Title	分散環境におけるフォールトトレラントソフトウエアの構成法に関する研究
Author(s)	伊関，浩
Citation	
Issue Date	1997-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1014
Rights	
Description	Supervisor:片山 卓也，情報科学研究科，修士

修 士 論 文

分散環境におけるフォールトトレラントソフトウエアの構成法に関する研究

指導教官　片山卓也 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

伊関 浩

1997 年 2 月 14 日

目 次

第 1 章

はじめに

1.1 研究の背景と目的

計算機の使用の増加に伴い、高い信頼性を持つ計算機の必要性が増している。高い信頼性を得るには、システムの一部に不具合が起きた時でもシステム全体がその影響を受けずに計算を進めるための技術が必要とされ、フォールトトレラントと呼ばれる。ソフトウエアによって、フォールトトレランスの技術を実現するものを実際の開発環境として実装したものは、手続き的モデルによるもののみであった。しかし関数的モデルによってフォールトトレランスを実現した場合、手続き的モデルと比較して、過去の計算状態の保存に複雑な処理が必要なく、誤りの原因分析や誤りからの回復等の処理が容易に行なえるという利点がある。

本研究の目的はFTAG の分散環境への実装を行なう事により、関数型モデルのフォールトトレランスにおける利点を実証すると共に、分散環境におけるフォールトトレラントソフトウエアの構成方法を

FTAG は属性文法に基礎をおく階層的関数型計算モデル HFP にフォールトトレラントの機構を拡張したモデルであり、モジュールと呼ばれる数学的な関数の集まりからなる計算木を用いて計算を行う。関数的な性質とモジュールの独立性によって、FTAG は共有メモリを持たない疎結合のマルチプロセッサシステム上への実装に適していると思われる。

本研究では、FTAG を基盤としてさまざまなソフトウエアフォールトトレラントのための実装技術を統合し実現する。そして実行可能なシステムを分散環境上に実装することで、フォールトトレラントソフトウエアの構成に関する研究を行なうことを目的とする。

1.2 論文の構成

本論文では、２章でフォールトトレランスに関する一般的概念および各種の定義を行なった後に、３章でソフトウエアフォールトトレランスを実現するための関数型モデルである FTAG について述べる。４章では FTAG モデルを実装するために、実装言語である CML や、実装するシステムのアーキテクチャ、特にノードマネージャ、ワークスペースマネージャ、アプリケーション実行部について詳しく触れる。５章では実装した FTAG 処理系においてフォールトトレラントソフトウエアを記述実行した例を示し、システムの動作について解説する。最後に本研究で構築したシステムの評価について述べる。

第 2 章

フォールトトレランス

フォールトトレランスとは、システムの一部の不具合に対して、それをシステム全体に波及させずに処理を続けさせる技術であり、ハードウエア的なアプローチと、ソフトウエア的なアプローチがある。本研究ではソフトウエアに関するフォールトトレランスのみをあつかう。

航空機管制システムや金融機関の管理システムなどが故障した場合、人命または財産が重大な被害を受ける可能性がある。こういったシステムのハードウエアとソフトウエアを、全く不具合の起きないものにしようとすると膨大なコストがかかる。そこで、システムにはある程度の不具合が起きるものとして、その不具合に対処する機構が必要とされる。そしてこの様な機構をフォールトトレランスという。

フォールトトレランスを実現する技法として、さまざまなものが提案されている。つぎに、一般的なフォールトトレラントの技法について説明をする。そして最後に、ソフトウエアフォールトトレランスを実現するモデルに必要な要件について考察する。

2.1 フォールトトレランスの技法

ここでは、一般的なフォールトトレランスの技法についていくつか紹介する。まず、誤りが発生した場合でもシステムの整合性を保証するための技法である、リカバリとアトミックアクションについて説明をする。

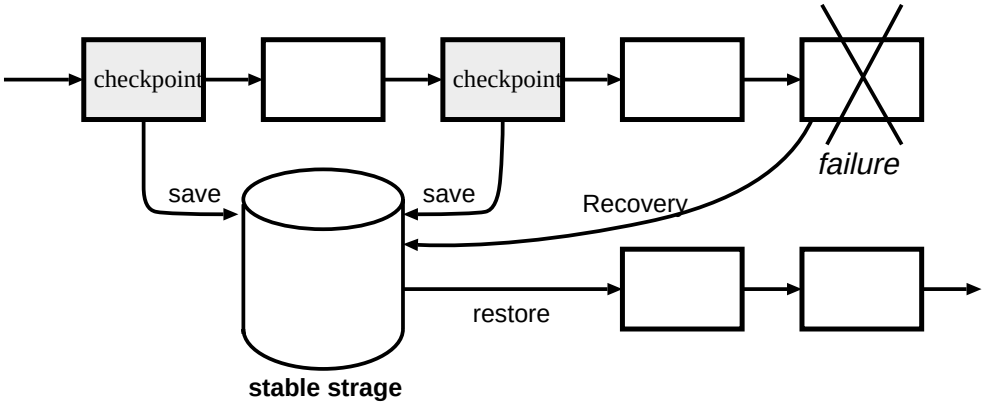

図 2.1: リカバリ

2.1.1 リカバリ

リカバリ とはフォールトトレランスの重要な技法の一つである。これは、システムに誤りがあった場合、それ以前の誤りがなかった状態に戻して処理を続ける方法である。リカバリを実現するためには、まず、システム実行中のある時点 (チェックポイント) で、そこでのシステムの状態を外部記憶に保存する。そして誤りが発生したら、一番最近のチェックポイントに戻り、外部記憶から誤り発生以前の状態を読み込んで、処理を続ける (図 2.1)。

2.1.2 アトミックアクション

アトミックアクションとは、ユーザがある操作を不可分として指定することを可能にする機構のことである。操作を不可分として指定することによって、その操作は、分割されることなく、外部からの干渉を受けることがなく、実行の順序が守られる、という利点を持つ。不可分として指定された操作において、その操作の内部でおきたシステムの状態の変化は、外部の操作には隠される。よって外側からは、操作が完全に終了した状態か、まだ処理が始まっていない状態か、ということのみが判別可能であり、この性質を”all–or–nothing” と呼ぶ。

もし不可分として指定された操作の内部で誤りが発生した場合は、この操作の始めの状態に戻す。もちろんこの時も、外部の操作には誤りの発生は知らされない。

2.1.3 Stable strage

フォールトトレランスの実現においては、リカバリ等で、システムのある時点での状態を保存する機構が必要となる。Stable strage とは、誤りが起きた後でも保存されているデータの意味が変わらないような性質をもった記憶装置のことである。通常 Stable strage は、ディスクシステムのような二次記憶装置によって実現される。そして Stable strage を実現するために、Disk Shadowing、Redundant Arrays of Disk 等の方法が存在する。

2.2 ソフトウエアフォールトトレランスの技法

ソフトウエアに関する誤りの発生は、ハードウエアの場合と違い、物理的な故障は存在しない。したがってソフトウエアの誤りは、常にソフトウエアの設計、実装、操作の誤りである。この設計上の誤りに対処するソフトウエアを、**フォールトトレラントソフトウエア**と呼ぶ。

ここでは、フォールトトレラントソフトウエアを実現するための技法である、リカバリブロック と N–バージョンプログラミングの説明をする。

2.2.1 リカバリブロック

リカバリブロックは、計算単位において複数の実現を用意しておき、順番に適用するものである。まず、計算すべき問題について、その計算をある方法で実現するモジュールを用意する。これを**主モジュール**と呼ぶ。そして主モジュールに計算を実行させる。つぎに主モジュールが計算した結果をテストにかけて、その計算結果が正しいものであるかどうかをチェックする。もし計算結果に誤りがあると思われる時は、主モジュールを実行する前の状態に戻される。そして**代替モジュール**と呼ばれる、主モジュールと同じ問題を別の方法で実装したモジュールに計算をさせて、また計算結果をチェックする。ここでも誤りが検出された場合は、まだ別の代替モジュールによって計算が行なわれる (図 2.2)。

2.2.2 N–バージョンプログラミング

N–バージョンプログラミングは、実装を同時に実行し結果を比較する技法である。この技法では、それぞれ違った方法によって実現された n 個の**バージョン**を用意し、並行

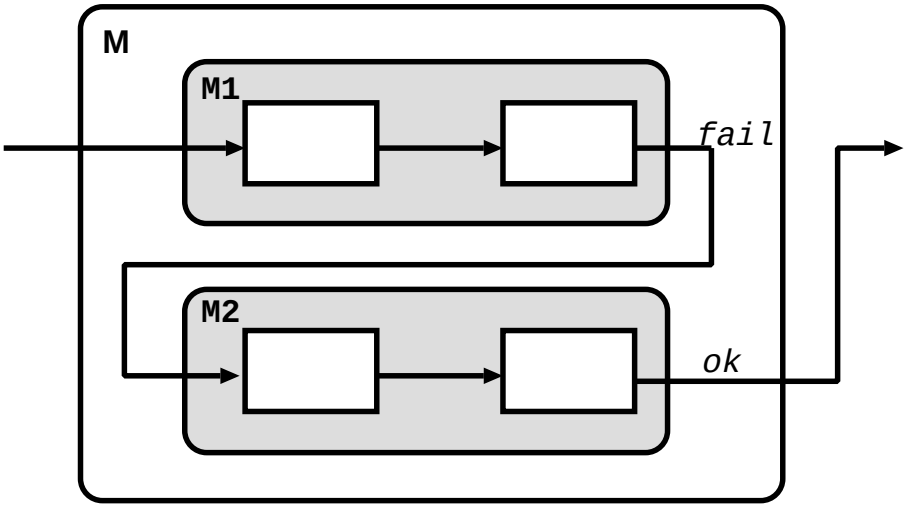

図 2.2: Recovery block

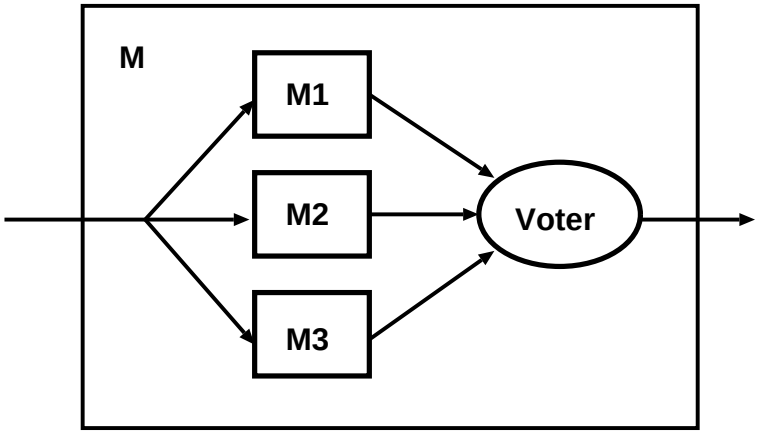

図 2.3: N–バージョンプログラミング

に動作するそれらのバージョンにすべて同じ入力を与る。そしてそれぞれのバージョンの出力を voter が調べて、正しいと思われる値を 1 つ選んで出力する (図 2.3)。

2.3 ソフトウエアフォールトトレランスを実現するモデルについて

ソフトウエアフォールトトレランスを実現するモデルには

- 過去のシステムの状態の保存

- 誤りの原因分析
- 誤りからの回復
- 現在のシステムの状態の監視

といった操作が容易に行なえることが必要となる。

そこで本研究では、ソフトウエアフォールトトレランスを実現するモデルとしてFTAG(Fault–Tolerant Attribute Grammars) モデルを採用した。FTAG モデルは次のような特徴を持つ。

1. 関数型モデルである
2. 属性文法を基にしている
3. 参照透過性が高い
4. モジュールが高い独立性を持つ

属性文法を基としているために、計算状態はすべて属性として表現される。よって、属性値を記憶する事により、システムの状態は容易に保存できる。FTAG では、計算は数学的な関数の集まりからなる計算木によって行なわれる。そこで、計算木を管理する機構を用意する事で、誤りの原因分析や誤りの回復も比較的用意に実現できる。同様に、現在のシステム状態の監視も、計算木の管理を行なう事で容易に実現可能である。そして、各モジュールが高い独立性をもつために、共有メモリを持たない疎結合のマルチプロセッサシステムのような、分散環境上での実装に適していると思われる。

第 3 章

FTAG モデル

この章では、本研究で扱う FTAG(Fault-Tolerant Attribute Grammars) モデルについて説明する。まず FTAG の機能として、基本モデルについて説明と、フォールトトレラントソフトウエアのための機能の説明をする。

3.1 FTAG モデルの機能

3.1.1 FTAG の基本モデル

FTAG では、すべての計算は、**モジュール**と呼ばれる数学的な関数の集合から構成される。関数の入出力として使用される値を**属性**という。入力 $x_1, \ldots, x_n$、出力 $y_1, \ldots, y_m$ をもつモジュール M は次のように表現される。

$$M(x_1, \ldots, x_n | y_1, \ldots, y_m)$$

この入力 $x_1, \ldots, x_n$ を入力属性、出力 $y_1, \ldots, y_m$ を出力属性と呼ぶ (図 3.1)。

モジュールの処理内容が十分に単純、すなわちモジュール内のみで計算を完了する事が可能なとき、そのモジュールを基本モジュールと呼び、次のように表す。

$$M(x_1, \ldots, x_n | y_1, \ldots, y_m) \Rightarrow \textbf{return where } E$$

ここで E は、$y_1, \cdots, y_m$ を $x_1, \cdots, x_n$ から計算する等式の集合である。

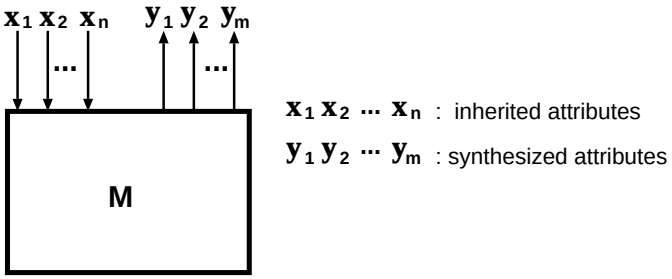

図 3.1: モジュールと属性

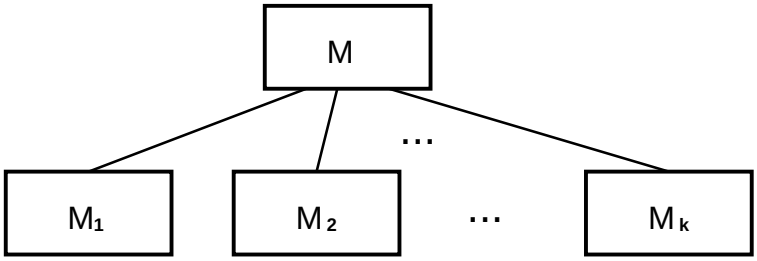

図 3.2: モジュールの分解

モジュールの処理内容が複雑であるとき、このモジュールはいくつかのサブモジュールに分解することができる。このようなモジュールを複合モジュールとよぶ。モジュール M を $M_1, \cdots, M_k$ というサブモジュールに分割したものは、以下のような式で表す。

$$M(x_1, \cdots, x_n \,|\, y_1, \cdots, y_m) \Rightarrow M_1, \cdots, M_k \ \textbf{where}\ E$$

このときの計算木の状態を図 3.2 に示す。ここで、E は M_i の入力属性と M の出力属性の間の関係を記述している等式の集合である。

モジュール分解を条件によって制御させることもできる。

$$\begin{array}{lll} M(x_1, \cdots, x_n \,|\, y_1, \cdots, y_m) \Rightarrow & & \\ [\ \ C_1 & \rightarrow & D_1 \\ |\ \ \vdots & & \\ |\ \ C_n & \rightarrow & D_n \\ |\ \ \textbf{otherwise} & \rightarrow & D_{def}\ \ ] \end{array}$$

上式では、条件 C_i が満たされるときのみ D_i の分解が行なわれる。条件は、C_1 から C_n まで順に調べられ、条件を満たすものがなければ D_{def} が選ばれる。

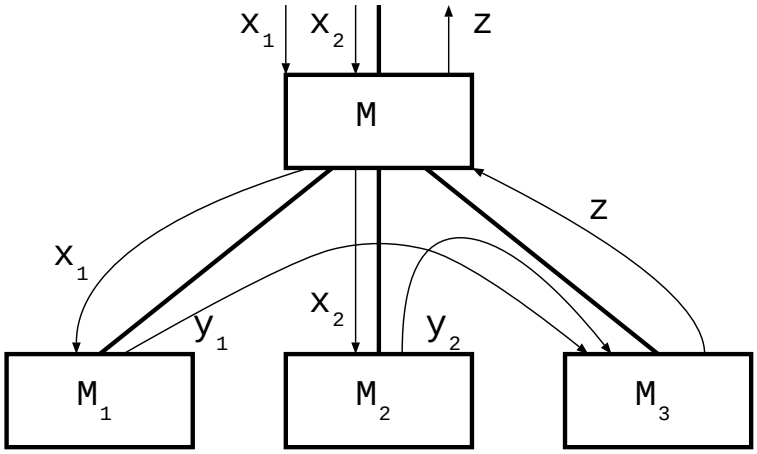

図 3.3: モジュール間のデータフロー

FTAG のプログラムの実行は、すべてのモジュールが、モジュールの分解によって基本モジュールになるまで行なわれる。そして各モジュールは出力属性として、計算した結果を返す。

例えば、次のようなモジュールがある。

$$
\begin{aligned}
M(x_1, x_2 \mid z) &\Rightarrow M_1(x_1 \mid y_1)\ M_2(x_2 \mid y_2)\ M_3(y_1, y_2 \mid z\ ) \\
M_1(x_1 \mid y_1) &\Rightarrow \textbf{return where}\ldots \\
M_2(x_2 \mid y_2) &\Rightarrow \textbf{return where}\ldots \\
M_3(y_1, y_2 \mid z\ ) &\Rightarrow \textbf{return where}\ldots
\end{aligned}
$$

このモジュール間のデータフローは図 3.3 のようになる。

3.1.2 フォールトトレランス実現のための拡張機能

FTAG は、フォールトトレランスを実現するために、いくつかの機構を持っている。ここではその機構についての説明と、その機構を使用する方法について説明する。

再実行

再実行とは、計算木の中で誤った計算をしている部分を、新しい計算木にいれかえる操作のことである。まず誤りを持った部分木を取り除き、つぎに回復作業をおこなう (図 3.4)。

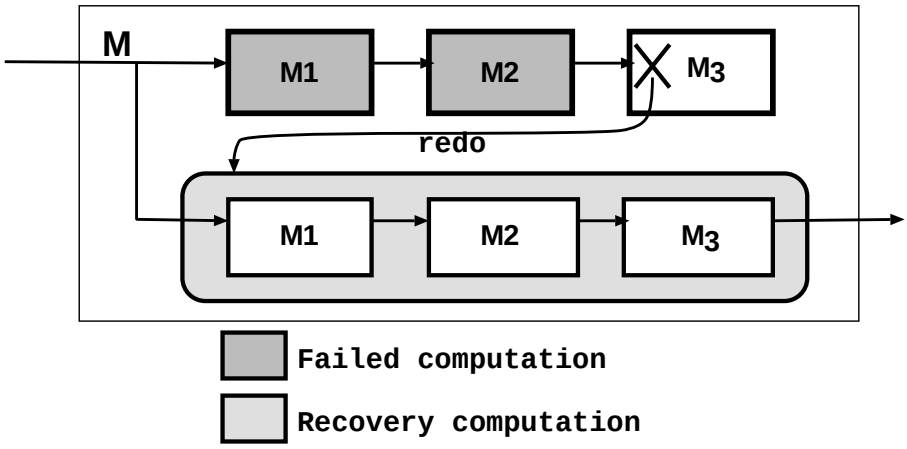

図 3.4: Redoing

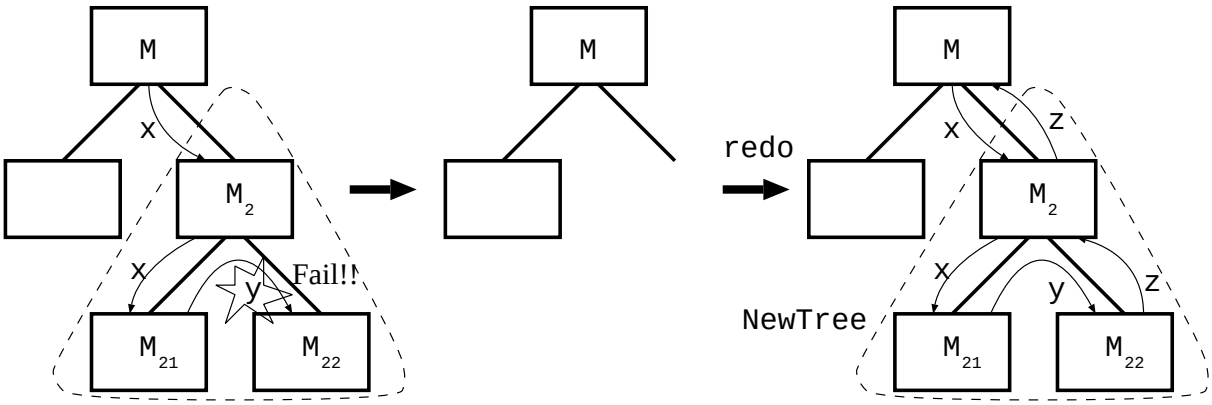

図 3.5: Redoing 時の計算木と属性

次のようなプログラムがある。このプログラムで redoing が発生した時の計算木の変化と属性値の流れは図 3.5 のようになる。

$$
\begin{aligned}
M_2(x|z) &\Rightarrow M_{21}(x|y)\ M_{22}(y|z) \\
M_{22}(y|z) &\Rightarrow [\text{not } fail(y) \rightarrow\ M_{22}body \mid \text{otherwise} \rightarrow redo\ M_2] \\
M_{22}body(y|z) &\Rightarrow \ldots
\end{aligned}
$$

Redoing を使えば、recovery block 等のフォールトトレランスの技法を容易に実現することができる。

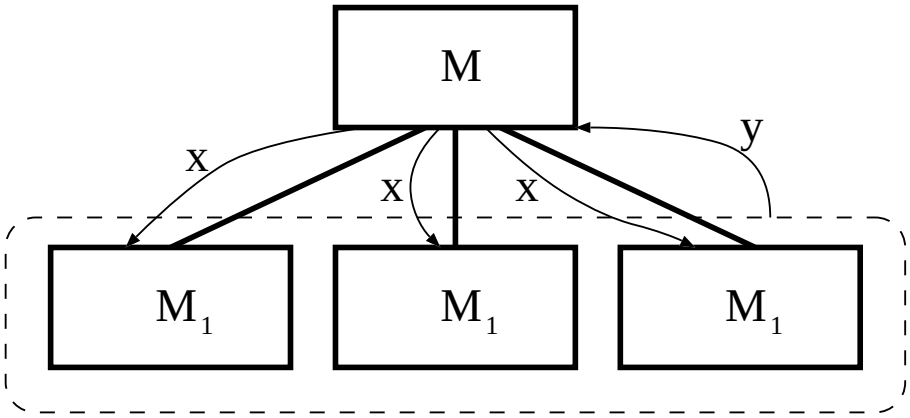

図 3.6: Replication

複製

複製とは、プログラムの一部分を複数個同時に実行させ、どれかに誤りが発生しても、全体として正しい計算を行なうための機構である。

FTAG では次のように記述する事により実現される。

$$M(x \mid y) \Rightarrow M_1(x \mid y), M_1(x \mid y), M_1(x \mid y)$$

同じ入出力属性値を持つサブモジュール (ここでは M_1) をレプリカと呼ぶ。モジュール M は、この複数個の**レプリカ**の出力属性から、正しいと思われる属性を M の出力として選んで M の出力属性として出力する (図 3.6)。

Replication をつかい、レプリカを複数の違った方法で実装することによって、N–version programing が簡単に実装できる。

Stable Object Base

FTAG では、redoing 発生時に次のような仕組みを用いて属性の回復が行なわれる。

1. 新しい値が保存されたら、その値を新しいバージョンとして保存する
2. 再実行が発生したら、もっとも最近のバージョンを復活させる
3. 古いバージョンは明示的な方法によって復活される

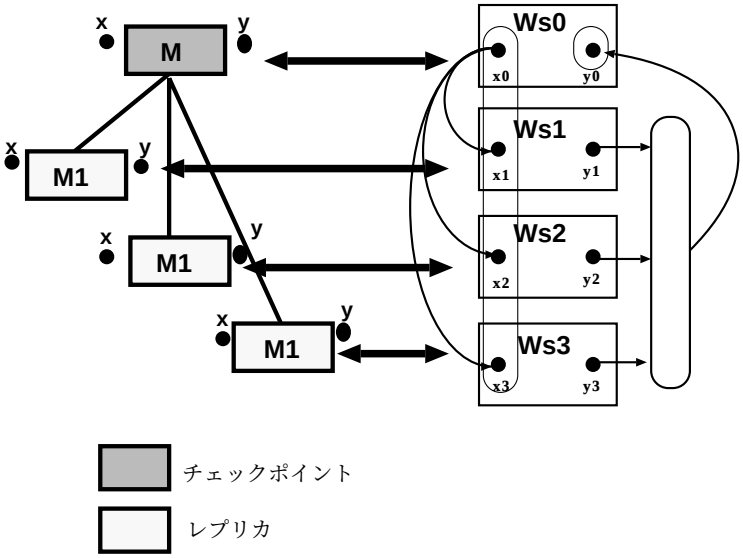

図 3.7: 複製の時のワークスペース

そして、stable strage の一部分として、再実行の際に必要となる最小限の属性値 (vital attribute) を保存する構造をワークスペースという。ワークスペースは、計算がチェックポイントに到達した時に作られ、チェックポイントの必要最小限の属性値を保存する。計算が複製に到達したら、このモジュールに対応したワークスペースが作られ、このモジュールの入力属性と出力属性は、どちらもワークスペースに保存される。上のモジュールのレプリカに対しても、各レプリカに一つずつワークスペースが割り当てられる。そして各レプリカの出力属性から、一つが選ばれ、この複製の出力となる (図 3.7)。

ワークスペースは、そのワークスペースに対応したモジュールの計算が終了した時、または再実行が起きて、モジュールが消された時に消去される。

第 4 章

FTAG モデルの実装

この章では、FTAG モデルを分散環境上に実装する方法について考察する。

4.1 方針

FTAG モデルは、階層的関数型計算モデル HFP を基にしており、計算をモジュール分解の繰り返しによって行う。このモジュールの計算順序はデータの依存関係のみによって決定され、依存関係を持たないモジュールは並列に実行される可能性がある。更にフォールトトレラントの機構を実現するために、階層的関数の分解によって作られる計算木を管理する操作と、属性値を保存するワークスペースを管理する操作を用意する。そしてこれらが互いに独立に通信しながら、計算を進める事が可能なようにする。

この性質を実現するためには、実装言語が並行計算の仕組みを持っている事が望ましく、ここではこれを満足した CML(concurrent ML) を用いて FTAG モデルの実装を行なう。

4.2 システムアーキテクチャ

4.2.1 Concurrent ML

Concurrent ML(CML) は、関数型言語である Standard ML(SML) 上で並列プログラミングをするためのシステムである。

CML は次のような性質をもつ

- 高階関数や多相型をサポートし、強く型づけされた言語である
- スレッドやチャネルを動的に作成可能である
- 統合されたI/O をサポートする
- スレッドやチャネルの割り当ては自動的に行なわれる

CML において新しいスレッドを作るには、次のような関数を用いる。

```
val spawn : (unit -> unit) -> thread_id
```

spawn を関数に適用する事によって、新しいスレッドは作られる。spawn の返す値は、新しく作られたスレッドを識別するためのスレッド ID である。スレッドは、スレッドの関数の適用が終るまで存在する。しかし、スレッドは次の関数 exit を呼び出す事によって自分自身で終了させる事ができる。

```
val exit : unit -> 'a
```

スレッドどうしがコミュニケートするためにはチャネルを用いる。

```
val channel : unit -> '1a chan
```

この関数を適用する事によって、メッセージの通り道であるチャネルが作られる。そしてチャネルを使ってメッセージをやりとりするために、次のような関数が用意されている。

```
val accept : 'a chan -> 'a
val send   : ('a chan * 'a) -> unit
```

accept はチャネルからメッセージを受け取り、send はチャネルにメッセージを流す。

FTAG の実装の際には

- モジュール → スレッド
- 属性値の流れ → チャネル

として表現する。

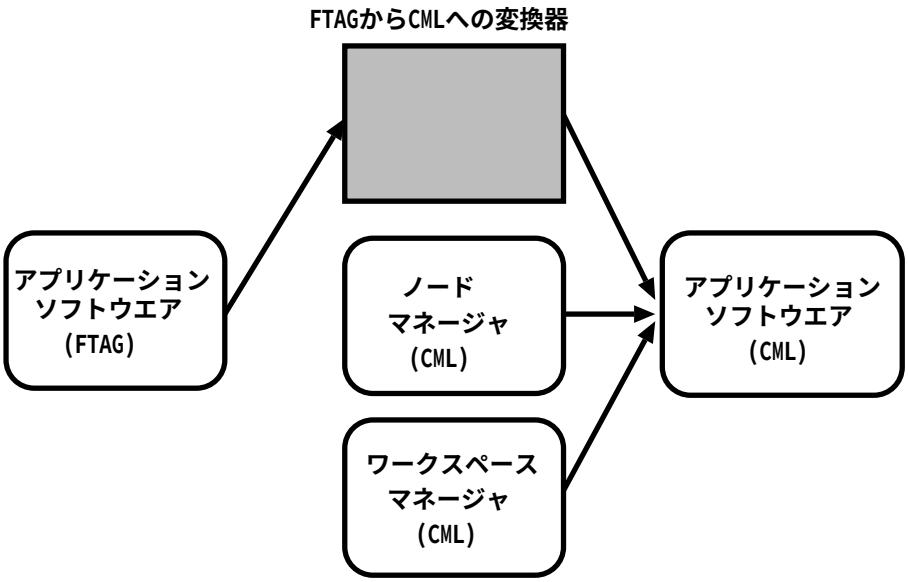

図 4.1: プログラムの出力までの流れ

4.2.2 処理の流れ

ここでは FTAG で書かれたプログラムリストを、実際に CML インタプリタ上で動作するプログラムリストに変換するまでの処理の流れを説明する。まず、FTAG で書かれたプログラムを、フォールトトレランスに関する処理を含む CML のプログラムリストに変換する。次に、プログラムが実行された時の計算木の管理をする**ノードマネージャ**と、再実行や複製の際に属性値を保存するために必要なオブジェクトベースの管理を行なう**ワークスペースマネージャ**を用意する。上記の２つのマネージャはともに CML で記述されている。これらのマネージャと、変換されたプログラムを合成する事によって、CML インタプリタ上で動作するプログラムが生成される (図 4.1)。

4.2.3 コンポーネントの動作の概要

プログラムを実際に動作させるうえで重要な、ノードマネージャ、ワークスペースマネージャ、そしてアプリケーションの実行部についての動作の概要を説明する。

- ノードマネージャ
  おもに、計算を進めることによってできる、計算木の管理を行う。計算木を管理することで、誤りが発生したときに不要になった部分木の削除や、ノードの削除に伴

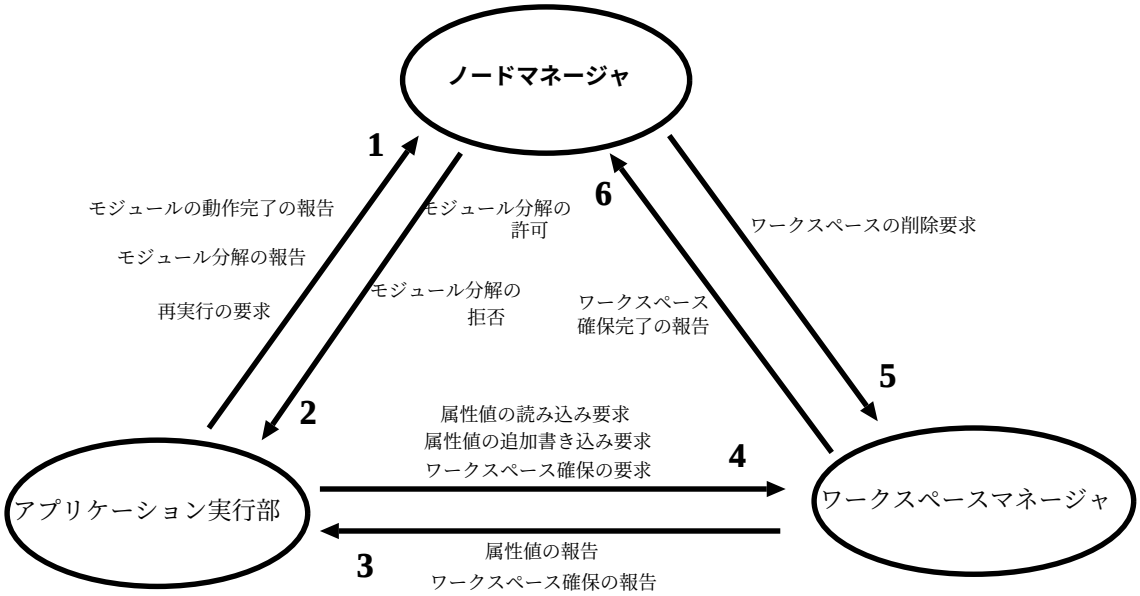

図 4.2: コンポーネント間のメッセージ

い発生するワークスペースの削除の指令等も行う。

- ワークスペースマネージャ
  ワークスペースの管理を行う。ワークスペースの作成と、ワークスペースへの属性値の書き込み、ワークスペースからの属性値の読み出し等の処理をして、他のコンポーネントに報告する。

- アプリケーション実行部
  実際の計算を行う。ほかにも、エラーの発見と報告、ワークスペース確保の指令、再実行の開始等の処理も行う。

4.2.4 コンポーネント間に流れるメッセージ

プログラムを実行するうえで必要なコンポーネントである、ノードマネージャ、ワークスペースマネージャ、アプリケーション実行部の間で交わされるメッセージについて説明する。

各コンポーネント間のメッセージは図 4.2 の様になる。

ワークスペースは文字列で識別可能であるから、識別子として文字列を使う。また、モジュールの識別はスレッド id を使う。

よって、メッセージに関するデータ型は次のようになる。

```
datatype ACTION  = DECOMP | DECOMP_OK? | REDO | COMPLETE | ACCEPT_DECOMP |
                   REFUSE_DECOMP | REP_ATT | REP_WSID | READ_ATT |
                   ADD_ATT | CREATE_WS | DEL_WS
datatype WS_ID   = None | string
datatype NodeID  = Null_id | thread id;
```

1. アプリケーション実行部からノードマネージャへのメッセージ

アプリケーション実行部からノードマネージャへは、

- モジュール分解の報告
- モジュール分解の許可要求
- モジュールの動作完了の報告
- 再実行の要求

が送られる。

メッセージには、行動と、自分のモジュールの ID、モジュール分解の際には新しく作られたモジュールの ID、そして、このメッセージに対する答えを返すためのチャネルを知らせる必要がある。

2. ノードマネージャからアプリケーション実行部へのメッセージ ノードマネージャからアプリケーション実行部へは、

- モジュール分解の許可
- モジュール分解の拒否

の 2 つのメッセージが送られる。

これらのメッセージは、このモジュールがこれ以上計算を続ける必要があるかどうかを判別するために用いる。再実行で、計算木から削除されたノードは、ノードマネージャによってモジュール分解が拒否される。

このメッセージでは、行動 (ACCEPT か REFUSE) のみが知らされる。

3. ワークスペースマネージャからアプリケーション実行部へのメッセージ

ワークスペースマネージャからアプリケーション実行部へは、

- 属性値の報告
- ワークスペース確保の報告

の 2 つがメッセージとして送られる。
属性値の報告は、動作 (REP_ATT) と属性値が送られて、ワークスペース確保の報告では、動作 (REP_WSID) と、ワークスペース ID が送られる。

4. アプリケーション実行部からワークスペースマネージャへのメッセージ

アプリケーション実行部からワークスペースマネージャへは、

- 属性値の読み込み要求
- 属性値の追加書き込み要求
- ワークスペース確保の要求

が送られる。
属性値の読み込み要求では、動作 (READ_WS) とワークスペース ID と読み込んだ値を返すチャネルが渡される。属性値の追加書き込み要求では、動作 (ADD_ATT) とワークスペース ID と属性値列が渡される。ワークスペース確保の要求では、動作 (CREATE) と作ったワークスペース ID を返すチャネルと属性値列とノードマネージャに知らせるときに使うスレッド ID が渡される。

5. ノードマネージャからワークスペースマネージャへのメッセージ

ノードマネージャからワークスペースマネージャへは、

- ワークスペースの削除要求

のみが渡される。
ワークスペースの削除要求は、動作 (DEL_WS) と削除するワークスペース ID が渡される。

6. ワークスペースマネージャからノードマネージャへのメッセージ

ワークスペースマネージャからノードマネージャへは、

- ワークスペース確保の報告

が送られる。このメッセージには、動作 (CREATE_WS) と、どのノードに属するかを判別するスレッド ID と、ワークスペース ID が含まれる。

ノードマネージャとワークスペースマネージャは、常に一つのチャネルからのメッセージを監視するだけで済むように、メッセージの型を統一する。
メッセージの型:
ACTION * thread_id * thread_id * 'a chan * attribute list * WS_ID

アプリケーション実行部が受け取るメッセージについては、各ノードが個別にメッセージを受け取るので、型の統一は不必要である。よって、必要最小限のメッセージのみ送る。

4.2.5 コンポーネントの動作の詳細

コンポーネントの動作の詳細について説明する。各コンポーネントは互いに前節のようなメッセージを交換し合い、そのメッセージによって動作する。

ノードマネージャ

ノードマネージャはアプリケーション実行部で作られた計算木の管理と、redoing 発生時に起きる動作の変更を各コンポーネントに指示する。ノードマネージャの動作は以下のようになる。

```
fun nodemanage(DECOMP_OK?,th_id,new_th_id,ch,_,_) =
      if member(redo_Tree,th_id) then
          send(ch,(REFUSE_DECOMP))
      else
          send(ch,(ACCEPT_DECOMP))
  | nodemanage(DECOMP,th_id,new_th_id,ch,_,None) =
          addnode(Tree,th_id,new_id))
  | nodemanage(DECOMP,th_id,new_th_id,ch,_,ws_id) =
          (addnode(Tree,th_id,new_id);
           add_ws(Tree,th_id,ws_id))
  | nodemanage(COMPLETE,th_id,_,ch,_,_) =
      if hasWS(Tree,th_id) then
         (send(WS_ch,DEL_WS);
          delnode(Tree,th_id))
```

```
    else if hasWS(redo_Tree,th_id) then
       (send(WS_ch,DEL_WS);
        delnode(redo_Tree,th_id))
| nodemanage(REDO,th_id,_,_,_,_) =
    addTree(redo_Tree,cutTree(Tree,th_id))
| nodemanage(REP_WSID,th_id,_,ch,_,ws_id) =
    (check_WS(Tree,th_id,ws_id);
     check_WS(redo_Tree,th_id,ws_id));
```

- アプリケーションマネージャから” モジュール分解の許可要求” をうけとったら、
  - もし再実行によって不要になったノードからのメッセージかどうかを調べて、不要なノードからのメッセージなら “モジュール分解の拒否” メッセージを送りこれ以上の無駄な計算木の拡大を防ぐ。
  - 通常ノードからのメッセージなら、” モジュール分解の許可” メッセージを送り、計算を進めさせる。

- アプリケーションマネージャから” モジュール分解の報告” をうけとったら、管理している計算木に新しく作られたノードを加える。
- アプリケーションマネージャから” モジュールの動作完了の報告” をうけとったら、
  - もしそのノードがワークスペースを持っていたら (そのノードがチェックポイントならば)、ワークスペースマネージャに” ワークスペースの削除要求” のメッセージを送る。
  - 再実行の際に不要になったノードが持っていたワークスペースに対しても同様に処理する。
- アプリケーションマネージャから” 再実行の要求” メッセージが来たら、再実行が起きるノードを根とする計算木の部分木を、不要な部分木として保存する。
- ワークスペースマネージャから” ワークスペース確保完了の報告” をうけとったら、指定されたノードにワークスペースの存在を記録する。

以上のことをすべてのノードが削除されるまで行う。

ワークスペースマネージャ

ワークスペースマネージャは、ワークスペースの確保や削除、ワークスペースへの属性値の保存や読みだし、を行なう。ワークスペースマネージャは以下のように振舞う。

```
fun wsmanage(CREATE_WS,th_id,_,ch,att,_) =
      (createfile(unique_name,att);
       send(ch,unique_name);
       send(nm_ch,(CREATE_WS,th_id,Null_id,my_ch,[],unique_name))
  | wsmanage(ADD_ATT,_,_,_,att,ws_id) =
      add_att(ws_id,att)
  | wsmanage(READ_ATT,_,_,ch,_,ws_id) =
      send(ch,read_ws(ws_id))
  | wsmanage(DEL_WS,_,_,_,_,ws_id) =
      delete_ws(ws_id);
```

- アプリケーション実行部から”ワークスペース確保の要求”メッセージを受け取ったら、
  - ワークスペースを確保し、そこにメッセージから得た属性値を保存する。
  - メッセージを送って来たノードにワークスペースの ID を知らせる。
  - ノードマネージャに”ワークスペース確保完了の報告”メッセージを送る事でワークスペースの ID を知らせる。
- アプリケーション実行部から、”属性値の追加書き込みの要求”メッセージを受け取ったら、指定されたワークスペースに属性値を追加する。
- アプリケーション実行部から”属性値の読み込み要求”メッセージを受け取ったら、指定されたワークスペースから属性値を読み込んで、結果を報告する。
- ノードマネージャから”ワークスペースの削除要求”のメッセージを受け取ったら、指定されたワークスペースを削除する。

実装の際は、一つのワークスペースについて、一つのファイルが割り当てられる事とする。

アプリケーション実行部

アプリケーション実行部は、FTAG のコードから生成され、当然扱うアプリケーションによって仕様が変化する。

アプリケーション実行部では、計算木の各モジュールはそれぞれ独立して並列に動作する。そして他のコンポーネントとのメッセージの受渡しは、各モジュールごとに行なわれる。

アプリケーション実行部でのさまざまな場合の cml のプログラムについて説明する。

- 追加される属性値

FTAG を CML に変換するときに、各モジュールは通常受け渡す入出力属性値のほかに２つの属性値を受渡しする。
１つはチェックポイントの集合をあらわす関数である。これは、計算の最中に変化するチェックポイントを記録したもので、再実行したいモジュール名を与えると、そのモジュールのスレッド ID を返す。詳細については後述する。
もう１つは、現在のモジュールが親モジュールに出力属性を渡してよいかを判定する属性である。もし自分が再実行の対称になっていて、属性値に誤りが含まれる可能性がある場合は、その値を親モジュールに返し、計算を終了させてしまうことはできない。よってこの属性によって、判定をする。詳細については後述する。

- モジュールの計算終了

モジュールの計算を終了させるときには、２つのことを確認する必要がある。
１つは、このモジュールは再実行によって切り離されていて、子どもは自分に対して出力属性を渡すことがないのかどうか。
これによって、無駄なモジュールの存在を消すことができる。もう１つは、上の場合以外で、自分のすべての子モジュールが出力属性を自分に対して送ったかどうかである。
この処理を行わないと子モジュールの計算終了前に、親モジュールが消えてしまうという不具合が起きる。

```
fun M(... ,out_ch,flag_ch) =let
    ...
    spawn(fn () => M1(... ,out_ch1,flag_ch1));
    spawn(fn () => M2(... ,out_ch2,flag_ch2));
    ...
    if accept(flag_ch1) andalso accept(flag_ch2) == false then
        (send(flag_ch,false);
         send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],ws_id)))
    else
        (val1 = accept(out_ch1);
         val2 = accept(out_ch2);
         ...
         calc
         ...
         send(out_ch, 計算結果))
```

- チェックポイントでのワークスペースへの属性の記入

チェックポイントで、ワークスペースに属性を記入するにはワークスペースには属性値を渡してやる必要がある。よって、一度属性値を受け取り属性値をワークスペースマネージャに渡した後に、子モジュールのチャネルに属性値の値を流す。

```
fun M(in_ch,...) = let
    val in_val = accept(in_ch)
    val ws_id = accept(my_id)
    ...
in
    send(WS_ch,(CREATE_WS,my_id,Null_id,my_ch,in_val,None));
    ...
    send(in_ch,in_val);
    spawn(fn () => M1(in_ch,....));
    ...
end
```

- モジュールの分解

モジュール分解を行うときには、ノードマネージャに対してモジュール分解の了解を得る必要がある。これは、再実行の際に切り離された部分木が、無駄な計算を進めるのを防止するためのものである。

```
    ...
    calc...
    ...
    send(nm_ch,(DECOMP_OK?,my_th,child_th,accept_ch,_,_));
    if accept(accept_ch) == ACCEPT_DECOMP then
         send(nm_ch,(DECOMP,spawn( ...), ... )
    else
       accept(accept_ch) == REFUSE_DECOMP then
         終了処理
\end
```

- チェックポイントの管理

チェックポイントは、チェックポイントの集合をあらわす関数を用意することで管理する。これは、計算の最中に変化するチェックポイントを記録したもので、再実行したいモジュール名を与えると、そのモジュールのスレッド ID を返す関数である。次のようにしてチェックポイントを増やすことができる。

```
fun M( ... , cps) = let
   val ...
   val cps_new = (fn n => (if n = "M" then my_id else cps(n)))
in


   ...
   send(nm_ch,(DECOMP,my_id,spawn(fn() => M1(...,cps_new))));
   ...


end;
```

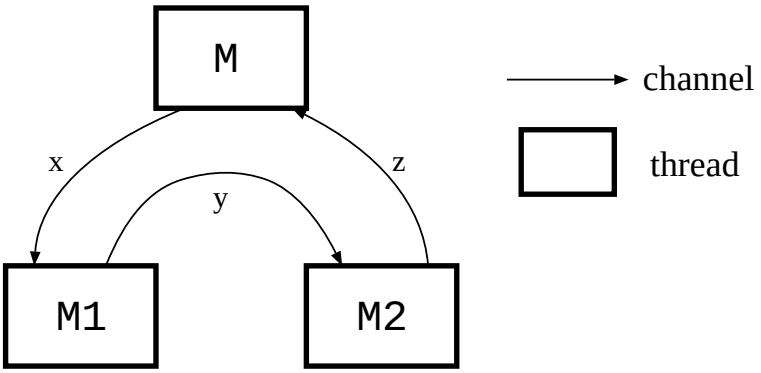

図 4.3: スレッドとチャネル

次節では、FTAG のプログラムから CML のプログラムを導き、再実行や複製などについて説明する。

4.2.6 FTAG から CML への変換

FTAG のプログラムを実行できる形にするには、まず、FTAG のプログラムを、再実行 等の操作を含まない CML のプログラムへ変換する必要がある。

FTAG は関数型のモデルであり、すべての計算はモジュールおよびモジュールの階層的分解により行なわれる。そこで各モジュールごとに別々の並列に動作するスレッドを生成する。そして、属性の受渡しは全て専用のチャネルを通じて行なわれる。 例えば次のような FTAG のプログラムでは、スレッドとチャネルの関係は図 4.3 の様になる。

```
M(x|z) =>  M1(x|y)  M2(y|z)
M1(x|y) => return  where y = x + 1
M2(y|z) => return  where z = y + 2
```

上記のプログラムを CML に変換すると次のようになる。

```
fun M1(x,y,cps,flag) = let
   val xv = accept x
   val my_id = getTid()
in
   send(flag,true);
   send(y,x+1);
```

```
   send(nm_ch,(COMPLETE,my_id,null_id)
end;

fun M2(y,z,cps,flag) = let
   val yv = accept y
   val my_id = getTid()
in
   send(flag,true);
   send(z,y+2);
   send(nm_ch,(COMPLETE,my_id,null_id))
end;

fun M(x,z,cps,flag) = let
   val y = channel()
   val z0 = channel()
   val my_ch = channel()
   val my_id = getTid()
   val zval = accept(z0)
in
   send(nm_ch,(DECOMP_OK?,my_id,Null_id,my_ch,[],None));
   if accept(my_ch) == REFUSE_DECOMP then
       (send(flag,false);
        send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None)))
   else
   (send(nm_ch,(DECOMP,my_id,spawn(fn() => M1(x,y,cps,flag1)));
    send(nm_ch,(DECOMP,my_id,spawn(fn() => M2(y,z0,cps,flag2)));
    if accept(flag1) andalso accept(flag2) = false then
        (send(flag,false);
         send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None)))
    else
```

```
        (send(z,zval);
         send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None)))
end;
```

つぎに、再実行におけるチェックポイントを含んだ FTAG のプログラムの変換について説明をする。

```
M2(x|z)  => M21(x|y) M22(y|z)
M21(x|y) => return when y = x+1
M22(y|z) => [y>0 -> return where z = y+2
             |otherwise redo M2]
```

モジュール M22 において、”redo M2”という記述があるので、M2 がチェックポイントになっている事がわかる。CML のプログラムは以下のようになる。

```
fun M21(x,y,cps,flag) = let
   val xv = accept x
   val my_id = getTid()
   val my_ch = channel()
in
   send(flag,true);
   send(y,xv+1);
   send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None))
end;


fun M22(y,z,cps,flag) = let
   val yv = accept y
   val my_id = getTid()
   val my_ch = channel()
   val cp_id = cps("M2")
   val (cp_x,cp_z,cp_cps,cp_flag) = accept(my_ch)
in
```

```
   if (yv>0) then
       (send(z,yv+2);
        send(flag,true);
        send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None)))
   else
       (send(ws_ch,(READ_ATT,my_id,Null_id,my_ch,[],ws_id));
        send(nm_ch,(REDO,cp_id,Null_id,my_ch,[],None));
        send(flag,false);
        send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None));
        spawn(fn() => M2(cp_x,cp_z,cp_cps,cp_flag)))
end
and M2(x,z,cps,flag)=let
   val y=channel()
   val z0=channel()
   val my_id=getTid()
   val my_ch=channel()
   val xv = accept x
   val flag1 = channel()
   val flag2 = channel()
   val accept_ch = channel()
   val ws_id=accept(my_ch)
   val z0_val = accept z0
   val cps_new = (fn n => (if n = "M2" then my_id else cps(n)))
in
   send(ws_ch,(CREATE_WS,my_id,Null_id,my_ch,xv,None));
   send(nm_ch,(DECOMP_OK?,my_id,Null_id,accept_ch,[],ws_id));
   if accept(accept_ch) == REFUSE_DECOMP then
       (send(flag,false);
        send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],ws_id)))
   else if accept(accept_ch) == ACCEPT_DECOMP then
```

```
        (send(nm_ch,(DECOMP,my_id,
                     spawn(fn()=>M21(x,y,cps_new,flag1)),my_ch,[],ws_id));
         send(x,xv);
         send(nm_ch,(DECOMP,my_id,spawn(fn()=>M22(y,z0,cps_new,flag2))));
         if accept(flag1) andalso accept(flag2) = false then
              (send(flag,false);
               send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],ws_id)))
         else
              (send(z,z0_val);
               send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],ws_id)))
end;
```

最後に複製の記述を含んだ FTAG のプログラムの変換について説明する。

```
M2(x|y) => M21(x|y1) M21(x|y2) M21(x|y3)
    return where pickup(y1,y2,y3);
M21(x|y) => return where y = x * 2
```

複製では、レプリカを複数個用意する。そして用意された全てのレプリカに対して属性値のコピーを用意して、同時に計算をさせる。最後にそれぞれのレプリカの計算結果から一つの値を選ぶ処理を挿入する。

CML に変換されたプログラムは以下のようになる。

```
fun M21(x,y,cps,flag) = let
   val xv = accept x
   val my_id = getTid()
   val my_ch = channel()
   val ws_id = accept(my_ch)
in
   send(ws_ch,(CREATE_WS,my_id,Null_id,my_ch,xv,None));
   send(flag,true);
   send(y,xv*2);
   send(ws_ch,(CREATE_WS,my_id,Null_id,my_ch,xv*2,None));
```

```
   send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None))
end;

fun M2(x,y,cps,flag) = let
   val xv = accept x
   val my_id = getTid()
   val ws_ch = channel()
   val my_ch = channel()
   val x1 = channel()
   val x2 = channel()
   val x3 = channel()
   val y1 = channel()
   val y2 = channel()
   val y3 = channel()
   val flag1 = channel()
   val flag2 = channel()
   val flag3 = channel()
   val ws_id = accept(ws_ch)
   val yv = pickup([accept y1,accept y2,accept y3])
in
   send(ws_ch,(CREATE_WS,my_id,Null_id,ws_ch,xv,None));
   send(nm_ch,(DECOMP_OK?,my_id,Null_id,my_ch,[],ws_id));
   if accept(my_ch) == REFUSE_DECOMP then
      (send(flag,false);
       send(nm_ch(COMPLETE,my_id,Null_id,my_ch,[],None)))
   else
      (send(nm_ch,(DECOMP,my_id,spawn(fn()=>M21(x1,y1,cps,flag1))));
       send(x1,xv);
       send(nm_ch,(DECOMP,my_id,spawn(fn()=>M21(x2,y2,cps,flag2))));
       send(x2,xv);
```

```
        send(nm_ch,(DECOMP,my_id,spawn(fn()=>M21(x3,y3,cps,flag3))));
        send(x3,xv);
        if accept(flag1) andalso accept(flag2)
                         andalso accept(flag3) = false then
             (send(flag,false);
              send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None)))
        else
             (send(flag,true);
              send(ws_ch,(CREATE_WS,my_id,yv));
              send(y,yv);
              send(nm_ch,(COMPLETE,my_id,Null_id,my_ch,[],None))))
    end;
end;
```

第 5 章

おわりに

5.1 まとめ

本論文では階層的関数型計算モデル HFP を基にした FTAG を、並列に動作するシステム上に実装することで、共有メモリを持たないマルチプロセッサ上への実装方法を与えた。

FTAG の各モジュールを、独立したスレッドとして扱ったことで、分散環境におけるフォールトトレラントソフトウエアの構成法を与えた。

次に FTAG のシステムでネームサーバを構築したことによって、関数型言語におけるフォールトトレラントソフトエアの優位性、例えば、過去の計算状態の保存の容易さ、障害の原因究明や障害からの回復の容易さを確認した。

5.2 今後の課題

今後の課題としては以下のようなものが考えられる。

- 各モジュール間の通信の最適化をはかる
- 各モジュールのノード割り付けの変更を行ないノードの割り当てに関する効率化をはかる
- 各コンポーネントの役割の変更を行ないコンポーネント間に流れるメッセージの効率化をはかる

- プロセッサ割当なども考慮して、実際の分散環境への実装

参考文献

[Suzuki94] M.Suzuki,T.Katayama,and R. D. Schlichting, "Implementing fault-tolerance with an attribute and functional based model," in *Proceedings of the 24th Symposium on Fault-tolerant Computing*,(Austin,TX),pp.244-253,Jun 1994

[Suzuki93] M.Suzuki,T.Katayama,and R. D. Schlichting, "A functional and attribute based computational model for fault-tolerant software,"Tech. Rep. TR 93-8, Dept of Computer Science,University of Arizona,Tucson,AZ,1993

[Pankaj] Pankaj Jalote "Fault Tolerance in Distributed Systems." P T R Prentice Hall,ISSBN 0-13-301367-7.

[Suzuki97] M.Suzuki,T.Katayama,and R. D. Schlichting, "FTAG: A Functional and Attribute Based Model for Writing Fault-Tolerant Software" IEEE Transaction of Software Engineering ,1997

[Suzuki96] M.Suzuki,T.Katayama,and R. D. Schlichting, "An Architecture for Software Fault Tolerance with a Function and Attribute-based Model FTAG," Proceedings on Workshop on Dependability in Advanced Computing Paradigm(DACP-96),1996.

[Avizienis85] A.Avizienis."The N-Version Approach to Fault-Tolerant Software." IEEE trans. on SOftware Engg., SE11(12):1491-1501,Dec 1985.