KLIC処理系における UNIXプロセス間通信を利用した例外処理の実装

金木 佑介† 加藤 紀夫 上田 和紀
† 早稲田大学大学院理工学研究科
lang@ueda.info.waseda.ac.jp

並行論理型言語 KL1 は GHC (Gaurded Horn Clauses) に基づいたプログラミング言語で、並行処理に必要な複雑な同期処理を自動化しているという特徴がある。もともと、KL1 は並列推論マシン (PIM) 用の言語だが、KL1 プログラムを C 言語にコンパイルし、汎用の計算機、つまり UNIX 環境上でも動作できるようにしたのが KLIC 処理系 [1, 2, 3] である。KL1 の特徴である、同期処理の自動化は並行処理だけではなく分散処理などにも有用である。しかし、現在の KLIC 処理系には例外処理機構が無いため分散環境での安定したアプリケーションの構築、またはそれを構築するためのサービスを提供する分散処理システムの構築が困難である。

本研究では UNIX プロセスを利用して、KLIC 処理系に例外処理機構を実装した。これにより分散環境でのサンドボックスモデルを記述することが可能になり、遠隔述語サーバなどのフォールトトレランスを意識しなければならない分散アプリケーションの構築が可能になった。

1 はじめに

現在の KLIC 処理系は、UNIX シグナルの割り込みや KL1 固有の例外 (永久中断、ゴール実行の失敗、単一化の失敗) などが起こると KLIC ランタイム自体が終了してしまい、それを回避する手段は用意されていない。そこで、本研究では KLIC 処理系に UNIX プロセスを利用して例外処理機構を実装し、例外を捕捉、報告することに成功した。これにより分散環境でのサンドボックスモデルを記述することが可能になり、遠隔述語呼び出しサーバなどのフォールトトレランスを意識しなければならない分散処理システムの構築が可能になった。

本論文は本節を含めて 8 節で構成されている。2 節では、例外処理の書式について述べる。3 節から 5 節にかけては、本例外処理機構の概要と実装の詳細を述べ、6 節は本例外処理機構のプログラム例を、7 節ではまとめ、そして 8 節では今後の課題について述べる。

2 例外処理の書式

ここでは、並行論理型言語 KL1 の言語特性を考えた例外処理の書式について考察する。書式について考えることは、例外処理の対象範囲をどう指定するかという問題も同時に含んでいる。まず、先に並行論理型言語 KL1 について簡単に触れる。

2.1 並行論理型言語 KL1

並行論理型言語 KL1 は GHC に基づいた一階述語論理のプログラミング言語である。そのプログラムの実行は他の手続き型言語とは大きく違っていて、項書き換え言語である Prolog に近い。

KL1 プログラムは、ガードつき節の集合からなる。ガードつき節とは下記のような形をしている。

$$h \ :- \ G \mid B.$$

ここで、h を節のヘッド、G をガード、B をボディと

いう。KL1 プログラムは、ゴールプール中のゴールの書き換えを繰り返すことで実行が進む。ゴールはヘッドにマッチする時、ボディで指定されたゴール列に書き換えられる。ガードは節の適用条件である。適用できる節が無いゴールは適用出来るようになるまでサスペンドする。ゴールプールが空になると実行は終了する。ゴール同士は論理変数によって通信して協調動作している。

2.2 他の言語との比較

例外処理は、古くは BASIC の時代から存在していて、現在では JAVA をはじめとして多くの言語がその機能を持つ。BASIC の例外処理は非構造化例外処理といわれ、行番号やラベルなどで例外発生時の処理を制御していた。現在の JAVA 処理系などが採用している try-catch 構文は構造化例外処理といわれ、現在の主流なプログラミング概念に沿ったものになっている。

しかし、KL1 ではプログラムソース上のゴールの位置関係と実行順序はまったく関係がないため、JAVA、BASIC などの例外処理の書式のように例外処理対象を行単位で指定することはできない。

Prolog 処理系の中には、失敗などの例外が起こるタイミングで呼び出されるゴールを指定する例外処理機構が組み込まれているものある。

2.3 例外処理機構のモデル

本研究の例外処理のモデルは、並列推論マシン PIM 上の KL1 処理系 [4] に実装されていた荘園モジュールを参考にしている。荘園は KL1 プログラムの実行のある部分 (荘園トップレベルのゴールと、そのゴールから派生した子孫ゴール群) を内部にもつ空間である。また、荘園の中にまた荘園を作ることができ、荘園は一般に木構造となる。荘園は実行制御、資源管理、例外処理の機能を備えていた。本例外処理機構を実装するにあたって荘園の書式、例外処理の機能などを参考にしている。

以下に荘園の書式を示す。

```
shoen:execute(Code,ArgV,M0,M1,Exc,C,Rep)
```

Code は生成する荘園のトップレベルのゴールの述語名で *ArgV* はそのゴールの引数、*Exc* は処理する例外事象を決めるマスクパターンを指定する。そして *Rep* は実行状態の報告で例外の報告はここに流れる。他の引数は本研究では参照しないので省略する。

荘園はゴールを指定することで、その荘園の適用範囲をそのゴールから派生する子孫ゴール群全てにしている。本例外処理系でも 3.2 節で述べるように、荘園と同じ指定方法で例外対象範囲を指定することにする。

3 例外処理機構の動作

3.1 例外処理機構の概要

現在の KLIC 処理系は、一つのゴールの実行失敗がランタイム全体を終了させてしまう。これを回避するのが本例外処理機構の一つの大きな目標である。本研究では UNIX プロセスを利用することによりこの問題を回避する。KLIC 処理系自体に大きく手をいれる方法もあったが、修正の量が膨大になるため本研究ではより実装が容易な方法を選んでいる。したがって KLIC 処理系にできるだけ手をいれないように実装することが一つのポリシーとなっている。

また、UNIX プロセスを利用することにより KLIC ランタイム自体が終了しないようにするのではなく、例外を発生して KLIC ランタイムが終了するという状況を逆に利用することが出来る。詳細は、4 節で述べる。

この例外処理機構の基本的な考え方は、例外処理対象のゴール群を別の UNIX プロセスの KLIC ランタイム上で実行させることによって、例外処理対象ゴール群をその他のゴールから切り離してしまうことにある。したがって、例外処理対象のゴール群が例外を発生して、その KLIC ランタイムが終了しても、その他のゴールがある KLIC ランタイムは存続する。また、この別プロセスにするという特徴を活かして、子プロセスが終了する時の終了コードに例外情報を流すという方法で例外の捕捉と報告を実現している。詳細は、5 節で述べる。

3.2 例外処理機構の書式

本例外処理機構が提供する述語を例外処理モジュールと呼ぶ。例外処理モジュールは通常の述語を呼び出すのと同じように呼び出される。書式を以下に示す。

```
exception:execute(GOAL,EXCEPTION)
```

引数 *GOAL* には *Module*:*Predicate*(*Arg*,...) という書式で例外対象にすべきゴール群のトップレベルのゴールを指定する。つまり、そのゴールから派生する子孫ゴール群が全て例外処理の範囲に入ることになる。

ゴールの引数には具体化される方向 (モード) があり、5.2 節で詳しく述べるように、例外処理時に通信路を閉じるためには変数のモード情報が必要になる。そこで、この例外処理モジュールの中に値が入って行く論理変数には + を、出て来る論理変数には − を明示的に付ける。実際にはモードが切り替わる時のみ明示的に付ければ良い。

引数 *EXCEPTION* は例外処理報告用の論理変数で、$err(N, C, M, P)$ という項で具体化され、以下の情報を報告する。

- N: *normal* か *abnormal* が返る。プロセス間の論理変数による通信が正常に終了したかどうか。
- C: エラーコード、整数値
- M: エラーメッセージ、文字列、正常終了なら *NORMAL* が返る。
- P: 子プロセスの *ID*、整数値

以下に使用例を示す。

```
exception:execute(main:test(+(A),-(B)),E)
```

この例では、test/2 を呼び出すゴールとそのゴールから派生するゴール群を例外処理の対象として実行する。test/2 が正常に終了、または例外を発生して終了するとその報告が E に具体化される。

3.3 例外処理機構の実行の流れ

以下に例外処理機構の実行の流れを示す。

1. 例外対象となるゴールを引数にした例外処理モジュールが呼ばれる。
2. 自分のプロセスのコピーである別の UNIX プロセス (子プロセス) を fork によって生成し、KLIC ランタイムを起動する。
3. 指定されたゴールとそのゴールから派生する子孫ゴール群をその別プロセス上の KLIC ランタイム上で実行する。
4. 子プロセス上のゴール群が正常終了もしくは、例外発生などで異常終了したらその情報を親プロセスに渡す。
5. 異常終了によって引き起こされたゴール間の通信障害を復旧する。具体的には通信路を閉じる (通信路内の未定義変数を具体化する) ことにより永久中断を起こすなどの障害を防ぐ。

4 UNIX プロセスによる例外処理

4.1 UNIX プロセスの生成

新しい UNIX プロセスを生成し KLIC ランタイムをその上で動作させるために、KLIC の組込み述語である fork_with_pipes を使用する。この述語は、内部で fork を使用していて、自分のプロセスのコピーを別の UNIX プロセス上に作成することができる。そのため、KLIC ランタイムごと別プロセスにコピーされるので、KLIC ランタイムをあえて実行する必要はなく、そのまま fork_with_pipes の返り値でプログラムを分岐することで、親プロセス、子プロセス上で各々動作させることができる。

ここで、一つの問題が生じる。前述したように自分のコピーを作成するため、コピーが始まる前にサスペンドまたはエンキューされたゴールは当然、子プロセスでもサスペンドまたはエンキューされた状態になっている。本来の動作では、子プロセスでは例外対象のゴール以外は実行させたくないので、これらの余計なゴールを削除する必要がある。

そこで例外処理モジュールが呼ばれるとすぐに子プロセスを作成し、ゴールプール内の例外処理対象ゴールを実行する処理に必要なゴールを除くサスペンド、エンキューしている全てのゴールをゴールプールから削除する。

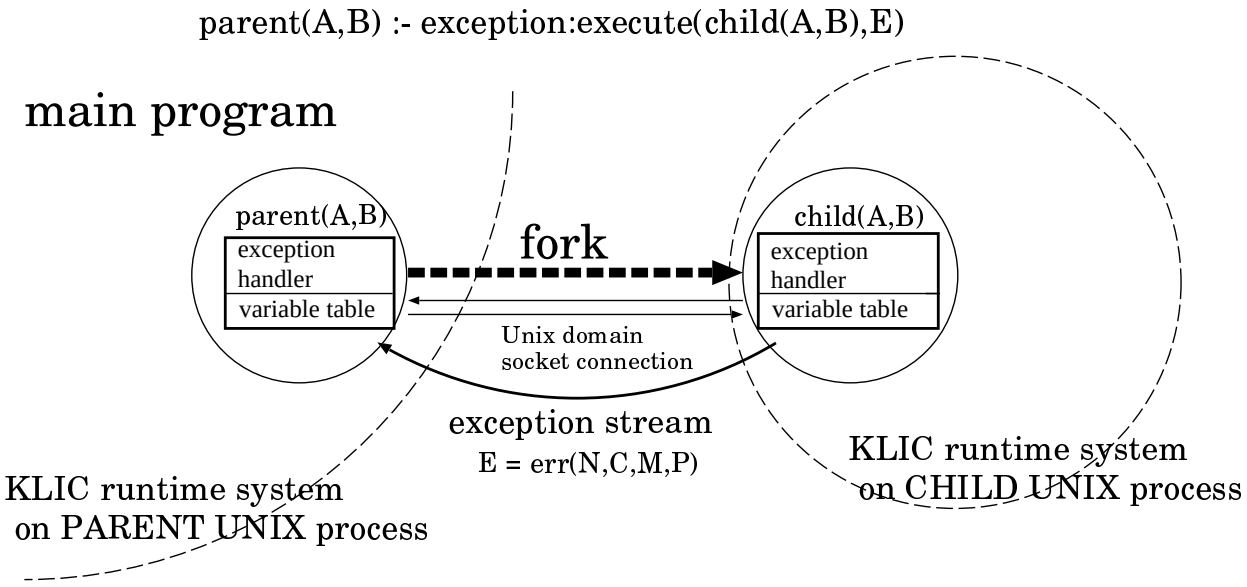

図 1: 例外処理システム 構成図

4.2 プロセス間通信と論理変数

KL1 はゴール同士が論理変数による通信によって協調動作しながら実行が進む。本例外処理機構では例外処理対象のゴールとそのゴールから派生する子孫ゴール群を、例外処理モジュールが作った UNIX プロセス上 (子プロセス) で実行する。そして、例外処理モジュールを呼び出している親プロセスと例外処理対象の子プロセス全体の実行は、通常の単一プロセス上での実行と同じ振舞いをしなければならない。つまり、子プロセス、親プロセスという二つの UNIX プロセス上のゴール間に論理変数による通信路を張る必要がある。

ここで、KL1 の分散言語処理系 DKLIC[5] の研究成果を利用する。DKLIC は分散論理変数表と 2 本のソケットを利用して異なるノード間に分散論理変数を提供し、ネットワーク透過な分散プログラミングを提供するミドルウェアである。本研究ではこの分散論理変数表を元にプロセス間論理変数表を実装した。プロセス間論理変数表については 4.3 節で詳しく述べる。

ソケットを TCP で張るとそのポート番号管理が非常に複雑になる。そこでソケットはポートの重複などを容易に避けるため UNIX ドメインのソケットを使用する。UNIX ドメインには”eXi” および “eXo” という文字列に子プロセスの *PID* を付加したものを使用する。例えば、“eXi0031335” と “eXi0031335” である。*PID* は同じ UNIX 上ならプロセスに一つの番号であるので、そのドメイン名が重複しないように管理する必要が無くなる。

4.3 プロセス間論理変数表

プロセス間論理変数表は DKLIC の分散論理変数表の一部に対して修正を加えたものである。修正は非常に微小であるので本論文では触れない。ここではプロセス間論理変数表の元になっている分散論理変数表について概要を述べる。

並行論理型言語 KL1 の変数は単一代入であり、未定義状態を持つ論理変数である。ゴール間通信には頭から徐々に具体化されるリストが利用されることが多い。しかし、KLIC 処理系には異なるノード間でも論理変数によって通信できるようなライブラリは実装されていなかった。

そこで、DKLIC は異なるノード間にソケットを張り、分散論理変数表を介して論理変数による通信を実現する。つまり、ノード A とノード B が論理変数 X による通信をするとする。まずノード A で論理変数 X が分散論理変数表に *ID* と共に登録される。そ

うするとノード B へその登録状態が送信されノード B でも *ID* と共に登録される。そして、変数表はその変数 *X* の具体化を監視していて、もしノード A で *X* が具体化されたならノード B にその変数 *X* が具体化されたこと知らせ、*ID* と具体値を送る。そうすると、ノード B でも変数 *X* が具体化されることになる。

以上のように分散論理変数表は、ノード間で共有される論理変数の ID を管理し、その変数の具体化を監視する。

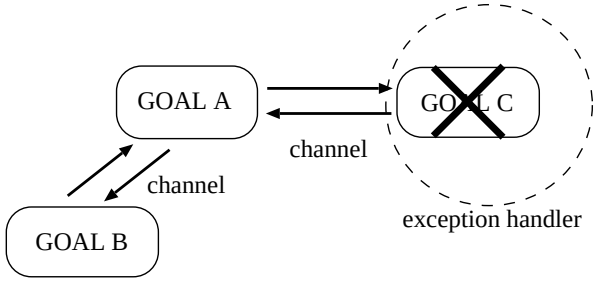

図 2: ゴール間通信と永久中断

5 例外の捕捉と報告の仕組み

本例外処理機構は、UNIX プロセスを作成しその上で別の KLIC ランタイムを起動することのより、そのランタイム上の例外が他の KLIC ランタイム上にあるゴール群に影響を及ぼすのを防ぐ他に、例外を捕捉、報告することも可能にしている。

fork を使用して作成された子プロセスには、wait という関数を発行できる。この wait というのは、子プロセスの終了を親プロセスが調べたり、待ったりする時などに使用される。この wait 関数にはさらに子の終了状態を得る機能があり、主に、exit の引数、または UNIX シグナルの値を wait は受け取る。つまり、wait を発行することによって、ある子プロセス上で実行されていた KLIC ランタイムの終了状態を知ることができる。

UNIX シグナル発生によって KLIC ランタイムが終了している場合には、そのまま wait 関数がシグナル番号を受け取ることができる。しかし、KLIC 特有の永久中断などの例外は、シグナル番号で区別できないため、そのままでは例外の種類を判別することができない。

そこで、exit 関数の引数を利用して KL1 固有の例外を判別する。つまり、例えば永久中断 (ゴール同士がお互いに論理変数による通信を待ち続けデッドロックを起こす状態) が起こるときには、永久中断を検知し KLIC ランタイムを終了させようとする部分がかならずあるので、KLIC ランタイムのその部分を修正して、exit 関数によって終了するようにし、さらにその引数に発生した例外特有の番号をとるようにする。実際にはシグナル発生以外の例外はある一つの関数で処理されているので、そこに例外の種類を整数に変換したものを引数に与え、exit 関数を発行するような修正を加える。

以上のように、KLIC 特有の例外は exit 関数の引数によって、UNIX シグナルは手を加えることなく wait 関数によって捕捉出来るようにした。

この方式の利点としてユーザが例外を定義できることが挙げられる。本例外処理機構は、exit 関数の引数に与えた番号をそのまま報告するようにしている。したがって、例外処理内で起動しているゴールを exit 関数で終了するようにし、その引数に適当な整数を与えればそのまま、その数字を例外報告として受けとることができる。

5.1 補足後の処理

KL1 は複数のゴールが論理変数を介した通信によって協調動作して実行が進んで行く。ところが、例外が発生して、あるゴールの実行が止まってしまうと当然そのゴールとの通信自体も止まってしまう。したがって、もし例外を発生して死んでしまったゴールからの通信を待っているようなゴールがいると、永久にそのゴールは実行を中断することになり、永久中断の原因になってしまう。例えば、図 2 のように、ゴール A、B、C がありそれらが論理変数による通信路 (channel) によって結ばれている時に、例外処理システム上のゴール C が例外を発生し終了してしまう状況を考える。ゴール C は例外処理システム上で終了するので、ゴール A、ゴール B は動作し続ける。しかし、ゴール A がゴール C からの通信を受け取って動作している場合にはゴール A は永久にその通信を待ってしまうことになる。これでは、例外処理対象側は処理できても、その他のゴールに例外発生の影響が及んでしまう。

この障害を回避するには二つの方法が考えられる。一つは、本例外処理機構がゴール C を復旧してゴール A からの通信路と再接続させ、ゴール A に通信障害の影響が及ぶのを防ぐ方法である。しかし、この方法はどの状態、いつの状態で例外発生ゴールを復帰させるかという問題や、全体のゴール実行の整合性などの複雑な問題を含んでいるため採用しなかった。もう一つは、通信障害を起こした通信路を例外発生が分かる形で閉じてしまう方法である。それによって、ゴール A に例外発生を検知させ、通信障害に対処させる方法である。本例外処理機構はこの方法を採用した。5.2 節で詳細を述べる。

5.2 通信障害への対処

ここでは論理変数による通信路をどうやって閉じるかについて述べる。

本例外処理機構の場合、問題となるゴール間の通信にはプロセス間論理変数表が仲介をしている。つまり、そのゴール間で具体化していない論理変数はプロセス論理変数表が管理しているので、例外が発生したら変数表をその未定義変数をなんらかの値 (例えば *exception*) で具体化してしまえばその通信路は閉じることになる。そして、他のゴールはその値を受け取るとこで例外発生を知り各々対処する。

しかし、全ての未定義変数を具体化すれば良いわけではなく、例外処理対象から出て行く通信が問題になっているのでその通信路だけを閉じる。逆に入って来る通信を閉じようとすると、他のゴールからの具体化と衝突が起きてシステム自体が単一化失敗の例外を起こしてしまう。

論理変数の通信の方向は動的に判別することができないので、現状では 3.2 節で述べたように論理変数の通信の方向、つまりモード情報をあらかじめつけて変数表モジュールに知らせるようにしている。

6 プログラム例

以下に本例外処理機構のプログラム例を示す。

遠隔述語サーバー側

```
rdc_server([call(GOAL,R)|M0]) :-
    exception:execute(ms:exec(GOAL),E),
    check(E,R),
    ...

check(err(_,0,_,_),R) :- R=ok.
check(err(_,105,_,_),R) :-
    R=fail("predicate name no match").
check(err(_,106,_,_),R) :-
    R=fail("module name no match").

exec(:(M,P)):-
    ...
```

遠隔述語呼出し側

```
rpc_client:-
    rpc_proxy(M),
    M=[
      test(math:gcdset(32,48,-(O1)),R1),
      test(math:gcd(32,48,-(O0)),R0)
      ],
    ...
```

このプログラムは本例外処理機構を使用した簡単な遠隔述語サーバである。クライアントと遠隔述語サーバの通信は DKLIC を用いて記述した。rpc_server/1 というのが遠隔述語サーバで、幾つかの述語を持っている。クライアントはサーバに対して call(GOAL,R) というゴールを投げることで、述語実行を依頼することが出来る。call/2 の第 1 引数 GOAL は実行する述語名である。第 2 引数の R は、実行に成功すれば *ok* に具体化し、正常に実行出来なければそのメッセージに具体化する。

本例外処理機構は遠隔述語サーバ側の rpc_server/1 内で使用される。

```
exception:execute(ms:exec(GOAL),E),
```

という部分である。exec/1 は call/2 で指定された述語を実行するゴールである。ここでは exec/1 を例外処理対象とすることで、遠隔述語呼び出しによる述語実行を例外処理システム内で行い、述語実行中に発生する例外を捕捉する。例えば、実行する述語が無かったり、述語自体の例外を捕捉する。例外が発生するとその報告が変数 E に返り、それを check/2 で処理をする。3.2 節で述べたように E は $err(\_, 0, \_, \_)$ のように

具体化される。第 2 引数はエラーコードで、check/2 はそれを元にその述語を試行した結果を test/2 の第 2 引数としてクライアントに返す。

この例において、クライアントは gcdset/3、gcd/3 という二つの述語を試行する。実際は、このサーバには gcdset/3 という述語はあるが、gcd/3 という述語は存在しない。つまり、最初の gcdset/3 の試行は正常に終了するが、gcd/3 は存在しないので例外処理システム内の exec/1 で例外が発生し、check/2 の引数に、存在しない述語が呼び出されたことを示すエラーコード 105 が返る。そして check/2 はクライアントに述語名の誤りを報告する。

上記のプログラム例のように本例外処理機構を用いることで、本来なら述語が存在するかどうか、その述語が正しく動作するかなどの検証ためのコードを本例外処理機構にまかせる事ができ、またそのような予期せぬ例外に対処できるようなコードをすっきり書く事ができた。

7 まとめ

本研究では並行論理型言語 KL1 の処理系 KLIC に例外処理機構を実装した。本例外処理機構は、異なる UNIX プロセスに例外処理対象となるゴール群を隔離することで、例外発生による影響がその他のゴールに及ぶことを防いだ。そして、UNIX シグナルの割り込みや KL1 固有の例外 (永久中断、ゴール実行の失敗、単一化の失敗など) を捕捉し、その例外を報告することができるようにした。これにより分散環境でのサンドボックスモデルを記述することが可能になり、遠隔述語サーバなどのフォールトトレランスを意識しなければならない分散アプリケーションの構築を可能にした。6 節では、本例外処理機構を分散言語処理系 DKLIC に導入し、フォールトトレランスを意識した分散プログラミング例を示した。

8 今後の課題

今回はゴールの復帰をユーザにまかせ、また、ゴールが接続していた通信路に関しても閉じるという方法で解決をした。今後は、例外処理機構がゴール実行のスナップショットをとったり、ゴール間通信の内容を保持するなどの拡張を施して、ゴール実行全体の整合性をとれたゴールの復帰を実現する。

また、現在の例外報告では例外の種類は判明するが、その例外が例外処理対象のどのゴールで起きたかまでは知る事ができない。今後はそのような詳細な例外報告ができるようにする。

謝辞

本研究の一部は、文部科学省科学研究費基盤 (C)(2)11680370 の補助を得て行った。

参考文献

[1] 関田大吾, Inside KLIC Version 1.0. *KLIC Task Group*, AITEC/JIPDEC, 1998.

[2] http://www.klic.org/

[3] 瀧和男 編, 第 5 世代コンピュータの並列処理. bit 別冊, 共立出版, 1993.

[4] ICOT PIMOS 開発グループ, PIMOS マニュアル (第 3.0 版), 1991.

[5] 松村 量, 高山 啓, 高木 祐介, 加藤 紀夫, 上田 和紀, 分散言語処理系 DKLIC の設計と実装, 日本ソフトウェア科学会第 19 回論文集, 2002.