修 士 論 文

設計モデル上でのメモリ量見積もりに関する研究

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

山下 幸之輔

2009 年 3 月

修 士 論 文

設計モデル上でのメモリ量見積もりに関する研究

指導教官　岸知二 特任教授

審査委員主査　岸知二 特任教授
審査委員　落水浩一郎 教授
審査委員　青木利晃 特任准教授

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

710074 山下 幸之輔

提出年月: 2009 年 2 月

概 要

本稿では，タスク設計の段階におけるタスクと割り込みの最大メモリ使用量見積り手法を提案する．本見積もりは，動的な振る舞いまで考慮して見積もることでより正確な見積もりを行うという点に特徴がある．その時点の設計情報を活用して，複数の設計方法から目的にあった設計方法を選択するための指標として活用することなどを想定としている．

目 次

第1章　序論

1.1　研究の背景

組み込みソフトウェアとは，組み込みシステムのためのソフトウェアのことである．組み込みシステムとしては，電子レンジやテレビや携帯電話などが挙げられる．センサーやモーターなどのハードウェアを制御する場合が多く，Linux や Windows などの OS を搭載しているパーソナルコンピュータのソフトウェアに比べてリアルタイム性が求められる場合が多い．製品を出荷したあとのソフトウェアの変更は，パーソナルコンピュータのソフトウェアのようにオンラインアップデートなどはできない場合が多く，変更をする場合はコストがかかる．また，組み込みソフトウェアでは，ハードウェアのリソースの制約が厳しい場合が多い．プロセッサやメモリなどのハードウェアは，パーソナルコンピュータに比べて性能が低い場合が多く，ハードウェアリソースに注意しながらソフトウェアを作成する必要がある．

組み込みソフトウェアにも，パーソナルコンピュータのように高機能な OS ではないが，ITRON 仕様や Linux や Windows CE といった OS が使用されることがほとんどである．IPA/SEC のアンケート調査によると [1]，ITRON 仕様の OS が全体使用比率の 29.4%を占めており，最も使用されている OS である．

近年，組み込みソフトウェアは大規模化が進んでおり，IPA/SEC のアンケート調査 [1] によると，新規開発工数は平均で 25.2 万行にも及んでいる．10 万行や 100 万行といった開発規模になってくると，ソフトウェアの設計はより重要になってくる．IPA/SEC のアンケート調査 [2] によると，実装工程の前の設計工程まででのソフトウェアの不具合率は，全工程の約 50%であり，その不具合の発見工程はソフトウェア実装・単体テストにおいて約 30%である．後々の工程での不具合の発見は手戻りのコストが大きいので望ましくないことである．そのため，ソフトウェアの設計をいかに高品質にするかが重要になってくる．

ソフトウェアの設計においては，機能要求だけでなく，非機能要求も考慮しなければならない．機能要求とは，提供するサービスのことで，例えば,「印刷する」「表示する」「計算する」などのことである．これに対して，非機能要求とは,「性能」「保守性」「資源効率性」など，ソフトウェアの品質に関わる特性に対する要求である．この品質に関する特性は，ISO9126 で定義されている．これに対して，非機能要求とは，ソフトウェアの品質に関わる特性のことで，例えば,「性能」「保守性」「資源効率性」などである．この品質に関する特性は，ISO9126 で定義されている．一般的には，ある機能を実装するときの設計方法は複数考えられ，設計ごとに非機能要求が異なってくる．要求されている非機能要

求を満たす設計方法を選択する必要があり，さらに，要求を満たす設計方法の中でもより良い非機能要求を満たす設計方法を選択する必要がある．

本研究では，設計段階における非機能要求の中のメモリ使用量に注目する．上記でも述べたように，組み込みソフトウェアではハードウェアリソースの制約が厳しい場合が多く，メモリ量においても制約が厳しいため，ソフトウェアの設計時にはメモリ使用量に注意しながら設計を行う必要がある．メモリ使用量の制約を満たす設計方法を選択していないと，後々の工程においてメモリ使用量の制約を満たさなかったがために，設計工程までの手戻りが発生することはコストがかかり望ましくない．そのため，設計段階における設計方法の選択が重要となる．いかに設計段階において同じ機能を実装するとしても非機能要求も完全に満たす設計方法を選択するかが重要となってくる．そこで，どの設計方法を選択すればよいかの指標として活用するために，最大メモリ使用量見積もり方式を提案するとともに，その方式に基づいたツールを作成し，最大メモリ使用量の見積もりを行った．

第2章　関連研究

2.1　メモリ量見積もり技術の概要

メモリ量を見積もる一般的な手段として，ソースコードレベルなどのプログラムに近い段階でアルゴリズムに基づいて見積もるアプローチ，あるいは処理手順や振る舞いを踏まえず静的に見積もる静的な見積もりアプローチなどがある．プログラムに近い段階でのアプローチは，設計段階に比べてより詳細な情報があるため，設計段階での見積もりに比べてより正確に見積もることができる．しかし，プログラムに近い詳細情報を活用するアプローチでは，タスク設計の選択のための指標とはできないという問題がある．静的な見積もりアプローチは，動的な振る舞いまで踏み込んで見積もるわけではないので，ある処理手続きの中での呼び出される関数が存在したり，複数のタスクから同時に呼び出される関数が存在する場合を考慮しないため，見積もり精度があまり良くないという問題がある．動的な振る舞いまで踏み込んで見積もるべきであり，動的な振る舞いまで踏み込んでメモリ量の見積もりが行われた研究について 2.2 で述べる．

2.2　設計段階でのメモリ量見積もりに関する研究

Jens Bek Jorgensen らにより，設計の段階におけるツールを用いてメモリ量を見積もる研究が行われている [3]．ここでは図 2.1 のように，シナリオにおいてアノテーションを用いてメモリ使用量を与える．そして，ツールを用いて図 2.2 のように全ての状態を網羅的に調べて全体の最大メモリ使用量を見積もる．

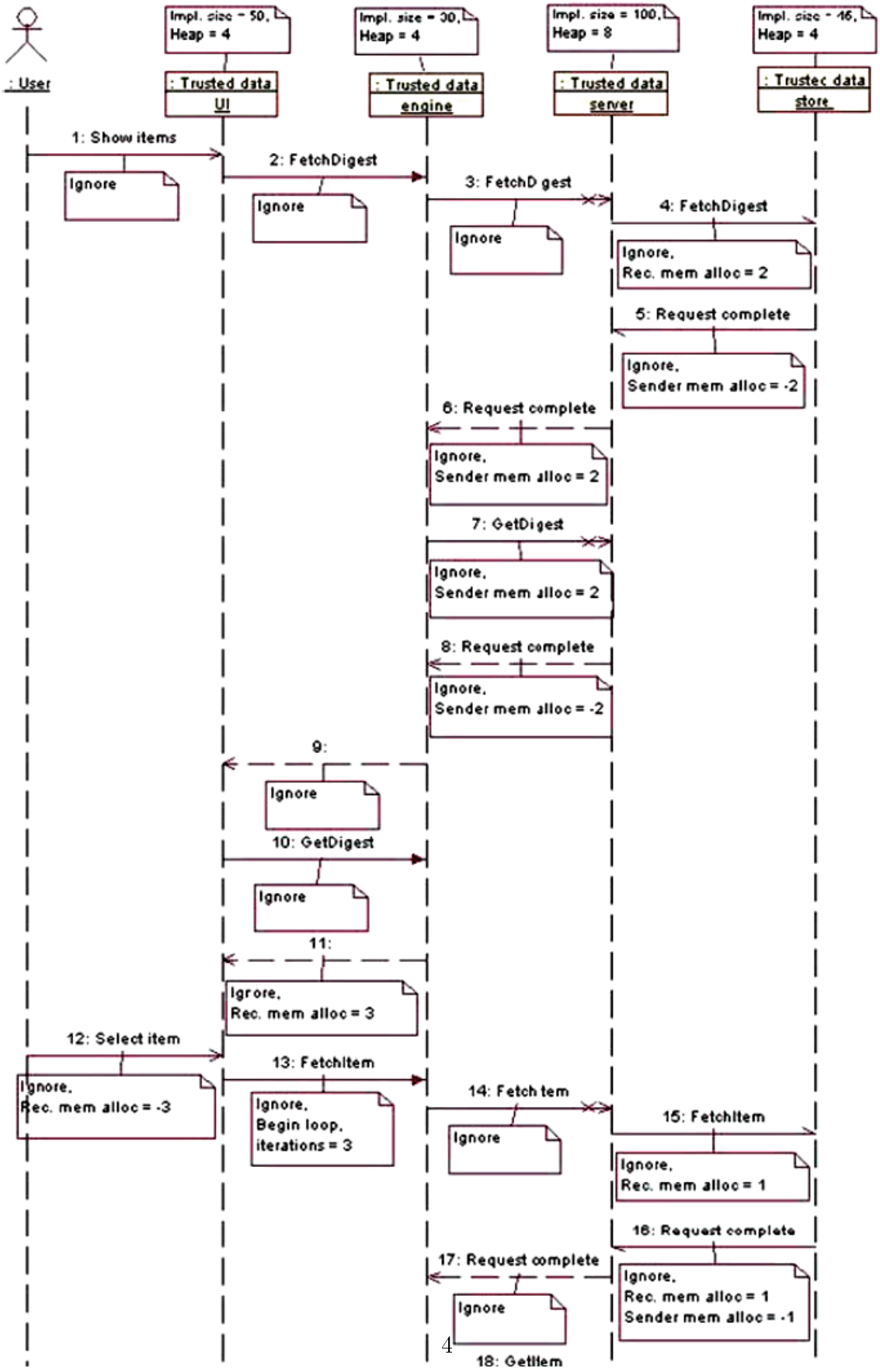

図 2.1: data scenario

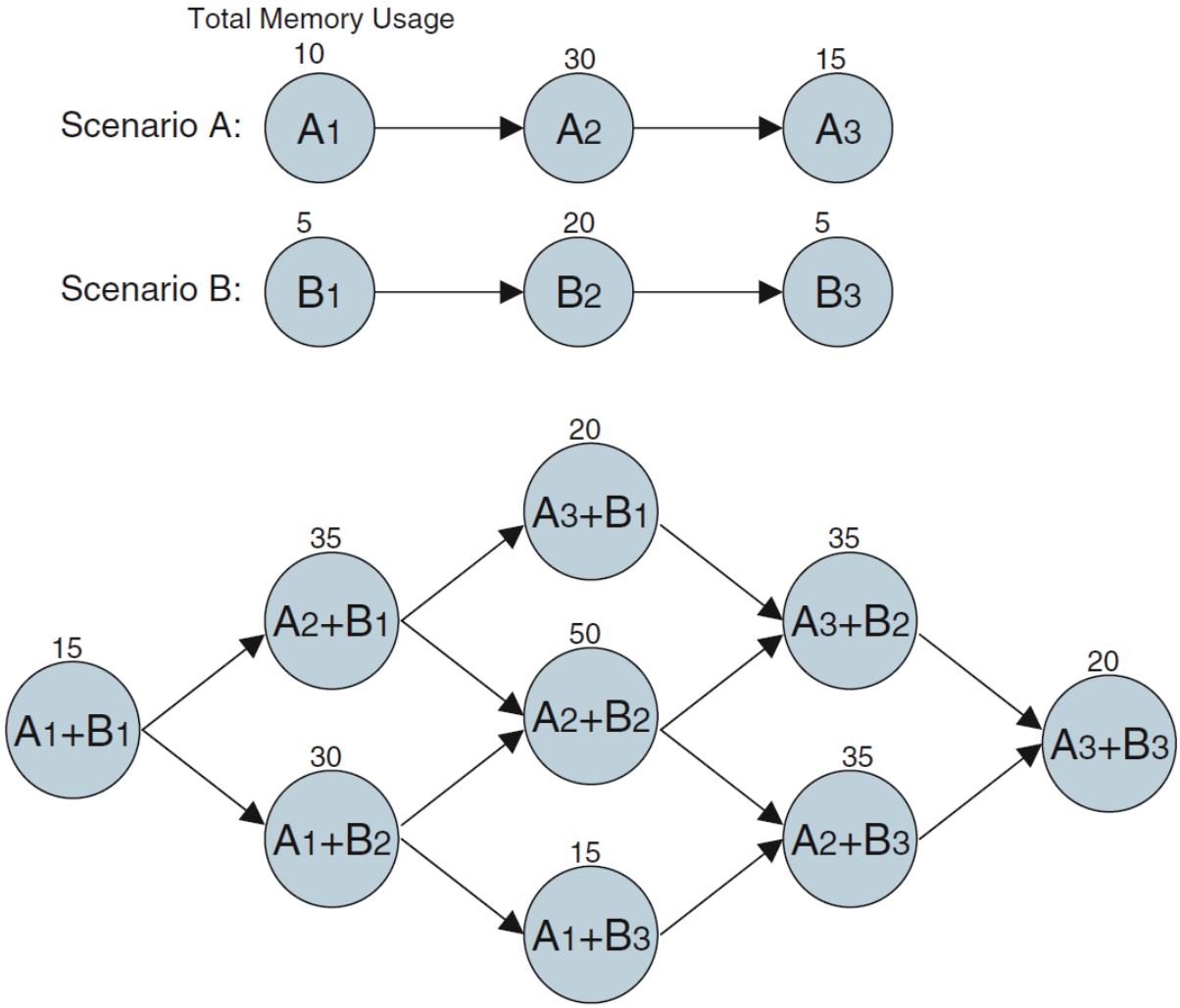

図 2.2: total memory usage

第3章　研究の目的

3.1　目的

設計段階における最大メモリ使用量の見積もり手法の提案を本研究の目的とする．特に，タスク設計の段階で，タスク分割などに関わる設計判断に活用することを狙っている．設計判断の段階で，その時点での振る舞い情報を踏まえることで静的見積もりよりも精度の高い見積もりができると考える．なお，見積もる対象は，タスクと割り込みの最大メモリ使用量とする．

3.2　解決したい問題点

1章で述べたように，組み込みソフトウェアでは，ハードウェアのリソースの制約が厳しい場合が多く，ハードウェアリソースに注意しながらソフトウェアを作成する必要がある．そのため，設計段階においてメモリ使用量を見積もることで，ハードウェアの制約を満たしているかどうかを調べる必要がある．

組み込みソフトウェアの大規模化にともない，設計段階における不具合が問題になってくる．設計以降の後々の工程において，設計段階での不具合による手戻りの発生は，非常にコストがかかる．そのため，設計以降にわかるのではなく，設計段階で一定の見通しを持つことが重要である．

メモリ使用量は，静的に見積もると，ある処理手続きの中での呼び出される関数が存在したり，複数のタスクから同時に呼び出される関数が存在する場合を考慮しないため，メモリ見積もり量は必要以上に大きくなる．特にタスクや割り込みは，設計者が明示的にメモリを利用するわけではないので適切に見積もらなければならず，これはタスク分割などの設計において大きく影響を受けてしまう．そのため，静的ではなく動的な振る舞いを見ることにより，呼び出されない関数や同時に呼ばれる関数までも考慮してメモリ使用量を見積もることがより有用であると考えた．

2章で述べた過去の研究では，ツールに与える入力はシナリオで与えていた．シナリオは，システム全体の振る舞いの一部であるため，システム全体のメモリ使用量は見積もることができないという問題がある．そのため，システム全体の振る舞いをステートマシン図で表し，そのステートマシン図に基づいてメモリ量を見積もれば，システム全体のメモリ使用量が見積もれると考えた．

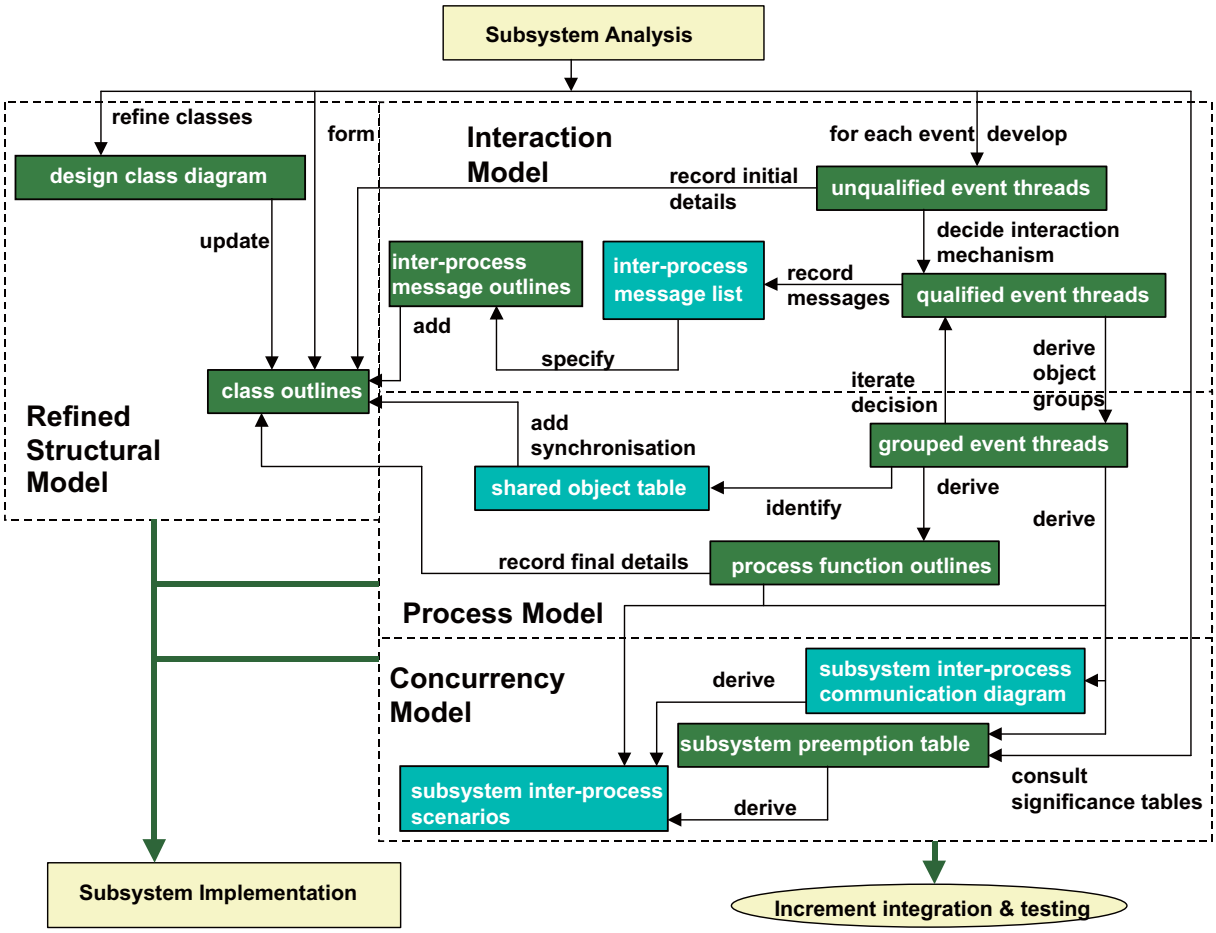

図 3.1: サブシステム設計の流れ

本研究が想定してる設計判断の段階を，具体的な例として，組み込み開発手法の典型的な OCTOPUS[4] で示す．OCTOPUS の開発は以下の流れで行う．

1. システム要求仕様
2. システムアーキテクチャ
3. サブシステム開発
    a. サイブシステム分析
    b. サブシステム設計
    c. サブシステム実装
4. サブシステムプログラム
5. システムプログラム

システム要求仕様において要求仕様を決定した後に，システムアーキテクチャでドメイン分割を行い，サブシステム分析ではクラス，ステートチャート，イベントリスト，アクションテーブルを作成する．サブシステム設計においては，共有オブジェクト表，メッセージフローごとの実相メカニズム (同期/非同期) の決定，プロセスのプライオリティオーダー

の決定を行い，サブシステムごとに実装を行いサブシステムプログラムを作成した後に，システムプログラムを作成する．

このとき，図 3.1 に示したサブシステム設計において，非限定イベントスレッド (unqualified event threads) から限定イベントスレッド (qualified event threads) を考える段階がある．非限定イベントスレッドの段階では，メッセージの同期・非同期やどれが並行動作単位かを考慮せずにシステムの基本的な振る舞いを考える．そして，限定イベントスレッドの段階で，メッセージの同期・非同期の決定 (add synchronisation) や並行動作単位 (derive object groups) の決定を行う．この限定イベントスレッドの段階において，どのメッセージを同期にするか非同期にするかなど複数な設計方法が考えられる．1.1 でも述べたように，ある機能を持ったソフトウェアの設計をする場合に，一般的には複数の設計方法が考えられ，設計ごとに非機能特性が異なる．そのため，どの設計を選択すればよいかという問題がある．このとき，非機能特性を見積もることで設計方法選択のための指標にすることができると考えた．また，どれが並行動作単位であるかを考える段階では，ステートマシン図などで基本的な振る舞いが決まっているので，そうした情報を活用・流用しながらメモリ量が見積もれるのではないかと考えた．

3.3 アプローチ

システムの構造を記述したクラス図とシステム全体の振る舞いを表したステートマシン図をタスク設計モデルとして与え，そのタスク設計モデルから網羅的に動的な振る舞いを調べ，最大メモリ使用量を見積もる．タスク設計モデルは，タスク設計段階で，その時点で通常定義されている情報をできるだけ活用しながら記述し，それにあたっては組み込み向け UML プロファイルである MARTE[5] を拡張したものを用いる．また，今回は，具体的な検討は μITRON4.0 を使用したソフトウェアを対象に行った．

第4章　最大メモリ使用量積もり方式の提案

4.1　最大メモリ使用量

本稿では，タスクと割り込みに関する最大メモリ使用量を対象とする．リアルタイムOSによって差異があるが，最大メモリ使用量は一般的には以下の式で表されると考える．

$$MM = \sum_{it=0}^{nt} TM_{it} + \sum_{ii=0}^{ni} IM_{ii} + \sum_{ir=0}^{nr} RM_{ir} + OM$$

$$TM = TCM + \sum_{i=0}^{m} FM_i$$

$$IM = ICM + \sum_{j=0}^{n} FM_i$$

MM：最大メモリ使用量
TM：タスクのメモリ使用量
IM：割り込みのメモリ使用量
RM：OSリソースのメモリ使用量
OM：それ以外のメモリ使用量
TCM：タスクコンテキストのメモリ使用量
ICM：割り込みコンテキストのメモリ使用量
FM：関数のメモリ使用量

本研究における見積もり対象はタスクと割り込みであり，上記の式においていうと$\sum_{it=0}^{nt} TM_{it}$と$\sum_{ii=0}^{ni} IM_{ii}$である．

TMを表す式中で，$\sum_{i=0}^{m} FM_i$は関数のネストによって使用されるメモリ使用量を意味している．TCMは，タスクが1つ生成されたときに使用される静的なメモリ使用量を意味している．

IMを表す式中で，$\sum_{i=0}^{m} FM_i$はTMのときと同様に関数のネストによって使用されるメモリ使用量を意味している．ICMは，割り込みが1つ使用されるごとに使用される静的なメモリ使用量を意味している．

4.2 μITRON4.0の特徴

今回はμITRON4.0を対象として具体的に検討した．基本的には4.1で述べた式で捉えられるが，以下に対する考慮が必要となる．
(1) タスクスタックと非タスクスタックがある．
(2) スケージュラがFCFS(First Come First Service)方式である．

(1)について，名前の通りタスクが使用するスタックがタスクスタック，タスク以外の割り込みなどが使用するスタックが非タスクスタックである．タスクスタックはタスクごとにそれぞれ割り当てられ，割り込みは非タスクスタックを供用して使用する．4.1で述べた式のTMとIMにあたる個所であり，見積もりの際には注意が必要である．つまり，タスクスタックはタスクが複数存在すれば同じだけ複数存在するが，割り込みスタックは割り込みが複数しても必ず1つしか存在しないということである．

(2)について，FCFS方式とは，最初に実行待ち状態に入ったタスクから実行する方式のことである．実行待ちのタスクは，先に待ち状態に入ったタスク順に待ち行列をつくる．そのため，振る舞いを検討する際にはこのスケジュール方式に従わなければならない．

μITRON4.0においては，タスクはそれぞれのタスクに個々に割り当てられたタスクスタックを使用し，割り込みは非タスクスタックを供用して使用する．つまり，4.1で述べた式におけるTMはタスク個々のタスクスタックの最大メモリ使用量，$\sum_{it=0}^{nt} TM_{it}$は全てのタスクスタックの合計の最大メモリ使用量，$\sum_{ii=0}^{ni} IM_{ii}$は全ての割り込みの非タスクスタックの最大メモリ使用量を表することになる．

4.3 最大メモリ使用量見積もり方式

4.3.1 提案する見積もり方式

今回の見積もりでは，4.1で述べた式の$\sum_{it=0}^{nt} TM_{it}$と$\sum_{ii=0}^{ni} IM_{ii}$を対象としている．一般的には，TCMとICMは使用するOSを決定すれば静的に決まるメモリ量であり，関数のネストのメモリ使用量である$\sum_{i=0}^{m} FM_i$が動的な振る舞いを見ないとわからないものである．この関数のネストのメモリ使用量を見積もる方式を以下に示す．

関数$f_1, f_2, ..., f_i (i >= 1)$，スレッド$thread_1, thread_2, ..., thread_j (j >= 1)$があるとする．関数それぞれが使用するメモリ使用量は$fm_1, fm_2, ..., fm_i$とし，それは設計者が与えるものとする．スレッドそれぞれに，現在使用しているメモリ使用量の値を保持するための変数$curmem_1, curmem_2, ..., curmem_j$を，現在までの最大メモリ使用量の値を保持するための変数$maxmem_1, maxmem_2, ..., maxmem_j$を置く．このとき，関数呼び出しを全てのスレッドを考慮した動的な振る舞いで網羅的に調べ，以下の(1)〜(3)の操作を行い，結果として得られる$maxmem_1, maxmem_2, ..., maxmem_j$がスレッドごとの最大メモリ使用

量となる．
(1) $thread_m$ において関数 f_n の呼び出しが行われた場合，
$curmem_m = curmem_m + fm_n$
(2) $thread_m$ において関数 f_n の実行が終了した場合，
$curmem_m = curmem_m - fm_n$
(3) $curmem_m$ の値の更新が行われたときに，$curmem_m > maxmem_m$ であったとき，
$maxmem_m = curmem_m$

今回対象とする μITRON4.0 においては，タスクと割り込みがスレッドに当たるものである．そのため，見積もりの際は，4.2 で述べたように，μITRON4.0 におけるスケジューリング方式はFCFS 方式であため，スレッドの実行順序や停止などを考慮しなければならないことに注意する必要がある．

4.3.2 見積もり方式の具体例

見積もり例を図 4.1 に示す．task1 のタスク関数のメモリ使用量が 10，func2 のメモリ使用量が 20，func3 のメモリ使用量が 30 であり，curmem を現在使用しているメモリ使用量を保持するための変数，maxmem を現在までの最大メモリ使用量を保持するための変数とする．このときの見積もりの流れを以下に示す．

1. func2 が呼び出されると，curmem に func2 のメモリ使用量 20 を追加し curmem=30 とし，curmem が maxmem よりも大きいので maxmem に curmem の値を代入し maxmem=30 となる．
2. func2 の実行が終了すると，curmem から func2 のメモリ使用量 20 をマイナスして curmem=10 とし，この場合は curmem が maxmem よりも大きくないので maxmem=30 のままとする．
3. func3 が呼び出されると，curmem に fucn3 のメモリ使用量 10 を追加し curmem=20 とし，この場合は curmem が maxmem よりも大きくないので maxmem=30 のままとする．
4. func3 の実行が終了すると，curmem から func3 のメモリ使用量 10 をマイナスし curmem=10 とし，この場合は curmem が maxmem よりも大きくないので maxmem=30 のままとする．
5. 最大メモリ使用量は maxmem=30 と求まる．

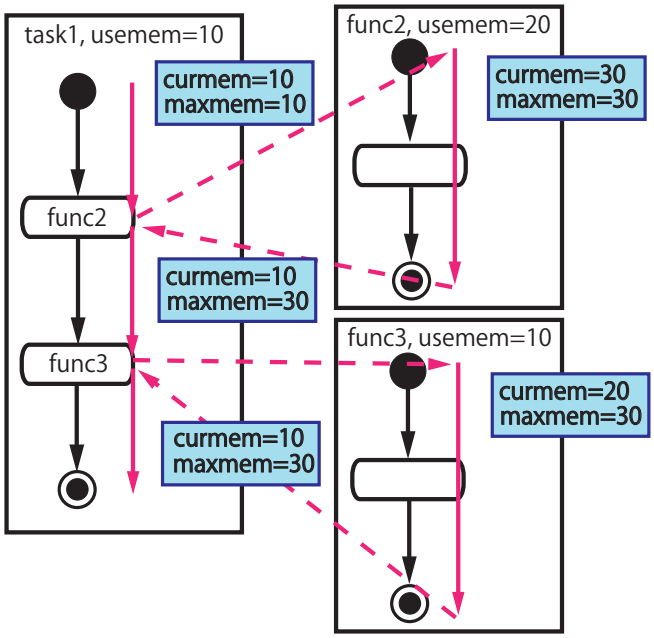

図 4.1: 見積もり例

4.4 見積もり手順の全体像

上記の見積もり方式を利用して，μITRON4.0 を対象とした見積もり手順を提案する．見積もり手順の全体像を図 4.2 に示す．

①では，システムの設計情報から見積もりに必要な情報を抽出・追記を行い，タスク設計モデルを作成する．タスク設計モデルについては 4.5 で述べる．次に，②では，タスク設計モデルに基づいて，メモリ見積もり方式を実行するためのマッピングルールから promela コードを記述したファイルを生成する．網羅的に調べる手段としてモデル検査技術を用いた．このことについては 5 章で述べる．また，マッピングルールについても 5 章で述べる．そして，③では，②で生成した promela ファイルを SPIN で検証実行して最大メモリ使用量を算出する．

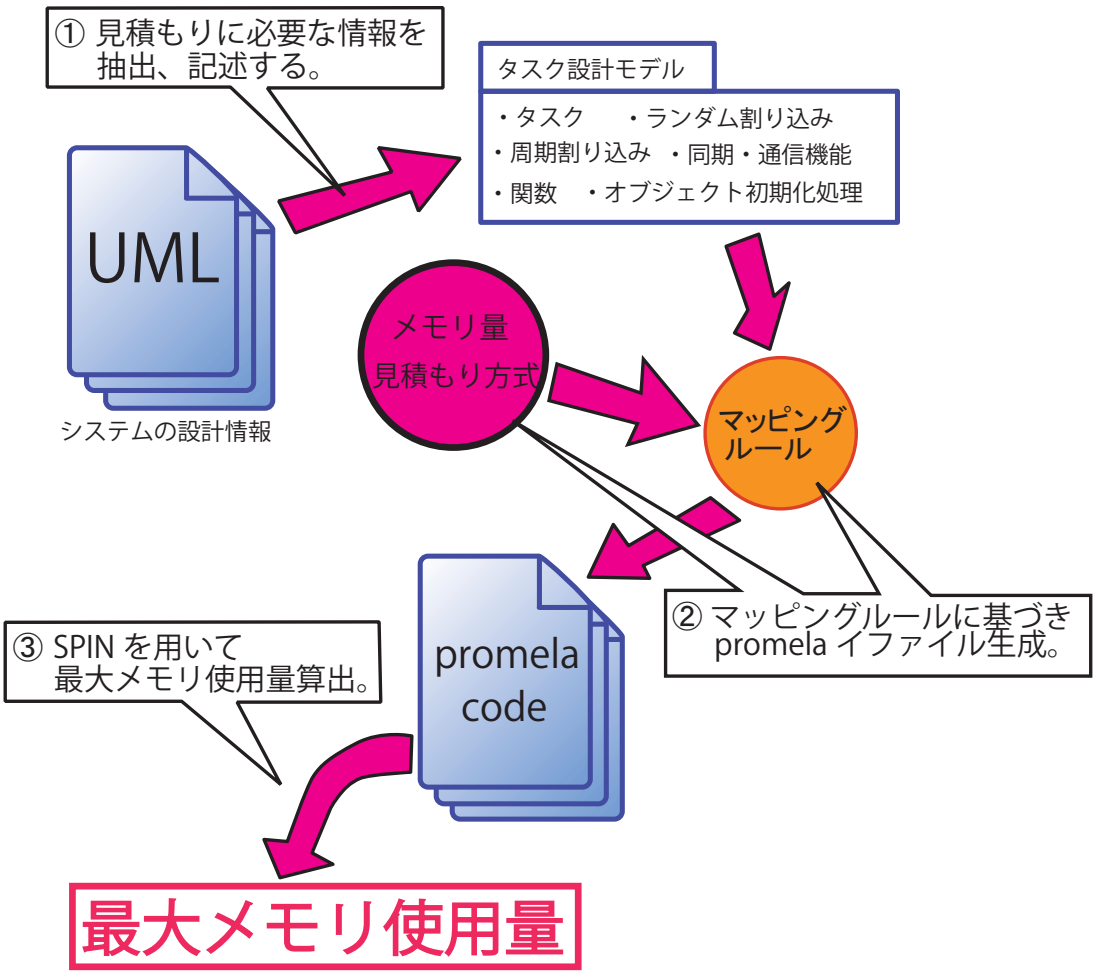

図 4.2: 見積もり手順の全体像

4.5　タスク設計モデル

見積もりに必要な情報を記述するためのモデルとしてタスク設計モデルを以下に定義する.

4.5.1　タスク設計モデルの位置づけ

メッセージの同期・非同期やどれが並行動作単位かを考慮するときを想定としている. 具体的にいうと，3 章で述べたような OCTOPUS における限定イベントスレッドに移った初期の段階を想定として考えている.

4.5.2　見積もりに必要な情報

見積もりに必要となる情報を以下に示す.
(1) タスク
(2) 同期・通信機能
　(a) セマフォ
　(b) ミューテックス
　(c) イベントフラグ
　(d) ランデブ
　(e) データキュー
　(f) メールボックス
　(g) メッセージバッファ
(3) ランダム割り込み
(4) 周期割り込み
(5) 関数
(6) オブジェクト初期化処理

これらそれぞれに μITRON4.0 の仕様書 [6] に記述されている OS のサービスを使用するために必要となるパラメータを与える必要がある．以下に与える必要があるパラメータを示す.
(1) タスク
タスクに関する情報は，タスク ID，優先度，タスクコンテキストサイズ，タスク関数のメモリ使用量.

(2.a) セマフォ
セマフォに関する情報は，セマフォID，資源数の初期値，最大資源数，待ち行列のポリシー.

(2.b) ミューテックス
ミューテックスに関する情報は，ミューテックス ID，待ち行列のポリシー，上限優先度.

(2.c) イベントフラグ
イベントフラグに関する情報は，イベントフラグ ID，待ち行列のポリシー，ビットパターンの初期値，待ち解除時にビットパターンクリアを行うかどうか.

(2.d) ランデブ
ランデブに関する情報は，ランデブ ID，呼び出し待ち行列のポリシー，呼び出しメッセージの最大サイズ，返答メッセージの最大サイズ.

(2.e) データキュー
データキューに関する情報は，データキュー ID，送信待ち行列のポリシー，データの個数.

(2.f) メールボックス
メールボックスに関する情報は，メールボックス ID，待ち行列のポリシー，メッセージキューのポリシー，メッセージの優先度の最大値.

(2.g) メッセージバッファ
メッセージバッファに関する情報は，メッセージバッファID，送信待ち行列のポリシー，メッセージの最大サイズ (バイト数)，メッセージバッファ領域のサイズ (バイト数).

(3) ランダム割り込み
ランダム割り込みに関する情報は，割り込みハンドラ ID，優先度，割り込みコンテキストサイズ，割り込み関数のメモリ使用量.

(4) 周期割り込み
周期割り込みに関する情報は，周期割り込み ID，優先度，割り込みコンテキストサイズ，割り込み関数のメモリ使用量，周期，位相.

(5) 関数
関数に関する情報は，メモリ使用量.

(6) オブジェクト初期化処理
オブジェクト初期化処理に関する情報は，オブジェクトの初期化情報.

4.5.3 タスク設計モデルの記法

タスクや割り込みなどのオブジェクトの構造の情報をクラス図で記述する．それぞれのオブジェクトにはステレオタイプとタグ付き値を与えることにより 4.5.2 で述べたような見積もりに必要な情報を記述する．ステレオタイプは組み込み向け UML プロファイルである MARTE[5] を拡張したものを与える．例えば，タスクであれば図 4.3 のように，タスクであるので SwSchedulableResource というステレオタイプ与え，アノテーションでタグ値を記述する．identifierElements や priorityElements などといった属性の意味は後述する．

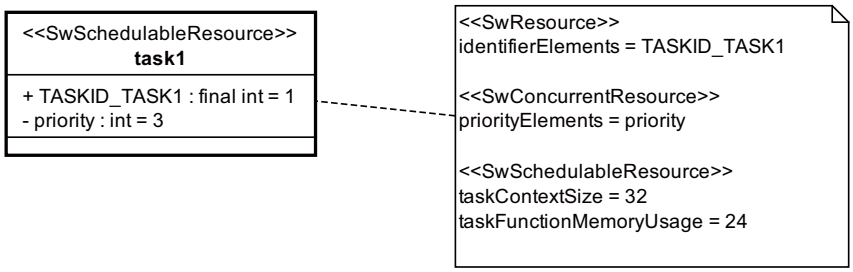

図 4.3: タスクに関する記法の例

タスク，ランダム割り込み，周期割り込み，関数，オブジェクト初期化処理の振る舞いに関する情報をステートマシン図で記述する．方針として，関数呼び出しに関わってくる事柄の振る舞いだけを書けば良いので，余計なものであると判断できる場合はそれを記述する必要はない．

4.5.4 スレテオタイプ一覧

以下が記述するオブジェクトに対応するステレオタイプ一覧である．

- タスク　SwSchedulableResource
- セマフォ　SwMutualExclusionResource
- ミューテックス　SwMutualExclusionResource
- イベントフラグ　NotificationResource
- ランデブ　NotificationResource
- データキュー　MessageComResouce
- メールボックス　MessageComResouce
- メッセージバッファ　MessageComResouce
- ランダム割り込み　InterruptResource
- 周期割り込み　Alarm
- 関数　Function

4.5.5 タスク設計モデルのメタモデル

MARTE を拡張して μITRON4.0 に対応させたプロファイルのメタモデルの構造を表したものをタスク，割り込み，関数に関連する部分を図 4.4 に，同期・通信に関連する部分を図 4.5 に示す．こららのステレオタイプやタグ付き値のうち，MARTE に対して本研究で追加したものは以下のとおりである．

・SwSchedulableResource::taskContextSize
・SwSchedulableResource::taskFunctionMemoryUsage
・NotificationResource::isClear
・NotificationResourceKind::rendezvous
・MessageComResourceKind::DataQueue
・MessageComResourceKind::MailBox
・MessageComResourceKind::MessageBuffer
・MessageComResource::messageQueueMaxPriority
・InterruptResource::interruptContextSize
・InterruptResource::interruptFunctionMemoryUsage
・Alarm::cycle
・Alarm::phase
・Function::memoryUsage
・Function::arguments

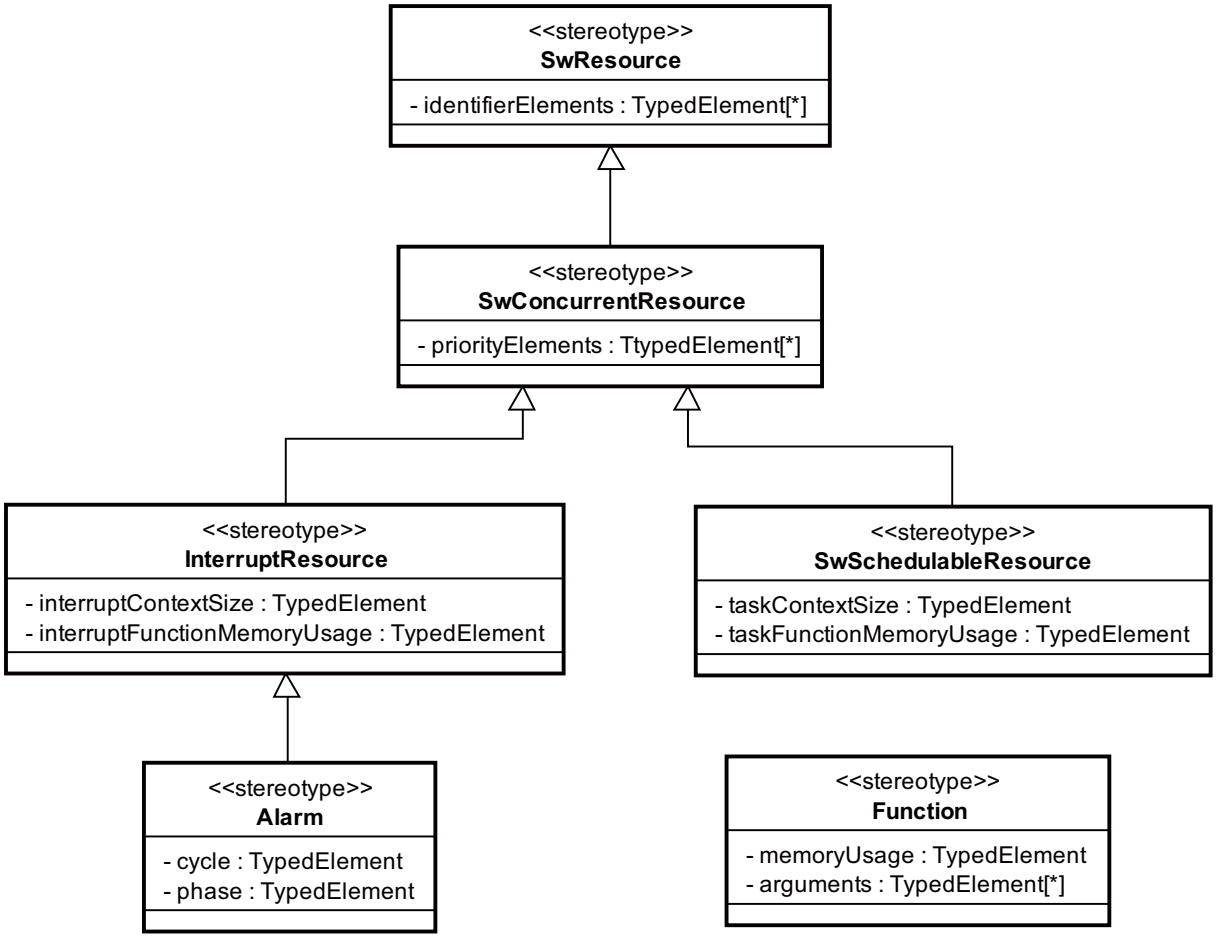

図 4.4: タスク設計モデルのメタモデル (タスク，割り込み，関数関連部分)

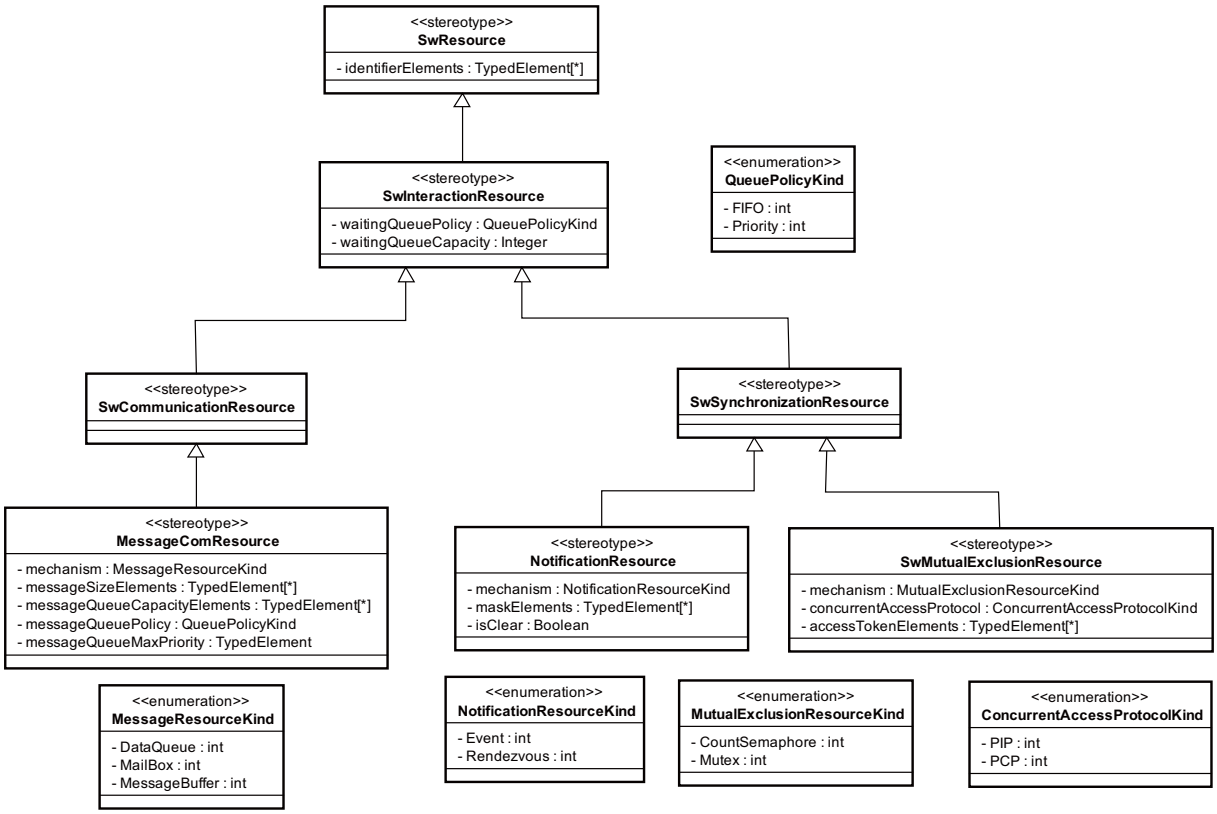

図 4.5: タスク設計モデルのメタモデル (同期・通信関連部分)

4.5.6　個々のオブジェクトに関するプロファイルと記述例

4.5.5で示したメタモデルから，タスク，セマフォ，ミューテックス，イベントフラグ，ランデブ，データキュー，メールボックス，メッセージバッファ，ランダム割り込み，周期割り込み，関数に関するそれぞれのプロファイルとそれを用いた記述例を示す．

タスク

タスクに関するプロファイルを図 4.6 に示す．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	タスク ID
SwConcurrentResource::priorityElements	優先度
SwSchedulableResource::taskContextSize	タスクコンテキスストサイズ
SwSchedulableResource::taskFunctionMemoryUsage	タスク関数のメモリ使用量

タスクの構造を記述した例を図 4.7 に示す．

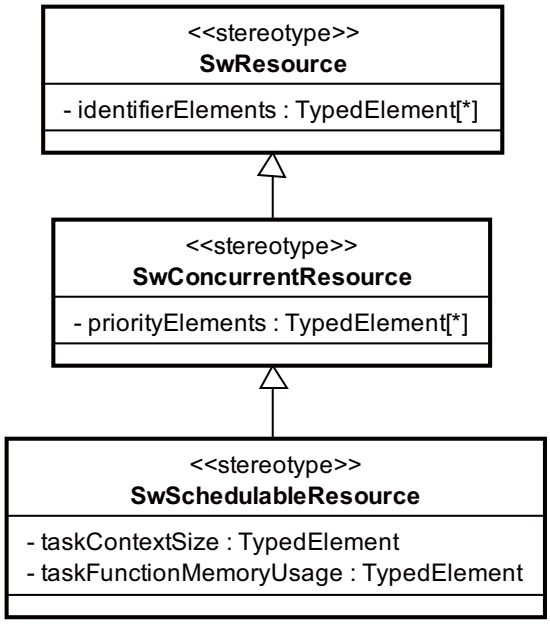

図 4.6: タスクに関するプロファイル

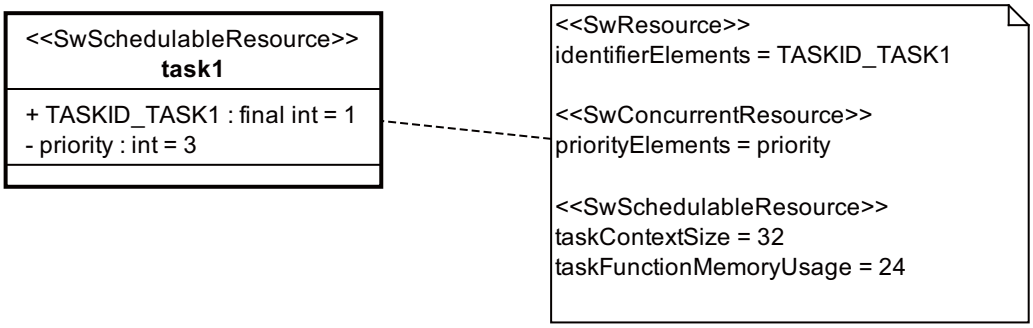

図 4.7: タスクの記述例

セマフォ

セマフォに関するプロファイルを図4.8に示す．セマフォであることを示すために，SwMutualExclusionResource::mechanism には CountSemaphore と記述する．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	セマフォID
SwMutualExclusionResource::accessTokenElements[0]	資源数の初期値
SwMutualExclusionResource::accessTokenElements[1]	最大資源数

セマフォの構造を記述した例を図 4.9 に示す．

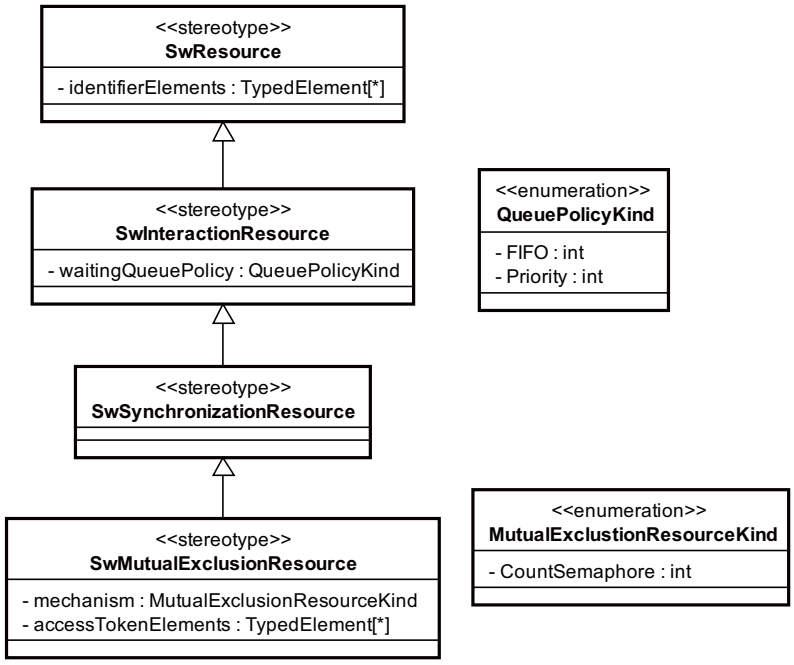

図 4.8: セマフォに関するプロファイル

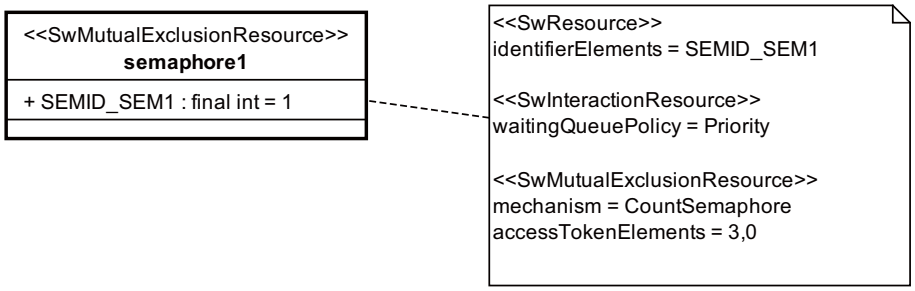

図 4.9: セマフォの記述例

ミューテックス

ミューテックスに関するプロファイルを図 4.10 に示す．ミューテックスであることを示すために，SwMutualExclusionResource::mechanism には Mutex と記述する．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	ミューテックス ID
SwInteractionResource::waitingQueuePolicy	待ち行列のポリシー
SwMutualExclusionResource::concurrentAccessProtocol	上限優先度
SwMutualExclusionResource::accessTokenElements	上限優先度

ミューテックスの構造を記述した例を図 4.11 に示す．

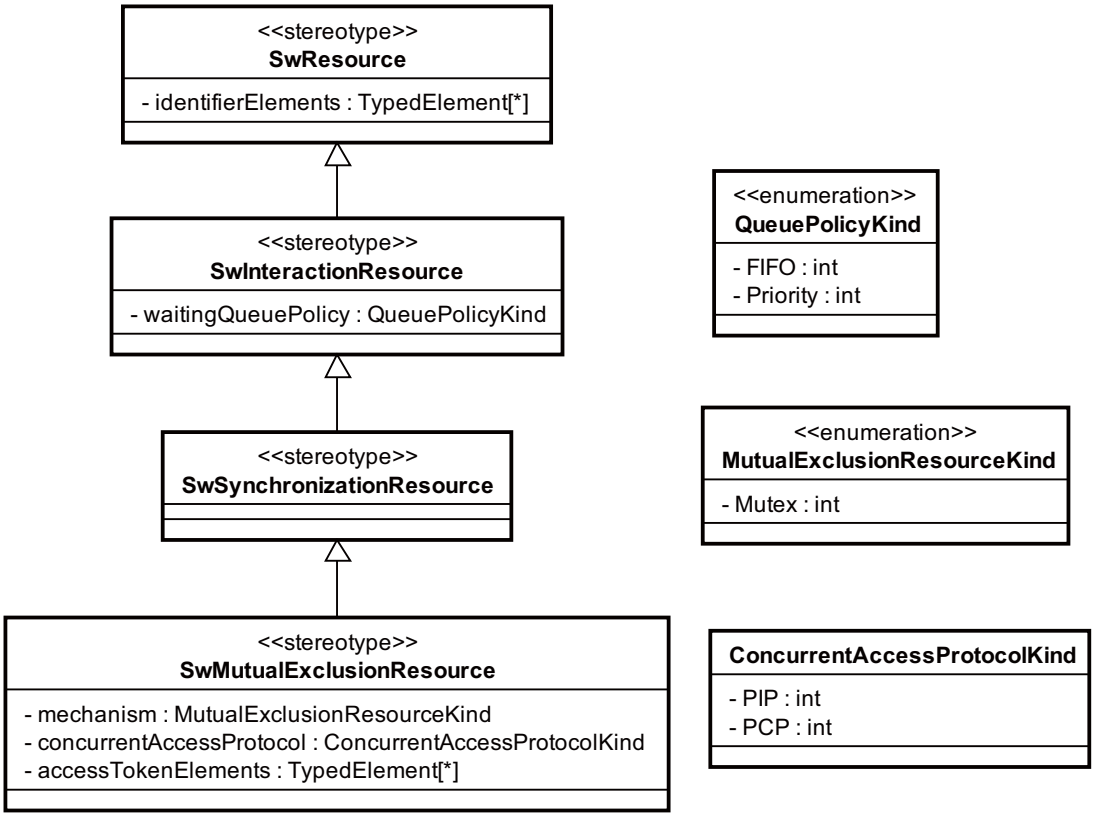

図 4.10: ミューテックスに関するプロファイル

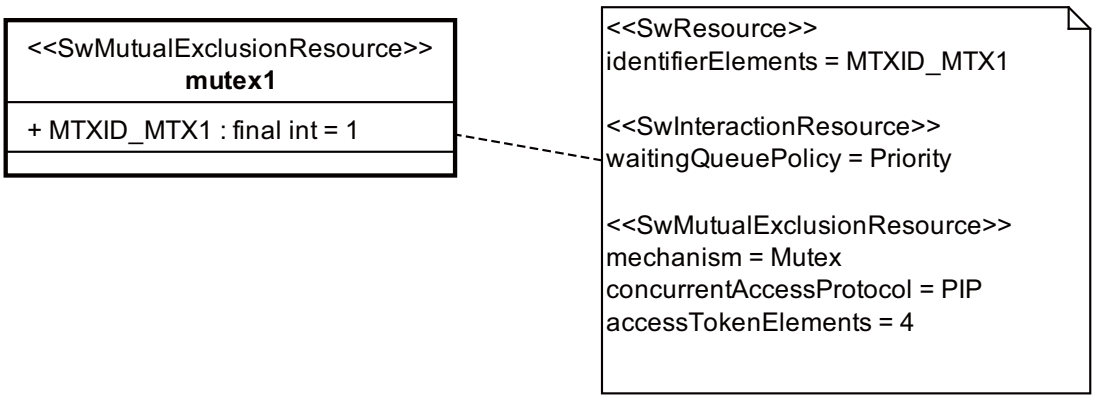

図 4.11: ミューテックスの記述例

イベントフラグ

イベントフラグに関するプロファイルを図4.12に示す．イベントフラグであることを示すために，NotificationResource::mechanismにはEventと記述する．以下に4.5.2で述べた情報に対応した属性を示す．

SwResource::identifierElements	イベントフラグID
SwInteractionResource::waitingQueuePolicy	待ち行列のポリシー
SwInteractionResource::waitingQueueCapacity	待ち行列のポリシー
SwInteractionResource::maskElements	ビットパターンの初期値
SwInteractionResource::isClear	待ち解除時にビットパターンクリアを行うかどうか

イベントフラグの構造を記述した例を図4.13に示す．

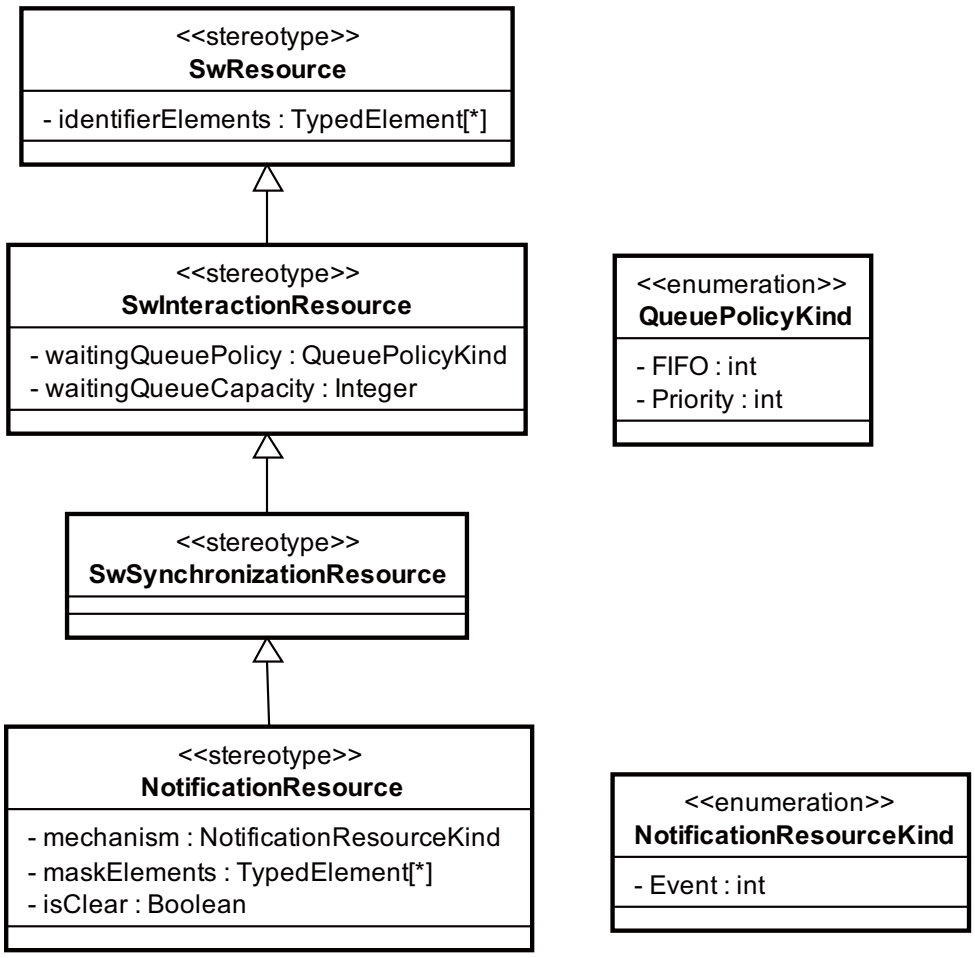

図 4.12: イベントフラグに関するプロファイル

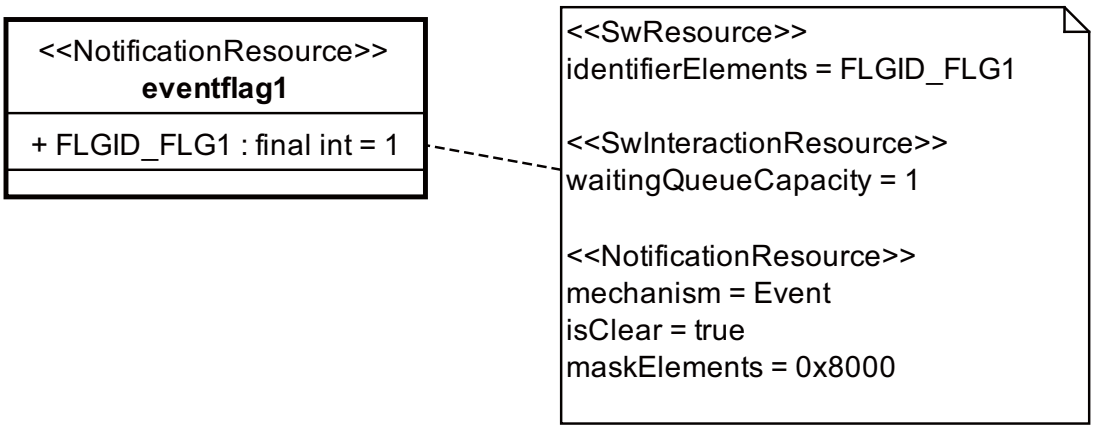

図 4.13: イベントフラグの記述例

ランデブ

ランデブに関するプロファイルを図 4.14 に示す．ランデブであることを示すために，NotificationResource::mechanism にはRendezvous と記述する．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	ランデブ ID
SwInteractionResource::waitingQueuePolicy	呼び出し待ち行列のポリシー
MessageComResource::messageSizeElements[0]	呼び出しメッセージの最大サイズ
MessageComResource::messageSizeElements[1]	返答メッセージの最大サイズ

ランデブの構造を記述した例を図 4.15 に示す．

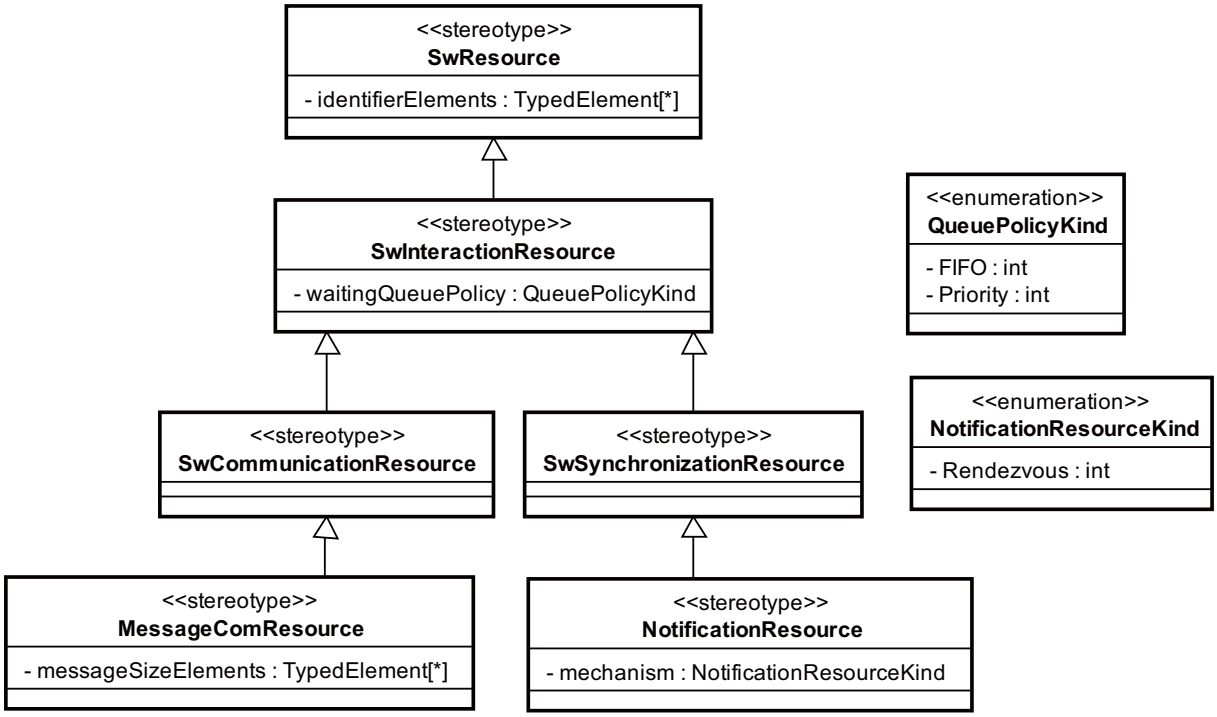

図 4.14: ランデブに関するプロファイル

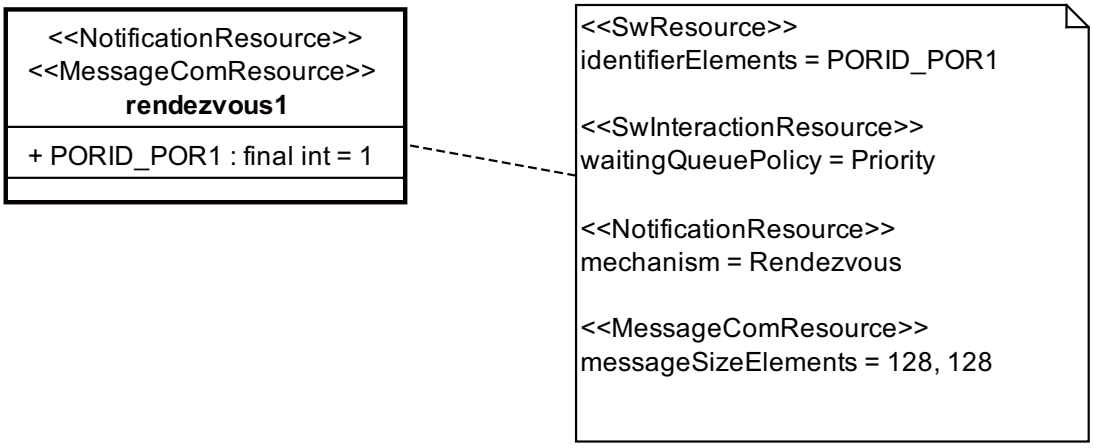

図 4.15: ランデブの記述例

データキュー

データキューに関するプロファイルを図 4.16 に示す．データキューであることを示すために，MessageComResource::mechanism には Dataqueue と記述する．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	データキュー ID
SwInteractionResource::waitingQueuePolicy	送信待ち行列のポリシー
MessageComResource::messageQueueCapacityElements	データの個数

データキューの構造を記述した例を図 4.17 に示す．

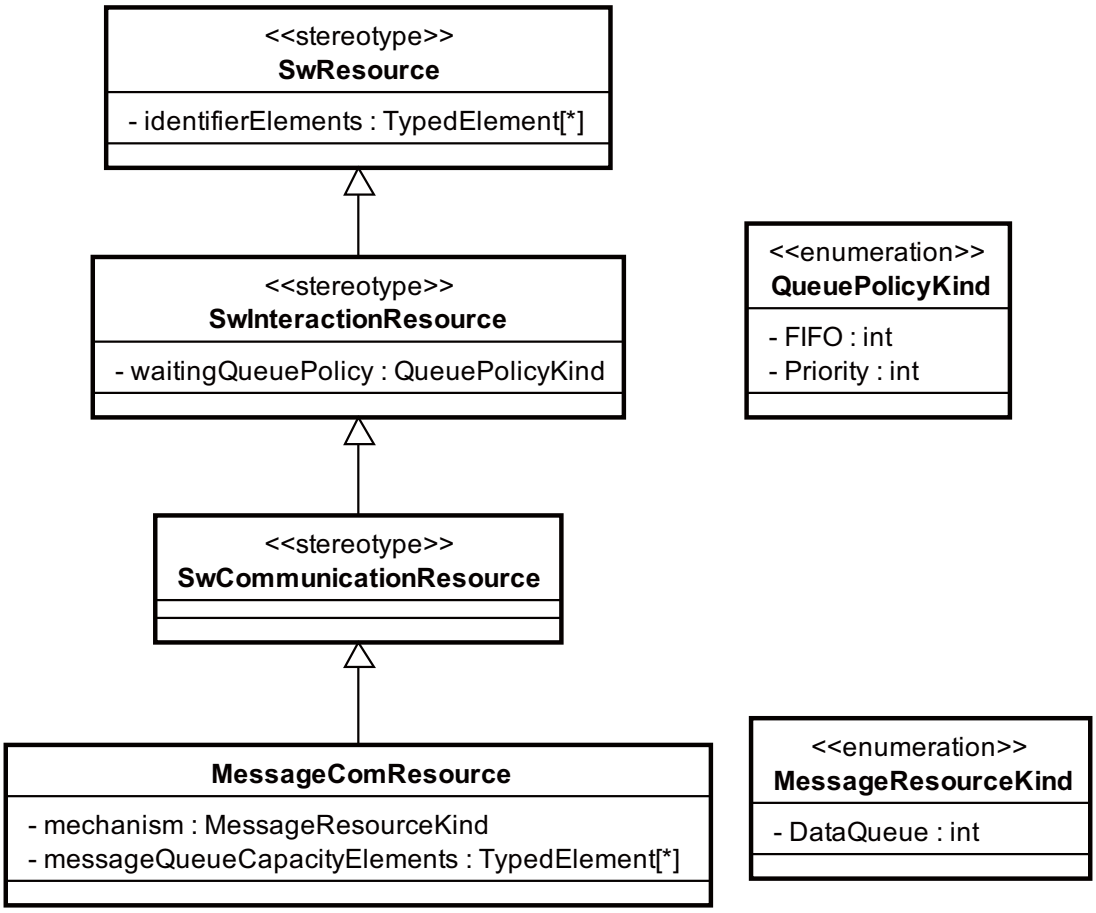

図 4.16: データキューに関するプロファイル

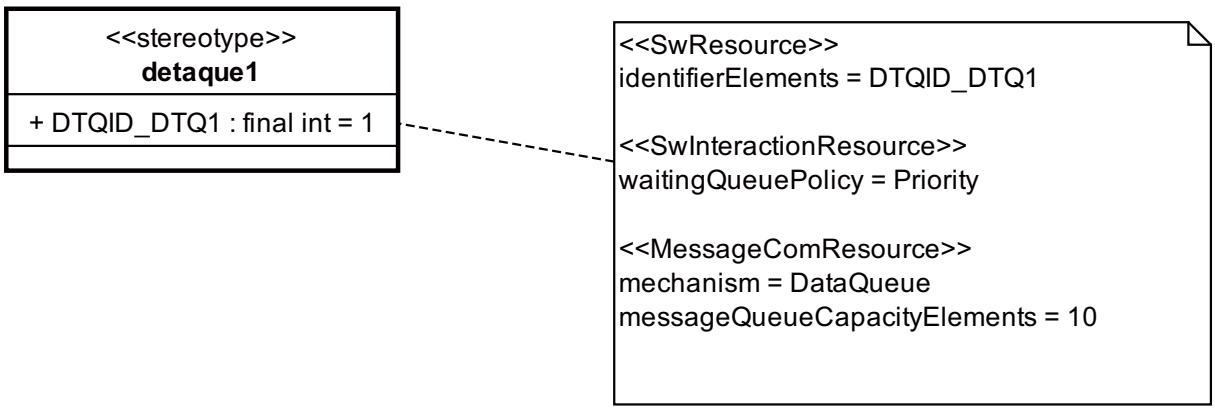

図 4.17: データキューの記述例

メールボックス

メールボックスに関するプロファイルを図 4.18 に示す．メールボックスであることを示すために，MessageComResource::mechanism には MailBox と記述する．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	メールボックス ID
SwInteractionResource::waitingQueuePolicy	待ち行列のポリシー
MessageComResource::messageQueuePolicy	メッセージキューのポリシー
MessageComResource::messageQueueMaxPriority	メッセージの優先度の最大値

メールボックスの構造を記述した例を図 4.19 に示す．

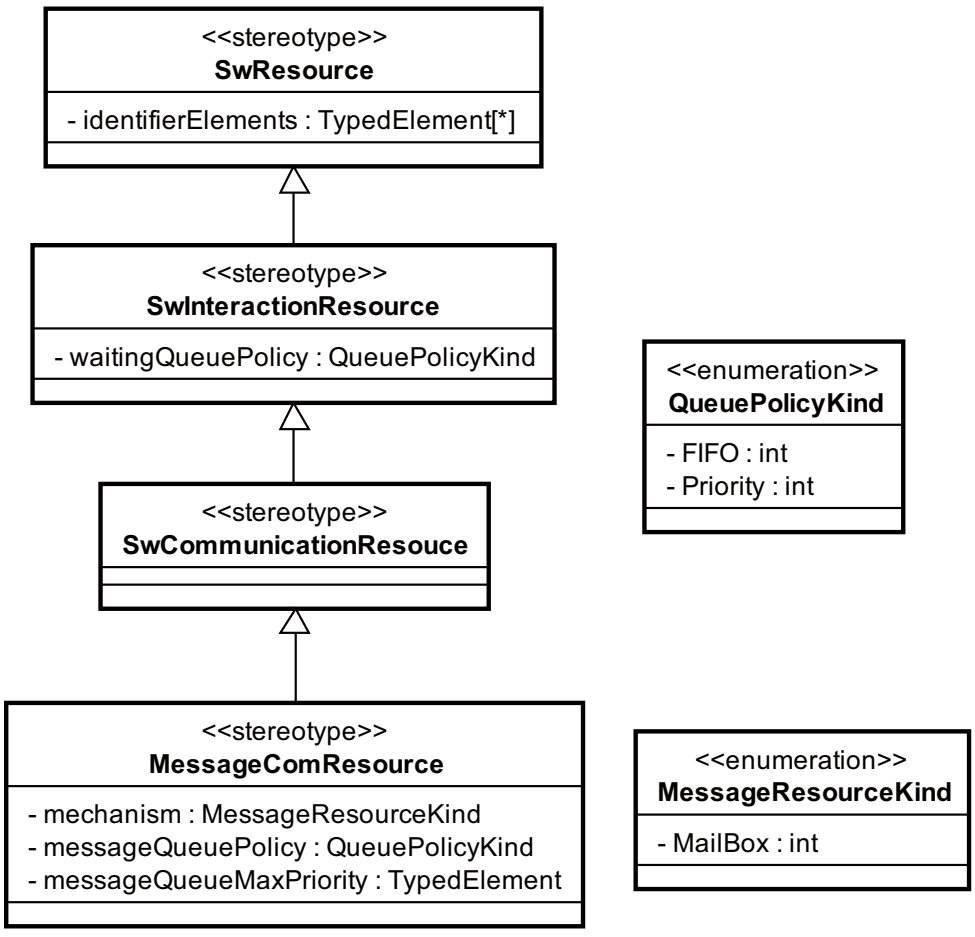

図 4.18: メールボックスに関するプロファイル

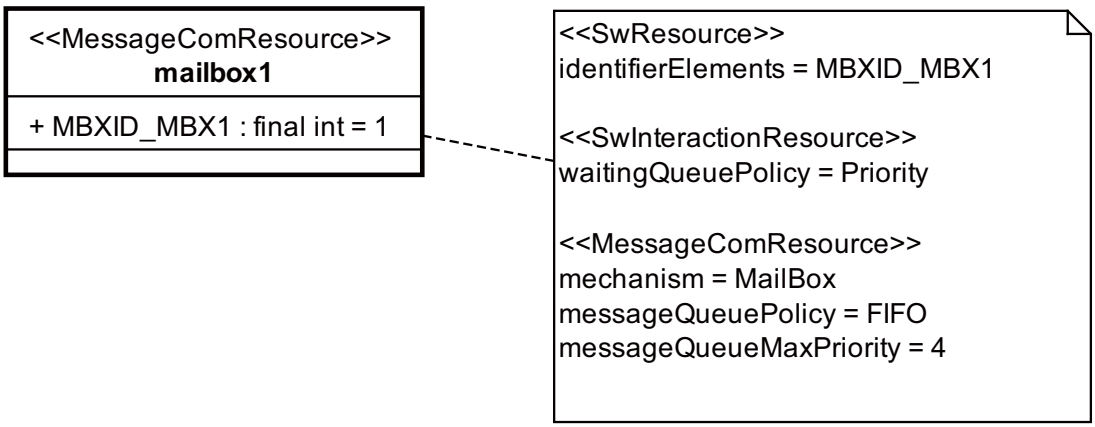

図 4.19: メールボックスの記述例

メッセージバッファ

メッセージバッファに関するプロファイルを図 4.20 に示す．メッセージバッファであることを示すために，MessageComResource::mechanism には MessageBuffer と記述する．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	メッセージバッファID
SwInteractionResource::waitingQueuePolicy	送信待ち行列のポリシー
MessageComResource::messageSizeElements[0]	メッセージの最大サイズ (バイト数)
MessageComResource::messageSizeElements[0]	メッセージバッファ領域のサイズ (バイト数)

メッセージバッファの構造を記述した例を図 4.21 に示す．

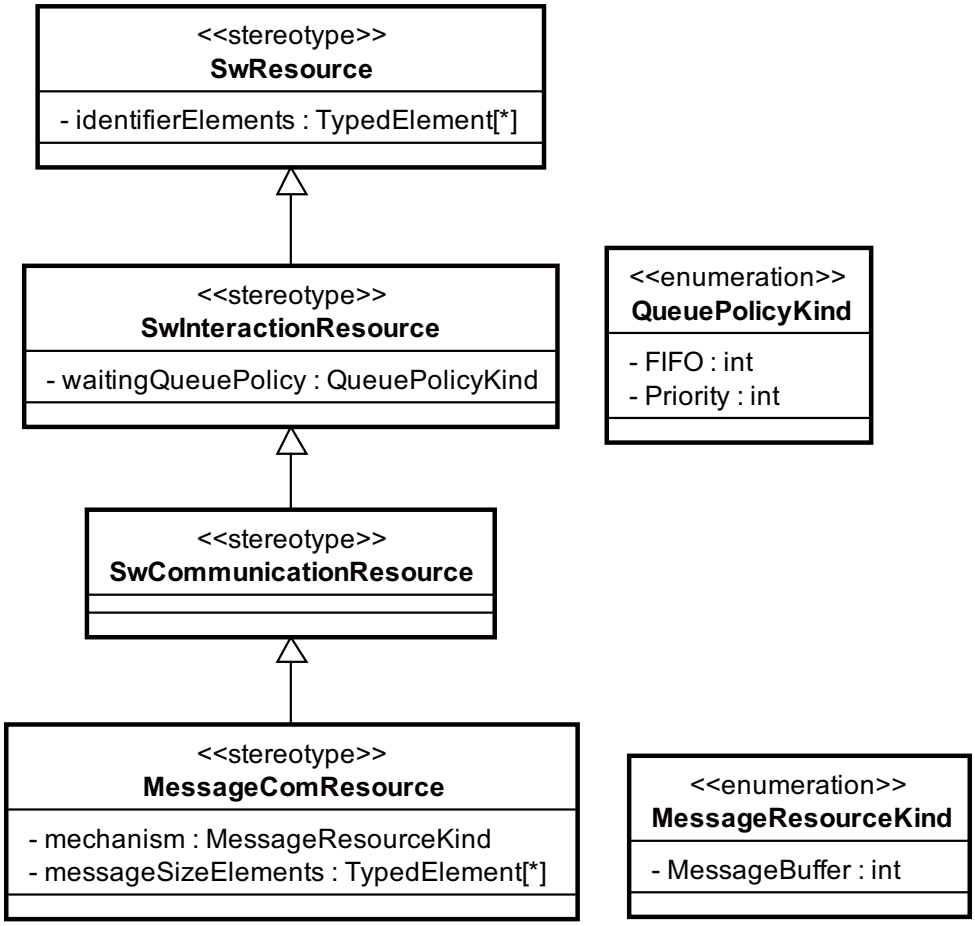

図 4.20: メッセージバッファに関するプロファイル

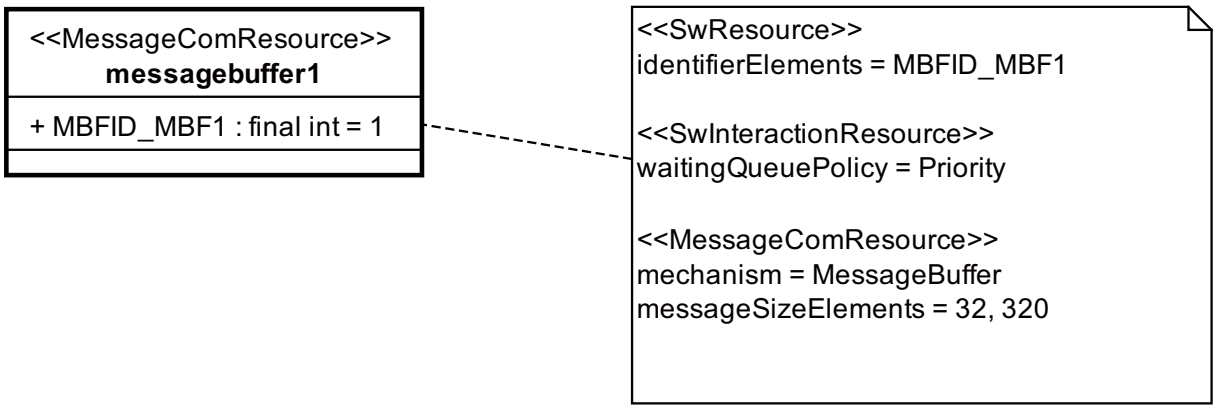

図 4.21: メッセージバッファの記述例

ランダム割り込み

ランダム割り込みに関するプロファイルを図 4.22 に示す．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	割り込みハンドラ ID
SwConcurrentResource::priorityElements	優先度
InterruptResource::interruptContextSize	割り込みコンテキストサイズ
InterruptResource::interruptFunctionMemoryUsage	割り込み関数のメモリ使用量

ランダム割り込みの構造を記述した例を図 4.23 に示す．

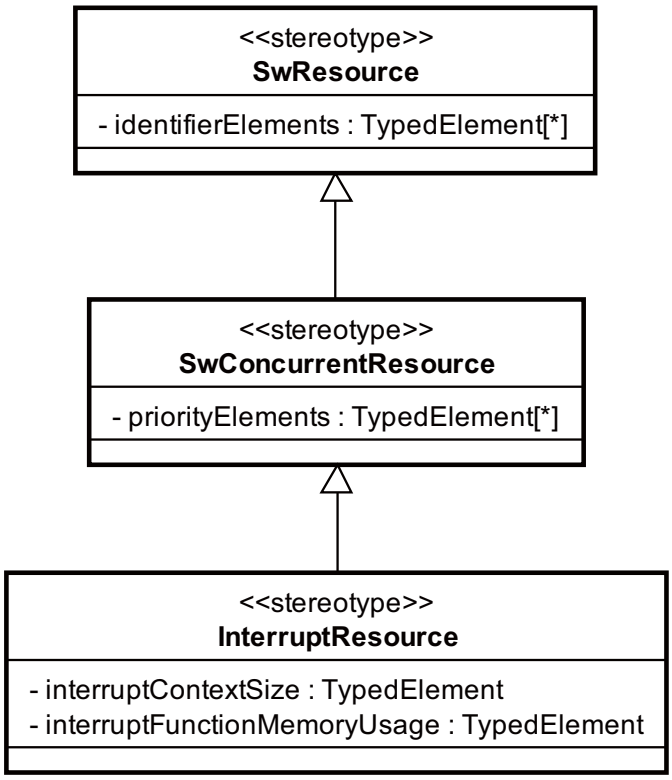

図 4.22: ランダム割り込みに関するプロファイル

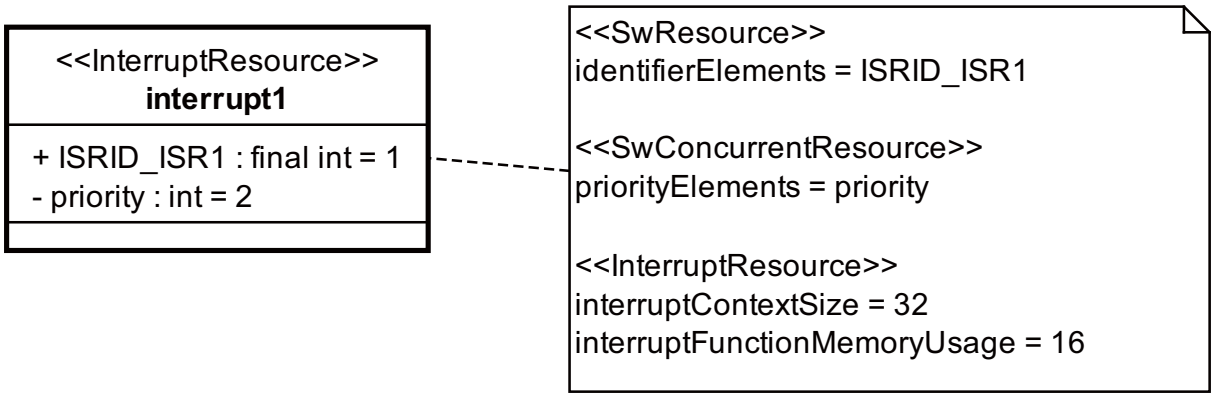

図 4.23: ランダム割り込みの記述例

周期割り込み

周期割り込みに関するプロファイルを図 4.24 に示す．以下に 4.5.2 で述べた情報に対応した属性を示す．

SwResource::identifierElements	周期割り込み ID
SwConcurrentResource::priorityElements	優先度
InterruptResource::interruptContextSize	割り込みコンテキストサイズ
InterruptResource::interruptFunctionMemoryUsage	割り込み関数のメモリ使用量
Alarm::cycle	周期
Alarm::pahse	位相

周期割り込みの構造を記述した例を図 4.25 に示す．

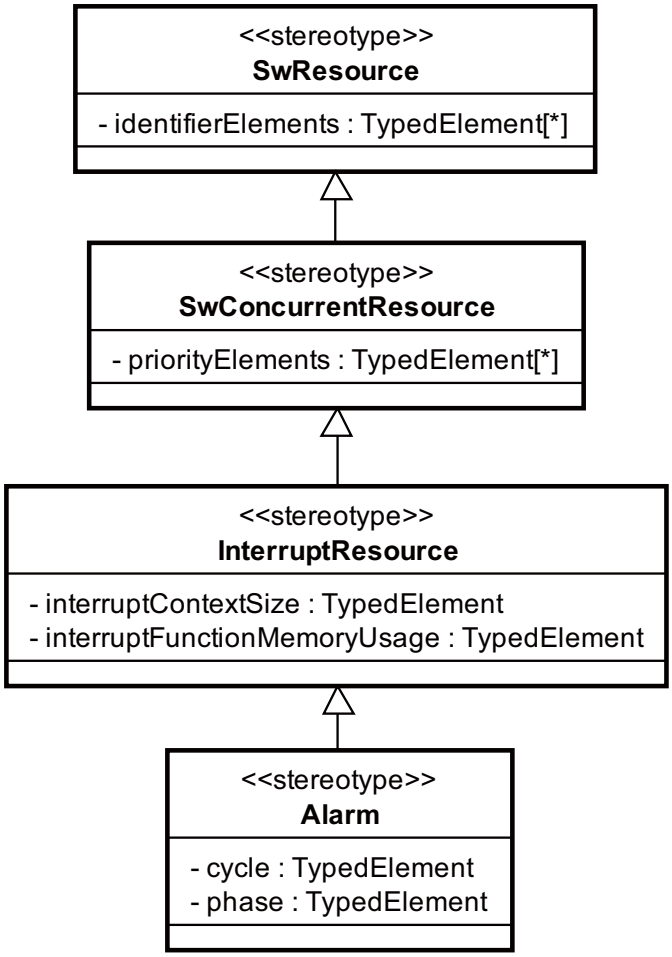

図 4.24: 周期割り込みに関するプロファイル

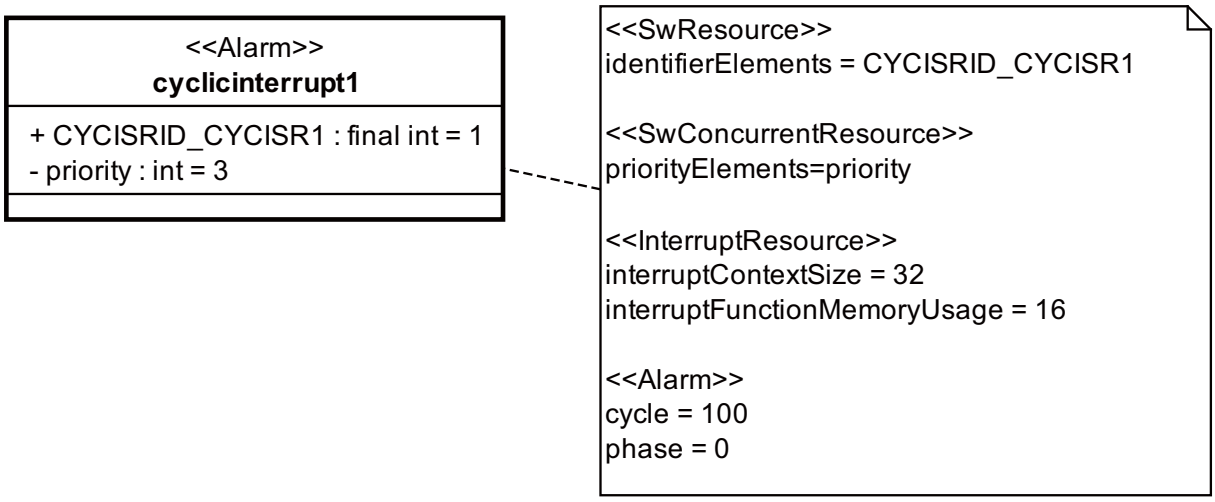

図 4.25: 周期割り込みの記述例

関数

関数に関するプロファイルを図 4.26 に示す．以下に 4.5.2 で述べた情報に対応した属性を示す．

Function::memoryUsageメモリ使用量

また，関数はクラスの操作として記述するか，もしくは，クラスとして記述するかの 2 通りあり，クラスとして記述する場合は Alarm::arguments にアーギュメントを記述する．

関数の構造を記述した例を図 4.27 と図 4.28 に示す．

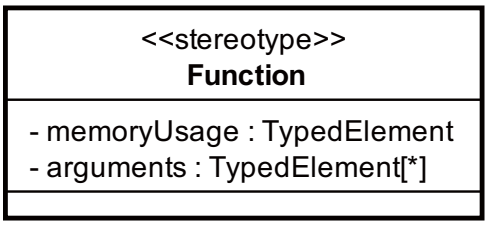

図 4.26: 関数に関するプロファイル

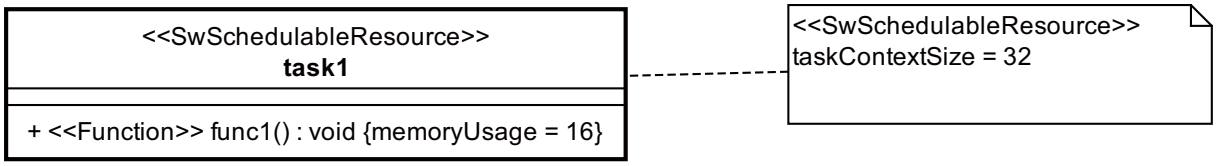

図 4.27: 関数の記述例 1

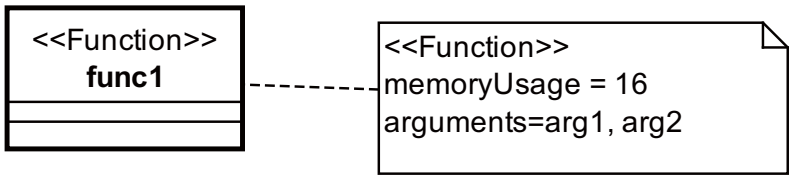

図 4.28: 関数の記述例 2

第5章　メモリ量見積もりツールの作成

5.1　ツールの概要

本研究で提案したメモリ量見積もり手法に従って，メモリを見積もるツールの作成を行った．

本見積もり方式では網羅的な解析が必要なので、モデル検査技術を用いて網羅的な解析を行った．ここではSPIN[7]を使用した．SPINでは，バックトラックの発生時で状態が以前の状態に戻ることに伴って使用している変数もその状態時の値に戻る．そのため，最大メモリ使用量を保持しておくための変数に通常の変数ではなくSPINの機能であるembedded C codeの変数を使用する．この変数は，状態が以前の状態に戻ることによる変数値の変更が行われず，探索開始から全ての探索終了まで自分で値を変更しない限りは値の変更が行われない．4.3.1においての$maxmem_1, maxmem_2, ..., maxmem_j$が，このembedded C codeの変数にあたるものである．また，4.2で述べたように，μITRON4.0のスケジューリング方式はFCFS方式である．そのため，SPINにおいてFCFS方式の振る舞いを行えるようにする必要がある．μITRON4.0におけるセマフォやデータキューといったサービスもSPINにおいてその振る舞いを行えるようにする必要がある．そこで，SPINにおいてμITRON4.0の振る舞いを実現するためのライブラリを作成した．このライブラリの使用方法は付録Aに示す．

メモリ量見積もりツールは，promelaファイルを生成するためのツールとそのツールを利用して実際にpromelaファイルを作成させて検証実行までしてメモリ見積もり量を画面に出力するためのツールから構成される．1つ目のツールはSPIN Code Generator，2つ目のツールはMemory Usage Estimatorと名付ける．SPIN Code Generatorは，タスク設計モデルに基づいて記述されたクラス図とステートマシン図のJUDEファイルからpromelaコードを生成する．Memory Usage Estimatorは，SPIN Code Generatorを利用してpromelaコードを生成してSPINで実行し，そのときの実行結果から最大メモリ使用量を画面に出力する．なお，なおタスク設計モデルの記述には，UMLモデリングツールであるJUDEを利用した．

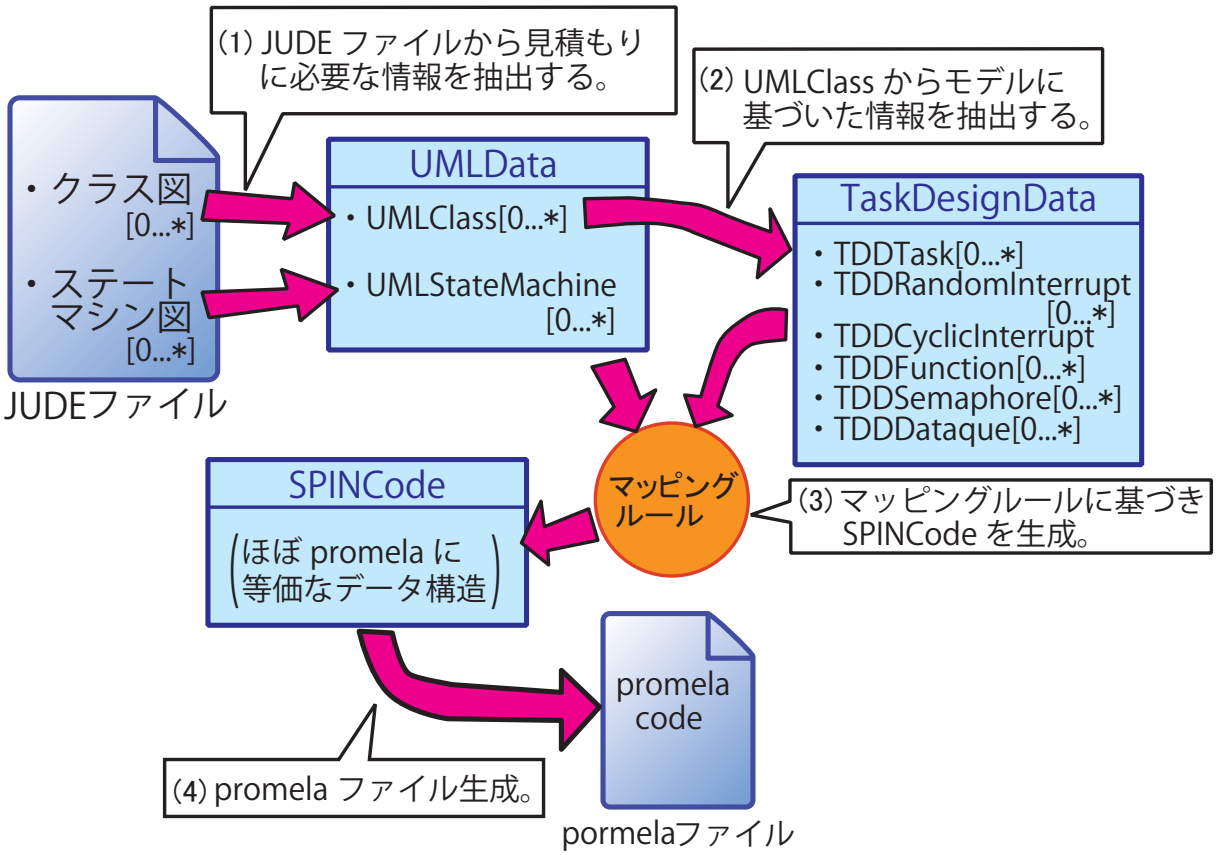

図 5.1: SPIN Code Generator の全体像

5.2 SPIN Code Generator

5.2.1 SPIN Code Generator の全体像

SPIN Code Generator の全体像を図 5.1 に示す.

(1) では，JUDE で記述されたデータファイルからデータ構造 UMLData に変換する．詳しくは 5.2.2 で述べる．

(2) では，UMLData の UMLClass から必要なデータをデータ構造 TaskDesignData に抽出する．詳しくは 5.2.3 で述べる．

(3) では，UMLData と TaskDesignData からマッピングルールに基づきデータ構造 SPINCode に変換する．詳しくは 5.2.4 で述べる．

(4) では，SPINCode から promela ファイルを生成する．詳しくは 5.2.5 で述べる．

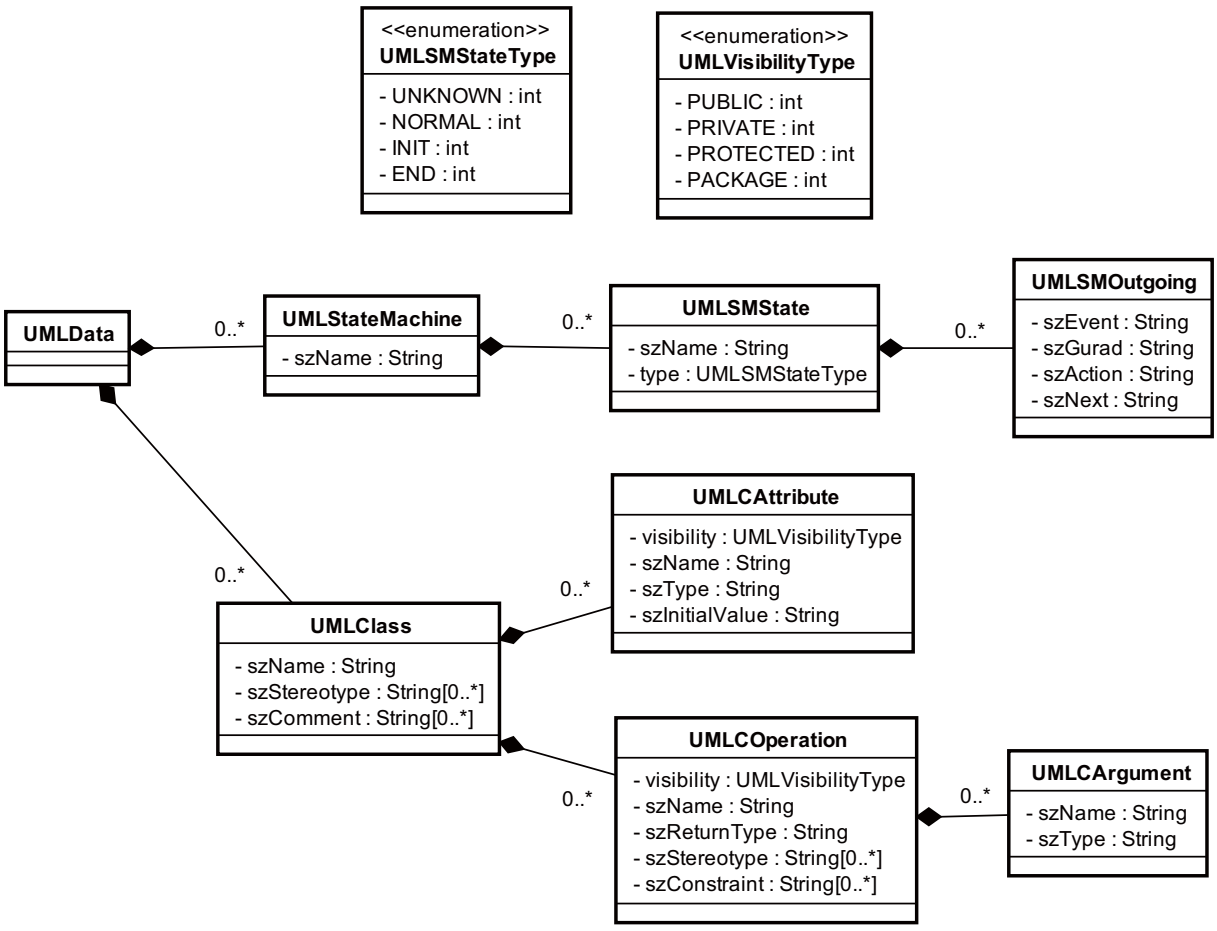

図 5.2: UMLData のデータ構造

5.2.2 (1) JUDE ファイルから UMLData への変換

タスク設計モデルに基づいて JUDE で記述されたデータファイルから，データ構造 UML-Data に変換する．変換といっても，特に複雑な処理は行わず，ステートマシン 1 つでステートマシンのためのクラス 1 つを割り当て，ステートマシンの状態 1 つに状態のためのクラスを 1 つを割り当てるというような素直な変換である．UMLData の構造を図 5.2 に示す．UMLData は，入力されたファイルのクラスとステートマシンの情報を等価に保持して置くためのデータ構造としている．UMLStateMachine がステートマシン図，UMLClass がクラス図に対応している．この段階では，このクラスはタスクを表すクラスであるだとか割り込みを表すクラスで表すだとかの識別は行わず，その識別は (2) で行う．UMLData のクラスと属性の意味を以下に示す．

・UMLSMStateType

ステートマシン図の状態の種類を表す列挙型．以下に属性の意味を示す．

UNKNOWN … 状態が不定

NORMAL ⋯ 開始状態と終了状態以外の状態
INIT ⋯ 開始状態
END ⋯ 終了状態

・UMLVisibilityType
可視性の種類を表す列挙型．以下に属性の意味を示す．
PUBLIC ⋯ public
PRIVATE ⋯ private
PROTECTED ⋯ protected
PACKAGE ⋯ package

・UMLData
このクラスは，UMLStateMachine を 0...*個，UMLClass を 0...*個持つクラス．

・UMLStateMachine
このクラスは，UML におけるステートマシンを意味する．szName はステートマシン名を表す属性である．

・UMLSMState
このクラスは，ステートマシンの状態にあたるクラスを意味する．以下に属性の意味を示す．
szName ⋯ 状態名
type ⋯ 状態の種類

・UMLSMOutgoing
このクラスは，ステートマシンにおける遷移を意味する．以下に属性の意味を示す．
szEvent ⋯ イベント
szGuard ⋯ ガード条件
szAction ⋯ アクション
szNext ⋯ 次の遷移先の状態名

・UMLClass
このクラスは，UML におけるクラスを意味する．以下に属性の意味を示す．
szName ⋯ クラス名
szStereotype ⋯ ステレオタイプ
szComment ⋯ アノテーション

・UMLCAttribute

このクラスは，クラスの属性を意味する．以下に属性の意味示す．
visiblity ••• 可視性
szName ••• 属性名
szType ••• 型
szInitialValue ••• 初期値

・UMLOperation
このクラスは，クラスの操作を意味する．以下に属性の意味を示す．
visiblity ••• 可視性
szName ••• 操作名
szReturnType ••• 返り値
szStereoType ••• ステレオタイプ
szConstraint ••• 制約

・UMLCArgument
このクラスは，捜査のパラメタを意味する．属性の意味を以下に示す．
szName ••• パラメタ名
szType ••• 型

5.2.3 (2) UMLClass からタスク設計モデルに基づいた情報を抽出

タスク設計モデルに基づいて情報を抽出し，データ構造 TaskDesignData にデータを格納する．どのクラスがタスクのクラスでどれが割り込みのクラスであるだとかは，ステレオタイプとタグ値を見て識別を行う．TaskDesignData の構造を図 5.3 に示す．TaskDesignData のクラスと属性の意味を以下に示す．合成集約で記述された関連は，集約する側が集約される側を 0...*個持つことを意味している．
・TDDTask
このクラスは，タスクの情報を保持するためのクラス．属性の意味を以下に示す．
szName ••• タスク名
szID ••• タスク ID
szPriority ••• 優先度
szContextSize ••• タスクコンテキストサイズ
szTaskFunctionMemoryUsage ••• タスク関数のメモリ使用量

・TDDDataque
このクラスは，データキューの情報を保持するためのクラス．属性の意味を以下に示す．
szName ••• データキュー名
szID ••• データキュー ID

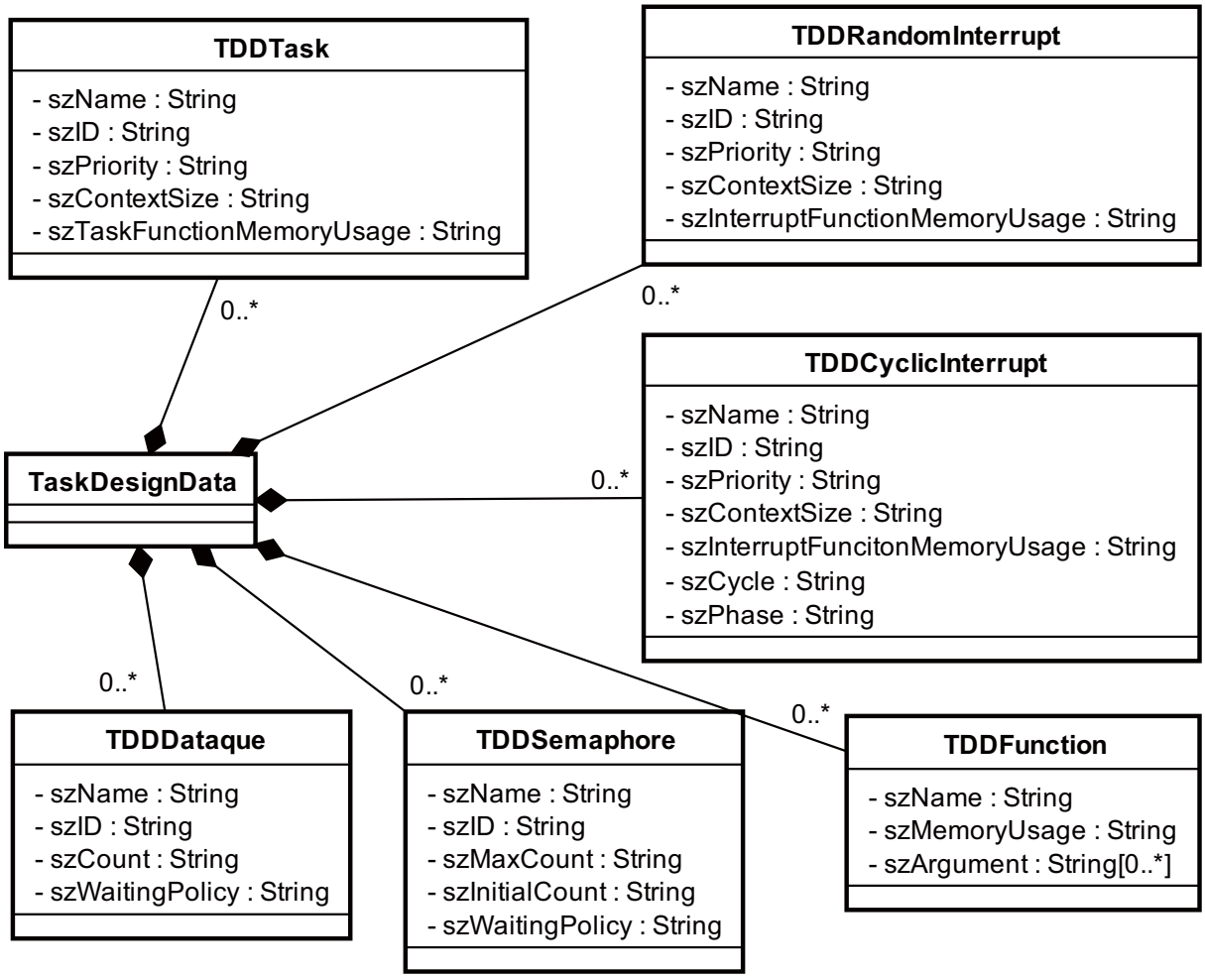

図 5.3: TaskDesignData のデータ構造

szCount ･･･ データの個数
szWaitingPolicy ･･･ 送信待ち行列のポリシー

・TDDSemaphore
このクラスは，セマフォの情報を保持するためのクラス．属性の意味を以下に示す．
szName ･･･ セマフォ名
szID ･･･ セマフォID
szMaxCount ･･･ 最大資源数
szInitialCount ･･･ 資源の初期値
szWaitingPolicy ･･･ 待ち行列のポリシー

・TDDRandomInterrupt
このクラスは，ランダム割り込みの情報を保持するためのクラス．属性の意味を以下に示す．
szName ･･･ 割り込み名
szID ･･･ 割り込みハンドラ ID
szPriority ･･･ 優先度
szContextSize ･･･ 割り込みコンテキストサイズ
szInterruptFunctionMemoryUsage ･･･ 割り込み関数のメモリ使用量

・TDDCyclicInterrupt
このクラスは，周期割り込みの情報を保持するためのクラス．属性の意味を以下に示す．
szName ･･･ 割り込み名
szID ･･･ 周期割り込み ID
szPriority ･･･ 優先度
szContextSize ･･･ 割り込みコンテキストサイズ
szInterruptFunctionMemoryUsage ･･･ 割り込み関数のメモリ使用量
szCycle ･･･ 周期
szPhase ･･･ 位相

・TDDFunction
このクラスは，関数の情報を保持するためのクラス．属性の意味を以下に示す．
szName ･･･ 関数名
szMemoryUsage ･･･ メモリ使用量
szArgument ･･･ 関数の引数

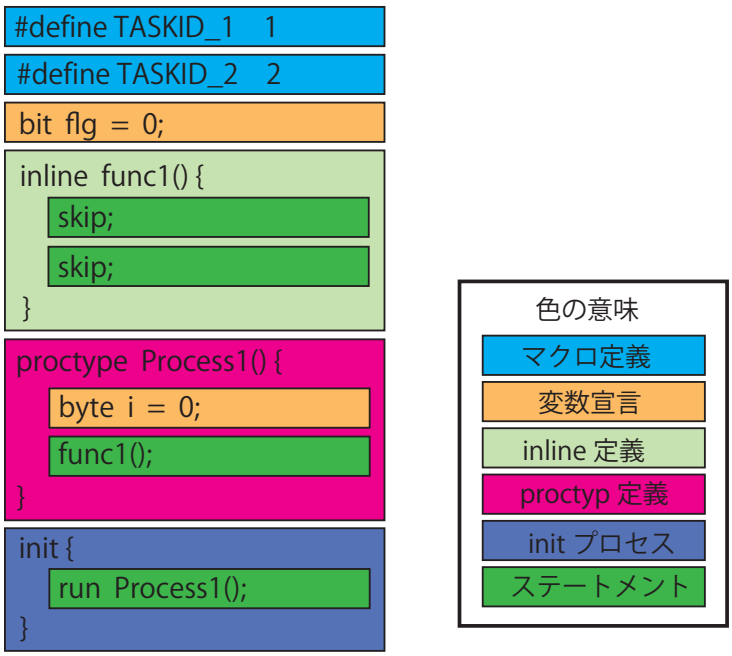

図 5.4: promela の構造の例

5.2.4 (3) マッピングルールに基づき SPINCode へ変換

(1) で作成した UMLData と (2) で作成した TaskDesignData からマッピングルールに基づいてデータ構造 SPINCode に変換する．

マッピングルールの説明をする前に，SPINCode の説明を行う．SPINCode は，このデータ構造にさえ変換すれば複雑な処理の必要なく promela ファイルが生成できるように作成した．ここで，今回の promela コード生成でとらえるべきだと考えた promela の構造は，マクロ定義，変数宣言，inline 定義，proctype 定義，init プロセス，ステートメントの 6 つの要素の列から成ると考えた．注意しなければならないのは，列であるので順番が存在するということである．なぜ順番が必要かというと，図 5.4 に示すように，promela においては各要素の順序が重要だからである．Process1 の中における「byte i = 0;」と「func1();」に関しても順番の情報は必要である．

これらを表現したものが図 5.5 と図 5.6 に示す SPINCode で，この構造は，上記で示したような列の構造を表すようにしているため，このデータ構造にすれば promela へのマッピングは容易にできるのは自明である．SPINCode のクラスと属性の意味を以下に示す．

・SPINCode

SPINCodeElement を順序付きで 0...*個持つ．

・SPINCodeElement

SPINMacro，SPINProc，SPINInitproc，SPINInline，SPINVardecl，SPINStmnt の 6 個

のクラスを継承してるが，図 5.5 に示しているのはその箇所で使用されないクラスの継承は省略している．

・SPINMacro
SPIN におけるマクロ定義を意味するクラス．属性の意味を以下に示す．
szMacro ••• マクロ定義

・SPINProc
SPIN におけるプロセスを意味するクラス．属性の意味を以下に示す．
szName ••• プロセス名
bActive ••• true であれば active
szArgument ••• プロセスの引数
szProvided ••• プロセスを実行させる条件

・SPINInitproc
SPIN おける init プロセスを意味するクラス．

・SPINInline
SPIN における inline 関数を意味するクラス．属性の意味を以下に示す．
szInline ••• インライン関数名
szArgument ••• インライン関数の引数

・SPINVardecl
SPIN における変数宣言を意味するクラス．属性の意味を以下に示す．
szVardecl ••• 変数宣言

・SPINStmnt
SPIN におけるステートメントを意味するクラス．属性の意味を以下に示す．
szLabel ••• ラベル名
szStmnt ••• ステートメント

次に，promela コードへのへのマッピングルールを説明する．

マッピングルールの流れを説明すると，TDDTask，TDDRandomInterrupt，TDDCyclicInterrupt をそれぞれ proctype にマッピングにする．TDDFucntion を inline にマッピングする．各クラスの属性を，可視性が public ならグローバル変数に，private なら各プロセスのローカル変数にマッピングする．ステートマシンの振る舞いは，outgoing = { event, guard, action, next } としたとき，outgoing_0, outgoing_1, ⋯ , outgoing_i を持った状態 state を以下のようにマッピングする．

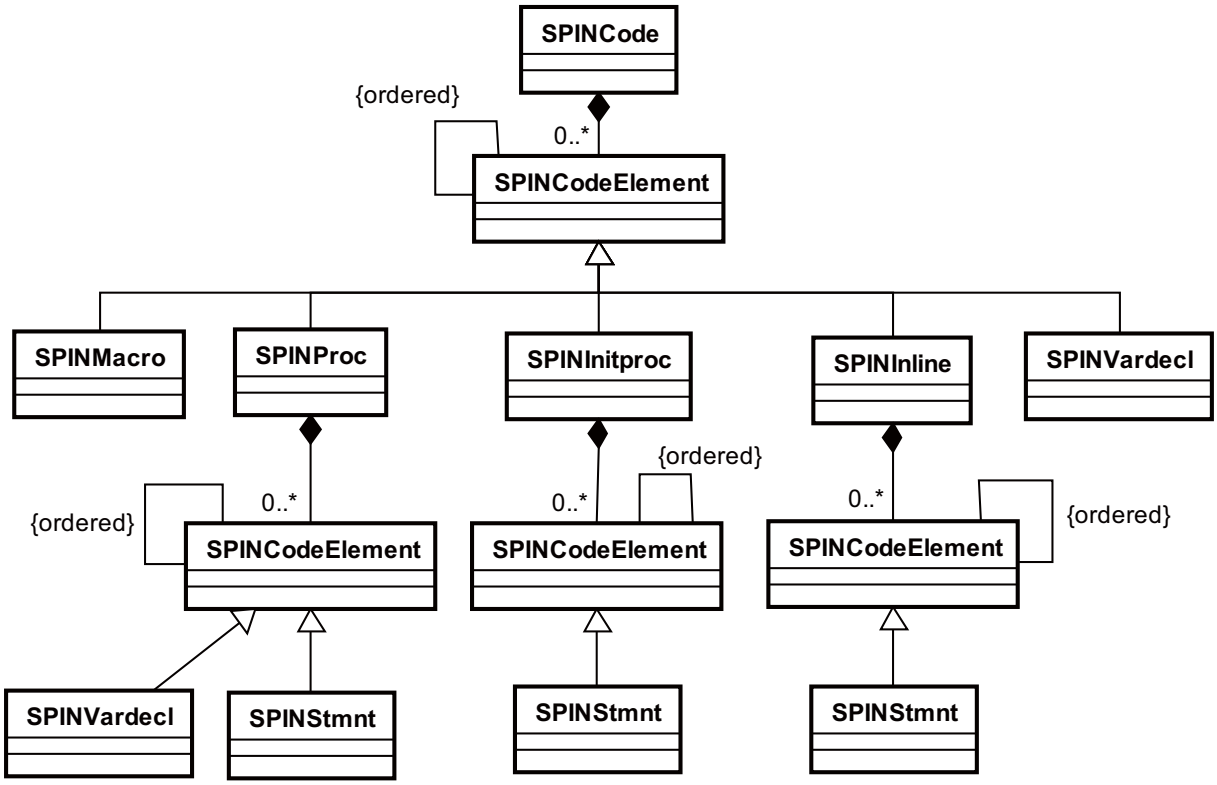

図 5.5: SPINCode のデータ構造 (属性省略)

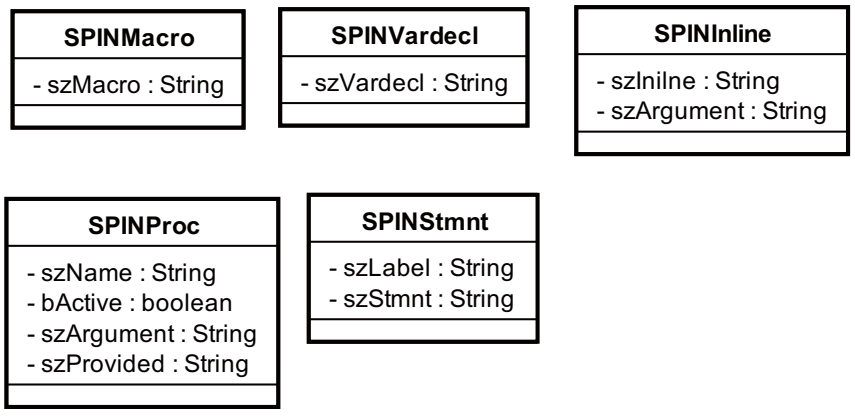

図 5.6: SPINCode の属性

ソースコード 5.1: ステートマシンのマッピングルール

```
1 state:
2         if
3         :: event_0 && guard_0 -> action_0; goto next_0
4         :: event_1 && guard_1 -> action_1; goto next_1
5                 ...
6         :: event_i && guard_i -> aciton_i; goto next_i
7         fi;
```

さらに詳細なマッピングルールは付録 C に示した.

5.2.5 (4) promela ファイル生成

(4) では，(3) で作成した SPINCode から promela ファイルを生成する．ここでは，特に複雑な処理はせず，(3) で述べたように，SPINCode から promela ファイルは容易に生成することができるので，promela ファイルを生成するのみでこのステップは終了する.

5.3 Memory Usage Estimator

5.3.1 Memory Usage Estimator の全体像

Memory Usage Estimator の全体像を 5.7 に示す．SPINCodeGenerator を用いて promela コードを生成して SPIN で実行し，そのときの出力結果から最大メモリ使用量を画面に出力する．このツールでは特に複雑な処理は無く，本来は人間が行う操作を自動実行するためのツールである.

STEP1 では，SPINCodeGenerator を用いて promela ファイルを生成する.

STEP2 では，生成した promela ファイルを SPIN で検証実行し，実行結果をファイルに出力する.

STEP3 では，出力結果から個々のタスクの最大メモリ使用量と割り込みの最大メモリ使用量の合計を得る.

STEP4 では，タスクの最大メモリ使用量の値を promela ファイルに反映し，再び SPIN で検証実行し，実行結果をファイルに出力する.

STEP5 では，出力結果からタスクのメモリ使用量の合計を得る．そして，タスクの最大メモリ使用量と割り込みのの最大メモリ使用量と個々のタスクの最大メモリ使用量を画面に出力する.

このツールのマニュアルは付録 B に示してある.

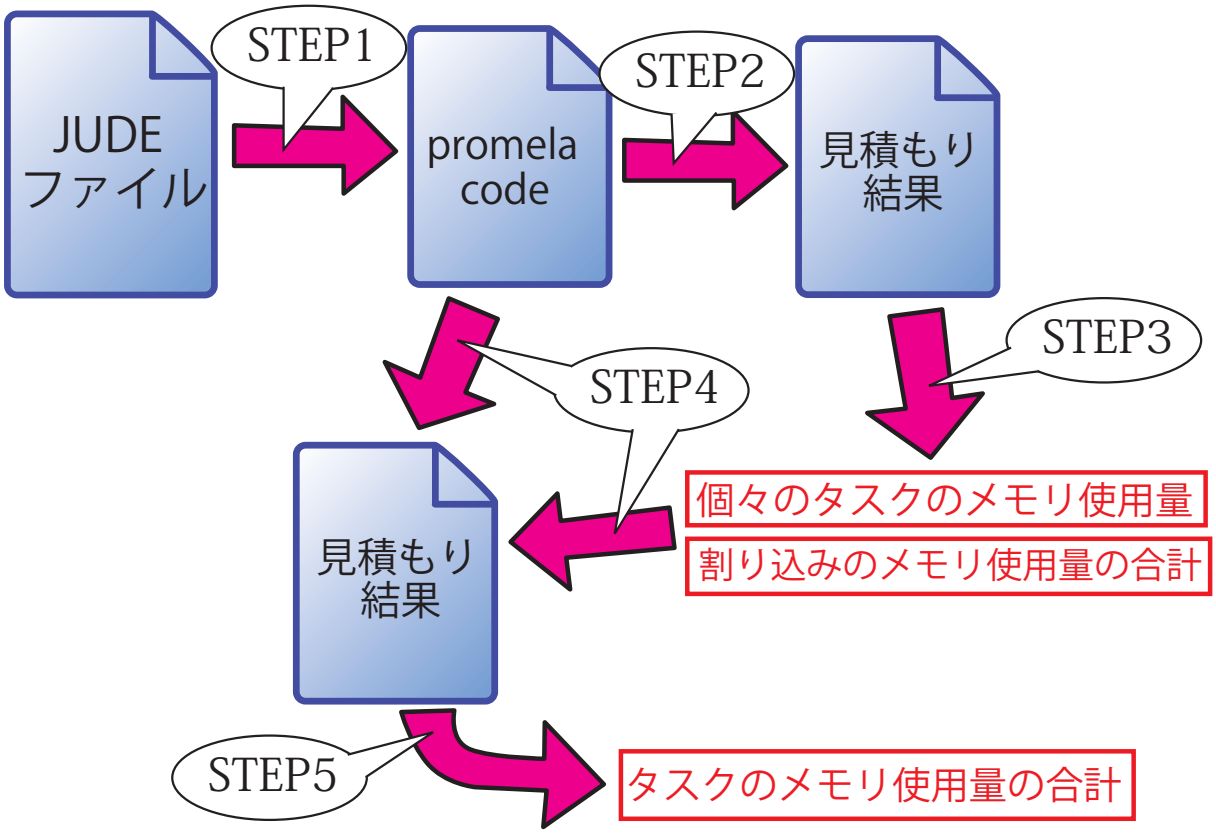

図 5.7: Memory Usage Estimator の全体像

第6章　評価

6.1　はじめに

本章では，4 章で述べたメモリ量見積もり手法を用いることで，タスク分割などを行った時のメモリ量の傾向を見ることができるかを評価する．具体的には，設計段階での見積もりとその設計で実際に実装を行ったときのメモリ使用量を実測し，設計での見積もり量においても実装後の実測したメモリ量においても同じ傾向が表れるかどうかの比較を行う．

6.2　評価に用いる設計

以下の要求を満たす設計を行う．
「センサー 1 とセンサー 2 が存在する．200ms ごとにセンサー 1 の値を 10 回読み，それが完了するとセンサー 2 の値を 200ms ごとに 10 回読み，再び 200ms ごとにセンサー 1 の値を 10 回読むというのを繰り返す.」

この仕様に対して，2 つの設計を行った．この 2 つの設計は，図 6.1 に示すような周期が同じ，もしくは，ほぼ周期が同じ周期割り込みが複数存在する場合に，それらの周期割り込みをまとめるかまとめないかの違いに観点を置いて設計を行った．これら 2 つの設計で，周期をまとめていない設計方法を「設計方法 1」，周期をまとめている設計方法を「設計方法 2」と名付け，以降で説明を行っていく．

設計方法 1 の全体像構造を図 6.2 に，それぞれの振る舞いを図 6.4〜6.12 に示す．これらの図は，メモリ量見積もりツールに与える情報の全てである．メモリ使用量をすでに決めてあるがこの値についは後述する．

設計方法 2 の全体像構造を図 6.13 に，それぞれの振る舞いで設計手法 1 から変更されたものを図 6.14 と図 6.15 に示す．それ以外の設計方法 1 と同様のものは省略する．メモリ使用量をすでに決めてあるがこの値についは後述する．

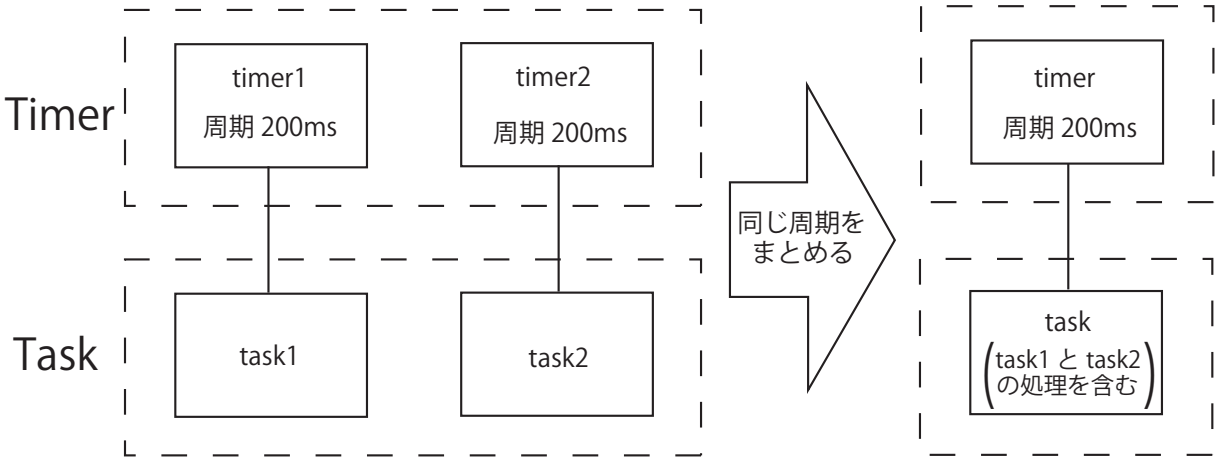

図 6.1: 今回のタスク分割

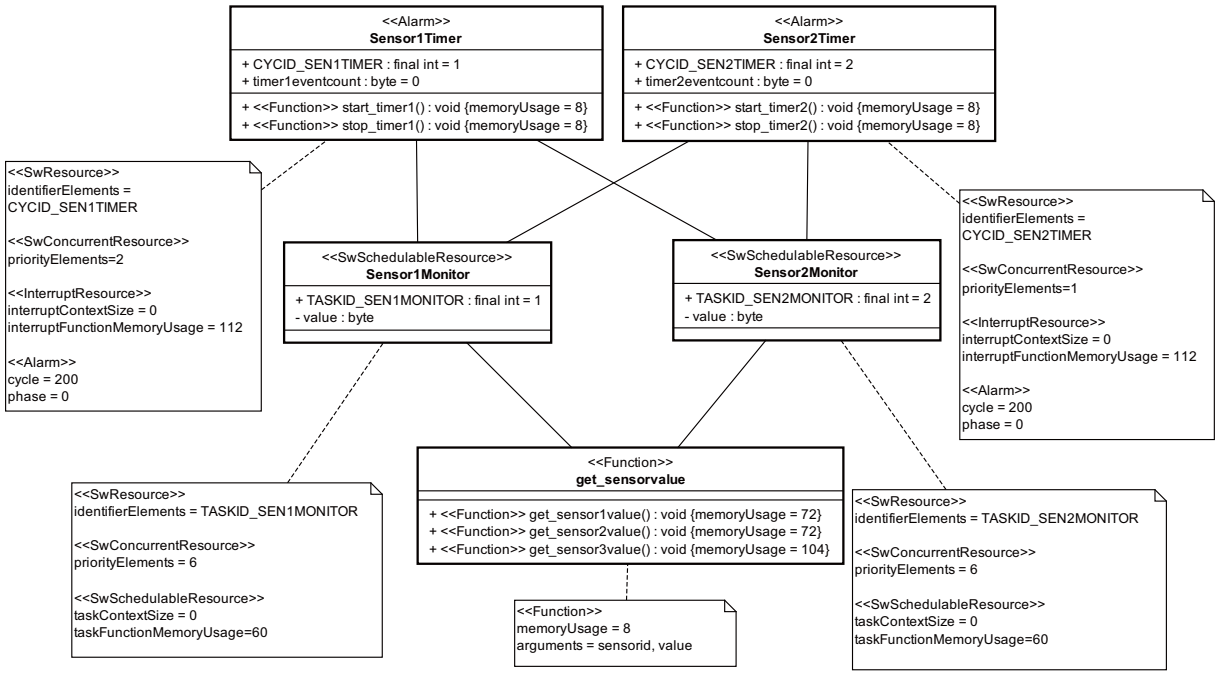

図 6.2: 設計方法 1：全体の構造

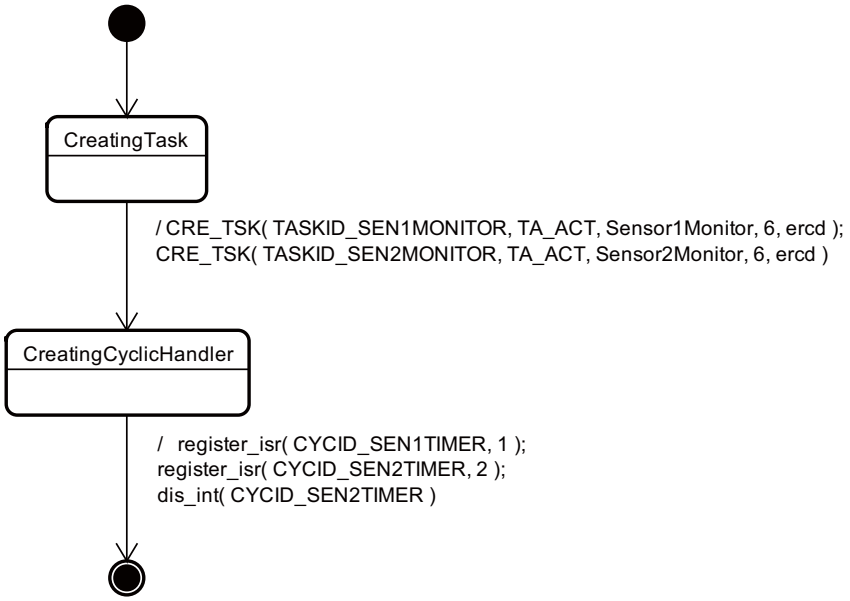

図 6.3: 設計方法 1：オブジェクト初期化

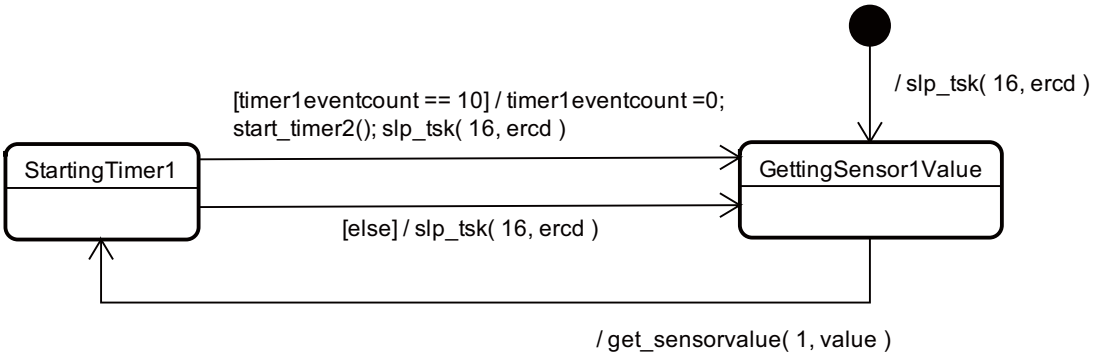

図 6.4: 設計方法 1：Sensor1Monitor

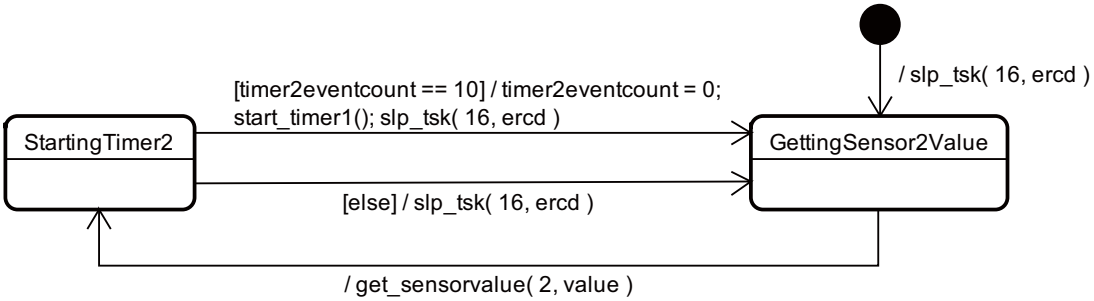

図 6.5: 設計方法 1：Sensor2Monitor

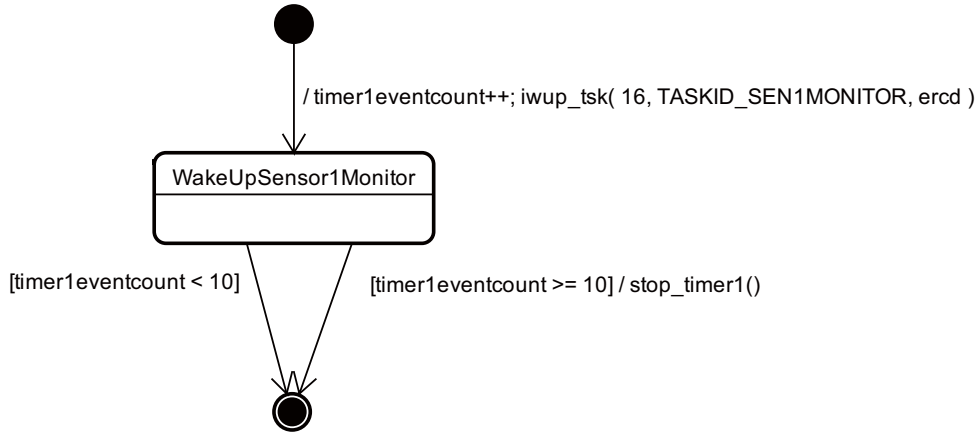

図 6.6: 設計方法 1：Sensor1Timer

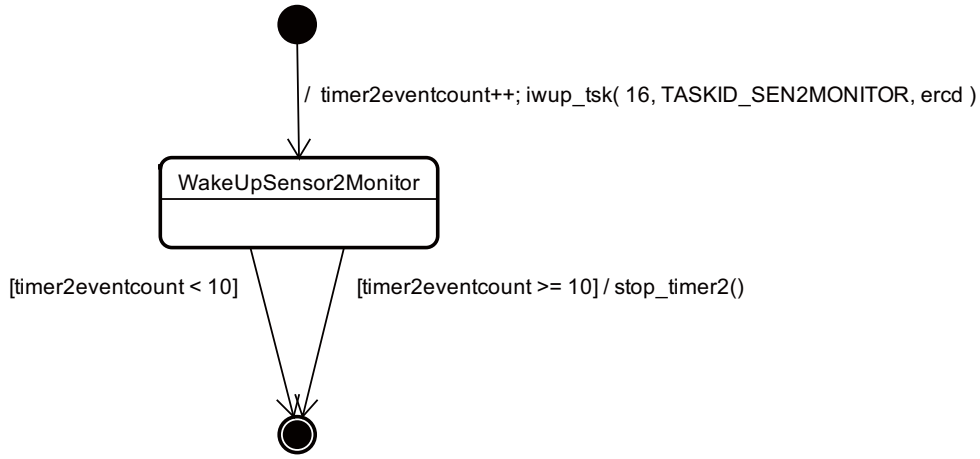

図 6.7: 設計方法 1：Sensor2Timer

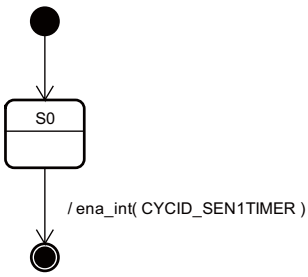

図 6.8: 設計方法 1：start_timer1

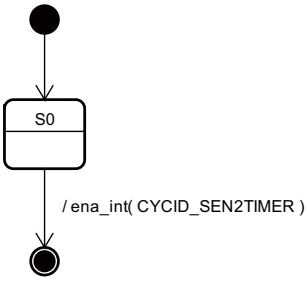

図 6.9: 設計方法 1：start_timer2

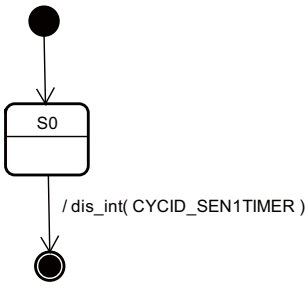

図 6.10: 設計方法 1：stop_timer1

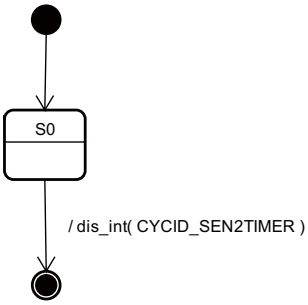

図 6.11: 設計方法 1：stop_timer2

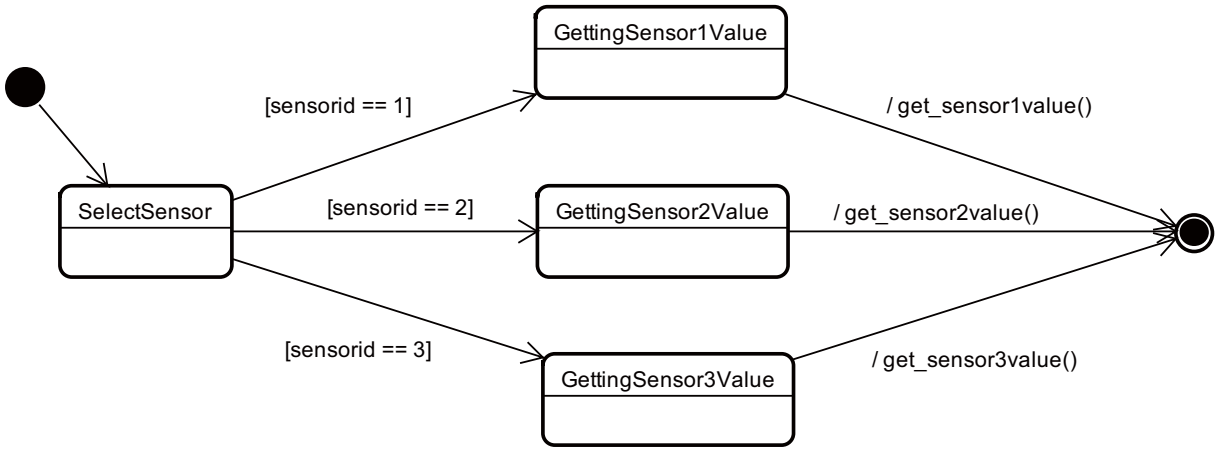

図 6.12: 設計方法 1：get_sensorvalue

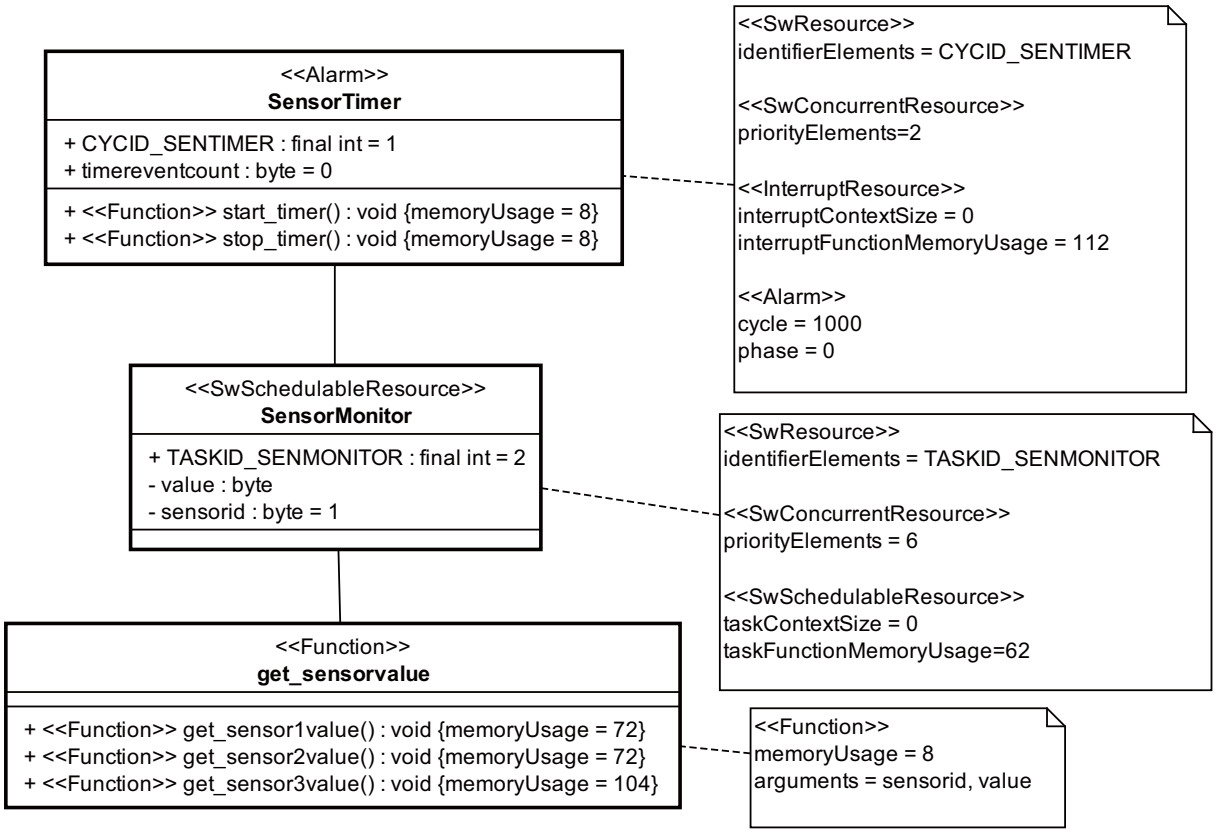

図 6.13: 設計方法 2：全体の構造

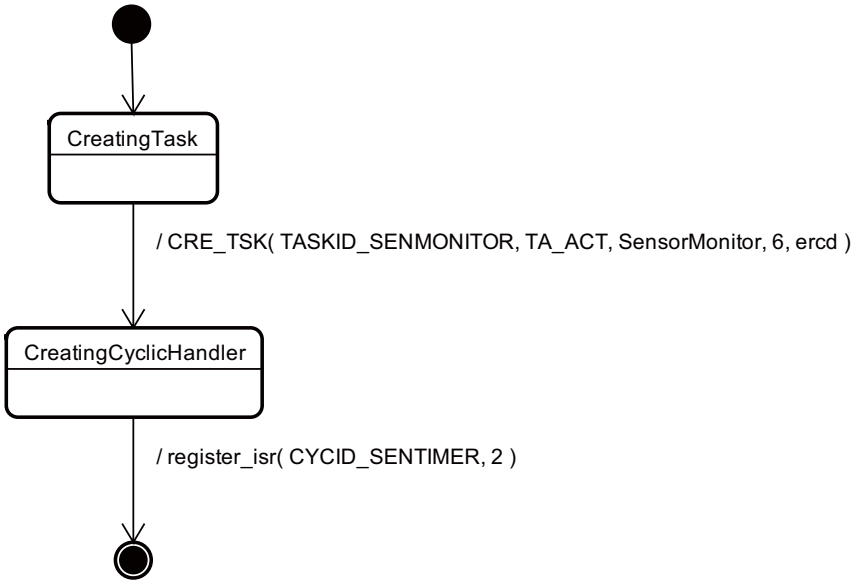

図 6.14: 設計方法 2：オブジェクト初期化

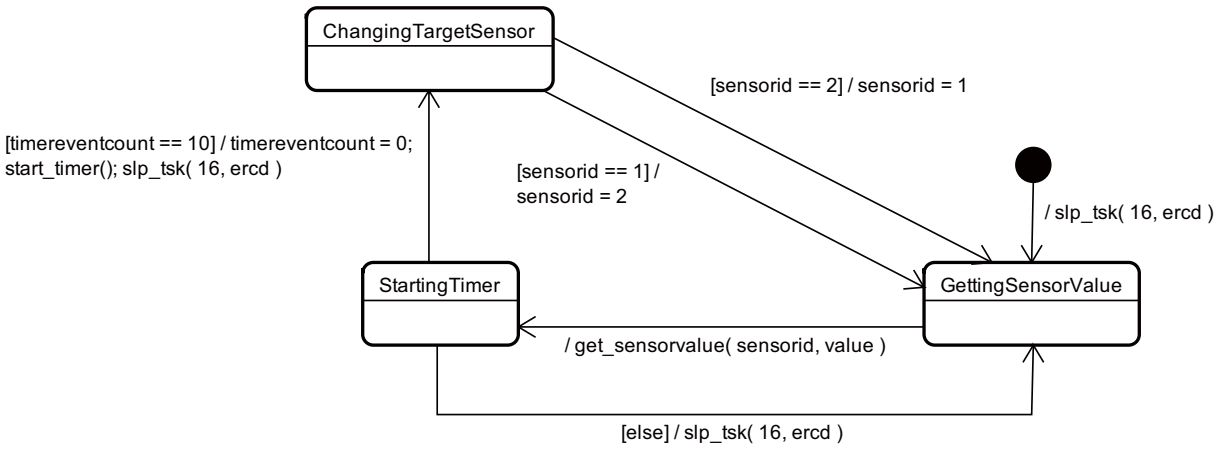

図 6.15: 設計方法 2：SensorMonitor

6.3 実測方法

あらからじめ特定の値でスタックを埋めておき，関数の呼び出し前と呼び出し後のメモリの値を直接見て変化した差を求める．実測例を図 6.16 に示す．左側が関数の呼び出し前のメモリの値，右側が関数の呼出し後のメモリの値を示している．この例では“ 0xee ”であらかじめスタックを埋めてある．呼び出し前では 0x501448 までが使用されていたスタックの上限で，呼出し後では 0x501440 が使用されていたスタックの上限である．このときの差は 8Byte であり，この値が関数の使用したメモリ量となる．

6.4 評価の流れ

以下の流れで評価を行う．
(1) 使用するサービスコールや関数などのメモリ使用量をあらかじめ決めておく．
(2) 今回作成したメモリ量見積もりツールでメモリ量を見積もる．
(3) 実際に実装してメモリ量を実測する．
(4) 評価を行う．

また，実装は以下のものを対象として行った．
・マイコン：秋月電子 AKI-H8/3069F
・OS：TOPPERS/JSP 1.4.3 (μITRON4.0 準拠)

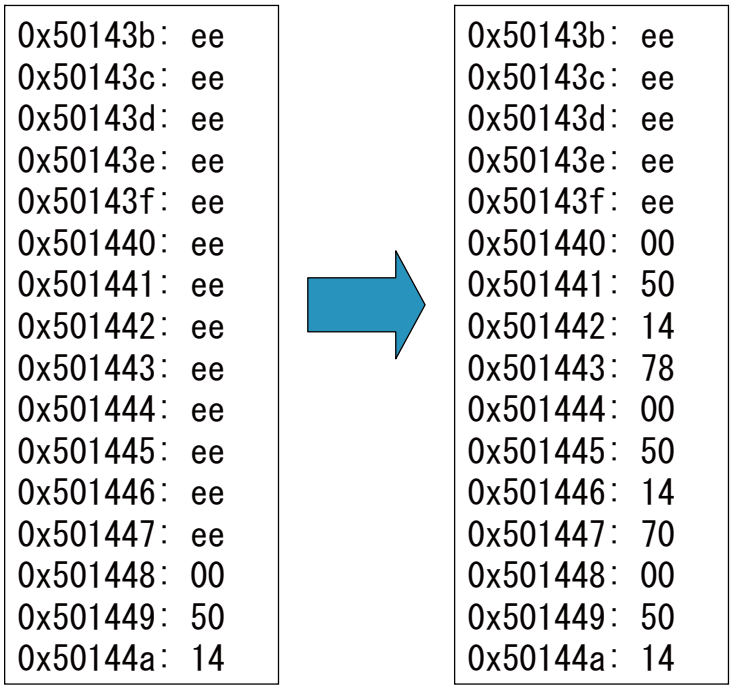

図 6.16: メモリ量実測方法の例

6.5 (1) 使用するサービスコールや関数のメモリ使用量の決定

TOOPERS/JSP1.4.3 のマニュアルにおいて以下に示すメモリ使用量について述べられていないのであらかじめ実測しておく必要がある．これはアプリケーション依存でない OS ごとに決まった基本的な値であり，アプリケーションの実測をしているのではないことに注意してほしい．

・タスク関数のメモリ使用量
・割り込み関数のメモリ使用量
・slp_tsk のメモリ使用量
・iwup_tsk のメモリ使用量
・関数呼び出しのメモリ使用量

6.3 で述べた実測方法を用いて，タスク関数と割り込み関数のメモリ使用量を実測した．実測値を以下に示す．

・タスク関数 … 32Byte
・割り込み関数 … 112Byte

今回使用するサービスコールは slp_tsk と iwup_tsk である．これも同様に 6.3 で述べた実測方法を用いて実測したがいずれも 0Byte となった．実測できなかったためこれらサービスコールのメモリ使用量は 16Byte と決め打ちで与えることとした．

関数呼び出しのみで仕様されるメモリ量は8Byteとした.

get_sensorvalue群の関数は，メモリ量の傾向を見るために，メモリ量の変化をつけさせるために，意図的にメモリを消費させる関数である．これは，実装時には，それぞれのメモリ消費を行うようコーディングを行う.

タスクコンテキストと割り込みコンテキストに関するメモリ使用量がマニュアルで述べられていないため，今回の見積もりにおいは，タスクスタックと割り込みスタックに関する見積もりのみを行うこととした.

また，マニュアルにおいて，割込みが発生すると，タスクコンテキスト用スタックにレジスタを一部保存した後，割込み用スタックに切り替えるため，タスクスタックが24Byte使用されると記述されていのるのでこれも考慮する必要がある.

以上これらを考慮して図6.2と図6.13におけるメモリ使用量を決定した.

6.6 (2) メモリ見積もりツールを用いての見積もり

今回作成したメモリ量見積もりツールを用いて設計手法1のメモリ使用量を見積もった結果を以下に示す.

・タスクのメモリ使用量の合計 … 280Byte
・割り込みのメモリ使用量の合計 … 128Byte
・Sensor1Monitor … 140Byte
・Sensor2Monitor … 140Byte

メモリ量見積もりツールを用いて設計手法2のメモリ使用量を見積もった結果を以下に示す.

・タスクのメモリ使用量の合計 … 142Byte
・割り込みのメモリ使用量の合計 … 128Byte
・SensorMonitor … 142Byte

6.7 (3) 実際に実装をしてのメモリ量の実測

設計手法1の実測結果を以下に示す.

・タスクのメモリ使用量の合計 … 224Byte
・割り込みのメモリ使用量の合計 … 112Byte
・Sensor1Monitor … 112Byte
・Sensor2Monitor … 112Byte

メモリ量見積もりツールを用いて設計手法2のメモリ使用量を見積もった結果を以下に示す.

・タスクのメモリ使用量の合計 … 120Byte
・割り込みのメモリ使用量の合計 … 112Byte
・SensorMonitor … 120Byte

6.8 (4) 評価

実際に実測したメモリ量の傾向

設計手法 1 と設計手法 2 の実測結果の傾向について述べる．タスクのメモリ使用量の合計は，設計手法 1 は 224Byte，設計手法 2 は 120Byte であった．タスクを 1 つにまとめたことで約 2 倍のメモリ量の違いとなった．割り込みのメモリ使用量の合計は，設計手法 1 と設計手法 2 で同じメモリ使用量となった．割り込みに関しては，設計手法の違いによる差異は見られなかった．

見積もりツールにおけるメモリ量の傾向

見積もりツールを用いたときの設計手法 1 と設計手法 2 のメモリ量の傾向について述べる．タスクのメモリ使用量の合計は，設計手法 1 は 280Byte，設計手法 2 は 142Byte であった．タスクを 1 つにまとめたことで約 2 倍のメモリ量の違いとなった．割り込みのメモリ使用量の合計は，設計手法 1 と設計手法 2 で同じメモリ使用量となった．割り込みに関しては，設計手法の違いによる差異は見られなかった．

見積もりと実測の比較

メモリ量の傾向においては，見積もりでも実測でも同様の結果が得られた．メモリ量の値については，見積もりのほうが実測よりも大きく出た．決め打ちの値を大きめにとっていたという理由もあると思われるが，コンパイラの最適化などの影響もあると思われる．

考察

タスク設計の段階で，実装後の実測値と同じ値を見積もるのはコンパイラの影響などもありほぼ不可能である．だが，本研究では，複数のタスク設計があるときの傾向の違いを見たいのであって，メモリ量を完璧に見積もるのは目的ではない．今回の評価においては，設計の違いによるメモリ使用量の傾向を捉えることができており，設計判断に使えると考える．ただし，使用する関数のメモリ使用量がほぼ正確に見積もれる，もしくは，どのくらい使用するかがほぼわかっているという条件においてである．OS 関連のメモリ使用量は，使用するメモリ使用量をきちんと公開している OS を購入すれば，さほど苦労なく OS 関連のメモリ使用量はわかると思われる．

6.9　関数のメモリ使用量が不明な場合の見積もり

あらかじめ主要な関数のメモリ使用量を見積もれない場合は，かなり余裕を持ってメモリ量を決め打ちで与えるしかない．このときに，設計手法 1 と設計手法 2 において，関数のメモリ使用量が見積もれる場合と同様の傾向が見られるかどうかを検討した．

以下の関数のメモリ使用量が見積もれないとし，今まで与えていたメモリ量の 5 倍のメモリ量を使用することとした．

・slp_tsk
・get_sensorvalue

6.9.1　slp_tsk のメモリ使用量が不明なときの見積もり結果

設計手法 1 と設計手法において，slp_tsk のメモリ使用量を 5 倍にして見積もりツールによる見積もりを行う．設計手法 1 の見積もり結果を以下に示す．

・タスクのメモリ使用量の合計 … 280Byte
・割り込みのメモリ使用量の合計 … 128Byte
・Sensor1Monitor … 140Byte
・Sensor2Monitor … 140Byte

メモリ量見積もりツールを用いて設計手法 2 のメモリ使用量を見積もった結果を以下に示す．

・タスクのメモリ使用量の合計 … 142Byte
・割り込みのメモリ使用量の合計 … 128Byte
・SensorMonitor … 142Byte

6.9.2　get_sensorvalue のメモリ使用量が不明なときの見積もり結果

設計手法 1 と設計手法において，get_sensorvalue のメモリ使用量を 5 倍にして見積もりツールによる見積もりを行う．設計手法 1 の見積もり結果を以下に示す．

・タスクのメモリ使用量の合計 … 776Byte
・割り込みのメモリ使用量の合計 … 128Byte
・Sensor1Monitor … 388Byte
・Sensor2Monitor … 388Byte

メモリ量見積もりツールを用いて設計手法 2 のメモリ使用量を見積もった結果を以下に示す．

・タスクのメモリ使用量の合計 … 390Byte
・割り込みのメモリ使用量の合計 … 128Byte
・SensorMonitor … 390Byte

6.9.3 評価

slp_tsk のメモリ使用量が不明なときのメモリ量の傾向

slp_tsk のメモリ使用量が不明としたときの見積もり結果は，不明でないときと同様の結果となった．同様になった理由としては，slp_tsk のメモリ使用量が 5 倍にしたとしても get_sensorvalue のメモリ使用量よりも小さかったためである．そのため，メモリ量の傾向も不明なときと不明でないときと同様である．

get_sensorvalue のメモリ使用量が不明なときのメモリ量の傾向

タスクのメモリ使用量の合計は，設計手法 1 は 776Byte，設計手法 2 は 390Byte であった．両者のメモリ量は約 2 倍の違いとなり，メモリ量の傾向は，不明でないときと同様の結果となった．割り込みのメモリ使用量の合計は，設計手法 1 と設計手法 2 で同じメモリ使用量となった．これも不明でないときと同様の結果となった．

考察

slp_tsk では，slp_tsk のメモリ使用量が不明なときと不明でないときで最大メモリ使用量に違いは見られなかった．かなり大きく見積もりをとったとしても，必ずしも最大メモリ使用量が変化するわけではないことがわかる．メモリ使用量の少ない関数を厳密に見積もったとしても，大きく余裕を持って見積もったときと比べてさほど影響がないので，メモリ使用量の少ない関数については厳密に見積もる必要はないとも考えられる．ただし，メモリ使用量の少ない関数であったとしても，メモリ使用量の少ない関数のネストが深く続いた場合には最大メモリ使用量に影響を及ぼす可能性があるので注意が必要である．

第7章　結論

本研究では，設計段階における最大メモリ使用量の見積もり方法の提案を目的として研究を行った．

第1章では，組み込みソフトウェアの現状について述べ，この現状から設計段階におけるメモリ使用量見積もりの必要性について述べた．

第2章では，メモリ使用量見積もりに関する一般的な手段とその問題点について述べ，設計段階の見積もりに関する関連研究について述べた．

第3章では，本研究の目的を述べた．解決した問題について述べ，想定とする設計段階はタスク分割を行う段階のあたりであることについて述べた．また，研究のアプローチとして，システムの構造を記述したクラス図とシステム全体の振る舞いを表したステートマシン図をタスク設計モデルとして与え，そのタスク設計モデルから網羅的に動的な振る舞いを調べ，最大メモリ使用量を見積もることを述べた．

第4章では，最大メモリ使用量見積もり方式の提案を行った．リアルタイムOSによって差異はあるが，一般的に考えられる最大メモリ使用量を表す式を定義し，最大メモリ使用量見積もり方式について述べた．さらに，タスク設計モデルを定義し，網羅的に調べる手段としてモデル検査技術を用いることを述べた．

第5章では，最大メモリ使用量見積もりツールの作成について述べた．最大メモリ使用量見積もりツールは，promelaファイルを生成するためのツールとそのツールを利用して実際にpromelaファイルを作成させて実行までしてメモリ見積もり量を画面に出力するためのツールから構成されることを述べた．

第6章では，タスク分割を行ったときに，作成した見積もりツールの見積もり値と実際に実装しての実測値に同じ傾向が見られるかどうかの評価を行った．タスク分割が異なる2つの設計方法を述べ，それらの設計方法に対して，作成した見積もりツールによって見積もった最大メモリ使用量と実際に実装を行い実測した最大メモリ使用量を比較した．結果は，見積もりでも実測でも同様のメモリ量の傾向が得られた．また，メモリ使用量の少なくなるだろうと思われる関数は，厳密にメモリ使用量を見積もらなくても最大メモリ使用量には影響がないことを述べた．ただし，メモリ使用量の少ない関数であったとしても，メモリ使用量の少ない関数のネストが深く続いた場合には最大メモリ使用量に影響を及ぼす可能性があるので注意が必要である．

本研究により，静的な見積もりに比べ設計段階における最大メモリ使用量のより精度の高い見積もりが可能となった．ただし，タスクと割り込みについてのみの見積もりであるので，今後の課題としては，動的に確保・解放が行われるメモリ量や必要となるキューの

上限値などの見積もりも可能にしていく必要がある．また，非機能特性に関しては，メモリ使用量だけでなく応答性や保守性などもあるのでこれに関しても，設計段階における見積もりを可能としていく必要がある．

付 録A　SPINのためのμITRON4.0ライブラリマニュアル

マニュアルの構成は以下の通り.

1. はじめに
2. ユーザ記述部分
   1. ユーザ定義部分の記述
   2. オブジェクト初期化の記述
   3. タスクの記述
   4. ランダム割り込みの記述
   5. 周期割り込みの記述
   6. 最大メモリ使用量のための記述
3. サービスコール一覧

A.1　はじめに

ライブラリファイルは，uitronlib.spin と uitronlib_header.spin と calcmem.spin の 3 つ. 主に，ユーザが変更を加える必要がある個所は uitronlib_header.spin の user define の部分である．このライブラリを用いることによって SPIN において μITRON4.0 の振る舞いをエミュレートが可能になる.

A.2　ユーザ記述部分

A.2.1　ユーザ定義部分の記述

uitronlib_header.spin における user define を書き換えることで，タスクの優先度の上限値やタスク ID の上限値などを変更できる．以下に user define 一覧を示す.

(1) タスク

PRIRANGE	タスク優先度の上限値
MAXTID	タスク ID の上限値

MAXACTQ	起動要求キューイング数の上限値
MAXSLPQ	起床要求キューイング数の上限値
MAXSUSQ	強制待ち要求ネスト数の上限値
(2) セマフォ	
TMAX_MAXSEM	セマフォの最大資源数の最大値
MAXSEMID	セマフォID の上限値
SEMQBUF	MAXTID と MAXSEMID を掛け合わせた値
(3) データキュー	
MAXDTQID	データキュー ID の上限値
MAXDTQCNT	データの個数の上限値
DTQQBUF	MAXTID と MAXDTQID を掛け合わせた値
DTQDATABUF	MAXDTQID と MAXDTQCNT を掛け合わせた値
(4) 割り込み	
MAXISRID	割り込み ID の上限値

A.2.2 オブジェクト初期化の記述

A.1 に示すものがオブジェクト初期化の最小セットである．A.2 に，タスク 1 つと割り込み 1 つの初期化例を示す．

ソースコード A.1: オブジェクト初期化の記述の最小セット

```
1  init
2  {
3          byte ercd;
4
5          atomic{
6                  enadisp = FALSE;
7                  enaisr = FALSE;
8          }
9
10         /* ここに初期化情報を記述する */
11
12         atomic{
13                 initstep = FALSE;
14                 enadisp = TRUE;
15                 enaisr = TRUE;
16                 TaskDispatch();
17         }
18 }
```

ソースコード A.2: オブジェクト初期化の記述例

```
1  init
2  {
3          byte ercd;
```

```
4
5           atomic{
6                   enadisp = FALSE;
7                   enaisr = FALSE;
8           }
9
10          /* タスク1作成 */
11          CRE_TSK( TASKID_TASK1, TA_ACT, task1, 4, ercd );
12          assert( ercd == E_OK );
13
14          /* 割り込み */
15          register_isr( ISRID_ISR1, 1 );
16
17          atomic{
18                  initstep = FALSE;
19                  enadisp = TRUE;
20                  enaisr = TRUE;
21                  TaskDispatch();
22          }
23  }
```

A.2.3 タスクの記述

A.3 に示すものがタスクの記述の最小セットである．A.4 に処理を記述した例を示す．

ソースコード A.3: タスクの記述の最小セット

```
1   proctype task2( TSKID id ) provided( _pid == runpid )
2   {
3           byte ercd;
4
5           begintsk();
6   again:
7
8           /* ここに処理を記述する */
9
10          endtsk();
11  }
```

ソースコード A.4: タスクの記述例

```
1   proctype task2( TSKID id ) provided( _pid == runpid )
2   {
3           byte ercd;
4
5           begintsk();
6   again:
7
8           sig_sem( SEMID_SEM1, ercd );
9           assert( ercd == E_OK );
```

```
10
11          endtsk();
12  }
```

A.2.4 ランダム割り込みの記述

A.5 に示すものがランダム割り込みの記述の最小セットである．A.6 に処理を記述した例を示す.

ソースコード A.5: ランダム割り込みの記述の最小セット

```
1   inline isr1( id )
2   {
3           /* ここに処理を記述する */
4   }
5
6   active proctype isr1_proc() provided( ( 0 == runisrid || enaisr ) && isrrun
        [ 0 ] )
7   {
8           byte ercd;
9           byte id = ISRID_ISR1;
10
11  again:
12          callcycisr( id, isr1, 64 );
13
14          goto again;
15  }
```

ソースコード A.6: ランダム割り込みの記述例

```
1   inline isr1( id )
2   {
3           iact_tsk( TASKID_TASK1, ercd );
4           assert( ercd == E_OK || ercd == E_QOVR );
5   }
```

A.2.5 周期割り込みの記述

A.7 に示すものが周期割り込みの記述の最小セットである．A.8 に処理を記述した例を示す.

ソースコード A.7: 周期割り込みの記述の最小セット

```
1   inline isr1( id )
2   {
3           /* ここに処理を記述する */
4   }
```

```
5
6  active proctype isr1_proc() provided( ( 0 == runisrid || enaisr ) && isrrun
       [ 0 ] )
7  {
8          byte ercd;
9          byte id = ISRID_ISR1;
10
11 again:
12         callcycisr( TRUE, isrcalled[ 0 ] = TRUE, id, isr1, 64 );
13
14         goto again;
15 }
```

ソースコード A.8: 周期割り込みの記述例

```
1  inline isr1( id )
2  {
3          iact_tsk( TASKID_TASK1, ercd );
4          assert( ercd == E_OK || ercd == E_QOVR );
5  }
```

A.2.6 最大メモリ使用量のための記述

最大メモリ使用量見積もりのために使用する関数を以下に示す.

・タスク関数用

IncreMem(id, mem)

DecreMem(id, mem)

・inline 関数用

IncreInlineMem(id, mem)

DecreInlineMem(id, mem)

タスク関数の最大メモリ使用量見積もりのための最小セットを A.9 に，inline 関数の最大メモリ使用量見積もりのための最小セットを A.10 に示す.

ソースコード A.9: タスク関数の最大メモリ使用量見積もりのための最小セット

```
1
2  c_decl{ char tskname[ TSKTYPECNT ][ MAXTSKTYPENAME ] = { "task1" }; }
3
4  #define MEMUSAGE_task1  56
5
6  proctype task1( TSKID id ) provided( _pid == runpid )
7  {
8          tsktype[ TID(id) ] = 0;
9
10         IncreMem( id, MEMUSAGE_task1 );
11
12         byte ercd;
```

```
13
14          begintsk();
15  again:
16
17          /* ここに処理を記述する */
18
19          endtsk();
20
21          DecreMem( id, MEMUSAGE_task1 );
22  }
```

ソースコード A.10: inline関数の最大メモリ使用量見積もりのための最小セット

```
1
2   inline start_timer()
3   {
4           IncreInlineMem( id, 16 );
5
6           /* ここに処理を記述する */
7
8           DecreInlineMem( id, 16 );
9   }
```

A.3 サービスコール一覧

以下にサービスコール一覧を示す.

・タスク管理機能
cre_tsk
CRE_TSK
del_tsk
act_tsk
iact_tsk
ext_tsk
exd_tsk
chg_pri

・タスク付属同期機能
slp_tsk
wup_tsk
iwup_tsk
rel_wai
irel_wai

sus_tsk
rsm_tsk

・同期・通信機能
cre_sem
del_sem
sig_sem
isig_sem
wai_sem
cre_dtq
del_dtq
snd_dtq
ipsnd_dtq
rcv_dtq

・システム状態管理機能
rot_dtq
irot_dtq
loc_cpu
iloc_cpu
unl_cpu
iunl_cpu
dis_dsp
ena_dsp

・割り込み管理機能
dis_int
ena_int

付 録B　メモリ量見積もりツールのマニュアル

マニュアルの構成は以下の通り.
1. はじめに
2. コンフィグレーションファイルの設定
3. ツールの出力結果の見方

B.1　はじめに

このツールは，タスク設計モデルに基づいて記述されている JUDE ファイルから，タスクのトータルと個々の最大メモリ使用量と割り込みのメモリ使用量を算出する．動作せるにはコンフィグレーションファイルの設定が必要となる．

B.2　コンフィグレーションファイルの設定

コンフィグレーションファイルの設定パラメータを B.1 に示す. step1_enabled 〜 step5_enabled は，0 であればそのステップをスキップし，1 であればそのステップの実行を行うというパラメータである．このパラメータは，このツールの実行環境において，SPIN が動作しなかったり，JAVA の動作が怪しい場合があるときのために使用するパラメータである．パラメータの記述例を B.2 にしめす.

ソースコード B.1: コンフィグレーションファイルの設定パラメータ

```
1  inputfilepath=<JUDEファイルのパスをここに記述する>
2  spinpath=<SPINの実行ファイルのパスをここに記述する>
3  ccpath=<Cコンパイラのパスをここに記述する>
4  workdirpath=<作業ディレクトリのパスをここに記述する>
5  maxsearchdepth=<SPINの探索の最大値をここに記述する>
6  step1_enabled=<0か1をここに記述する>
7  step2_enabled=<0か1をここに記述する>
8  step3_enabled=<0か1をここに記述する>
9  step4_enabled=<0か1をここに記述する>
10 step5_enabled=<0か1をここに記述する>
```

ソースコード B.2: コンフィグレーションファイルの記述例

```
1 inputfilepath=C:/jaist/kishilab/research/タスク分割の例題/TaskInversion/例
      題3/TTI2.jude
2 spinpath=spin
3 ccpath=gcc
4 workdirpath=C:/cygwin/home/tsuboi/spin/research/generate_test/
5 maxsearchdepth=800000
6
7 step1_enabled=0
```

B.3 ツールの出力結果の見方

ツールの出力結果はB.3 のように見る.

ソースコード B.3: ツールの出力結果

```
1 estimate result:
2 total task stack memory usage = <メモリ使用量>
3 total interrupt stack memory usage = <メモリ使用量>
4 <以下に個々のタスクスタックのメモリ使用量が出力される>
```

付 録C　promelaコードへのマッピングルールの詳細

promelaコードへのマッピングルールを以下に示す.

1. ファイルの先頭に以下を記述.
#include ”calcmem.spin”
#include ”uitronlib.spin”

2. 全てのUMLClassの各属性を，以下のようにマッピングする．このとき，nameを属性の名前，typeを属性の型，valueを属性の初期値とする.
(1) final型を持っている属性をグローバル領域に以下のように記述する.
#define name value

(2) (1)に該当しない，public型を持っている属性をグローバル領域に以下のように記述する.
・valueがemptyのとき
type name;
・valueがemptyでないとき
type name = value;

(3) (2)に該当しない属性を対応する各プロセスのローカル領域に以下のように記述する.
・valueがemptyのとき
type name;
・value
type name = value;

3. TDDTask，TDDRandomInterrupt，TDDCyclicInterrupt，TDDFunctionにそれぞれ対応するステートマシン図の振る舞いを以下のようにマッピングする．このとき，退場をoutgoing，イベントをevent，ガード条件をguard，アクションをaction，次の状態をnextとする．Outgoingは，event，guard，actionをそれぞれ0〜1個持ち，終了状態以外はnextを必ず1つ持つ．また，ある状態stateは，0個以上の退場ougoingを持つことと

する．
(1) 初期状態 state を以下のように記述する．

```
state:
   if
   :: guard_0 -> action_0; goto next_0
   :: guard_1 -> action_1; goto next_1
              …
   :: guard_i -> action_i; goto next_i
   fi;
```

(2) (1) の記述が終了した後に，初期状態と終了状態以外の状態 state を以下のように記述する．

```
state:
   if
   :: event_0 && guard_0 -> action_0; goto next_0
   :: event_1 && guard_1 -> action_1; goto next_1
              …
   :: event_i && guard_i -> action_i; goto next_i
   fi;
```

(3) (2) の記述が終了した後に，終了状態 state を以下のように最後に記述する．

```
state:
```

4. 全ての UMLClass のステレオタイプ <<Function>> の付いた各操作を，以下のようにマッピングする．このとき，name を操作の名前，arg を操作のパラメータ，mem を関数のメモリ使用量とする．
(1)

```
inlne name( arg )
{
   IncreMem( id, mem );

   ここに 3. で述べたようにステートマシンの振る舞いを記述する．

   DecreMem( id, mem );
}
```

5. 全ての TDDRandomInterrupt と TDDCyclicInterrupt に基づいてそれぞれの inline 関数を記述する．

(1) 名前を name とする．
inline name_inline(id)
{
　skip;

　ここに 3. で述べたようにステートマシンの振る舞いを記述する．

}

6. 全ての TDDTask に基づいてそれぞれのプロセスを記述する．このとき，TDDTask の名前を name，cotextsize と taskfunctionmemoryusage を足し合わせた値を mem，タスクタイプ ID を typeid とする．
(1)
proctype name(TSKID id) provided(_pid == runpid)
{
　　tsktype[TID(id)] = typeid;

　　byte ercd;

　　IncreMem(id, mem);

　　begintsk();
again:

　　ここに 3. で述べたようにステートマシンの振る舞いを記述する．

　　endtsk();

　　DecreMem(id, mem);
}

7. 全ての TDDRandomInterrupt に基づいてそれぞれのプロセスを記述する．このとき，TDDRandomInterrupt の名前を name，contextsize と interruptfunctionmemoryusage を足し合わせた値を mem，ID を isrid，ID より 1 小さい値を isrindex とする．
(1)
active proctyp name_proc()
　　provided((isrindex == runisrid || enaisr) && isrrun[isrindex])
{

```
    byte ercd;
    byte id = isrid;

again:
    callisr( id, name_inline, mem );

    goto again;
}
```

8. 全ての TDDCyclicInterrupt に基づいてそれぞれのプロセスを記述する．このとき，TDDCyclicInterrupt の名前を name，contextsize と interruptfunctionmemoryusage を足し合わせた値を mem，ID を isrid，ID より 1 小さい値を isrindex，周期割り込みインデックスを cycindex，cycindex より 1 小さい値を cycprevindex とする．

(1) cycindex が 0 のとき

```
active proctype name_proc()
                    provided( ( isrindex == runisrid || enaisr ) && isrrun[ isrindex ] )
{
    byte ercd;
    byte id = isrid;

again:
    callcycisr( TRUE, isrcalled[ 0 ] = TRUE, id, name_inline, mem );

    goto again;
}
```

(1) cycindex が 0 でないとき

```
active proctype name_proc()
                    provided( ( isrindex == runisrid || enaisr ) && isrrun[ isrindex ] )
{
    byte ercd;
    byte id = isrid;

again:
    callcycisr( isrcalled[ cycprevindex ] == TRUE,
                srcalled[ cycprevindex ] = FALSE; isrcalled[ cycindex ] = TRUE,
                id, name_inline, mem );
```

```
    goto again;
}
```

9. TDDInitproc に基づいてプロセスを記述する.

(1)

```
init
{
    byte ercd;

    atomic enadisp = FALSE; enaisr = FALSE;

    ここに 3. で述べたようにステートマシンの振る舞いを記述する.

    atomic initstep = FALSE; enadisp = TRUE; enaisr = TRUE; TaskDipatch();
}
```

謝辞

北陸先端科学技術大学院大学情報科学研究科 片山 卓也学長には，非常にお世話になりました．夏期合同研究報告会に出席して頂いて貴重な意見を頂きありがとうございました．ここに深く感謝申し上げます．

北陸先端科学技術大学院大学情報科学研究科 岸 知二教授には，研究の機会を与えて頂きありがとうございました．ソフトウェア設計に関する様々な知識，研究に対する姿勢や取り組み方を熱心に教えていただきました．ここに深く感謝申し上げます．

北陸先端科学技術大学院大学情報科学研究科 青木 利晃准教授には，非常にお世話になりました．SPIN の使用方法や，ソフトウェア検証論の講義，青木研での輪講などにおいて様々なことを教えていただきました．ここに深く感謝申し上げます．

北陸先端科学技術大学院大学情報科学研究科 デファゴ先生には，非常にお世話になりました．夏期合同研究報告会に出席して頂いて貴重な意見を頂きありがとうございました．ここに深く感謝申し上げます．

博士課程の金井 勇人氏，細合 晋太郎氏には，行き詰まったときにはいつも心良く相談に乗って頂きました．また，悩んでいるときには向こうから声を掛けて頂き励ましてもらいました．ここに深く感謝申し上げます．

同輩の朝倉 功太，小坂 浩之太氏とは，2 年間仲良く研究をすることができました．2 年間頑張れたのはこの仲間がいたからかもしれません．

参考文献

[1] IPA/SEC. 2008 年版組み込みソフトウェア産業実態調査報告書. 2008.

[2] IPA/SEC. 2006 年版組み込みソフトウェア産業実態調査報告書. 2006.

[3] Jens Bek Jorgensen, Soren Christensen, Antti-Pekka Tuovinen, and Jianli Xu. Tool support for the estimating the memory usage of mobile phone software. *Software Tools for Technology Transfer*, 2006.

[4] M. Awad, J. Kuusela, and J. Ziegler. Object-oriented technology for real-time systems. 1996.

[5] OMG. A uml profile for marte. 2008.

[6] 社団法人トロン協会. μ ITRON4.0 仕様 Ver.4.03.00. ホクエツ印刷株式会社, 2006.

[7] Gerard J.Holzman. *THE SPIN MODEL CHECKER: Primer and Reference Manual*. Addison-Wesley, 2003.