態は右側の 3 列で表現している。遷移後の統合状態が表になければ表に追加する。すべての統合状態がチェックされたら終了する。

環境統合アルゴリズムで得られたステートマシン図記述の検査モデルを図 4.44 に示す。

表 4.5: タスクのインスタンスレベルの遷移条件表 (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$)

$Task_1$

	obf_1	obf_2	obf_3	obf_4	obf_5	obf_6	obf_7
obc_1	T2==SUS		−	−	−	−	R1==FREE
obc_2	T2==RUN	TRUE	−	−	−	−	R1==FREE
obc_3	T2==RUN	FALSE	−	−	−	−	−
	T2==RUN	−	−	−	−	−	R1==OCC
obc_4	−	−	T2==SUS	−	−	−	R1==FREE
obc_5	−	−	T2==READY	−	−	−	R1==FREE
obc_6	−	−	T2==SUS	−	−	−	R1==FREE
	−	−	T2==READY	TRUE	−	−	R1==FREE
obc_7	−	−	T2==READY	FALSE	−	−	R1==FREE
obc_8	−	−	T2==SUS	−	T2==SUS	−	R1==FREE
	−	−	T2==READY	−	FALSE	FALSE	R1==FREE
obc_9	−	−	T2==READY	−	T2==SUS	FALSE	R1==FREE

$Task_2$

	obf_1	obf_2	obf_3	obf_4	obf_5	obf_6	obf_7
obc_1	T1==SUS		−	−	−	−	R1==FREE
obc_2	T1==RUN	FALSE	−	−	−	−	R1==FREE
obc_3	T1==RUN	TRUE	−	−	−	−	−
	T1==RUN	−	−	−	−	−	R1==OCC
obc_4	−	−	T1==SUS	−	−	−	R1==FREE
obc_5	−	−	T1==READY	−	−	−	R1==FREE
obc_6	−	−	T1==SUS	−	−	−	R1==FREE
	−	−	T1==READY	FALSE	−	−	R1==FREE
obc_7	−	−	T1==READY	TRUE	−	−	R1==FREE
obc_8	−	−	T1==SUS	−	T1==SUS	−	R1==FREE
	−	−	T1==READY	−	FALSE	FALSE	R1==FREE
obc_9	−	−	T1==READY	−	T1==SUS	FALSE	R1==FREE

表 4.6: 資源のインスタンスレベルの遷移条件表 (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$)

$Resource_1$

	obf_1	obf_2	obf_3	obf_4
obc_1	T1==RUN	-	-	-
obc_2	T1==RUN	-	-	FALSE
	T1==RUN	T2==READY	TRUE	TRUE
	T1==RUN	T2==SUS	-	TRUE
obc_3	T1==RUN	T2==READY	FALSE	TRUE

表 4.7: インスタンスレベルの誘導遷移表 (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$、

$Task_1$

	YOb	YS
oby_1	T2	READY
oby_2	T2	RUN
oby_3	T2	RUN
oby_4	T2	RUN
oby_5	Null	-

$Task_2$

	YOb	YS
oby_1	T1	READY
oby_2	T1	RUN
oby_3	T1	RUN
oby_4	T1	RUN
oby_5	Null	-

$Resource_1$

	YOb	YS
oby_1	Null	-

表 4.8: インスタンスレベルの遷移集合 (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$)

Ob.S	ObO	ObC	ObY	Ob.S
$Task_1$:SUS	ActivateTask(T1)	T2==SUS && R1==FREE && R2==FREE	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	T2==RUN || (R1==OCC || R2==OCC)	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	T2==SUS && R1==FREE && R2==FREE	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	T2=READY && R1==FREE && R2==FREE	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	T2==SUS && R1==FREE && R2==FREE	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	T2==READY && R1==FREE && R2==FREE	(T2,RUN)	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	T2==SUS && R1==FREE && R2==FREE	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Task_2$:SUS	ActivateTask(T2)	T1==SUS && R1==FREE && R2==FREE	Null	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	T1==RUN || (R1==OCC || R2==OCC)	Null	$Task_2$:READY
$Task_2$:RUN	TerminateTask(T2)	T1==SUS && R1==FREE && R2==FREE	Null	$Task_2$:SUS
$Task_2$:RUN	TerminateTask(T2)	T1==READY && R1==FREE && R2==FREE	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T2)	T1==SUS && R1==FREE && R2==FREE	Null	$Task_2$:RUN
$Task_2$:RUN	ChainTask(T2)	T1==READY && R1==FREE && R2==FREE	(T1,RUN)	$Task_2$:READY
$Task_2$:RUN	ChainTask(T1)	T1==SUS && R1==FREE && R2==FREE	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T1)	FALSE	-	$Task_2$:SUS
$Resource_1$:FREE	GetResource(R1)	T1==RUN || T2==RUN	Null	$Resource_1$:OCC
$Resource_1$:OCC	RelseaseResource(R1)	T1==RUN && (R2 ==OCC || (R2==FREE && (T2==READY || T2==SUS))) ||		
		T2==RUN && (R2 ==OCC || (R2==FREE && (T1==READY || T1==SUS)))	Null	$Resource_1$:FREE
$Resource_1$:OCC	RelseaseResource(R1)	FALSE	-	$Resource_1$:FREE
$Resource_2$:FREE	GetResource(R2)	T1==RUN || T2==RUN	Null	$Resource_2$:OCC
$Resource_2$:OCC	RelseaseResource(R2)	T1==RUN && (R2 ==OCC || (R2==FREE && (T2==READY || T2==SUS))) ||		
		T2==RUN && (R2 ==OCC || (R2==FREE && (T1==READY || T1==SUS)))	Null	$Resource_2$:FREE
$Resource_2$:OCC	RelseaseResource(R2)	FALSE	-	$Resource_2$:FREE

表 4.9: 環境統合アルゴリズムの実行例

us			ObO	ObC	真偽	us		
$Task_1$	$Task_2$	Res_1				$Task_1$	$Task_2$	Res_1
SUS	SUS	FREE	ActivateTask(T1)	T2==SUS && R1==FREE	○	RUN	SUS	FREE
			ActivateTask(T1)	T2==RUN && R1==FREE				
			ActivateTask(T1)	T2==RUN || R1==OCC				
			ActivateTask(T2)	T1==SUS && R1==FREE	○	SUS	RUN	FREE
			ActivateTask(T2)	FALSE				
			ActivateTask(T2)	T1==RUN || R1==OCC				
			GetResource(R1)	T1==RUN				
RUN	SUS	FREE	TerminateTask(T1)	T2==SUS && R1==FREE	○	SUS	SUS	FREE
			TerminateTask(T1)	T2=READY && R1==FREE				
			ChainTask(T1)	(T2==SUS || T2==READY) && R1==FREE	○	RUN	SUS	FREE
			ChainTask(T1)	FALSE				
			ChainTask(T2)	T2==SUS && R1==FREE	○	SUS	RUN	FREE
			ChainTask(T2)	FALSE				
			ActivateTask(T2)	T1==SUS && R1==FREE				
			ActivateTask(T2)	FALSE				
			ActivateTask(T2)	T1==RUN || R1==OCC	○	RUN	READ	FREE
			GetResource(R1)	T1==RUN	○	RUN	SUS	OCC
SUS	RUN	FREE	ActivateTask(T1)	T2==SUS && R1==FREE				
			ActivateTask(T1)	T2==RUN && R1==FREE	○	RUN	READ	FREE
			ActivateTask(T1)	T2==RUN || R1==OCC				
			TerminateTask(T2)	T1==SUS && R1==FREE	○	SUS	SUS	FREE
			TerminateTask(T2)	T1==READY && R1==FREE				
			ChainTask(T2)	T1==SUS && R1==FREE	○	SUS	RUN	FREE
			ChainTask(T2)	T1==READY && R1==FREE				
			ChainTask(T1)	T1==SUS && R1==FREE	○	RUN	SUS	FREE
			ChainTask(T1)	FALSE				
			GetResource(R1)	T1==RUN				
RUN	READY	FREE	TerminateTask(T1)	T2==SUS && R1==FREE				
			TerminateTask(T1)	T2=READY && R1==FREE	○	SUS	RUN	FREE
			ChainTask(T1)	(T2==SUS || T2==READY) && R1==FREE	○	RUN	READ	FREE
			ChainTask(T1)	FALSE				
			ChainTask(T2)	T2==SUS && R1==FREE				
			ChainTask(T2)	FALSE				
			GetResource(R1)	T1==RUN	○	RUN	READ	OCC
RUN	SUS	OCC	TerminateTask(T1)	T2==SUS && R1==FREE				
			TerminateTask(T1)	T2=READY && R1==FREE				
			ChainTask(T1)	(T2==SUS || T2==READY) && R1==FREE				
			ChainTask(T1)	FALSE				
			ChainTask(T2)	T2==SUS && R1==FREE				
			ChainTask(T2)	FALSE				
			ActivateTask(T2)	T1==SUS && R1==FREE				
			ActivateTask(T2)	FALSE				
			ActivateTask(T2)	T1==RUN || R1==OCC	○	RUN	READ	OCC
			RelseaseResource(R1)	T1==RUN && (T2==READY || T2==SUS)	○	RUN	SUS	FREE
			RelseaseResource(R1)	FALSE				
RUN	READY	OCC	TerminateTask(T1)	T2==SUS && R1==FREE				
			TerminateTask(T1)	T2=READY && R1==FREE				
			ChainTask(T1)	(T2==SUS || T2==READY) && R1==FREE				
			ChainTask(T1)	FALSE				
			ChainTask(T2)	T2==SUS && R1==FREE				
			ChainTask(T2)	FALSE				
			RelseaseResource(R1)	T1==RUN && (T2==READY || T2==SUS)	○	RUN	READ	FREE
			RelseaseResource(R1)	FALSE				

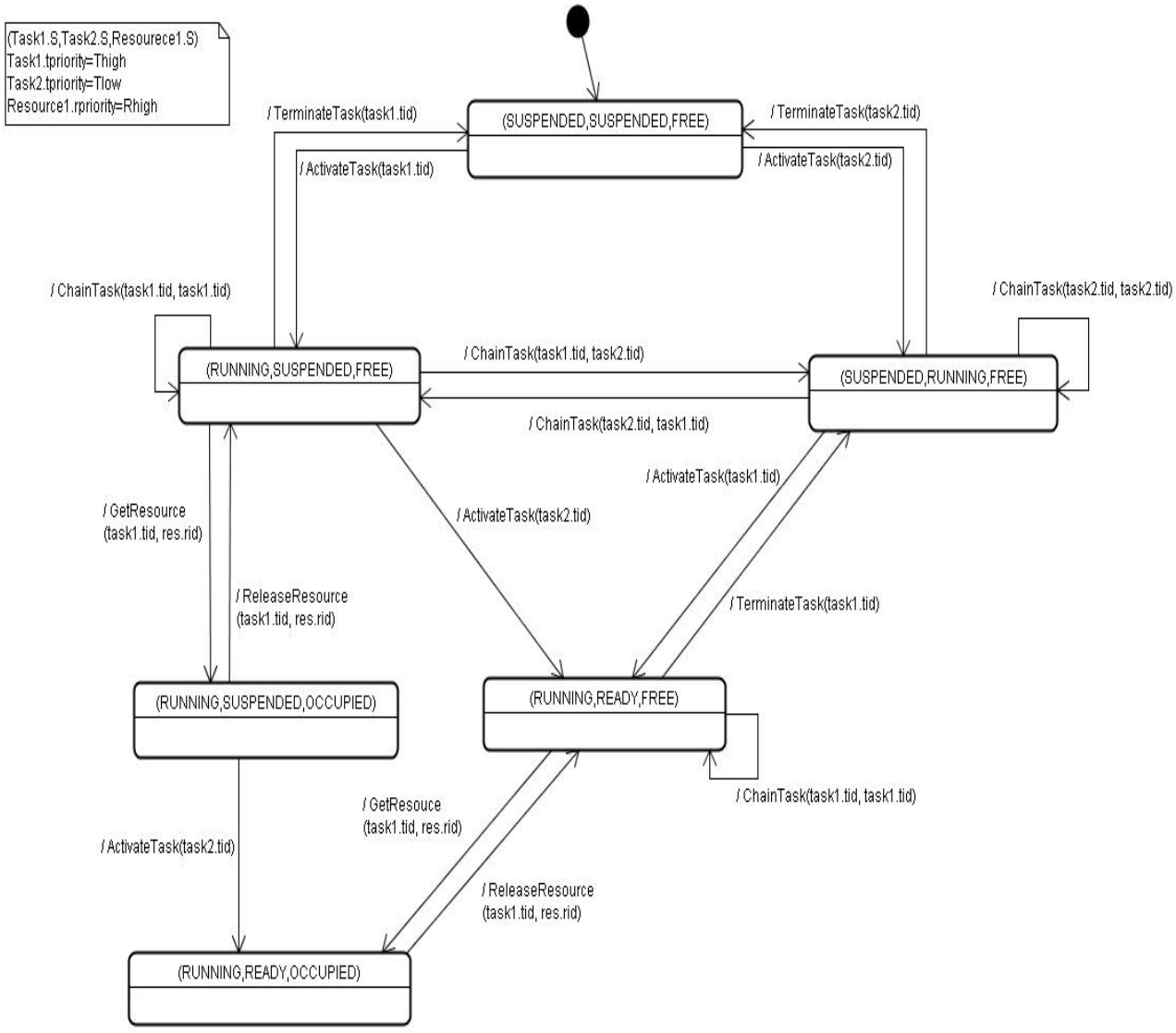


図 4.44: ステートマシン記述の検査モデル (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$)

4.6 実験:Promela 変換

図 4.44 で得られたステートマシン図記述の検査モデルをを 4.10 の対応表をもとに変換した。Promela 記述の検査モデルを図 4.45 に示す。

```
active proctype T2_R1_A_HLH(){
SUS_SUS_FREE:
  if
  ::ActivateTask(_task1._tid) -> goto RUN_SUS_FREE;
  ::ActivateTask(_task2._tid) -> goto SUS_RUN_FREE;
  fi;

RUN_SUS_FREE:
  if
  ::TerminateTask(_task1._tid) -> goto SUS_SUS_FREE;
  ::ChainTask(_task1._tid, _task2._tid) -> goto SUS_RUN_FREE;
  ::ActivateTask(_task2._tid) -> goto RUN_READY_FREE;
  ::GetResource(_task1._tid, _res1._rid) -> goto RUN_SUS_OCC;
  fi;

SUS_RUN_FREE:
  if
  ::TerminateTask(_task2._tid) -> goto SUS_SUS_FREE;
  ::ChainTask(_task2._tidk _task1._tid) -> goto RUN_SUS_FREE;
  ::ActivateTask(_task1._tid) -> goto RUN_READY_FREE;
  fi;

RUN_SUS_OCC:
  if
  ::ActivateTask(_task2._tid) -> goto RUN_READY_OCC;
  ::ReleaseResource(_task1._tid, _res1._rid) -> goto RUN_SUS_FREE;
  fi;

RUN_READY_FREE:
  if
  ::TerminateTask(_task1._tid) -> goto SUS_RUN_FREE;
  ::GetResource(_task1._tid, _res1._rid) -> goto RUN_READY_OCC;
  fi;

RUN_READY_OCC:
  if
  ::ReleaseResource(_task1._tid, _res1._rid) -> goto RUN_READY_FREE;
  fi;
}
```

図 4.45: Promela 記述の検査モデルの例

4.7 検査実験

Promela 記述の検査モデルを用いて OSEK/VDX 仕様の検査を行った。検査するにあたり、図 3.12 の記述では不十分である。不十分であると考えられる理由は 2 点ある。1 つめに、検査項目の記述がない点である。そのためサービスコールや優先度によって正しい状態変化が行われているか確認するため、assert 表明を追加した。状態変化、優先度による動作を確認するためのサービスコールとして GetTaskState と GetDPriority を追加した。これらのサービスコールは状態変化の起点にはならないため、検査モデルの各状態に追加しても振舞いには影響しない。2 つめに Promela で実装されている検査対象モデルの記述に合わせる必要がある点である。各変数は検査対象モデルの記述と整合性がとれるように記述した。以上の点を踏まえて図 3.12 にサービスコール、表明、変数などを追加した。

検査した結果を図 4.46 に示す。

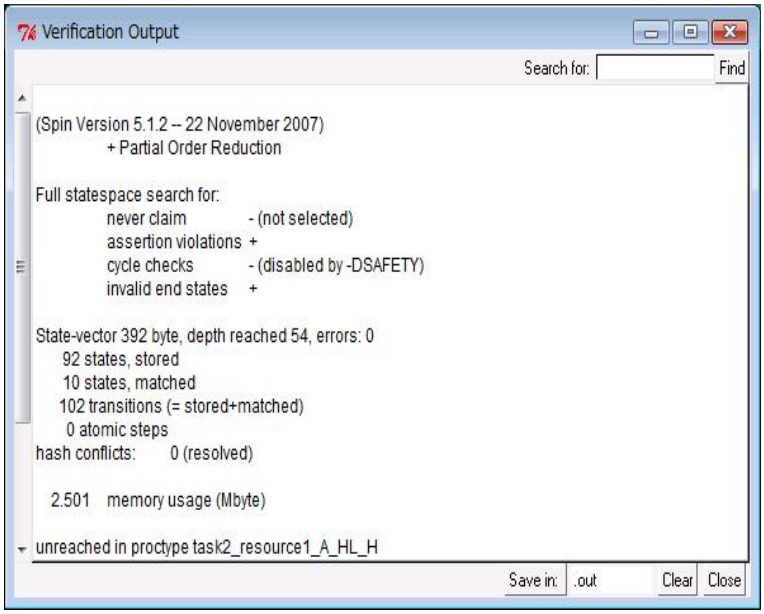


図 4.46: 検査結果 (エラーなし)

次に、不具合を検出できるか確認するために検査対象モデルの一部をコメントアウトして検査した。図 4.47 に示すようにサービスコールの GetResource 内のタスクの優先度を資源のシーリング優先度にする部分をコメントアウトした。

検査した結果を図 4.48 に示す。

```
 * id:自分自身のタスク ID
 * rid:リソース ID
 */
inline _GetResource(id,rid){
  if
  :: (rid < 0) || (N_RES - 1 < rid) ->
     ercd = E_OS_ID;
     ErrorHook

  :: else ->
    if
    :: res[rid].exist_flag ->
       get_index_state(id);
       if
       :: ret_ix == NOINDEX ->
          LibErrorHook("Cannot find task index\n")

       :: else ->
          if
          :: tsk_state[ret_ix].tstat == READY ->
             deq_prio(tsk_state[ret_ix].dpriority, id);
             enq_prio(res[rid].rpriority, id);
             /* タスクの実行時優先度をリソースの優先度にする. */
             if
             :: tsk_state[ret_ix].dpriority < res[rid].rpriority ->
             /*   tsk_state[ret_ix].dpriority = res[rid].rpriority  */
             :: else
             fi;
             res[rid].rtask = id

          :: else ->
             LibErrorHook("Task calling GetResource is not in READY or RUN.\n")
          fi
       fi
    :: else ->
       ercd = E_OS_ID;
       ErrorHook
    fi
  fi
}
```

図 4.47: コメントアウトしたサービスコール GetResource

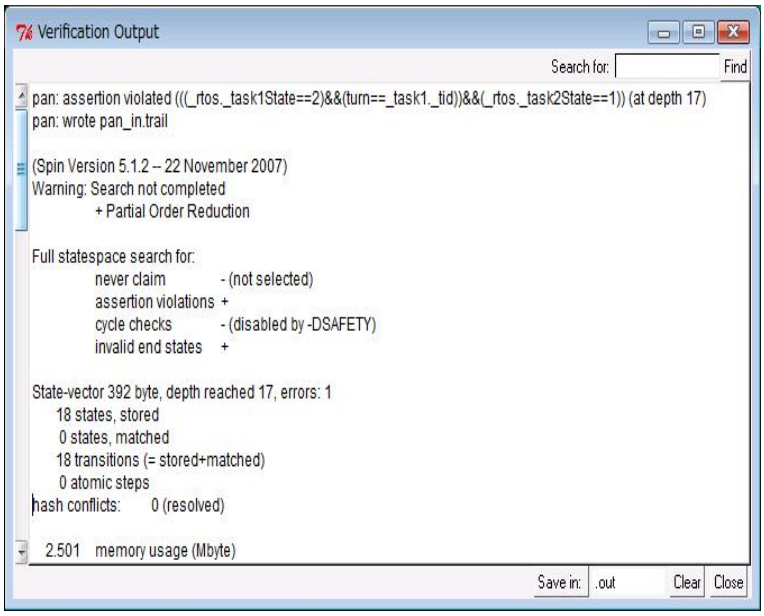


図 4.48: 検査結果 (エラーあり)

4.8 評価・考察

提案手法により検査したい機能毎の外部環境を任意に統合した検査モデルを生成した。検査モデルの状態数はタスク 2 つ、資源 2 つのパターン D、優先度が共に等しい場合が最も多く、17 状態であった。計算上は $3 \times 3 \times 2 \times 2 = 36$ であるため、約半分の状態に遷移しない事が確認できた。提案手法により、統合する環境毎に制約を考える必要がなくなり、効率的に検査モデルを生成できる。

OSEK/VDX 仕様ではそれぞれの同意義パターン定義により組み合わせパターン数は表 4.10 のように減少した。表ではタスク、資源の属性を 2 値で実体化した場合について計算した値を表記している。表より、同意義パターン定義により組み合わせパターン数が大きく減少している事が読み取れる。よって、同意義パターン定義は多重度、属性を多値にわたって実体化する場合など有効であると考えられる。しかしながら実体化したオブジェクトに対して、同意義パターンとするものは曖昧であり、検査対象によって異なる。また、同意義パターンと定義しても実際には違う振舞いをする事も十分考えられる。よって、同意義パターン定義はあくまで組み合わせパターン数が膨大になると想定され、そのすべての検査モデルを生成する事が困難であると場合に適用するのが有効である。

環境モデリング手法のメリットとして、複雑な環境統合を検査モデルごとに分析する労力を削減できる点である。また、人手による分析漏れ、記入ミスなどのリスクも削減できると考えられる。しかしながら、提案手法ではモデリング段階で遷移の種類・遷移条件・誘導遷移の情報を正確に分析しなければ、期待する検査モデルを得ることはできない。同意義パターン定義によるメリットとして、計算上膨大な数になると考えれられる検査モデルに対して生成する検査モデルの数自体を減らす事により、人手の負担を減らす事になる

表 4.10: 同意義パターン定義による組み合わせパターン数の変化 (OSEK/VDX 仕様例)

タスク多重度	資源多重度	実体化後の組み合わせパターン数	同意義パターン除去後の組み合わせパターン数
1	0	2	1
1	1	8	1
1	2	32	2
2	0	4	2
2	1	32	5
2	2	256	24

と考えられる。同意義パターン定義において、検査モデルの生成の作業量を削減できると考えられるが、それはすべての組み合わせパターンをすべて検査するという事でないと理解して導入すべきである。小規模な検査対象システムの場合、検査モデルも少数で良いと考えられる。そのような少数の検査モデルの生成時には従来手法のモデリングで十分であると考えられる。しかしながら、本研究で適用した OSEK/VED 仕様のような大規模な検査システムの場合、多重度、属性の実体化の与え方によっては大量の検査モデルが必要になる。また、個々の検査モデルのついて振舞いを分析して生成する必要がある。よって大規模な検査対象が大規模な場合、提案手法による環境モデリングと同意義パターンの除去は有効であると考えられる。

検査実験の結果より、提案手法で生成した検査モデルにより不具合を発見できる事を確認した。提案手法により発見できる不具合は、状態や属性が考慮しない値になった場合を確認することができると考えられる。つまり assert 表明で記述したエラーに到達する場合である。しかしながら、提案手法は正しい動作をする事を前提に検査モデルを生成するため、エラーフックや並行動作による不具合は発見が出来ない。

第5章　まとめ

5.1　研究のまとめ

提案手法では検査モデル生成の困難さの解決法として、環境モデリング手法、同意義パターン定義の2つを提案した。

環境モデリング手法では、状態から発生する遷移の種類を抽出する遷移抽出図のダイアグラムを提案した。遷移抽出図を元に、モデル検査用クラス図とモデル検査用ステートマシン図の2種類を用いて環境モデリングを行った。各遷移のクラスレベルの遷移条件と誘導遷移を分析し、遷移条件表と誘導遷移表として表記した。実体化したオブジェクトをモデル検査用オブジェクトと呼ぶこととした。各モデル検査用オブジェクトの情報とクラスレベルの遷移の情報を元に、遷移の実体化を行った。実体化した遷移の情報を環境統合アルゴリズムによって、機械的に統合し検査モデルを生成した。生成した検査モデルを、モデル検査器 SPIN の記述言語である Promela に変換するための対応関係を明示した。

同意義パターン定義では、同じ振舞いすると考えられるオブジェクトを同意義パターンとして定義する事を提案した。多重度実体化時点と属性実体化時点の2段階で同意義パターン定義を行った。

提案手法を OSEK/VDX 仕様に適用し、検査モデルを生成した。生成した検査モデルにより不具合を発見できる事を確認した。

5.2　今後の課題

本手法をより、実用的にするためにはモデル化において記述する内容を拡張すべきである。具体的に、条件遷移表、誘導遷移表に必要な情報もモデリング段階で記述すべきである。本研究ではクラスレベルの遷移条件表、誘導遷移表を作成し、同意義パターンを定義した。それらを作成、定義するために関数や集合を用いた。こういった情報も、モデリング段階で記述すべきであり、表記能力の拡張を行うべきである。また、本手法では多重度2値、属性2値による実体化を想定している。よって、3値以上の場合はこれらの情報の修正が必要である。タスク1、タスク2、タスク3のタスクが3つ場合を想定する。タスク1が RUNNING 状態でもう2つが READY 状態であるとする。TerminateTask が発行された場合、タスク1が SUSPENDED 状態へ遷移し、タスク2かタスク3の内、優先度の高いタスクが SUSPENDED 状態のタスクが RUNNING 状態に遷移される。。タスク2とタスクの3の優先度が同じ場合は先に READY 状態になったタスクが RUNNING 状態

に遷移される。よって、キュー構造を準備すべきである。こういった情報もまた、モデリングの段階で記述すべきである。

提案手法では2.2.4の2つめに挙げた検査したい振舞いと検査できる振舞いが明確になっている点は不十分である。よって、検査できる振舞いを明確に記述できるようにUMLを拡張すべきである。モデル検査には状態爆発問題により検査できない性質も多数あるため、検査不可能な性質について明確になっている事が望ましいと考えられる。

本手法は割り込みや並行動作を想定としていない。RTOSの性質上、これらの動作の信頼性の向上は必須であると考えられる。よって、それらの性質を検査するための拡張を提案手法に追加する必要があると考えられる。

本手法で作成した検査モデルは、取りうる状態のみをモデル化した。しかしながら、エラーハンドリングも信頼性の向上には不可欠な要素である。よって、エラーを含ませた検査モデルを生成により、想定するエラーが検出できるかの確認が行えると考えられる。

謝辞

北陸先端科学技術大学院大学 安心電子研究センター 青木利晃特任准教授には、本研究を進めるにあたり親切丁寧な御指導を頂き、厚く御礼申し上げます。また有益な助言を頂いた、同大学院 岸知二特任教授、矢竹健朗助教に感謝の意を表明します。

本研究を行う上で、相談にのっていただいた青木研究室 土肥 雅俊さんに感謝いたします。また、他研究室でありながら、貴重なアドバイスをいていただいた片山研究室 Chaiwat Sathawornwichit さん、谷崎裕明さん、岸研究室 金井隼人さん、細合晋太郎さんに感謝いたします。

その他、有意義な研究生活を過ごす事ができた青木研究室、岸研究室、片山研究室、Defago 研究室の皆様に深く感謝いたします。

参考文献

[1] 中島震, SPIN モデル検査 検証モデリング技法, 近代科学社, 2008.

[2] 萩原昌己 吉岡信和 青木利晃 田原康之 共著, SPIN による設計モデル検証, 近代科学者,2008.

[3] 穴田啓樹, 日経エレクトロニクス:組み込みアカデミー 2, 2008.

[4] http://portal.osek-vdx.org/.

[5] H.E. エリクソン M. ペンカー 監訳 杉本宣男 落合修 武田多美子, UML ガイドブック, 株式会社トッパン, 1998.

付 録A　遷移集合

表 A.1: インスタンスレベルの遷移集合 (Task1 つ、Resource なし、$Task_1.priority = Thigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	TRUE	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	-	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	TRUE	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	FALSE	-	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	TRUE	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	FALSE	-	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS

表 A.2: インスタンスレベルの遷移集合 (Task1 つ、Resource1 つ、$Task_1.priority = Thigh$、$Resource_1.rpriority = Rhigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	TRUE	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	-	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	TRUE	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	FALSE	-	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	R1==FREE	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	FALSE	-	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Resource_1$:FREE	GetResource(R1)	T1==RUN	Null	$Resource_1$:OCC
$Resource_1$:OCC	RelseaseResource(R1)	T1==RUN	Null	$Resource_1$:FREE
$Resource_1$:OCC	RelseaseResource(R1)	FALSE	-	$Resource_1$:FREE

表 A.3: インスタンスレベルの遷移集合 (Task1 つ、Resource2 つ、$Task_1.priority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resource_2.rpriority = Rhigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	(R1==FREE && R2==FREE)	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	-	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	(R1==FREE && R2==FREE)	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	FALSE	-	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	(R1==FREE && R2==FREE)	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	FALSE	-	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Resource_1$:FREE	GetResource(R1)	T1==RUN	Null	$Resource_1$:OCC
$Resource_1$:OCC	RelseaseResource(R1)	T1==RUN	Null	$Resource_1$:FREE
$Resource_1$:OCC	RelseaseResource(R1)	FALSE	-	$Resource_1$:FREE
$Resource_2$:FREE	GetResource(R2)	T1==RUN || T2==RUN	Null	$Resource_2$:OCC
$Resource_2$:OCC	RelseaseResource(R2)	T1==RUN	Null	$Resource_2$:FREE
$Resource_2$:OCC	RelseaseResource(R2)	FALSE	-	$Resource_2$:FREE

表 A.4: インスタンスレベルの遷移集合 (Task2 つ、Resource なし、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	T2==SUS	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	T2==RUN	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	T2==SUS	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	T2=READY	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	T2==SUS	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	T2==READY	(T2,RUN)	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	T2==SUS	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	–	$Task_1$:SUS
$Task_2$:SUS	ActivateTask(T2)	T1==SUS	Null	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	FALSE	–	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	T1==RUN	Null	$Task_2$:READY
$Task_2$:RUN	TerminateTask(T2)	T1==SUS	Null	$Task_2$:SUS
$Task_2$:RUN	TerminateTask(T2)	T1==READY	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T2)	T1==SUS	Null	$Task_2$:RUN
$Task_2$:RUN	ChainTask(T2)	T1==READY	(T1,RUN)	$Task_2$:READY
$Task_2$:RUN	ChainTask(T1)	T1==SUS	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T1)	FALSE	–	$Task_2$:SUS

表 A.5: インスタンスレベルの遷移集合 (Task2 つ、Resource なし、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	T2==SUS	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	T2==RUN	(T2,READY)	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	T2==SUS	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	T2=READY	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	T2==SUS || T2==READY	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	FALSE	–	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	T2==SUS	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	–	$Task_1$:SUS
$Task_2$:SUS	ActivateTask(T2)	T1==SUS	Null	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	FALSE	–	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	T1==RUN	Null	$Task_2$:READY
$Task_2$:RUN	TerminateTask(T2)	T1==SUS	Null	$Task_2$:SUS
$Task_2$:RUN	TerminateTask(T2)	T1==READY	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T2)	T1==SUS	Null	$Task_2$:RUN
$Task_2$:RUN	ChainTask(T2)	T1==READY	(T1,RUN)	$Task_2$:READY
$Task_2$:RUN	ChainTask(T1)	T1==SUS	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T1)	FALSE	–	$Task_2$:SUS

表 A.6: インスタンスレベルの遷移集合 (Task2 つ、Resource1 つ、パターン B、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	T2==SUS && R1==FREE	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	T2==RUN || R1==OCC	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	T2==SUS && R1==FREE	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	T2=READY && R1==FREE	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	T2==SUS && R1==FREE	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	T2==READY && R1==FREE	(T2,RUN)	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	T2==SUS && R1==FREE	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	–	$Task_1$:SUS
$Task_2$:SUS	ActivateTask(T2)	T1==SUS && R1==FREE	Null	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	T1==RUN || R1==OCC	Null	$Task_2$:READY
$Task_2$:RUN	TerminateTask(T2)	T1==SUS && R1==FREE	Null	$Task_2$:SUS
$Task_2$:RUN	TerminateTask(T2)	T1==READY && R1==FREE	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T2)	T1==SUS && R1==FREE	Null	$Task_2$:RUN
$Task_2$:RUN	ChainTask(T2)	T1==READY && R1==FREE	(T1,RUN)	$Task_2$:READY
$Task_2$:RUN	ChainTask(T1)	T1==SUS && R1==FREE	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T1)	FALSE	–	$Task_2$:SUS
$Resource_1$:FREE	GetResource(R1)	T1==RUN || T2==RUN	Null	$Resource_1$:OCC
$Resource_1$:OCC	RelseaseResource(R1)	T1==RUN || T2==RUN	Null	$Resource_1$:FREE
$Resource_1$:OCC	RelseaseResource(R1)	FALSE	–	$Resource_1$:FREE

表 A.7: インスタンスレベルの遷移集合 (Task2 つ、Resource2 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$、$Resource_2.rpriority = Rhigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	T2==SUS && (R1==FREE && R2==FREE)	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	T2==RUN && (R1==FREE && R2==FREE)	(T2,READY)	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	T2==RUN && (R1==OCC ‖ R2==OCC)	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	T2==SUS && (R1==FREE && R2==FREE)	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	T2=READY && (R1==FREE && R2==FREE)	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	(T2==SUS ‖ T2==READY) && (R1==FREE && R2==FREE)	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	FALSE	(T2,RUN)	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	T2==SUS && (R1==FREE && R2==FREE)	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Task_2$:SUS	ActivateTask(T2)	T1==SUS && (R1==FREE && R2==FREE)	Null	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	T1==RUN	Null	$Task_2$:READY
$Task_2$:RUN	TerminateTask(T2)	T1==SUS && (R1==FREE && R2==FREE)	Null	$Task_2$:SUS
$Task_2$:RUN	TerminateTask(T2)	T1==READY && (R1==FREE && R2==FREE)	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T2)	T1==SUS && (R1==FREE && R2==FREE)	Null	$Task_2$:RUN
$Task_2$:RUN	ChainTask(T2)	T1==READY && (R1==FREE && R2==FREE)	(T1,RUN)	$Task_2$:READY
$Task_2$:RUN	ChainTask(T1)	T1==SUS && (R1==FREE && R2==FREE)	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T1)	FALSE	-	$Task_2$:SUS
$Resource_1$:FREE	GetResource(R1)	T1==RUN	Null	$Resource_1$:OCC
$Resource_1$:OCC	RelseaseResource(R1)	T1==RUN && (R2==OCC ‖ (T2==READY && R2==FREE))	Null	$Resource_1$:FREE
$Resource_1$:OCC	RelseaseResource(R1)	FALSE	-	$Resource_1$:FREE
$Resource_2$:FREE	GetResource(R2)	T2==RUN	Null	$Resource_2$:OCC
$Resource_2$:OCC	RelseaseResource(R2)	T2==RUN && (R2 ==OCC ‖ T1==SUS)	Null	$Resource_2$:FREE
$Resource_2$:OCC	RelseaseResource(R2)	T2==RUN && T1==READY && R1==FREE	(T1,READY),(T2,RUN)	$Resource_2$:FREE

表 A.8: インスタンスレベルの遷移集合 (Task2 つ、Resource2 つ、パターン D、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resource_2.rpriority = Rhigh$)

OB.S	OB.O	OBC	OBY	OB.S
$Task_1$:SUS	ActivateTask(T1)	T2==SUS && R1==FREE && R2==FREE	Null	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_1$:SUS	ActivateTask(T1)	T2==RUN || (R1==OCC || R2==OCC)	Null	$Task_1$:READY
$Task_1$:RUN	TerminateTask(T1)	T2==SUS && R1==FREE && R2==FREE	Null	$Task_1$:SUS
$Task_1$:RUN	TerminateTask(T1)	T2=READY && R1==FREE && R2==FREE	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T1)	T2==SUS && R1==FREE && R2==FREE	Null	$Task_1$:RUN
$Task_1$:RUN	ChainTask(T1)	T2==READY && R1==FREE && R2==FREE	(T2,RUN)	$Task_1$:READY
$Task_1$:RUN	ChainTask(T2)	T2==SUS && R1==FREE && R2==FREE	(T2,RUN)	$Task_1$:SUS
$Task_1$:RUN	ChainTask(T2)	FALSE	-	$Task_1$:SUS
$Task_2$:SUS	ActivateTask(T2)	T1==SUS && R1==FREE && R2==FREE	Null	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	FALSE	(T2,READY)	$Task_1$:RUN
$Task_2$:SUS	ActivateTask(T2)	T1==RUN || (R1==OCC || R2==OCC)	Null	$Task_2$:READY
$Task_2$:RUN	TerminateTask(T2)	T1==SUS && R1==FREE && R2==FREE	Null	$Task_2$:SUS
$Task_2$:RUN	TerminateTask(T2)	T1==READY && R1==FREE && R2==FREE	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T2)	T1==SUS && R1==FREE && R2==FREE	Null	$Task_2$:RUN
$Task_2$:RUN	ChainTask(T2)	T1==READY && R1==FREE && R2==FREE	(T1,RUN)	$Task_2$:READY
$Task_2$:RUN	ChainTask(T1)	T1==SUS && R1==FREE && R2==FREE	(T1,RUN)	$Task_2$:SUS
$Task_2$:RUN	ChainTask(T1)	FALSE	-	$Task_2$:SUS
$Resource_1$:FREE	GetResource(R1)	T1==RUN || T2==RUN	Null	$Resource_1$:OCC
$Resource_1$:OCC	RelseaseResource(R1)	T1==RUN && (R2 ==OCC || (R2==FREE && (T2==READY || T2==SUS))) || T2==RUN && (R2 ==OCC || (R2==FREE && (T1==READY || T1==SUS)))	Null	$Resource_1$:FREE
$Resource_1$:OCC	RelseaseResource(R1)	FALSE	-	$Resource_1$:FREE
$Resource_2$:FREE	GetResource(R2)	T1==RUN || T2==RUN	Null	$Resource_2$:OCC
$Resource_2$:OCC	RelseaseResource(R2)	T1==RUN && (R2 ==OCC || (R2==FREE && (T2==READY || T2==SUS))) || T2==RUN && (R2 ==OCC || (R2==FREE && (T1==READY || T1==SUS)))	Null	$Resource_2$:FREE
$Resource_2$:OCC	RelseaseResource(R2)	FALSE	-	$Resource_2$:FREE

付 録B　検査モデル

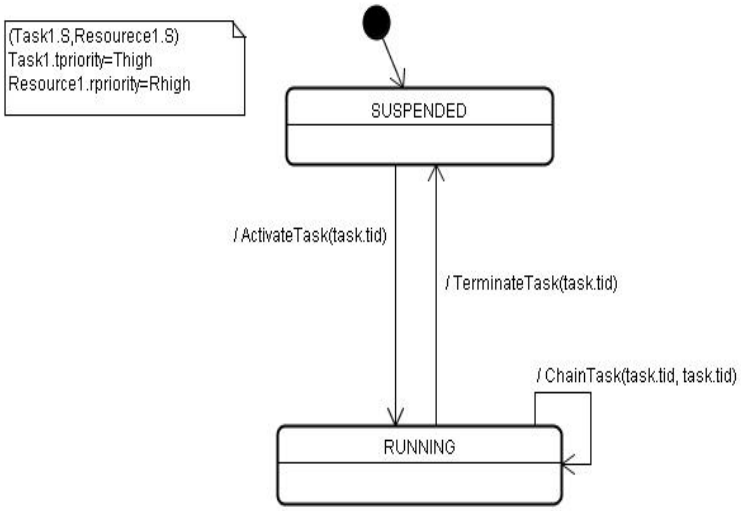


図 B.1: 検査モデル (Task1 つ、Resource なし、$Task_1.priority = Thigh$)

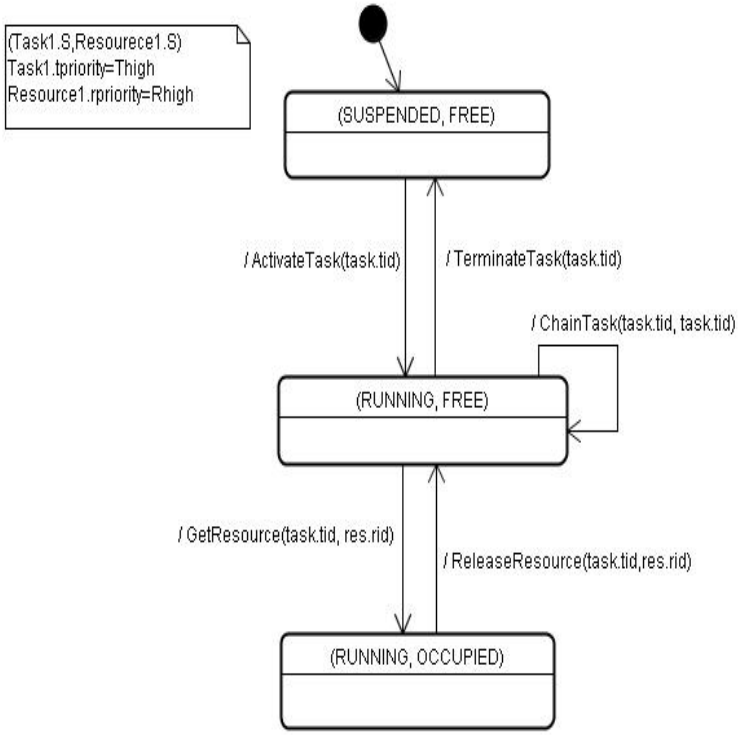


図 B.2: 検査モデル (Task1 つ、Resource1 つ、$Task_1.priority = Thigh$、$Resource_1.rpriority = Rhigh$)

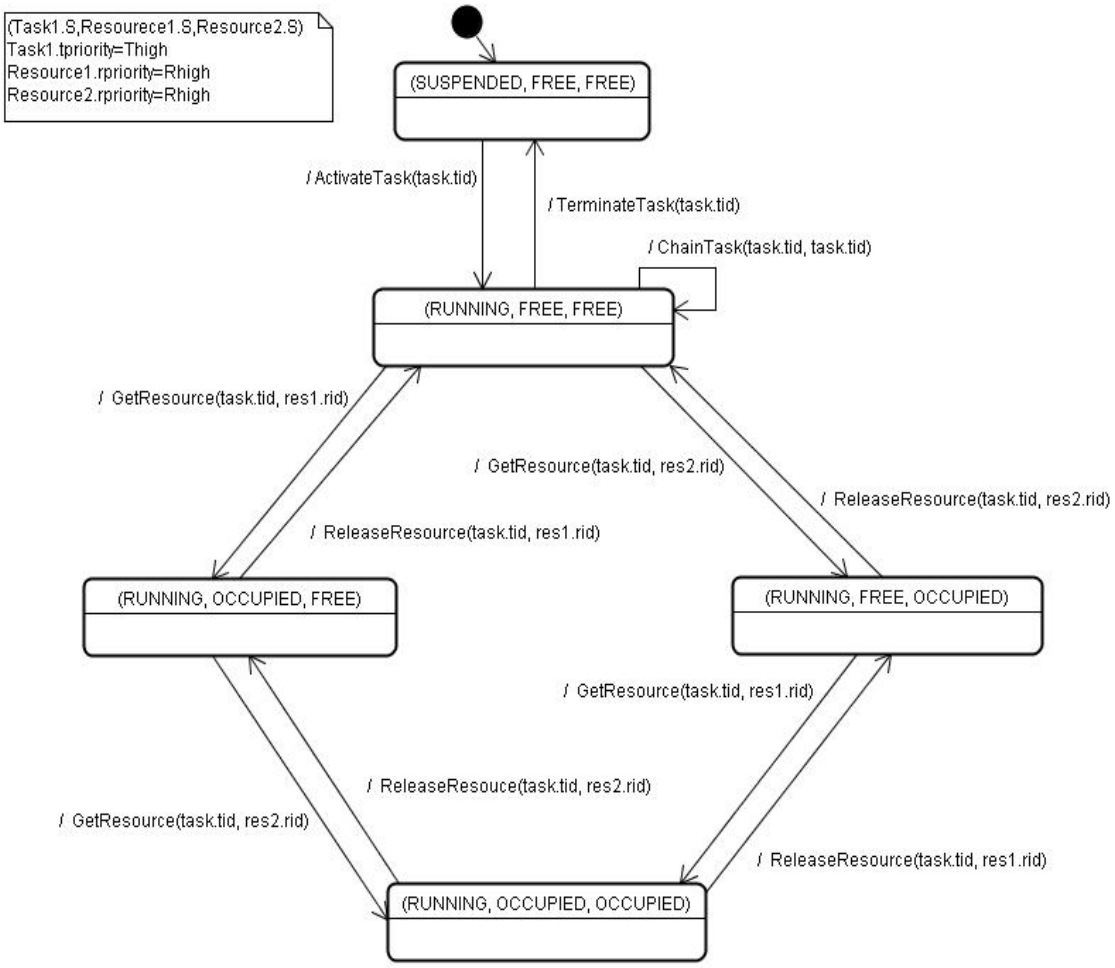


図　B.3:　　検査モデル (Task1 つ、Resource2 つ、$Task_1.priority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

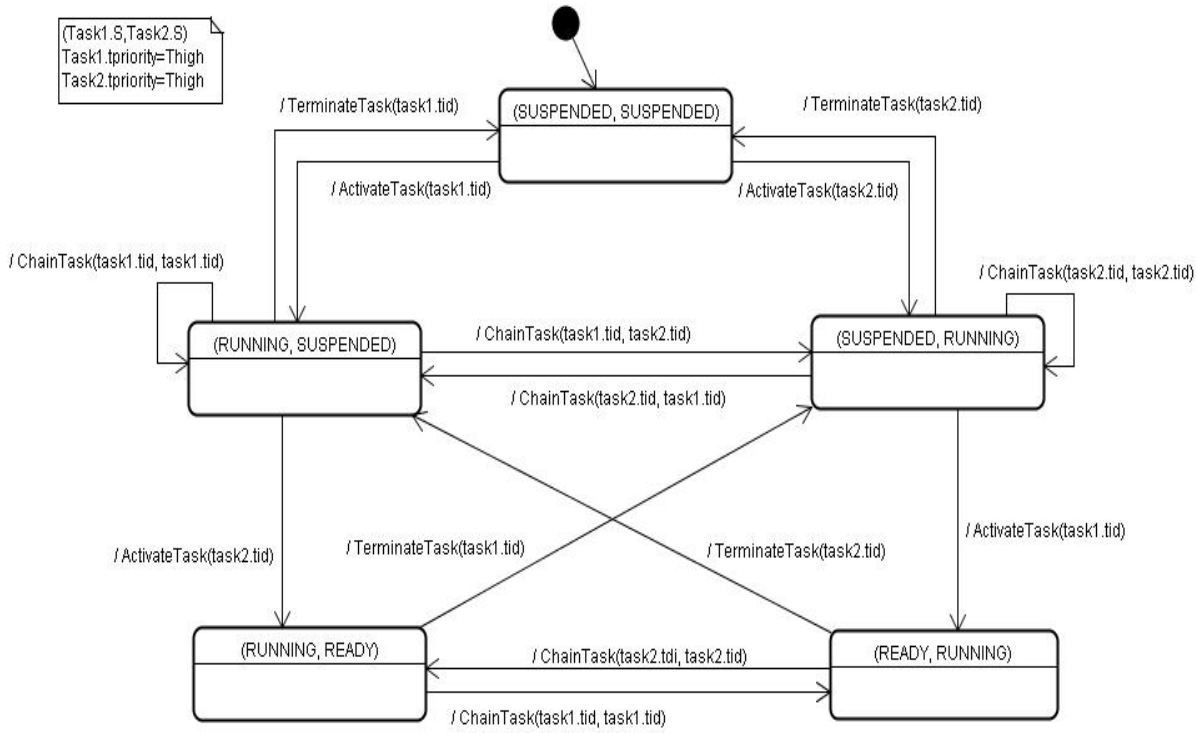


図 B.4: 検査モデル (Task2つ、Resourceなし、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$)

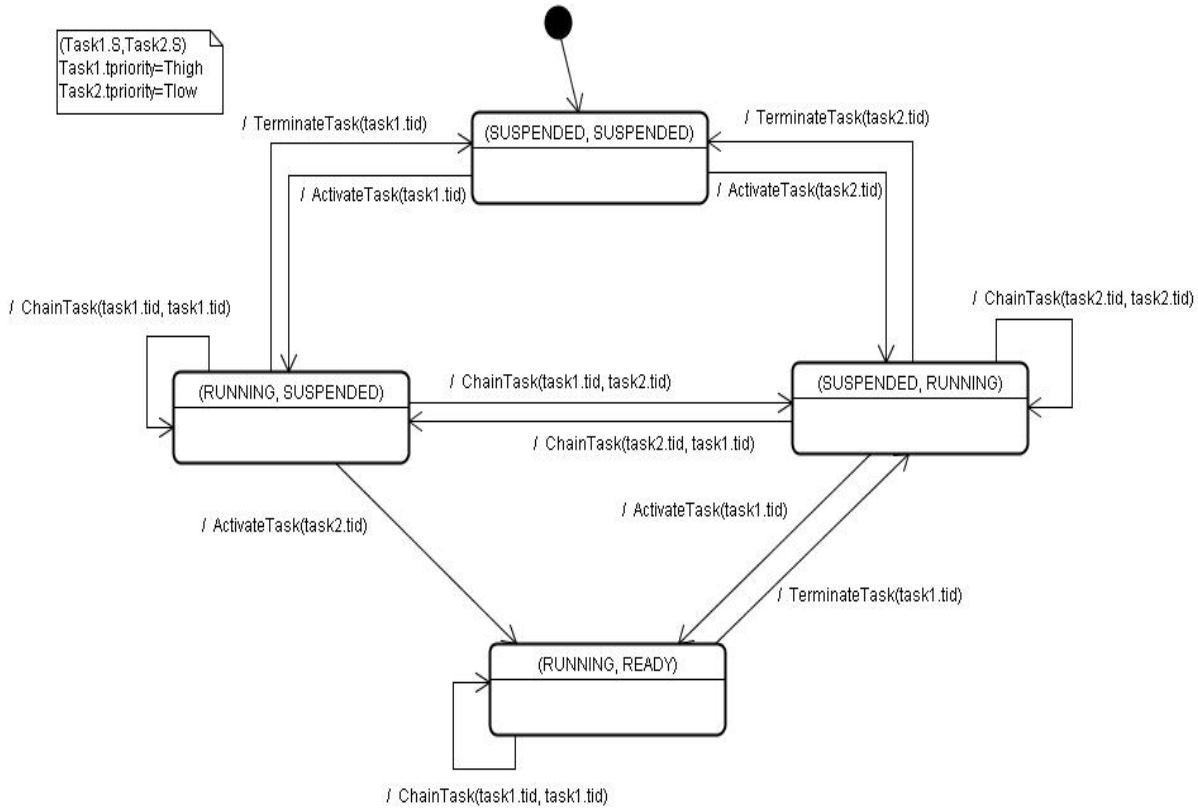


図 B.5: 検査モデル (Task2 つ、Resource なし、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$)

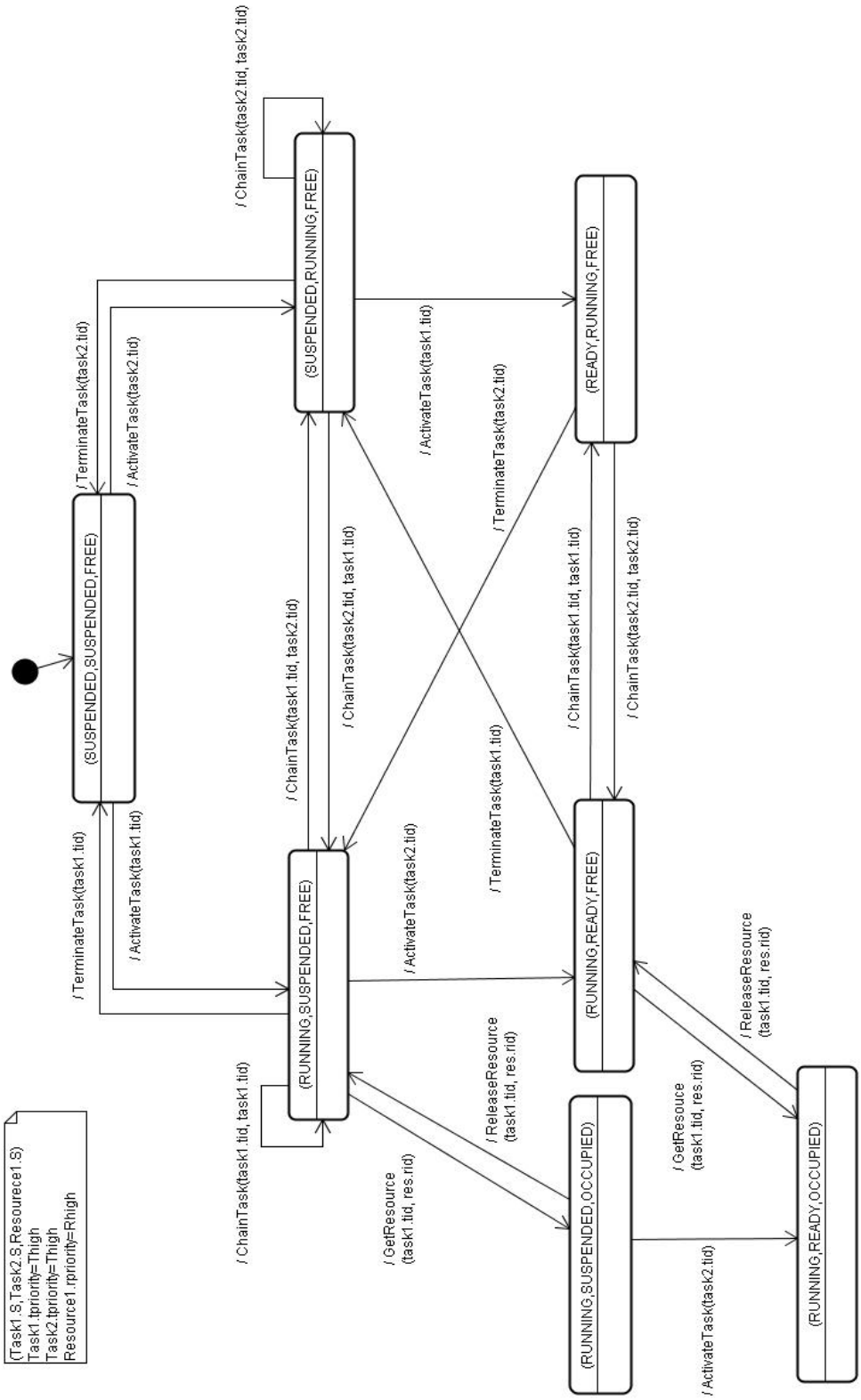


図 B.6: 検査モデル (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$)

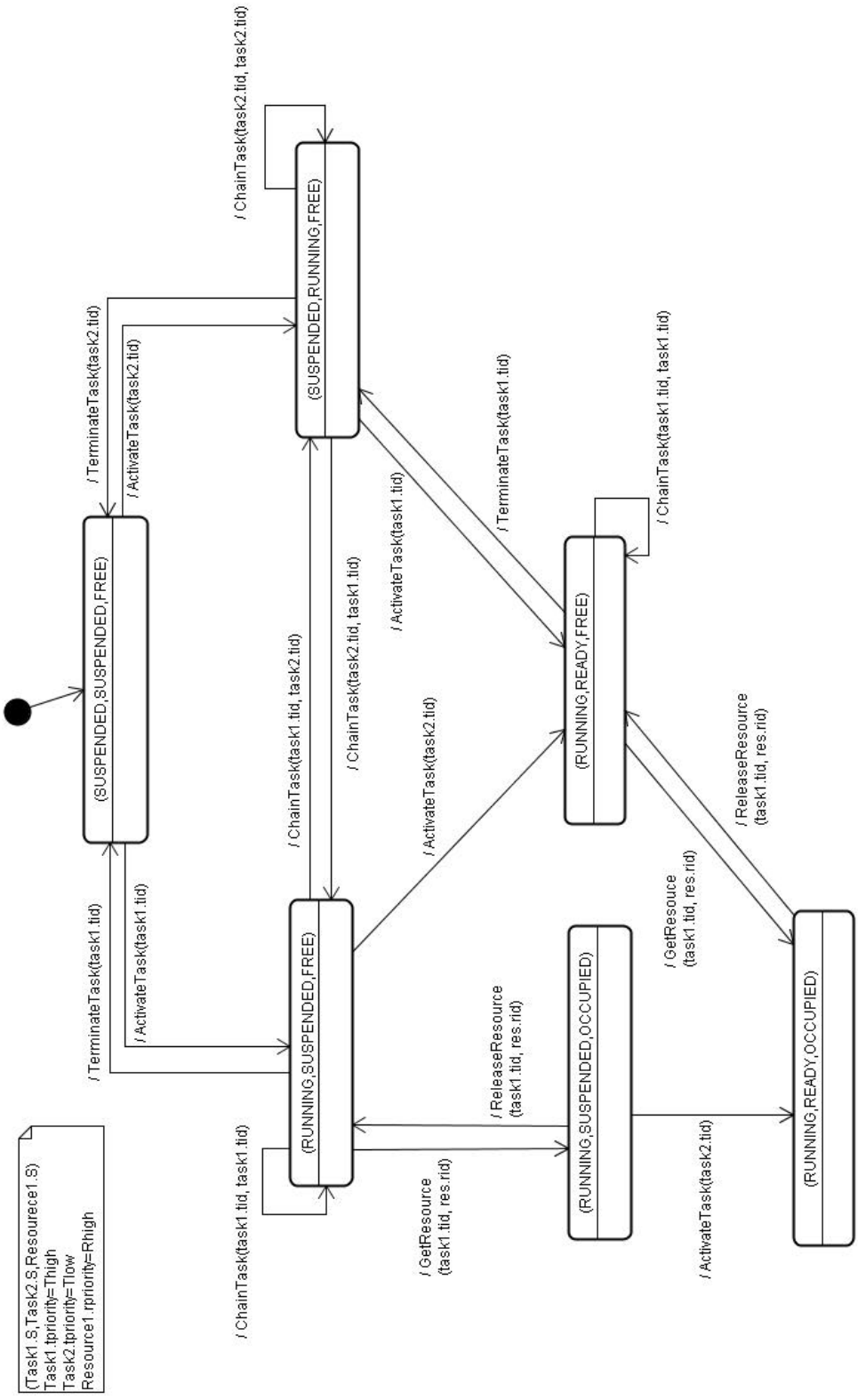


図 B.7: 検査モデル (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$)

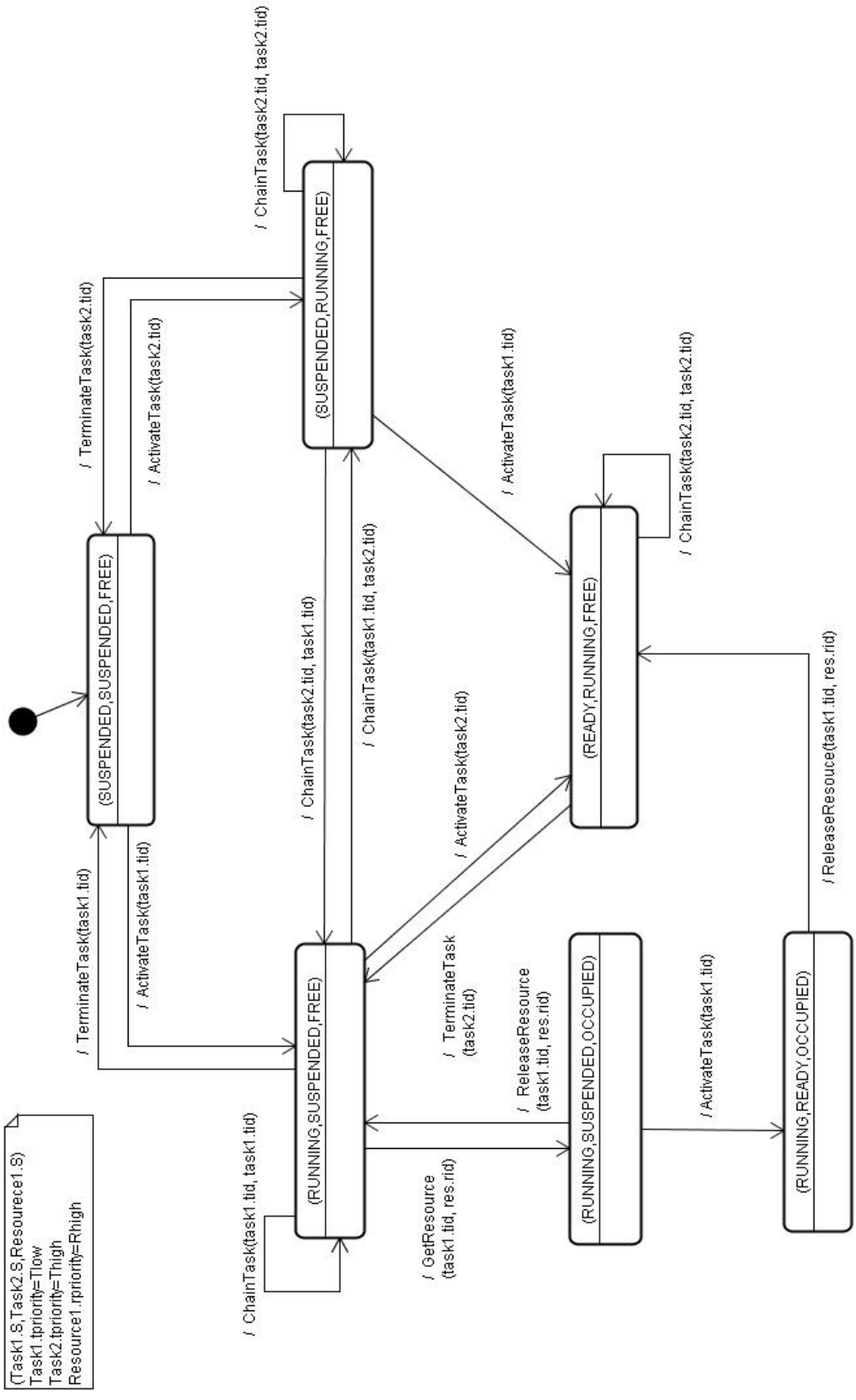


図 B.8: 検査モデル (Task2 つ、Resource1 つ、パターン A、$Task_1.priority = Tlow$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$)

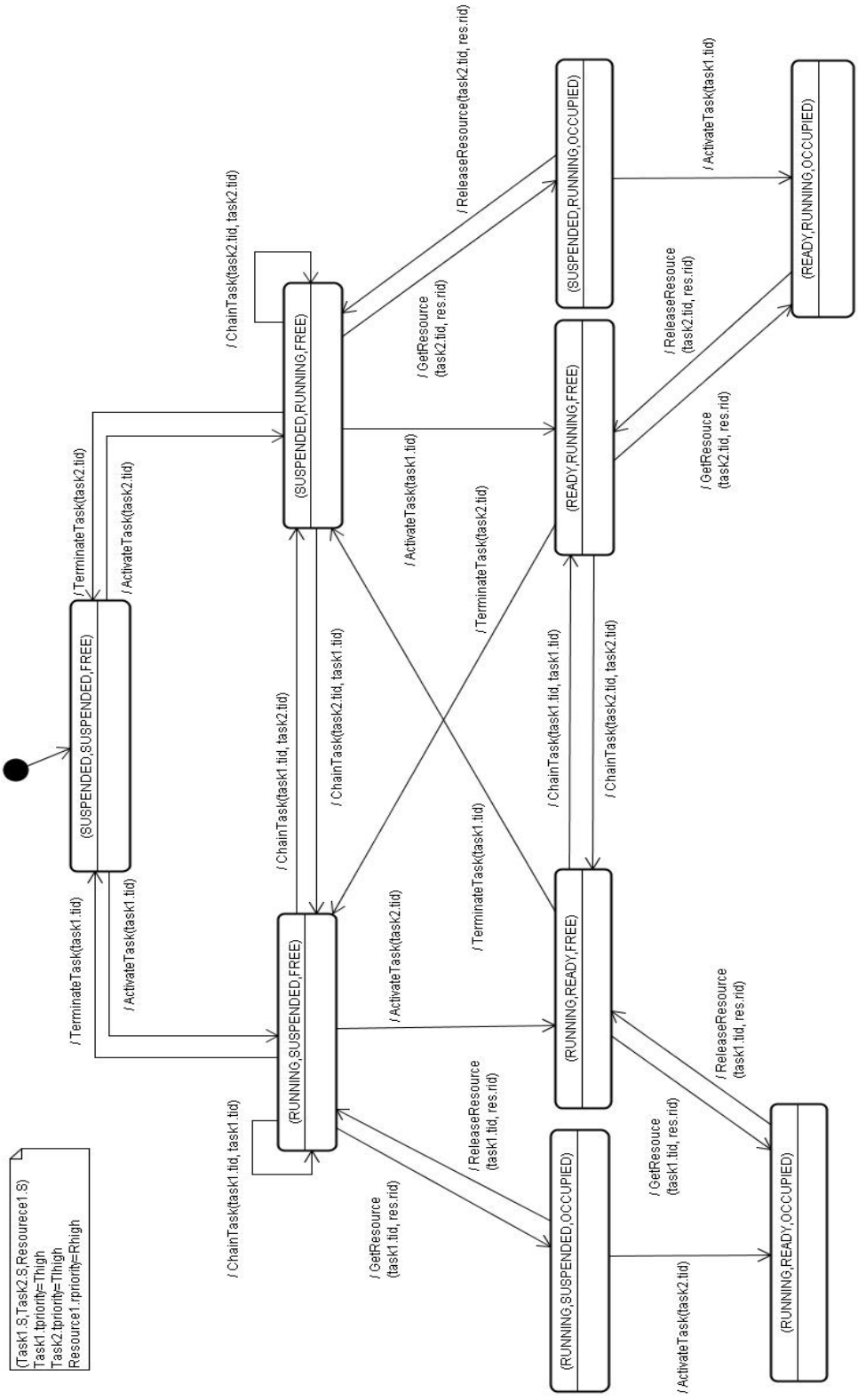


図 B.9: 検査モデル (Task2 つ、Resource1 つ、パターン B、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$)

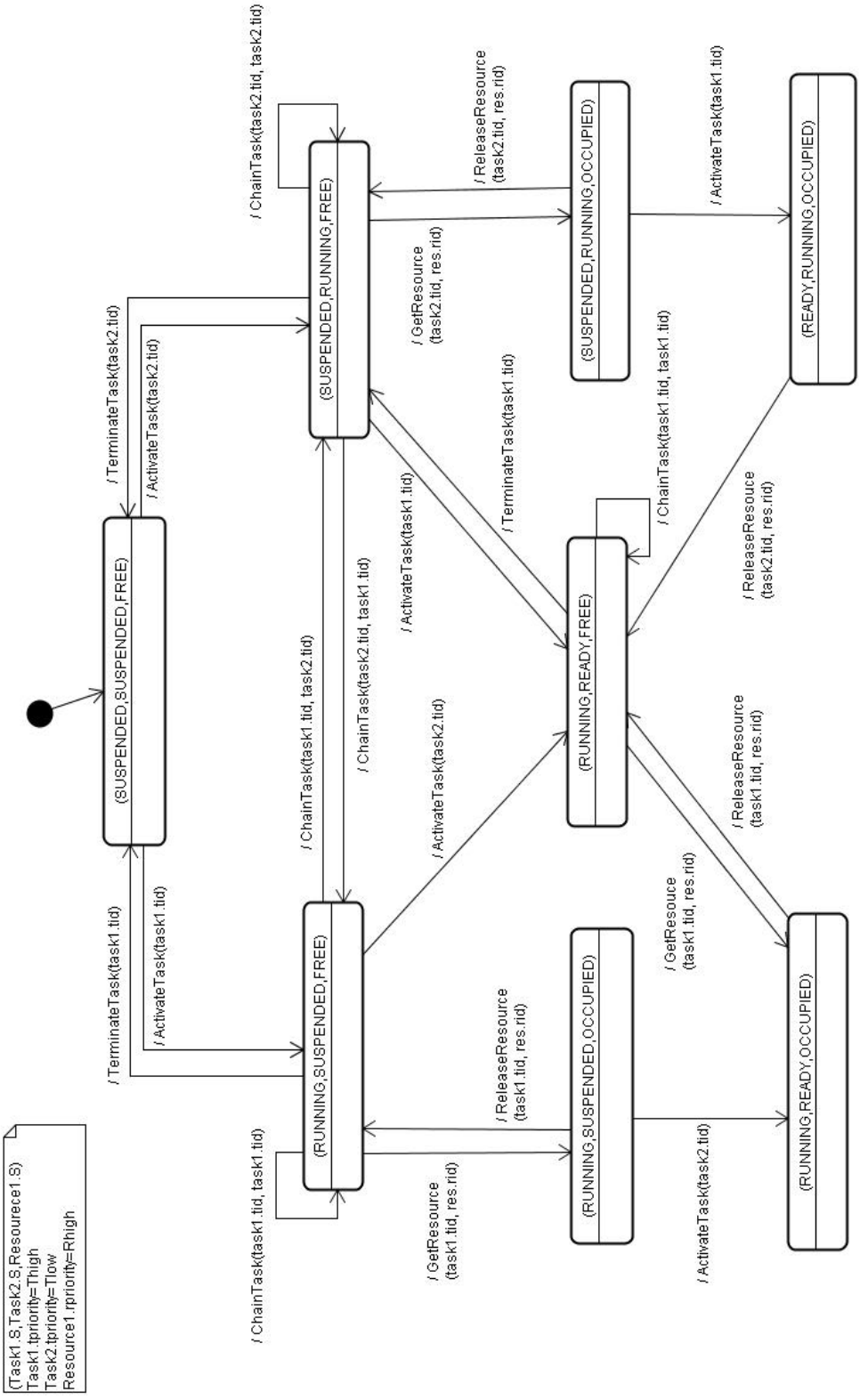


図 B.10: 検査モデル (Task2 つ、Resource1 つ、パターン B、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$)

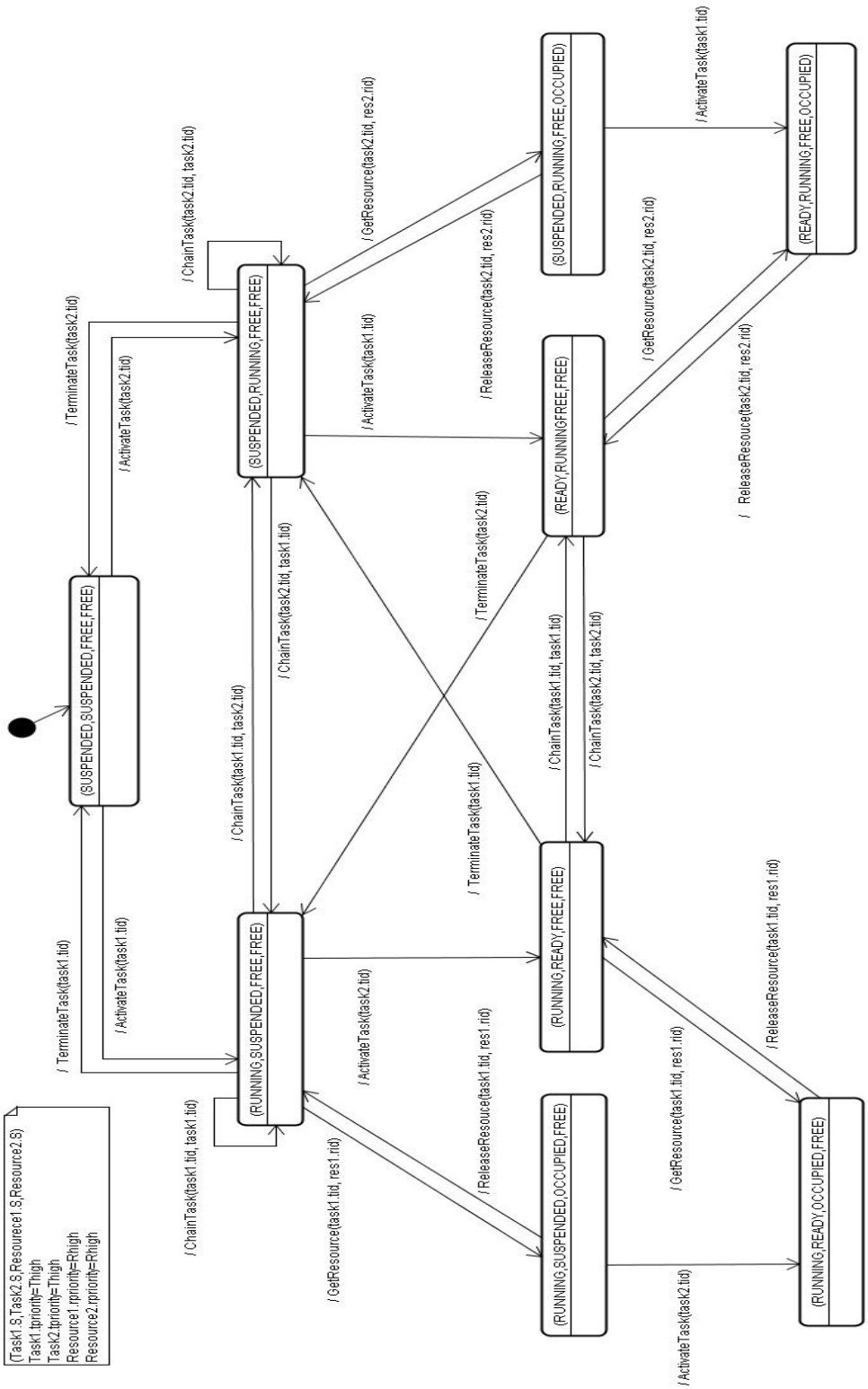


図 B.11: 検査モデル (Task2 つ、Resource2 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

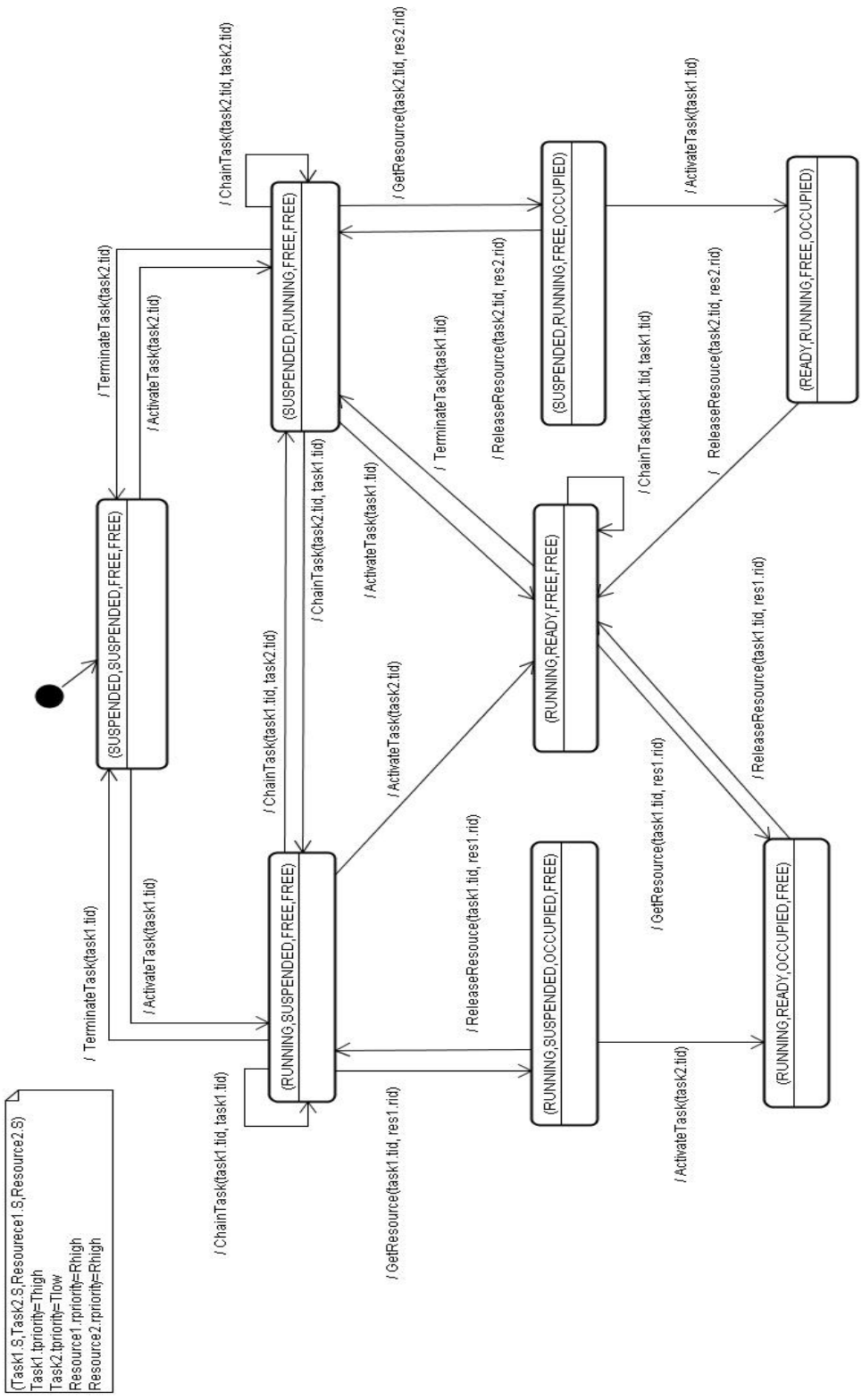


図 B.12: 検査モデル (Task2 つ、Resource2 つ、パターン A、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

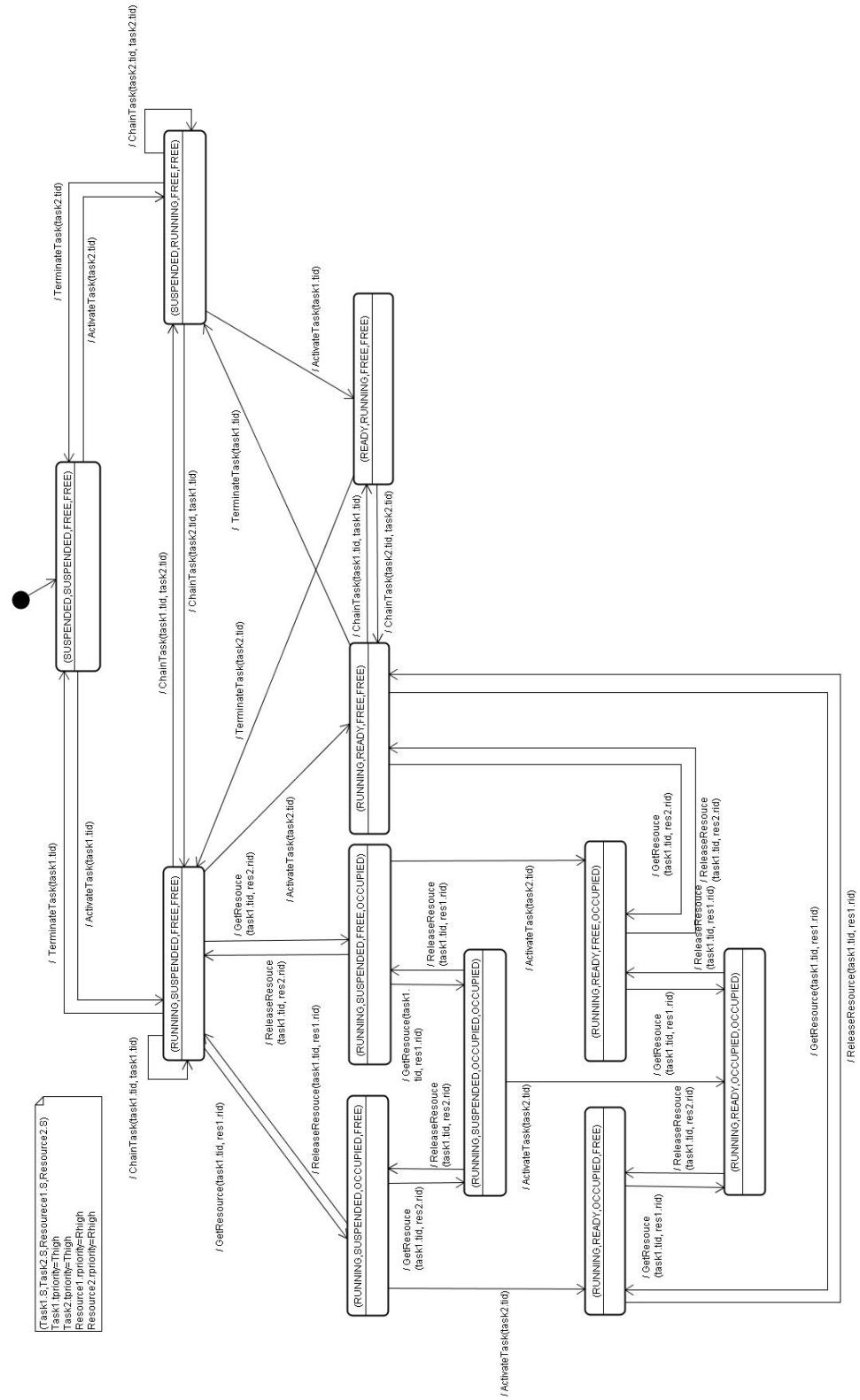


図 B.13: 検査モデル (Task2 つ、Resource2 つ、パターン B、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

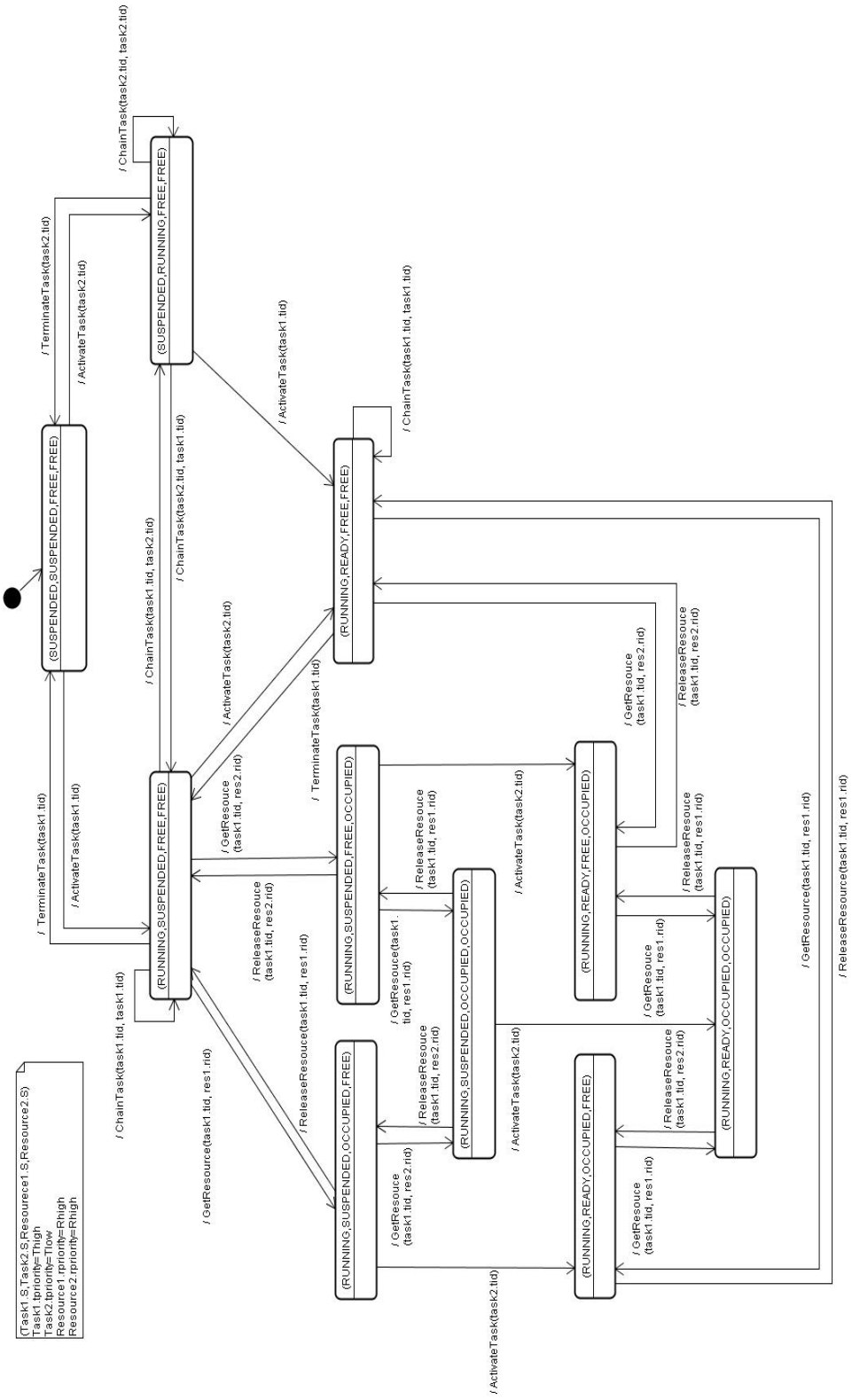


図 B.14: 検査モデル (Task2 つ、Resource2 つ、パターン B、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

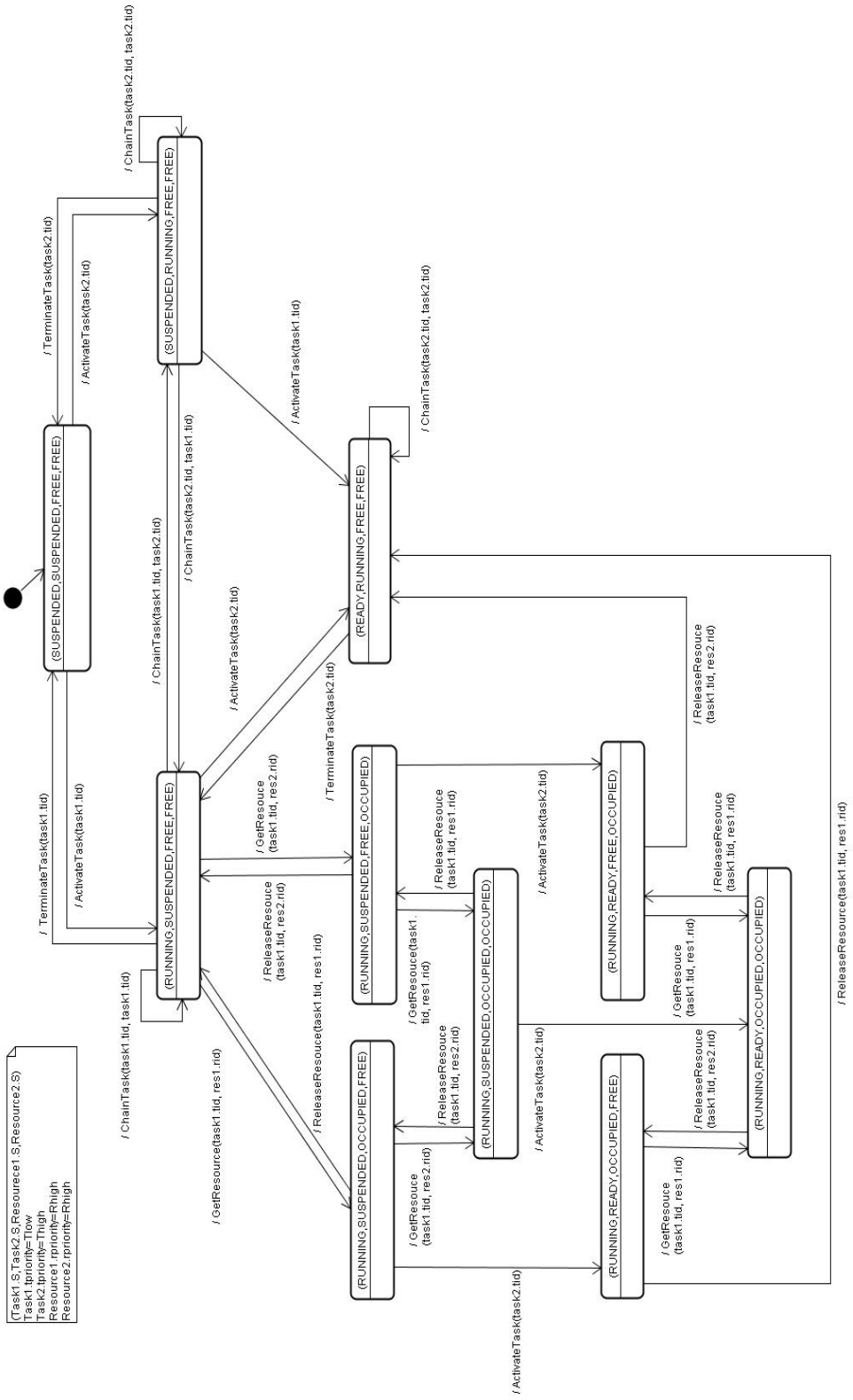


図 B.15: 検査モデル (Task2 つ、Resource2 つ、パターン B、$Task_1.priority = Tlow$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

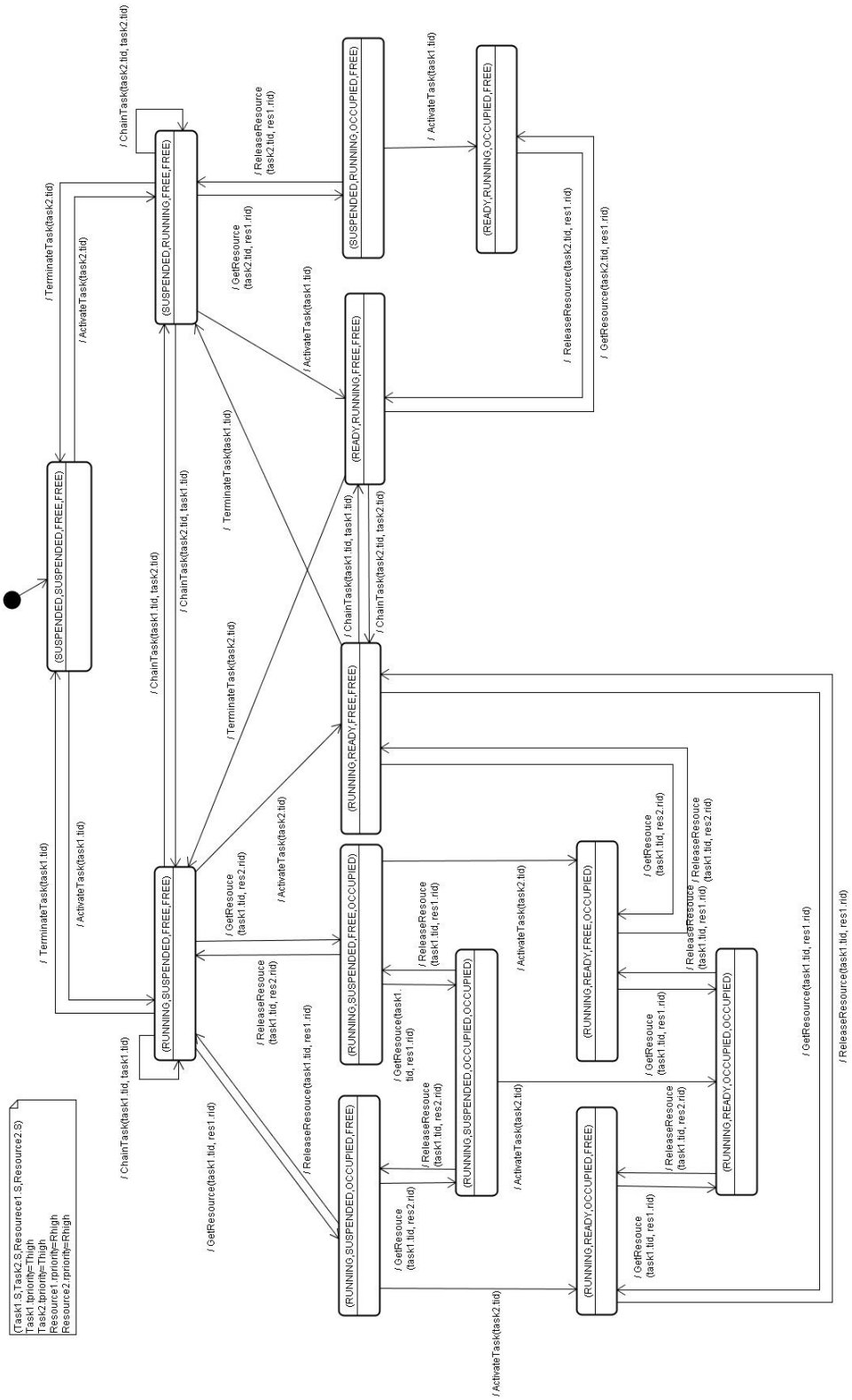


図 B.16: 検査モデル (Task2 つ、Resource2 つ、パターン C、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

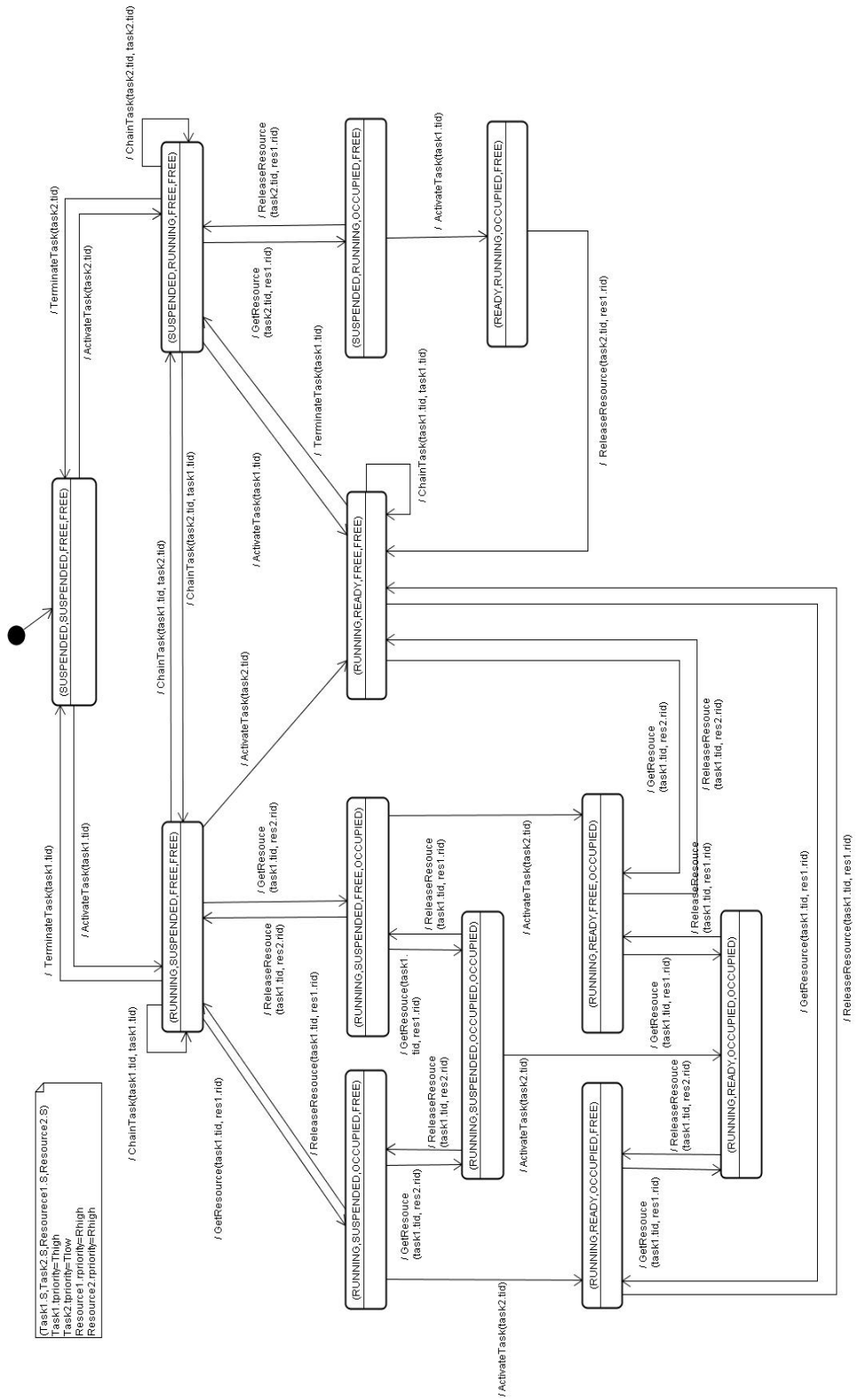


図 B.17: 検査モデル (Task2 つ、Resource2 つ、パターン C、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

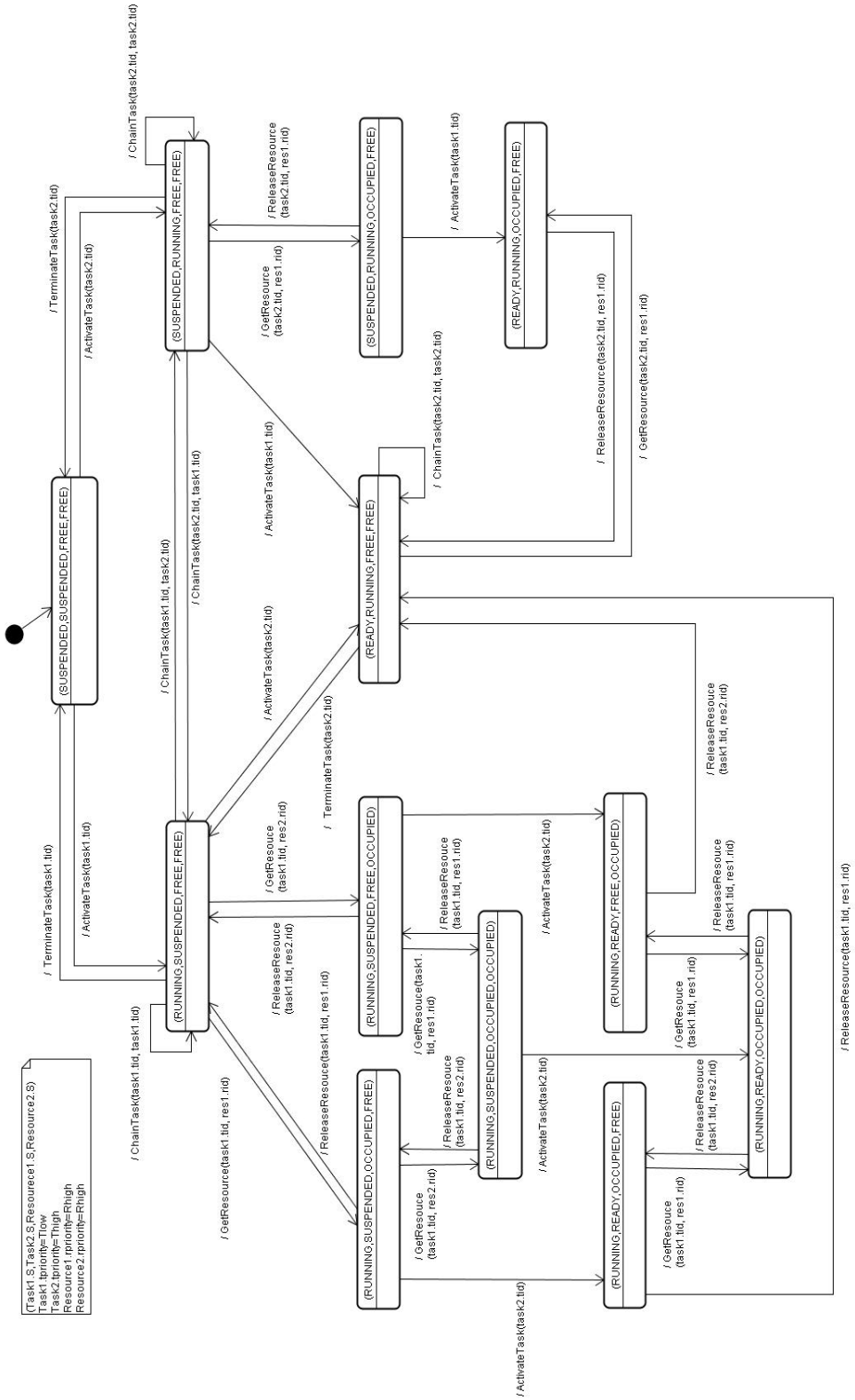


図 B.18: 検査モデル (Task2 つ、Resource2 つ、パターン C、$Task_1.priority = Tlow$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

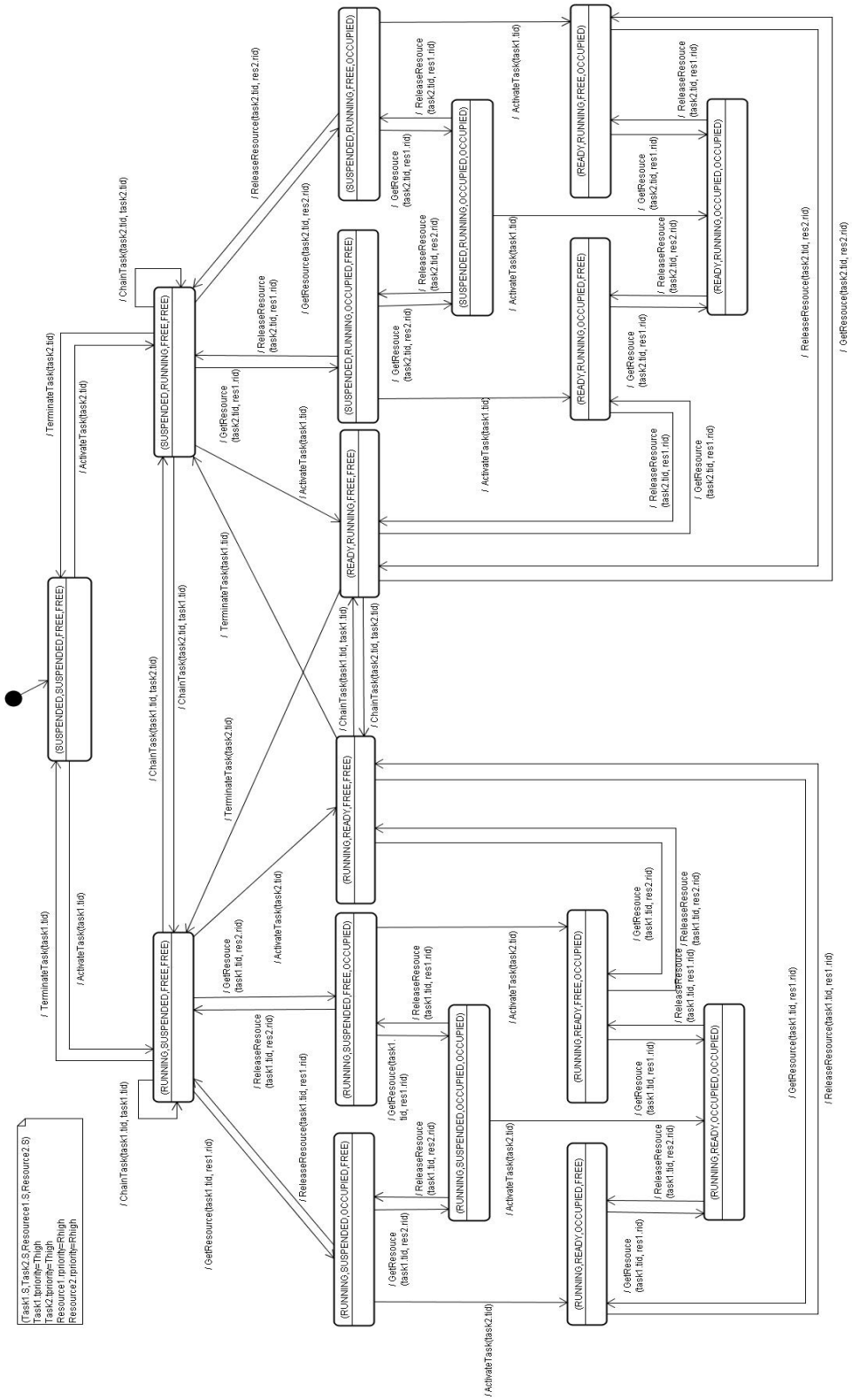


図 B.19: 検査モデル (Task2 つ、Resource2 つ、パターン D、$Task_1.priority = Thigh$、$Task_2.tpriority = Thigh$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)

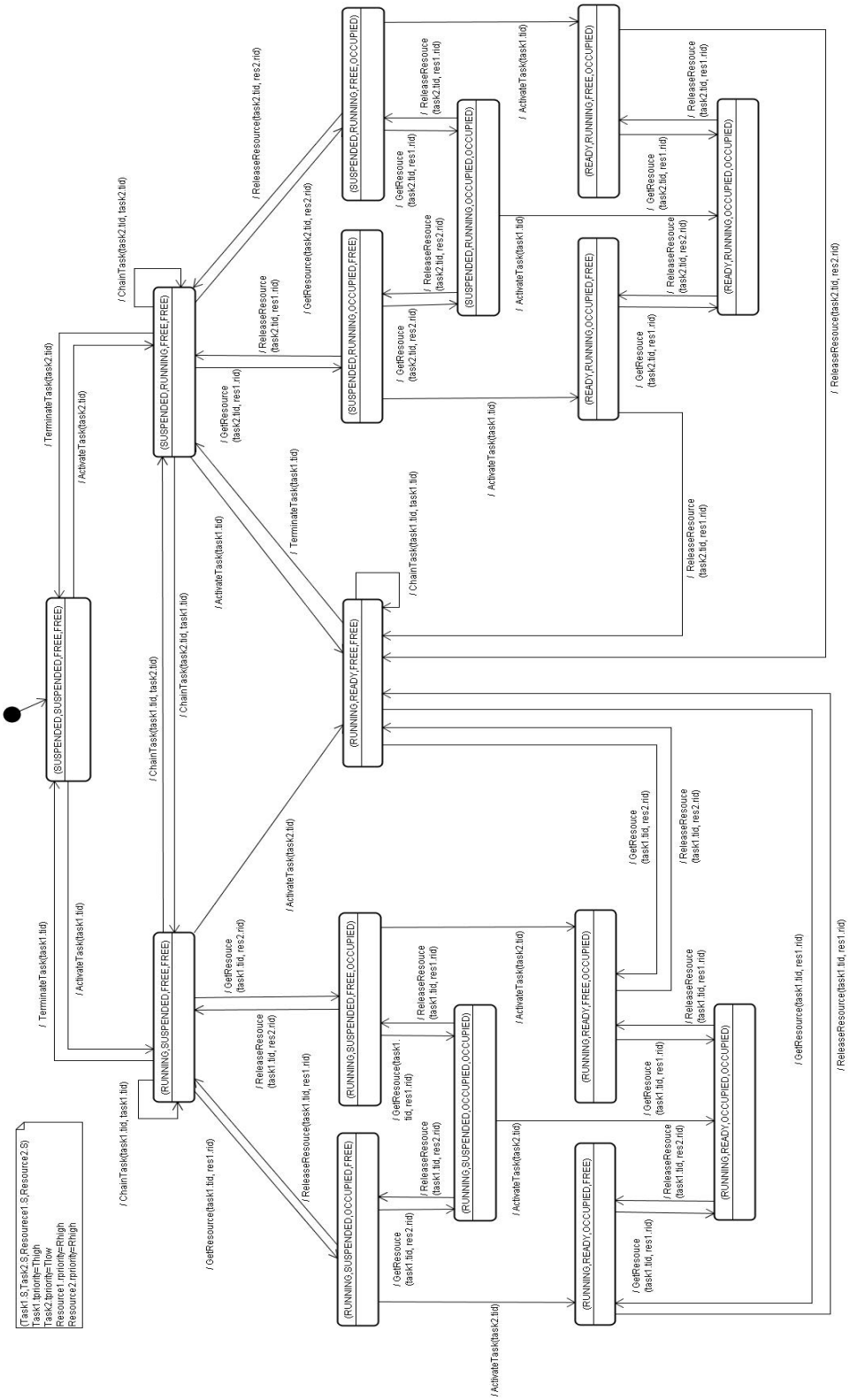


図 B.20: 検査モデル (Task2 つ、Resource2 つ、パターン D、$Task_1.priority = Thigh$、$Task_2.tpriority = Tlow$、$Resource_1.rpriority = Rhigh$、$Resourece_2.rpriority = Rhigh$)