自己計測機能を含む並行プログラムの開発における計測仕様書とその応用

野 中 裕 介[†,☆]　程　京 徳[††,☆☆]　牛 島 和 夫[†††]

並行システムの信頼性を確保するためにその振舞いを監視・計測することは必要不可欠である．並行システムの全体性・不確定性といった性質から導かれる自己計測原理によると，並行システムの振舞いを正しく監視，計測するためには，システム自身が計測機能を備えていなければならない．本論文で提示する計測仕様書とは，自己計測機能を含む並行システムの系統的な開発を行うための１つの手法で用いるもので，本来の機能を果たす構成要素（機能部）の実行時情報のうち，計測の役割を持つ構成要素（計測部）が役割を果たすために必要な部分の要求を記述するものである．計測仕様書の具体例として，プログラミング言語 Ada を対象とした仕様記述法を提案した．また，計測仕様書を解釈し，計測部と機能部から自己計測機能を含む並行システムの生成を自動的に行うことによりその開発を支援するツールを実装した．また，その応用例として，一般的な並行プログラムを対象としたタスキングデッドロック検出機能およびタスク実行履歴の視覚化機能，特定の並行プログラムを対象としたバッファ監視機能のための計測仕様書および計測部の設計を示した．

Measurement Specifications and Their Applications for Development of Concurrent Self-measurement Programs

Yusuke Nonaka,[†,☆] Jingde Cheng[††,☆☆] and Kazuo Ushijima[†††]

Large-scale and highly reliable concurrent systems are more required and it is indispensable for them to keep on measuring and monitoring in order to ensure their reliability. The self-measurement principle can be deduced from wholeness and uncertainness which are behavioral characteristics of concurrent systems. According to it, any concurrent system has to include some function to measure itself for acculate monitoring and measuring. The measurement specifications are used for systematic development of concurrent systems including measurement functions. In the measurement specifications, we can describe requirements for monitoring and measuring a subject. As a concrete example of them, we decided on the specification of the measurement specifications for Ada. And we designed and implemented a tool which can understand the measurement specifications for Ada and support development of self-measurement programs based on them. As examples of practical applications, we showed sample functions for tasking deadlock detection, creating a tasking execution history, and monitoring buffering function.

1. はじめに

近年，並列処理システムや分散システムのような，大規模な並行システムに対する要求が増大している．社会基盤となっているソフトウェアシステムにも並行システムが多く用いられており，現代社会の信頼性は並行システムの信頼性に依存している．しかしながら，現在実用とされている並行システムの多くは高い信頼性を持っているとはいえないため，それらに依存している社会基盤を，信頼性が高いと認めることができな

† 九州大学
Kyushu University
†† 埼玉大学
Saitama University
††† 九州システム情報技術研究所
Institute of Systems and Information Technologies/KYUSHU
☆ 現在，九州大学大学院システム情報科学研究科
Presently with Graduate School of Information Science and Electrical Engineering, Kyushu University
☆☆ 現在，埼玉大学大学院理工学研究科
Presently with Graduate School of Science and Engineering, Saitama University

本研究の一部は文部科学省科学研究費萌芽的研究（2）課題番号 13878057「自己計測機能を持つ並行処理ソフトウェアの系統的開発法に関する研究」（平成 13 年度）の補助を受けて行った．

い．社会基盤の信頼性を高めるためにも，信頼性の高い並行システムをどのように設計・開発・保守するかということは，非常に重要な問題である．

並行システムの振舞いに関する最も本質的な特徴は，その全体性である．すなわち，並行システムの振舞いは，並行に動作する構成要素の単なる集合ではなく，機械的に分解して元に戻しても同じ振舞いを再現することのできない「全体」である．これが並行システムにおける全体性原理[5]である．

全体性原理から，並行システムの計測や監視における不確定性原理[5]を導くことができる．すなわち，実行時モニタのような観測者の振舞いは，その対象となる並行システムの振舞いと切り離して考えることはできない，ということである．ここで，システムの振舞いの計測とは，何らかの方法でシステム中の特定の対象の属性を検出し，明確に定義された法則によって属性に対して数値あるいは記号を割り当てることにより，システムの実行時情報をとらえることをいう．また，システムの振舞いの監視とは，計測によってとらえられた実行時情報を収集，報告することをいう．

現存するほぼすべての並行システムは，その機能のみが重視され，分離計測方式に基づいて設計，開発，保守されている．ここでいう分離計測方式とは，システムが，システムに要求されている本来の機能を果たすいくつかの構成要素（以下，機能部）のみで構成されていて，システム内には，計測のために恒久的に存在する構成要素（以下，計測部）が含まれていないことをいう．結果として，並行システムの振舞いを計測・監視する必要があるときには，対象システムとは独立に存在する，別の計測/監視ツール（たとえば，デバッガ）を用いなければならない．しかし，この方法で，対象システムの正しい振舞いを知ることは不可能である．機能重視である分離計測方式における最も深刻な問題は，この方式に基づいて設計，開発，保守された並行システムについて，たとえその開発，保守段階でテストおよびデバッグがなされたとしても，運用段階での信頼性を完全に保証することは不可能である，ということである．すなわち，並行システムの計測，監視が開発中のある時点でのみ行われて，その後システムが運用に入る際に計測部が取り除かれたとしたら，不確定性原理により，開発時に得たシステムに関する情報は運用時のものと異なる情報となり，そのような情報に基づいて開発時に確保した信頼性は運用時には失われてしまうのである．

この問題の解決のためには，自己計測原理[4]に従って並行システムの開発を行わなければならない．自己計測原理は，信頼性の高いシステムはいくつかの機能部と，それらと並行に動作して，ある要求に従ってシステム自身を計測，監視し，システムの振舞いに関する実行時情報をシステムの外部に伝える，いくつかの（あるいは 1 つの）恒久的な計測部（自己計測のためのコンポーネント）で構成されなければならないということをいっている．

自己計測原理に従い，ソフトウェアシステムとしての自己計測機能を含む並行システム（以下，自己計測並行システム）を開発するためには，自己計測並行システムを構成する並行プログラムに自己計測機能を持たせなければならない．以下，自己計測機能を持つ並行プログラムのことを自己計測並行プログラムと呼ぶ．

プログラミング言語や開発支援ツール，実行時環境を含んだ，従来のソフトウェア開発環境およびソフトウェア開発手法は，自己計測並行プログラムの開発にそのまま利用すると様々な問題が生じる．そこで，自己計測並行プログラムの開発に適した開発環境，系統的な開発手法の確立が本研究の目的である[13),14]．

発生する問題の一例として，計測部の再利用の問題がある．計測部の再利用が重要である理由の 1 つとしては，計測部自身が高い信頼性を持っている必要がある，ということである．機能部の動的な振舞いに関する正しさは，その機能部のための計測部が保証するが，その計測部の動的な振舞いに関する正しさを保証するものはない．ソフトウェアの高い信頼性を確保するための手段の 1 つとして，ソフトウェア部品の再利用をあげることができる．すなわち，高い信頼性を持つ既存の計測部を新たな機能部のために用いることで，計測部の信頼性の問題を解決することができる．

信頼性の問題以外にも，計測部の再利用による利益はある．同じ領域を対象とした 2 つの異なるシステムについて，計測に対する要求が，同じか，あるいは似たものになることがある．また，動的なデッドロック検出のように，並行プログラム一般の問題として，すべての並行プログラムに有効な種類の計測も存在するので，一般的な要求のための計測部を再利用することはソフトウェア開発の効率化のために有効である．

本論文の構成を示す．2 章では，ソフトウェア開発環境の中から，開発支援ツールを取り上げ，自己計測並行プログラムの系統的な開発を支援する手法を提案し，3 章で，その手法のプログラミング言語 Ada[1),10]への適用を示す．4 章では自己計測並行プログラムの開発支援ツールの実装についての概要を説明する．5 章では，3 つの応用例を用いて，開発支援ツールによる自己計測並行プログラム開発をいかにして行うべき

なのかという実例を示し，6 章では結論を述べる．

2. 計測部の再利用を実現するための計測仕様書

本論文で提案する自己計測並行プログラム開発支援ツールについて，その基盤となる概念である計測仕様書[13]について説明する．計測部は，計測によって得られた実行時情報を受け取り，その情報を加工して，あるいはそのままシステムの外部に伝える役割（監視）を行うプログラムである．したがって，それぞれの計測部には，機能部の動的な振舞いについての情報の中でどの情報が必要なのかに関する要求がある．計測仕様書とは，機能部の実行時情報についての計測部からの要求を記述したものである．

図 1 は，計測仕様書に基づく並行プログラム開発支援ツール（以下，開発支援ツール）についての概略を示している．自己計測並行プログラムに含まれている計測部は，対応する計測仕様書を持っている．この計測仕様書には，機能部が実行されている間に，その計測部が計測処理のためにどの実行時情報を必要とするかということが記述してある．そして，センサとは，機能部から計測部へ実行時情報を送信するための，プログラム中の命令文である．開発支援ツールを用いる場合であれば，機能部内のセンサは，機能部を担当している開発者が記述する必要はない．開発者はセンサについて考慮しなくても，センサの位置や渡すべき情報は計測仕様書によって機械的に決定することができるので，開発支援ツールを用いて，センサを含む機能部を自動生成することができる．生成された，センサを含む機能部と計測部をリンクすることにより，自己計測機能を含んだ並行プログラムを生成することができる．

すなわち，計測仕様書とは，従来の開発環境を用いて自己計測並行プログラムの開発を行おうとした際に，計測のためのプログラムの一部でありながら，機能部の方に含まれざるをえなかったセンサを分離して，再利用可能な形にまとめたものである．この分離によって，計測のためのプログラム部品である計測部と計測仕様書を，別のシステムに再利用することができる．デバッガのような，機能部に関する振舞いを，計測部の要求に関係なく可能な限りすべてを集めるような方法をとれば，計測仕様書は必要ないかもしれない．しかし，デバッグを行うときのみという，デバッガの用いられる場面と違って，自己計測システムを構成する並行プログラムなので，計測という行為は運用時も含めて恒久的に続く．したがって，計測による性能低下

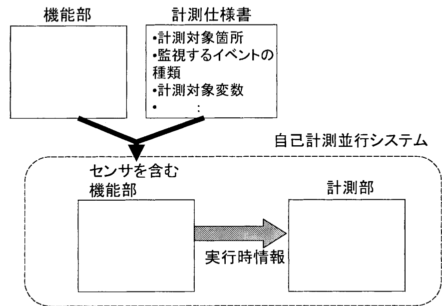

図 1 計測仕様書に基づく自己計測並行プログラム開発支援ツールの概略

Fig. 1 The architecture of the supporting tool with measurement specifications.

を無視して考えることはできない．性能の低下を抑えるためにも，個々のアプリケーションの機能部にとって必要最小限なセンサを用いて計測を行いたい．そのためにも，そのアプリケーションの信頼性を確保するにあたって必要な最小限の計測要求を指定できる仕組みが必要であり，計測仕様書はその役割を果たす．

3. Ada のための計測仕様書

現在我々が考えている計測仕様書は，計測部および機能部を記述する特定のプログラミング言語に依存した仕様を持つ．ここでは，計測仕様書の具体的な例を示すために，プログラミング言語 Ada 95[10] のための計測仕様書についての詳細を説明する．Ada 95（以下，単に Ada）は，オブジェクト指向並行プログラミング言語であり，高い信頼性が要求されるシステムの開発に多く用いられている．以下，計測仕様書とは，Ada のための計測仕様書のことを指す．

計測部が必要とする機能部の実行時情報についての，計測仕様書に記述することができる要求は以下のとおりである．

- 計測部はどの種類のタスキングイベントの発生を監視するか．
- それぞれのタスキングイベントについて，どのスコープの範囲での発生を監視するか．
- サブプログラム呼び出しの発生について，監視対象となるサブプログラム群．
- エントリ呼び出しの発生について，監視対象となるエントリ群．
- 定義および参照がなされたかどうかの監視を行う変数群．
- 監視イベント発生時に，機能部から計測部に提供される情報（例：イベントが発生したタスクの識別子（タスク ID），変数の値，イベントが発生し

た場所（スコープ情報））.

- イベント発生を通知する条件となる論理式.

ここでいうタスクIDとは，Ada Systems Programming Annexの Task_Identification パッケージで定義されている，走行中のタスクを一意に識別する Task_ID 型の識別子のことである[10]．

計測仕様書の文法としては，Adaのパッケージ仕様と同じ文法を採用した．これは，機能部から実行時情報を受け取るインタフェースとなる部分に相当するパッケージの仕様部として計測仕様書をそのまま利用できるようにするためである．また，計測仕様書の字句解析，構文解析を行うために新たにパーサを用意する必要がなく，既存のAdaパーサをそのまま利用することができる．以下に示すプログラムは，計測仕様書として利用できるパッケージ仕様部の例である．Adaのパッケージ仕様部を計測仕様書として利用する場合，サブプログラム宣言およびプラグマのみ仕様部内に列挙することができるという文法的な制限がある．

```
package MS_Example is
   procedure RENDEZVOUS_START
             (Caller_Task : Task_ID);
   procedure RENDEZVOUS_END
             (Caller_Task : Task_ID);
end MS_Example;
```

列挙されたサブプログラム宣言におけるサブプログラム名は，計測部が計測を要求するタスキングイベントの種類を表す．ある種類のタスキングイベントを計測したいという要求があれば，その種類に対応する名称のサブプログラムを計測仕様書に宣言し，計測部内に本体を実装すればよい．計測仕様書中に，要求する計測の種類を示すために記述するこのようなサブプログラム宣言文のことをクエリと呼ぶ．タスキングイベントの種類とサブプログラム名との対応関係は仕様記述法として定められており，たとえば，エントリ呼び出しによるタスク間のランデブが開始されたときに，計測部がそのことを知り何らかの処理を行いたければ，RENDEZVOUS_START という名称のサブプログラムを計測仕様書となるパッケージ仕様内に宣言すればよい．

クエリは，現状では42種類が使用可能である．これは，上記の，計測部が必要とする機能部の実行時情報への要求を表現できるように揃えた数である．**表1**に，クエリ一覧のうちの一部を示す．紙面の都合でそのすべての詳細を説明することはできない．

クエリとなっている各サブプログラム宣言における引数は，クエリに対応するイベントの発生時に，機能部から計測部にどのような実行時情報を渡すのかを決定する．実行時情報の種類と引数名との対応関係は仕様記述法として定められており，たとえば，RENDEZVOUS_START によって計測される，ランデブ開始イベントのクエリに含まれる引数として Task_ID 型の Caller_Task という名称の引数が指定されていれば，ランデブの契機となったエントリ呼び出しの，呼び出し元タスクのタスクIDを計測部が機能部から受け取ることを意味する．

表1 計測仕様書におけるクエリ（抜粋）
Table 1 Some of queries for the measurement specifications.

クエリ	説明
function BLOCK_ELABORATION_START	ブロック宣言の仕上げの開始
function BLOCK_ELABORATION_COMPLETION	ブロック宣言の仕上げの完了
procedure BLOCK_EXECUTION_START	ブロックの実行開始
procedure BLOCK_EXECUTION_COMPLETION	ブロックの実行完了
procedure SIMPLE_ENTRY_CALL	単純エントリ呼び出しの直前
procedure PROTECTED_ENTRY_CALL	プロテクト付きエントリ呼び出しの直前
function TASK_ACTIVATION_START	タスクの活性化開始
procedure ACCEPT_START	accept 文の直前
procedure RENDEZVOUS_START	ランデブの開始
procedure RENDEZVOUS_END	ランデブの完了
procedure FUNCTION_CALL	関数呼び出しの直前

クエリに対して補助的な役割を果たす，特別なプラグマ（計測仕様書のために導入したプラグマ．Adaの標準ではない）を用いることができる．そのような特別なプラグマをクエリに付加することによって，そのクエリに以下のような補助的な要求を加えることができる．

- タスキングイベントが発生した際に，そのイベントの種類に対応するクエリがそのイベントを取り扱う，機能部内の範囲（スコープ）.
- タスキングイベントが発生した際に，そのイベントの種類に対応するクエリが実行されるための条件となる論理式.
- サブプログラム呼び出しに関わるイベントについて，クエリが取り扱うサブプログラムの集合.
- エントリ呼び出しに関わるイベントについて，クエリが取り扱うエントリの集合.
- 変数の定義/参照に関わるイベントについて，ク

表 2 計測仕様書におけるプラグマ
Table 2 Pragmas for the measurement specifications.

プラグマ
Designate_Subject(Scope_Name1, ...) 後に続くクエリの有効範囲を指定する．引数はドット表記で，複数を指定した場合には後方の引数の方が優先する．
Designate_Entry(Entry_Name1, ...) 後に続く，エントリ呼び出しに関するクエリの，対象となるエントリを指定する．
Designate_Subprogram(Subprogram_Name1, ...) 後に続く，サブプログラム呼び出しに関するクエリの，対象となるサブプログラムを指定する．
Designate_Condition(Condition_Expression1, Block_Name1, ...) 後に続くクエリに対して，有効となる条件式を付加する．
Variable_Refer(Variable_Name, [Scope_Name1, ...]) 直後に宣言されるサブプログラムを，Variable_Name によって指定された識別子の変数が参照されたというイベントを監視するクエリにする
Variable_Assign(Variable_Name, [Scope_Name1, ...]) 直後に宣言されるサブプログラムを，Variable_Name によって指定された識別子の変数が定義されたというイベントを監視するクエリにする

エリが取り扱う変数の集合．

表 2 は，特別なプラグマの一覧である．変数の定義/参照に関わるイベントに関しては特別で，たとえば，特定の変数が定義されたときに呼び出されるクエリは，対象となる変数名を指定する Variable_Refer プラグマの直後に宣言されたサブプログラム定義である．すなわち，変数ごとにクエリを設定することができる．変数の定義/参照に関わるイベント以外のクエリの場合，イベントの発生した場所（スコープ）によって異なる処理を行う必要がある場合には，クエリの本体内で条件分岐をする必要がある．

クエリの中には，特定の用途に用いるための特殊なものがある．Get_CDT クエリがその 1 つである．CDT（Communication Dependence Task set）とは，あるタスクのあるエントリに関して，そのエントリを呼び出す可能性のあるタスクの集合である．これは，たとえば，動的なデッドロック検出のような目的のために必要になる集合であるが，CDT を求めるためには，ソースコードの静的な解析が必要となる．

プラグマを付加したクエリおよび引数の具体例をあげる．“main” というスコープ（手続き）に含まれている，エントリ “main.task1.get” への呼び出しの発生を計測したいとする．その際，呼び出し先（ここでは task1.get）のタスク ID を知る必要があるとするならば，計測仕様書内に以下のようなクエリを記述する．

```
pragma Designate_Entry("main.task1.get");
pragma Designate_Subject("main*");
procedure Simple_Entry_Call
  (Callee_Task : in Task_ID);
```

あるプラグマの有効範囲は，プラグマの直後に宣言されたクエリから，同種のプラグマが現れる直前のクエリまでである．クエリの有効範囲を指定する種類のプラグマは，スコープを表す引数としてドット表記を用いる．2 つ以上のスコープをクエリの有効範囲として指定することができる．また，正規表現によってクエリの有効範囲を指定することもできる．

計測仕様書と機能部から，センサを含んだ機能部を生成する作業を自動的に行うツールを実装するためには，計測仕様書の解析と理解，およびそれに基づいた機能部のプログラムの変換を実現すればよい．

4. Ada のための自己計測並行プログラム開発支援ツール

これまでに示した仕様記述法に基づいて，我々は，Ada プログラムのための，計測仕様書に基づく自己計測並行プログラム開発支援ツール（以下，本ツール）の設計および実装を行った．**図 2** は，本ツールの構造を示している．

我々は，本ツールにおいて，ソースコード変換アプローチを採用した．それは，ツールが計測仕様書の解析および理解を行った後で，解析結果を用いて，計測対象となる機能部のソースコードを変換して，センサを含んだ機能部のソースコードを出力する，という方法である．

ほかに，実行時環境に手を入れて，機能部から情報を得るという方法も考えることができるが，それと比較して，ソースコード変換アプローチの利点は，すべての計測対象が静的に決定することによる性能低下の抑制をあげることができる．もし，自己計測並行プログラムが，自己計測を行うための特別な実行時環境の上で動作し，計測対象が動的に決まるのであれば，すべての要素が計測可能であるように準備しておいたことによるオーバヘッドは無視できない．

もう 1 つの利点は，ソースコード変換ツールであれば，特定の処理系に依存せず，Ada プログラムの処理系であればどれにでも適用することができるということである．

逆に，欠点としては，ソースコード変換によって実現できる計測の種類が，対象言語の表現力の範囲に限定されてしまうことである．さらに，実行時環境内の記憶領域と計測部の記憶領域で，取り扱う実行時情報の重複が避けられない可能性がある．

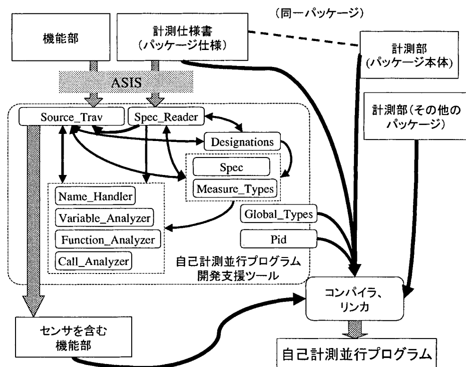

図 2 計測仕様書に基づく自己計測並行プログラム開発支援ツールの構造

Fig. 2 The supporting tool for development of self-measurement systems.

図 2 に示した自己計測並行プログラム開発支援ツールの入力情報は，機能部と計測仕様書である．前者は一般的な Ada 並行プログラムのソースコード，後者は Ada パッケージ仕様部のソースコードである．出力情報は，入力された，機能部としての並行プログラムに計測機能を加えたプログラムのソースコードである．出力情報に加えて，入力情報として用いた計測仕様書パッケージ（仕様部，本体部），および，必要に応じて用意する，実行時情報の処理を行う計測部のパッケージを組み合わせることで，自己計測並行プログラムがコンパイル可能となる．

図 2 中の自己計測並行プログラム開発支援ツールの内部構造は，機能分担ごとのパッケージに分割されている．Spec_Reader パッケージは，入力された計測仕様書を解析して，利用しやすい内部表現に変換して保持する．Designations パッケージは，計測仕様書の記述のうち，計測範囲を指定するプラグマの引数となる文字列（正規表現）の解釈を行う．これらの情報は，機能部のプログラムを入力情報として，実際に変換作業を行う Source_Trav パッケージの要求に応じて提供される．Spec および Measure_Types パッケージは，仕様記述法を定義する部分である．Measure_Types パッケージは，クエリの型を定義する．また，Spec パッケージは，各クエリの型に付随する情報のほかに，プラグマの動作を決定する手続きが含まれている．Name_Handler，Variable_Analyzer，Function_Analyzer，Call_Analyzer は，スコープの識別子の管理，変数の定義/参照の検索，関数の検索，呼び出し関係の解析といった，Source_Trav パッケージの補助となるパッケージ群である．Global_Types，Pid パッケージは，本ツールと，変換結果として出力される自己計測並行プログラムとで共通に用いられるパッケージである．この中に，計測部が受け取るデータの型が定義されている．

本ツールでは，Source_Trav，Spec_Reader パッケージが Ada のソースコードから情報を得るために，字句解析，構文解析をツール自身が行うのではなく，ASIS[6),7),11)] を用いている．ASIS は，各 Ada コンパイラが中間表現として保持する，Ada プログラムの構文/意味情報にアクセスするためのインタフェース規格である．Source_Trav パッケージは，Ada コンパイラ Gnat[16)] のための ASIS ライブラリである ASIS-for-GNAT[15)] の使用例として付属している “display source” プログラムに含まれている 1 つのパッケージを基にして作成した．

変数の定義/参照に関わるクエリを対象とするソースコード変換については，競合状態についての特別の配慮が必要である．計測対象となった変数は，Ada において共有変数の排他制御を実現する言語機能であるプロテクト付き型オブジェクトに変換することによって，競合状態から解放している．たとえば，Main 手続きで宣言されている Integer 型の変数 A が更新されるたびにその値を取得したいのであれば，計測仕様書には，以下のように記述する．

```
pragma Variable_Assign("main.a");
procedure Get_A(Target : Integer);
```

このクエリが指定された場合，A の値を定義する命令は，以下のように変換される．

(変換前) `A := 5;`

(変換後) `VAR_00000.Mutex.Set(5);`

Mutex はプロテクト付き型オブジェクトであり，Set は Mutex に含まれる手続きの 1 つである．手続き Set は，A に 5 を代入するだけではなく，手続き Get_A の呼び出しを行って，計測部に更新された A の値を渡すことも行う．この一連の操作は Mutex によって相互排除が行われるので，競合状態に陥る，すなわち，A の更新と A の値の伝達が独立に行われることがなくなり，その局面において正しい値を伝えることができる．VAR_00000 は Mutex を宣言しているパッケージ名で，複数の変数が扱えるように変数ごとにユニークなパッケージが変換後のソースコードに自動生成される．

5. 計測仕様書の応用

本章では，計測仕様書を用いた自己計測並行プログラム開発の実例を示す．

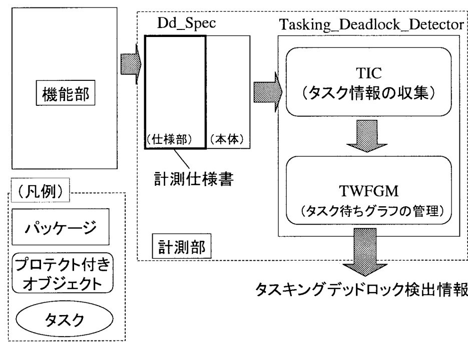

図 3　タスキングデッドロック検出機能
Fig. 3　Tasking deadlock detector.

5.1　タスキングデッドロック検出

並行プログラムにおいて，デッドロックの発生は，信頼性を低下させる深刻な問題である．この例では，実行時に，デッドロックの原因となるタスク間の同期待ち関係の監視を行い，デッドロックが発生する直前にそれを検出することを目的とする．

動的なデッドロックの検出方法として，タスク待ちグラフに基づくデッドロック検出法[3),12)]を用いる．この方法では，Ada 95[10)]プログラムにおいて発生する危険性のあるすべての種類のデッドロックを検出することができる．タスク待ちグラフとは，タスキングオブジェクトを示す節点と，タスキングオブジェクト間の同期待ち関係を示す枝からなる有向グラフである．

ある並行プログラムのある実行時状態の同期待ち関係を表すタスク待ちグラフ内にサイクルが存在するということは，その状態においてデッドロックを生じているタスク群がプログラム内に存在することを意味する．ここでいうサイクルについての詳細は，文献 3)を参照されたい．

Adaプログラムの実行時に，タスク間の同期待ち関係を表現する形式的表現としてタスク待ちグラフを用いることにより，タスキングイベントの監視，タスク待ちグラフの生成・管理，タスク待ちグラフ内のサイクルの検出・報告を行えば，タスキングデッドロック検出機能が成立する．

図 3 は，タスキングデッドロック検出機能のための計測部および計測仕様書の構成を示している．計測部を構成するパッケージのうち，Dd_Spec パッケージの仕様部が計測仕様書に相当し，タスク待ちグラフの生成・管理に必要な実行時情報の要求が記述してある．この計測仕様書は，特定の機能部に依存するものではなく，すべての並行プログラムに対して適用することができる．

Dd_Spec パッケージの本体には，機能部から受け取った実行時情報を，タスキングデッドロック検出のための他のパッケージ（後述の Tasking_Deadlock_Detector）に伝える役割をするコードが記述してある．一般的に，計測仕様書の再利用性を考えて，計測仕様書となるパッケージの本体には，単に他のパッケージに対する呼び出し文のみが記述されるのが好ましい．

Tasking_Deadlock_Detector がタスキングデッドロック検出機能の中心的役割をなすパッケージである．このパッケージの主な構成要素は，2 つのプロテクト付き型オブジェクト（TIC（Task Information Collector），TWFGM（Task-Wait-For Graph Manager））である．

TIC は，機能部で発生したタスキングイベントを TWFGM に渡す役割をしているが，同時に 1 つのタスキングイベントしか渡さないための相互排除を行っている．並行プログラムを対象とする計測では，複数のタスキングイベントが同時に発生することがあり，計測部の設計においてもそのことを留意しておく必要がある．特に，タスキングデッドロック検出という目的のためにはタスキングイベントの発生順序が重要な意味を持ち[2)]，TIC は，これらの順序を保存する役割を持つ．

TWFGM は，タスク待ちグラフを保有し，TIC から受け取ったタスキングイベントの情報に基づいてタスク待ちグラフの変更を行う．その際，変更によってサイクルができる場合であれば，直後にデッドロックが発生することをユーザに通知する．各タスキングイベントが計測部に通知されるのは発生する直前であるため，デッドロックが発生する直前にそのことを検出できるのである．

デッドロック検出のための計測部のソースコードは約 3,500 行であった．また，本パッケージを用いることによる性能低下については，対象プログラムにおいてタスキングイベントの発生する頻度によって異なってくる．たとえば，10 個のタスクを用いて並列にクイックソートを行うプログラムの場合，タスキングデッドロック検出機能を加えると 1.3 倍の実行時間を要するようになった．性能の低下を抑えるためには，それぞれの対象プログラムに合わせて計測仕様書を修正してデッドロックの発生に影響しないことが静的に保証できるタスキングイベントを計測しないようにすべきである．

5.2　タスク実行履歴の視覚化

複数のタスクが同時に走行する，複雑な制御の流れを持った並行プログラムの動作を理解するにあたって，実行時状態を視覚的に認識することは非常に効果的で

図 4 ヒストリビューの例
Fig. 4 An example of history view.

ある．

並行プログラムの実行時状態を様々な視点で視覚化するモデルが提案されているが[8),17),18)]，本研究における例題として，ヒストリビューの実行時生成ツールの実装を行った．ヒストリビューとは，スレッドの実行履歴の表現である．これは Gthread[18)] に含まれる出力結果のうちの 1 つである．Gthread とは，Pthread[9)] ライブラリを用いた並行プログラムの実行時に，その振舞いに関する履歴のデータを収集し，それに基づいて対象プログラムの振舞いを視覚的に再現するツールである．

Gthread は，実行時には単に各イベントにタイムスタンプの付いた振舞いに関する履歴のデータを収集し，事後にそのデータを入力として各種の視覚化を行うツールであるのに対して，本研究の例題として用いたヒストリビュー生成ツールは，対象プログラムの実行中に，実時間で視覚化を行う．それゆえに，表現方法に若干の相違がある．それは，Gthread では時間軸をユーザが操作することによって，視覚化する時刻を指定することができるのに対して，本例題では，表示できるのは，対象プログラムの実行中の時点と同じ状態および一定の過去の状態である．

図 4 は，ヒストリビューの例である．これは，本例題の出力結果である．図中の帯グラフの各行は，実行中のプログラムの各タスクに対応している．横軸は時間軸で，左から右に向かっている．すなわち，右端は現在の状態である．各タスクに対応する帯グラフの色が変化しているのは，それぞれ特定のサブプログラムを実行していることを意味している．タスク間の垂直な矢印は，タスクの生成および消滅を意味している．タスク生成の場合，矢印の始点が親タスクであり，終点が新しく生成されたタスクを表す．タスク消滅の場合，矢印の始点が消滅するタスクであり，終点が親タスクである．

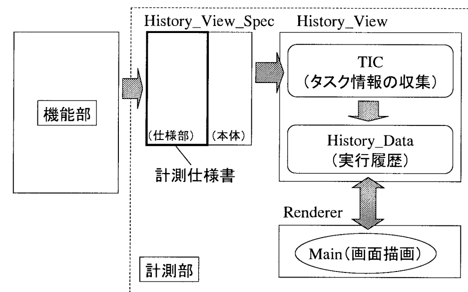

図 5 ヒストリビュー生成機能
Fig. 5 History view visualizer.

図 5 は，ヒストリビュー生成機能のための計測部および計測仕様書の構成を示している（凡例は図 3 参照）．計測部を構成するパッケージのうち，History_View_Spec パッケージの仕様部が計測仕様書に相当する．

図 6 は，History_View_Spec 計測仕様書である．たとえば，(β) の Task_Activation_Start 関数の宣言は，タスクの活性化が開始したという情報を受け取るためのクエリである．タスクが生成された際には，親タスクから新しいタスクに向けて矢印を引くため，Parent_Task 引数（親タスクのタスク ID）を要求している．

Designate_Subject プラグマは，イベント発生を監視する範囲を選択するためのプラグマである．(α) は，Task_Activation_Start クエリから Block_Execution_Completion クエリまでの範囲で有効であり，この場合，引数の “all” ですべてのブロックが範囲であることを指定した後，“-map” “-map.busstop.view” のように先頭にマイナス記号をつけた文字列の引数によって，例外としてイベント発生の監視を行わない範囲を決定する．(γ) は，環境タスクによって呼び出される手続きの場合のみ監視するイベントの範囲を指定している．これらの 2 カ所を，対象プログラムによって変更する．

History_View_Spec パッケージの本体には，後述の History_View パッケージへ，機能部から得られた実行時情報を伝える役割がある．また，タスキングデッドロック検出機能と同様に，History_View パッケージにはプロテクト付き型オブジェクト TIC が含まれており，同時に 2 つ以上のタスキングイベントが処理されないことを保証することによって，イベントの順序が保存されるようにしている．

ヒストリビュー生成機能の適用例として，バス運航

```
-- History View 生成のための 計測仕様書
with Ada.Task_Identification; use Ada.Task_Identification;
with Global_Types; use Global_Types;
with Ada.Calendar; use Ada.Calendar;
with Renderer;
package History_View_Spec is
   pragma Designate_Subject("all", "-map",
     "-map.busstop.view"); …(α)
   function Task_Activation_Start(Now : Time;
     Parent_Task : Task_ID; Block_Name : String)
     return Boolean; …(β)
   procedure Procedure_Call(Now : Time;
     Full_Subprogram_Name : String);
   procedure Procedure_Call_Completion(Now : Time);
   procedure Function_Call(Now : Time;
     Full_Subprogram_Name : String);
   procedure Function_Call_Completion(Now : Time);
   procedure Protected_Procedure_Call(Now : Time;
     Full_Subprogram_Name : String);
   procedure Protected_Procedure_Call_Completion
     (Now : Time);
   procedure Protected_Function_Call(Now : Time;
     Full_Subprogram_Name : String);
   procedure Protected_Function_Call_Completion
     (Now : Time);
   procedure Protected_Entry_Call(Now : Time;
     Full_Subprogram_Name : String);
   procedure Protected_Entry_Call_Completion(Now : Time);
   procedure Block_Execution_Completion(Now : Time;
     This_Unit_Class : Unit_Class);
   -- 以下の引数は main procedure に応じて変更
   pragma Designate_Subject("main"); …(γ)
   function Block_Elaboration_Start(Now : Time;
     Block_Name : String) return Boolean;
end History_View_Spec;
```

図 6 History_View_Spec 計測仕様書
Fig. 6 The measurement specifications of History_View_Spec.

システムのシミュレータを用いた例を紹介する．このプログラムは，各バスに対応する 20 個のタスクが運航状況をサーバに伝え，各停留所では，サーバからバスの接近情報を知ることができるシステムをシミュレートするものである．このプログラムに対して 0.5 秒間隔でヒストリビューーを生成した場合，80.54 秒の実行時間のうち，4.2% (3.40 秒) がヒストリビュー生成機能のためのプロセッサによる処理時間であった．いずれの数値も，5 回の試行による平均値である．実行中に発生した，計測対象となったイベントの回数は 4552 回で，毎秒 56.5 回であった．ヒストリビュー生成機能に関しても，対象プログラムの性能に与える影響はイベントの発生頻度によって異なる．

計測仕様書に基づく自己計測並行プログラム開発支援ツールを用いる利点の 1 つとして，計測のための多様なツールが個別に実行時情報の収集のための機構を持つ場合と比べて，コーディング量という観点から見た，計測部の開発者の負担が少ないことがある．これは，字句・構文解析のような，ソースコードの静的解析が必要なツールが共通に行う部分を記述する手間が省けるという ASIS の利点と同様である．本例題の計測仕様書を用いたヒストリビュー生成機能を実装するために要した計測部のソースコードは，わずか 700 行弱であった．

5.3 バッファ監視

これは，前述の 2 例とは異なり，特定のアプリケーションで発生する問題を対象とした計測部の例である．ここでは，有限サイズのバッファを持ち，バッファのオーバフローによって破綻を招くという問題をかかえている例題を取り扱う．バッファサイズの決定について，オーバフローを起こさない十分なサイズを事前に予測することが不可能である場合に，運用時に破綻が起こる前にその危険性を指摘することによって，システムの安全な停止などの措置を講じることができる．

本例題の対象プログラムとしては，オンラインオークションシステムのサーバプログラム（以下，オークションサーバ）を取り上げた．入札者はクライアントとして，オークションサーバに対して金額を送信して入札を行う．オークションサーバでは，クライアントからの要求は，いったん有限サイズのバッファに格納し，別のタスクがバッファから要求を取り出して金額を検査し，最高値の更新を行うという処理を行う．クライアントからの入札が一時に集中し，サーバの処理能力を超えれば，バッファ内に待っている入札が増加し，バッファのサイズを超える数になったときオーバフローを起こし，システムは破綻をきたす．

図 7 は，オークションサーバの構成を示している．オークションサーバの機能部を構成する主なパッケージは，バッファの管理を行う Buffer パッケージと，バッファから要求を取り出して処理を行う Server パッケージである．計測部を構成する主なパッケージは，Buffer パッケージの計測を担当する Ms_Buffer パッケージと，Server パッケージの計測を担当する Ms_Server パッケージ，および計測した情報を管理する Ms_Info パッケージである．Ms_Buffer パッケージおよび Ms_Server パッケージの仕様部はそれぞれ計測仕様書の役割を持っている．すなわち，機能部のパッケージごとに異なる計測仕様書を用いている例である．

機能部に含まれるバッファ操作に関するサブプログラムの開始，終了というイベントの発生時刻を知る，

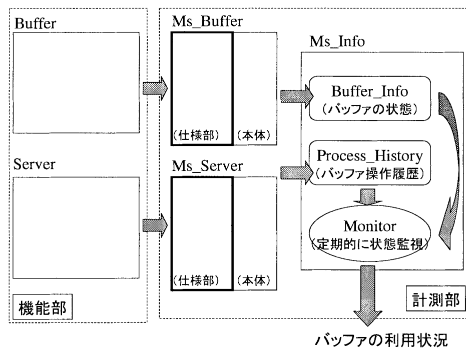

図 7 バッファ監視機能
Fig. 7 Monitor for buffer.

という要求を各計測仕様書が持つ．これによりバッファ内に含まれるデータ数の管理，およびバッファに含まれるデータを 1 つ処理するのに必要としている時間，バッファに入札が格納される間隔（頻度）を知ることができ，計測部は，過去にバッファに含まれていたデータ数の最大値，過去の一定量の記録に基づいたサーバの利用率を定期的に出力する．

この計測部を別の対象プログラム（機能部）に対して再利用する場合には，計測仕様書の Designate_Subject プラグマによって指定されている，バッファ操作に関するサブプログラムの識別子と，Ms_Buffer パッケージの本体に含まれている，開始したという通知を受けたサブプログラムがバッファに対する出し入れのどちらの操作なのかを判定するための識別子を変更すればよい．

この計測部のソースコードは約 200 行であり，計測による負荷は処理の負荷に比べて十分小さく，測定しても有意な差として現れない程度であった．

6. おわりに

本論文では，自己計測並行プログラムの開発を系統的に行うための 1 つのアプローチとして，開発支援ツールを用いる方法を示した．その中心となる考え方は，計測部がその役割を果たすために必要な計測仕様書を用いることである．ここには，機能部の実行時情報の種類や計測対象範囲が記述される．計測部と機能部をそれぞれプログラム部品として考える際に，機能部から計測部に情報を渡すという処理に関するプログラムは，計測部の要求に基づくものであり，それらのプログラムは，本質的には機能部という部位に含まれない形で存在するのが自然である．このような，プログラムの構造という観点以外にも，計測による性能低下の抑制や開発者の負担軽減という実用的な利点も備えている．また，実際の自己計測並行システムの開発において，計測仕様書を用いた計測部と機能部がどのようにシステムを構成するのかを示すための，いくつかの具体例を示した．

今後の研究にとって必要となることは，さらにより多くの，自己計測並行システムの開発の具体例を蓄積することである．クエリの種類は，計測仕様書を応用する対象によっては十分でないかもしれない．多くの種類の応用を対象とした事例研究を重ねて，さらに必要となるクエリを明らかにしなければならない．また，本論文において示したいくつかの例を見ても，それらのシステム構成には共通の特徴があることがうかがえる．また，信頼性の高い計測部の再利用を実現するためには，多くの具体例から再利用可能な計測部を，収集整理する必要がある．システム構成や個々の計測部を構成する部品を含めて，自己計測という行為をパターン化することは，信頼性の高い自己計測並行システムの開発にとって有効だからである．

また，3 章で，ソースコードの静的な解析が必要になる応用が存在することを述べたが，今後も，このような特定の用途の計測のために，事前の静的な解析が必要とされることが予測される．特定の用途のために仕様記述法を拡張しなければならないという状況は好ましくないため，事前の静的な解析を含めて記述できる計測仕様書を実現するための手法を検討しなければならない．

参 考 文 献

1) Barnes, J.: *Programming in Ada 95*, p.702, Addison-Wesley (1995).
2) Cheng, J. and Ushijima, K.: Partial Order Transparency as a Tool to Reduce Interference in Monitoring Concurrent Systems, *Distributed Environments*, Ohno, Y. (Ed.), pp.156–171 Springer-Verlag, (1991).
3) Cheng, J. and Ushijima, K.: Tasking Deadlocks in Ada 95 Programs and Their Detection, *Reliable Software Technologies — Ada-Europe '96*, Strohmeier, A. (Ed.), Lecture Notes in Computer Science, Vol.1088, pp.135–146, Springer-Verlag (1996).
4) Cheng, J.: The Self-Measurement Principle: A Design Principle for Large-scale, Long-lived, and Highly Reliable Concurrent Systems, *Proc. 1998 IEEE-SMC Annual Int. Conf. on Systems, Man, and Cybernetics*, Vol.4, pp.4010–4015 (1998).
5) Cheng, J.: The Wholeness Principle of Concurrent Systems and the Uncertainty Princi-

ple in Measuring Concurrent Systems, *Proc. Int. Sympo. on Future Software Technology '98*, pp.311–314 (1998).
6) Colket, C., et al.: Architecture of ASIS: A tool to Support Code Analysis of Complex Systems, *ACM Ada Letters*, Vol.XVII, No.1, pp.35–40 (1997).
7) Cooper, C.D.: ASIS-Based Code Analysis Automation, *ACM Ada Letters*, Vol.XVII, No.6, pp.65–69 (1997).
8) Heath, M.T. and Finger, J.E.: ParaGraph: A Tool for Visualizing Performance of Parallel Programs, Technical Report Oak Ridge National Lab (Sept. 1993).
9) Institute of Electrical and Electronic Engineers: Information Technology — Portable Operating Systems Interface — Part 1: System Application Program Interface (API) — Amendment 2: Threads Extensions [C Language] (1995).
10) International Organization for Standardization: Information Technology — Programming Language — Ada, ISO/IEC 8652:1995(E) (1995).
11) International Organization for Standardization: Information Technology — Programming languages — Ada Semantic Interface Specification (ASIS) ISO/IEC 15291:1999 (1999).
12) Nonaka, Y., Cheng, J. and Ushijima, K.: A Tasking Deadlock Detector for Ada 95 Programs, *Ada User Journal*, Vol.20, No.1, pp.79–92 (1999).
13) Nonaka, Y., Cheng, J. and Ushijima, K.: A Supporting Tool for Development of Self-Measurement Ada Programs, *Proc. Ada-Europe 2000*, Lecture Notes in Computer Science, Vol.1845, pp.69–81, Springer-Verlag (2000).
14) Nonaka, Y., Cheng, J. and Ushijima, K.: Monitoring Facilities in Languages Supporting Development of Concurrent Self-Measurement Programs, *Proc. 7th Asia-Pacific Software Engineering Conference*, pp.92–99 (2000).
15) Rybin, S., Strohmeier, A. and Zueff, E.: ASIS for GNAT: Goals, Problems and Implementation Strategy, *ACM Ada Letters*, Vol.XVI, No.2, pp.39–49 (1996).
16) Schonberg, E. and Banner, B.: The GNAT Project: A GNU-Ada 9X Compiler, *Ada Europe News*, No.20, pp.10–19 (1995).
17) Waheed, A. and Rover, D.T.: Instrumentation Systems for Parallel Tools, *Chapter 3 in the book Modeling and Simulation of Advanced Computer Systems*, pp.35–54, Gordon and Breach Publishers (1996).
18) Zhao, Q.A. and Stasko, J.T.: Visualizing the Execution of Threads-based Parallel Programs, Technical Report GIT-GVU-95-01, College of Computing, Georgia Institute of Technology (1995).

(平成 13 年 3 月 12 日受付)
(平成 13 年 12 月 18 日採録)

野中　裕介（学生会員）

1975 年生．1999 年九州大学大学院システム情報科学研究科情報工学専攻修士課程修了．同年 4 月より同専攻博士後期課程に在籍．ソフトウェア工学，並行システム開発手法に興味を持つ．IEEE-CS 会員．

程　　京徳（正会員）

1982 年中国清華大学計算機科学技術系卒業．1989 年九州大学大学院工学研究科博士後期課程情報工学専攻修了．工学博士．1989 年九州大学工学部助手．1991 年同助教授．1996 年九州大学大学院システム情報科学研究科情報工学専攻教授．1999 年埼玉大学大学院理工学研究科情報数理科学専攻教授．ソフトウェア工学，知識工学，および情報セキュリティ工学の研究に従事．日本ソフトウェア科学会，人工知能学会，IEEE-CS，ACM，AAAI 各会員．

牛島　和夫（正会員）

1937 年生．1961 年東京大学工学部応用物理学科（数理工学専修）卒業．1963 年九州大学工学部講師．1977 年同教授．1996 年九州大学大学院システム情報科学研究科長併任．2001 年 4 月から ISIT 九州システム情報技術研究所長．本会理事，監事，九州支部長を歴任．本会フェロー．現在本会アクレディテーション委員会委員長，工学博士．専門：計算機科学，ソフトウェア工学，日本語インタフェース．