UML スクラップブックと
スナップショットプログラミング環境の実現

佐藤治[*]	Richard Potter[†]	山本光晴[‡]	萩谷昌己[§]
東京大学	科学技術振興事業団 さきがけ	千葉大学	東京大学

概 要

User-mode Linux の実行状態を保存、再実行できるシステム SBUML を実装した。本論文では SBUML の設計と手法について説明し、SBUML の各機能を提供するコマンドラインプログラムを組み合わせたスナップショットプログラミングについて実例とともに提案する。

1 はじめに

現在、システムソフトウェアの仮想化の分野に大きな着目が寄せられているということは周知の事実であろう。その顕著な例が VMware[11] に代表されるような「仮想計算機」と呼ばれるソフトウェア群である。これらのソフトウェアでは、仮想化されたハードウェアリソースを実際の計算機上に構成する。これによって、実際の計算機 (ホスト計算機) 上で全く別の独立した仮想的な計算機をいくつも走らせることが可能になる。

そのような仮想計算機に付与される機能の一つに、計算機の実行状態をホスト側のファイル形式として保存させるものがある。例えば、VMware においても仮想マシンのサスペンド機能は実装されていて、ユーザは任意のタイミングで仮想マシンを一時停止、再開させることが可能である。

一般に、仮想計算機において最も重要とされる機能は、「一つの物理的なコンピュータで複数のコンピュータの役割を果たす」というものである。この要求を満たす仮想計算機の多くには、状態保存機能は付加的なものとして付けられているに過ぎない。

[*] osamu-s@is.s.u-tokyo.ac.jp
[†] potter@is.s.u-tokyo.ac.jp
[‡] mituharu@math.s.chiba-u.ac.jp
[§] hagiya@is.s.u-tokyo.ac.jp

しかし、仮想計算機の状態保存機能に着目して考えると、実に様々な用途が期待出来る。例えば、計算機の状態を自在に保存、再実行できるこのようなシステムが得られると、一般的には以下に挙げるような用途が可能になると考えられる。

1. サーバシステムの障害からの復旧
   現在広く提供されているインターネットサービスのほとんどは、障害からの復旧時間が長ければ長い程、運営者の損害が大きくなる。あらかじめ障害が発生する前の時点での実行状態を保存しておき障害時にそれを復元することで、復旧時間を最小限に縮めることが可能である。
2. 教育用環境の構築
   サーバ管理者、またはソフトウェア開発者の教育の際に、各個人に対して全く同一の環境を提供する作業は非常に煩雑である。実行状態のスナップショットをコピーして配布することにより、被教育者全員に対して同じ環境を簡単に提供することが可能になる。
3. デスクトップ環境の利便性の向上
   仮想計算機の利用はサーバ環境のみにはとどまらない。近年の PC の性能の向上により、仮想計算機をデスクトップ環境で用いることももはや現実的である。オフィスの PC の実行状態をファイル形式で自宅にメール送信し、自宅で作業を継続する、といったようなことも可能である。
4. 侵入検知システム、ハニーポットの実現
   仮想計算機のスナップショットを静的に解析することで、その時点での仮想計算機の状態を容易に知ることが出来る。そのため、侵入検知システム、ハニーポットに対して応用することで、悪意のある攻撃によりサーバの状態がどのように変化したのかを知ることが可能となる。

表 1: SBUML 機能一覧

コマンド名	機能
sbumlfreeze	全 UML プロセスをサスペンドする
sbumlcontinue	全 UML プロセスをレジュームする
sbumlsave	UML の実行状態をスナップショットとして保存する
sbumlrestore	スナップショットの実行状態から UML を再実行する
sbumlanalyzesnapshot	スナップショットを静的に解析して情報を取り出す
sbumlschedule	指定した UML プロセスの優先度を変更する
sbumldonext	状態が遷移するまでの間指定した UML プロセスを実行し続ける
sbumlenumeratestate	指定した UML プロセスの全状態を列挙してスナップショットの木を構築する

5. サンドボックス技術への応用
   一般的に、仮想計算機からのホスト計算機の計算資源へのアクセスは制限される。よって、仮想計算機を一種のサンドボックスとみなすことが出来る。Java サンドボックスなどとは異なり、ネイティブコードを実行できる点、ファイルやネットワーク資源も仮想化される点が特徴として挙げられる。さらには、仮想計算機サンドボックス内で実行中のコードをスナップショットとして別のマシンに配布することも可能になるため、大きな利便性の向上が期待できる。

今回我々は、仮想オペレーティングシステム User-mode Linux(以下 UML) に対して拡張を行い、「Scrapbook for UML」(以下 SBUML) を開発した。このシステムによって、仮想 OS のプロセス、メモリ、ファイルシステムなどの全情報を自在に保存したり、情報のスナップショットから仮想 OS を再実行させたりすることが可能となる。

状態保存、再実行機能に加えて、SBUML に実装した全ての主要な機能に対して、コマンドラインプログラムとして外部インターフェースを提供した (表 1)。これらを用いることによって、シェルスクリプトなどのスクリプト言語などでも SBUML の操作を行えるため、状態保存、再実行を使った複雑なシステムの構築も容易に行うことができる。つまり我々は、SBUML の実装に加えて、スナップショットのレベルにおける新たなプログラミング環境を提供することになる。これは、スナップショットを利用したデバッギング環境を与える SBDebug[4] のアイデアを一般化したものと考えることが出来る。

我々が、既存の仮想計算機状態保存システムを用いることなく、UML を用いて新たな実装を行った理由がここにある。すなわち、VMware のような商用ソフトウェアにおいては、その内部実装が明らかでない為、それらを部品として新たなシステムを構築する際に自由度が大きく制限されてしまうからなのである。もちろん、先に列挙した状態保存機能の用途のいくつかは、既存の仮想計算機によっても充分実現可能である。しかし、状態保存機能を活用した更なるシステムを将来的に考える際に、実装がオープンな仮想計算機をベースに実装を行う方が都合が良い。

本論文の後半では、スナップショットプログラミングの一例として、実環境におけるモデル検査システムを挙げる。これは、SBUML の機能をシェルスクリプトによって統合することにより、SBUML 内部で実際に動作するアプリケーションプログラムの状態を網羅させるという試みである。これを実現するには、検証対象のプログラム内の幾多の状態のうち我々が着目する一部のみを取り出し、その遷移に基いたプログラム実行の制御を行う必要がある。これは既存の仮想計算機の状態保存機能を用いては実現不可能なものであり、我々が UML を用いて新しく実装を行った意義がここにある。

本節においては、我々の研究の背景と意義について述べた。第 2 節では、SBUML 実装の詳細を理解するのに必要とされる User-mode Linux 自体の実装を紹介する。第 3 節においては SBUML における状態保存の手法ならびに実装の詳細について説明する。第 4 節は、スナップショットプログラミングに必要なその他の付随する機能の実装について述べる。そして、以上の機能を統合する一例として、実環境モデル検査システムの実現について第 5 節で説明する。第 6 節以降では、SBUML の最適化の可能性や、SBUML モデル検査システムのこれからの見通しについて概略を述べる。

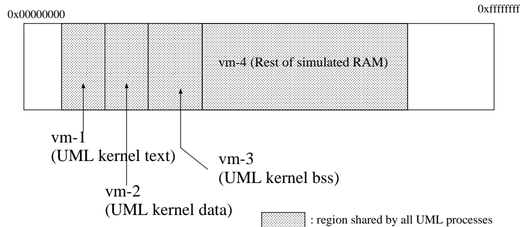

図 1: UML プロセスのアドレス空間

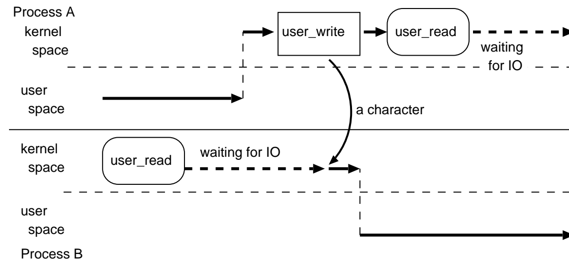

図 2: UML におけるコンテクスト切替機構の概略図

2 User-mode Linux

本節では、SBUML のベースとなる仮想オペレーティングシステム User-mode Linux について紹介し、重要ないくつかの実装について述べる。

User-mode Linux は、Jeff Dike によって開発されている Linux の拡張であり、Linux カーネルをユーザモードプログラムとして実行できる [1]。これを用いると、実際の OS(ホスト OS) の内部で、複数の仮想的な OS(ゲスト OS) を実行させせることができる。UML の拡張のほとんどは、アーキテクチャ依存コード (arch/um 以下) として記述されている。すなわち Linux のオペレーティングシステムとしての機構 (プロセス管理、メモリ管理など) は一般的な Linux カーネルと同一のものである。

2.1 メモリの仮想化

実際の計算機における物理メモリに相当するものは、UML においてはホスト OS 上のいくつかのファイルとして構成される。

具体的には、1 つの UML が起動される際に、

- カーネル実行形式の text 領域 (vm-1)
- カーネル実行形式の data 領域 (vm-2)
- カーネル実行形式の bss 領域 (vm-3)
- ユーザプロセス用として用いられる領域 (vm-4)

の 4 つのファイルが生成される。

vm-1, vm-2, vm-3 の領域は UML 内で実行されるプロセス (UML プロセス) の全てに共通して利用されるものである。また vm-4 ファイルは、UML のページサイズごとに分割され、UML のページング機構によって各 UML プロセスに細切れに割り当てられる (図 1)。

2.2 UML プロセスの構成

通常のオペレーティングシステムにおいてプロセスアドレス空間は、OS によって割り当てられたメモリ上に構築される。一方、UML プロセスのアドレス空間はホスト OS のプロセス (ホストプロセス) のアドレス空間の一部として構成される。UML 内部で 1 プロセスが新しく作成される度にホスト OS 側でも新たに 1 プロセス生成され、各 UML プロセスには、ホスト OS の mmap() システムコールにより UML カーネル実行形式の text, data, bss の各セグメントに相当する UML 仮想メモリが割り当てられる。また、UML のページング機構に従い、vm-4 ファイルの一部が UML ユーザプロセス用の領域として割り当てられる。

カーネル実行形式用の領域、すなわち vm-1, vm-2, vm-3 の 3 つのファイルは、実質全 UML プロセス間で共有される。このことは UML プロセスの実行モード切替のエミュレーションと深く関連している。実は、UML プロセスの SIGUSR1 シグナルハンドラ用スタックには、カーネル領域に存在するプロセスのカーネルモード用スタックが割り当てられている。すなわち、UML のカーネルモードはシグナルハンドラとして実装されているのである。

2.3 UML プロセスのコンテクスト切替

2.2 節で述べたとおり、1 つの UML プロセスは 1 つのホストプロセスのアドレス空間の中に構築され

る。しかしこのままでは、UML カーネルは UML プロセスの実行コンテクストを操作することができない。そのため UML ではホスト OS のパイプデータ構造を用いた特殊な手法でコンテクスト切替の強制を行っている。

UML カーネルが保存する task_struct 構造体の中には、1 プロセスにつき 1 組のパイプデータ構造 (switch_pipe) が含まれている。アクティブ状態にない全てのプロセスは、switch_pipe からの文字の読み込み待ち状態になっている。アクティブなプロセスは、UML スケジューラによって選択された次のプロセスの switch_pipe に対して 1 文字書き込みを行う。すると書き込まれたプロセスは読み込み待ち状態から解放され、アクティブ状態となり、書き込んだ側のプロセスは代わりに読み込み待ちを行う。

図 2 の概略図を用いて説明する。図に表されている user_read, user_write は、コンテクスト切替えをエミュレーションしている部分で呼ばれている内部関数であり、それぞれ switch_pipe から文字を読み書きしようとするだけのものである。アクティブなプロセス以外の全てのプロセスは、Process B で表されているようにパイプの読み込み待ち状態で停止している。Process A から Process B にコンテクストが切り替わるとき、A はまず始めに user_write を呼んで B の switch_pipe に一文字書き込みを行う。すると B はパイプから文字を読み込むことに成功し、アクティブ状態に移行する。一方 A は、user_write の後に呼ばれる user_read によって再び待ち状態に入ることになる。

このような仕組みにより、UML のユーザプロセスは常にアクティブなプロセスが 1 個だけ存在するという状態を保ち続けることが出来る。

2.4 UML のファイルシステム

仮想メモリファイルと同様に、ファイルシステムもまた、ホスト OS のファイル上に仮想ディスクとして構成されている。仮想ディスクへのアクセスというハードウェア的な部分をエミュレートするために、UML には IO thread と呼ばれるカーネルスレッドが用意されている。この IO thread によって、仮想ディスクへの読み書き及び後述する COW ディスクの処理などが行われている。

2.5 tracing thread プロセス

UML においては、ユーザモードプロセス、カーネルモードプロセスの他に、もう 1 つ、tracing thread と呼ばれるプロセスがホスト OS 上に存在する。tracing thread が存在する最大の理由は、UML カーネルが、ptrace() システムコールによるシグナルの捕捉を行っているためである。シグナルの捕捉を行わない場合、UML プロセスに対してシグナルが行われるとホスト側で設定されたシグナルハンドラが呼ばれてしまう。これを防ぐために、tracing thread はあらかじめ全 UML プロセスに対して ptrace() システムコールを発行する。tracing thread はループ状態で待ち、UML プロセスが受けたシグナルを捕捉し、UML で設定されたハンドラを呼ぶように UML プロセスに指示を与えるのである。

3 SBUML 基本機能の実装

前節で記述した UML の実装の特徴に基づき、我々は始めに SBUML の基本機能、すなわち状態保存、再実行の実現に取り組んだ。本節においては、SBUML 基本機能の内部実装の詳細について解説する。

3.1 UML 状態保存に必要な情報

UML の状態保存システムを実現するためには、保存しなければならないいくつかの情報がある。

1. UML 仮想メモリ
   前述したとおり、UML 仮想メモリは vm-1〜4 の 4 つのホストファイルとして実現されている。これら 4 つのメモリファイルを保存することでメモリ状態の保存は実現できる。

2. UML ファイルシステム
   UML のファイルシステムも、仮想ディスクとしてホストのファイル上に構築されているため、仮想ディスクファイルを保存することで状態保存は容易に行える。しかし、UML 仮想ディスクのファイルサイズは Gbyte 単位のもので、これを丸々保存することはホストの計算機への大きな負荷となる。よって、今回我々は、UML に元々実装されている COW(Copy on Write) ファイルシステムの利用を選択した。COW ファイ

ルシステムは、ベースとなるファイルシステムからの差分を構築するもので、作成されたディスクファイルの差分はホスト OS 上で sparse file となる為、非常に小さな領域しか占有しない。そのため、数百個単位ものスナップショットを保存することも現実的となる。

3. UML プロセスの状態
ここでいう「UML プロセスの状態」というフレーズにはいくつかの意味がある。

   - プロセスアドレス空間の復元
   前述したとおり、UML プロセスアドレス空間には、ユーザプロセスアドレス空間として vm-4 ファイルがページサイズの単位で細切れになっていくつもマッピングされている。原理的には、このマッピング情報を取得して再実行時に完全に復元できるようにすればよい。マッピング情報は、/proc ディレクトリの内容を参照するなどの方法でホスト OS 側から取得する方法が最も直接的である。しかし、UML プロセスのファイルマッピングは完全に UML カーネルのページング機構の振舞いに従っている為、ホスト OS 側からの情報取得の必要は無い。実装的な面でより簡潔に済ませる事が出来るという理由からも、UML カーネルの内部データを用いる方法を選択した。
   - 「ホストプロセスとしての」コンテクスト
   プロセスアドレス空間を保存したのみでは、UML プロセスの状態を完全に保存したことにはならない。全ての UML プロセスは、ホスト OS 側から見れば単なる 1 ユーザプロセスに過ぎず、ホスト側でのプロセスのレジスタ値などの実行コンテクストも保存しなければならないのである。この部分は、setjmp() 関数によって取得したコンテクストをスナップショット内に保存しておき、復元時に longjmp() 関数を呼ぶという方法で実現されている。

4. UML が使用しているファイルディスクリプタに関する情報
仮想メモリ、ファイルシステム、仮想端末用の xterm ウィンドウなど、UML 自体がホストのいくつものファイルディスクリプタを開いている。これらのディスクリプタに関する情報は UML の実行状態によって異なるため、スナップショットとして保存しておく必要がある。我々は、ファイルディスクリプタ番号とその形式に関する情報をディスクリプタの作成の度ごとに保存するような拡張を UML に対して加えた。これによって、現在の UML のファイル使用状況を保存することが可能になり、復元時に用いることが出来るようになった。

3.2 control thread プロセスの導入

我々が SBUML を実装するに際して直面した最初の問題は、外部とのインターフェースにまつわるものである。スナップショットの保存、再実行などの命令を UML に対して与える為には、UML の内部に我々の命令を受け付ける部分が存在しなくてはならない。UML には元々mconsole と呼ばれる機能が存在し、これによって UML 自体を操作することが出来た。これを拡張することが最も単純な手法ではあるが、mconsole 自体が UML カーネルスレッドとして動作している点、mconsole による接続は仮想的なソケット接続である点などを考えると、実装面での問題が多い。tracing thread プロセスの流用というアイデアも考えられるが、SBUML まわりの機能を明確にモジュール化するという意味も含めて、我々は全く別の手法を用いることにした。それが control thread プロセスの導入である。

control thread プロセスは、UML カーネルのイメージを共有しながらも UML カーネルスレッドとは全く異なる構造を持ったプロセスである。control thread プロセスは UML スケジューリングとは独立して動作し、対応する task_struct 構造体も持たない。control thread プロセスの大きな役割は、

- 外部からの命令 (サスペンド、レジューム、スナップショット保存などのコマンド) を受け付け、実際の処理を行う。
- スナップショット再実行の際に、全ての UML プロセスの雛型となり新しい UML プロセスを生成する。(詳しくは次節で述べる)

の 2 つである。SBUML に関わるこれらの重要な役割を control thread プロセスに分離して持たせることにより、実装をより明解にすることが可能になる (図 3)。

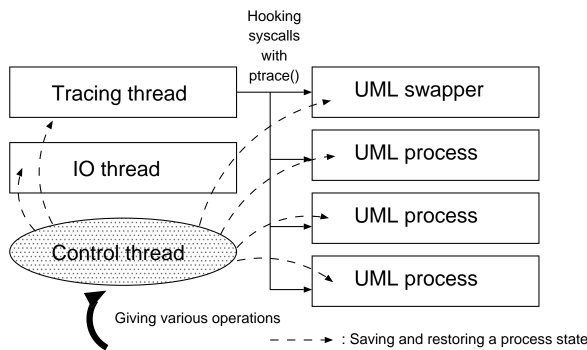

図 3: UML プロセスと control thread の位置付け
新たに導入した control thread は、SBUML の状態保存、再実行に関する総合的な処理を行う。

3.3 基本機能の実装

今回我々が SBUML を実装するにあたり、いくつかの実装過程を経た。ここではその実装過程を詳細に述べる。これらの全ての機能は、シェルスクリプトによって記述されたコマンドラインインターフェースとともに実装されている。各項目の括弧内にそれらのコマンド名を記述しておく。

1. サスペンド、レジューム機能 (sbumlfreeze, sbumlcontinue)
UML のスナップショットを作成する際には、各 UML プロセスが「安全な状態」であるということを保証する必要がある。ここでいう「安全な状態」とは、スナップショットの保存実行中にメモリ、ファイルシステムへの書き込みなどの変更が起こらないような状態のことである。その他にも、前述したパイプによるコンテクスト切替に関しての問題も起こる可能性がある。切替前のプロセスが switch_pipe に文字を書き込んでから、次のプロセスがその文字を読み込むまでの間の瞬間にスナップショット保存が行われてしまうと、スナップショットを復元した際にどの switch_pipe にも文字が存在しないこととなり、結果としてレジューム後の UML は正常な動作が得られない。つまり、単純に状態をファイルとして保存するだけではうまくはいかないのである。

   そのため我々は、全 UML プロセスを安全な状態にするための機能を付加した。前述した通り、全 UML プロセスは UML カーネルイメージを共有している。そのため、UML カーネル領域に 1 つフラグを用意し、外部からのインターフェースを介して control thread プロセスがフラグの切替えを行う。各 UML プロセスは、そのフラグが立ったときにのみ一定の場所でループを繰り返すようにする。これで、サスペンド、レジューム機能を実装することが出来たことになる。

2. スナップショットの保存 (sbumlsave)
sbumlfreeze, sbumlcontinue コマンドにより、UML プロセスを自在に安全な状態にすることが可能となった。3.1 節で述べた全状態のバックアップを取ることによってスナップショットの保存を容易に行うことが出来る。

3. スナップショットからの UML 再実行 (sbumlrestore)
今回実装した機能の中で最も煩雑なものが、UML 再実行機能である。これは、あらかじめ保存したスナップショットと同じ実行状態をもった UML を新しく作成する機能である。新しく作成された UML は、スナップショット作成時の状態から再び実行を開始する。我々の実装においては以下に挙げるようなステップを踏んでいる。

   (a) UML カーネル起動直後に、ファイルディスクリプタ情報を復元する。これにより、仮想メモリ、仮想ディスク上のデータに関しては即座に使用できるようになる。

   (b) UML 実行時に最初に作られる tracing thread プロセスから、control thread、及びスナップショットからは作成できないプロセス (swapper プロセス、IO thread。これらは UML スケジューリングに無関係であり、かつ他の UML プロセスとはアドレスマッピングの方式が異なる為。) を作り出す。

   (c) control thread は、スナップショットのプロセスに関する部分の情報を参照して、自らが雛型となりホストの clone() システムコールを呼び出す。control thread 自身には既にカーネルイメージがマッピングされ

ている為、ここまででカーネル部分の復元は完了する。

(d) clone された各プロセスは、まずカーネルの情報を用いて自らのプロセスアドレス空間を復元する。その後、共有されたカーネルの大域変数を介して、tracing thread に対して自らを ptrace するように伝える。最後に、スナップショット内部に保存された jmp_buf を取り出し、それを用いて longjmp() 関数を呼び出す。

(e) 全てのプロセスの復元が完了した後、UML 再実行コマンドを実行し、新しい UML の実行状態を継続する。

これらの操作を UML 内部に実現することにより、UML 再実行を実現した。

我々が開発に用いたノート PC 環境 (Pentium 4 2GHz, 1GByte RAM) では、UML スナップショットの保存、再実行ともに約 10～20 秒程度の処理で行うことが出来た。同程度のスペックの環境で VMware のレジューム機能を用いた時とほぼ同じ位の実行時間がかかる。これは SBUML に未だ最適化の余地が残されている事を考慮すると充分に高速な処理と考えられる。

また、VNC[12] を用いた Linux デスクトップの保存、再実行 (図 4) も行った。すなわち、VNC サーバを稼働させた状態の SBUML の状態保存を行い、Linux デスクトップ環境を保存させるものである。これに関しては、VNC サーバの状態保存のみを行う SBUML の場合約 10～20 秒程度とほぼ変わらないのに対して、ディスプレイ周りのエミュレートを行っている VMware では倍以上の処理時間を要した。

4 付随する機能の実装

以上までで、UML の状態をスナップショットとして保存、再実行する機能を実現することができた。本節においては、この SBUML によるスナップショットプログラミング環境を実現する為に必要な、いくつかの追加機能について説明する。特に 4.2 節に挙げる機能は、UML 内部の 1 プロセスの状態を操作、抽出するためのもので、スナップショットプログラミングの一例として後に挙げる SBUML モデル検査システムを実現するために非常に重要なものである。

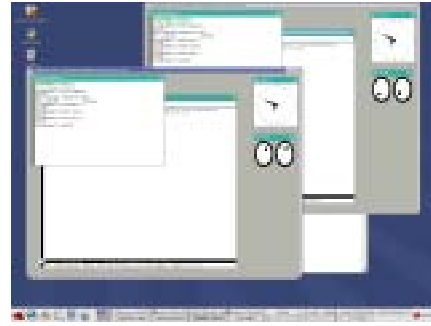

図 4: SBUML による VNC デスクトップの保存
1 つのスナップショットから全く同じ内部状態を持つ 2 つ以上の UML 仮想機械を作ることも可能。

4.1 UML スケジューラの操作

我々は付随機能の一つとして、UML スケジューラを操作できる機能を付加した。これを導入した一つの大きな理由は、我々のシステムの応用例である、後述する SBUML 実環境モデル検査システムの実現の為には、UML の各プロセスのうち指定されたもののみを実行させる、という手段が必要だったからである。

そのために我々は、Linux カーネルのスケジューラに拡張を施し、指定したプロセスの優先度を上下させる機能を追加した。

4.2 UML スナップショットからの静的情報抽出

SBUML システムにおけるスナップショットは、フォーマットが明らかである為、UML カーネルの情報、あるいは各 UML プロセスに関する情報を容易にスナップショットから抽出出来る。SBUML の再実行のコストを削減する為に、UML 自体を実行することなしにその内部情報を取り出す機能、すなわち静的な情報の抽出機能は、我々の目的を実現するには重要な要素であると考えられる。以下に我々が用いた例を挙げる。以下のコードはいわゆる Dining Philosopher Problem と呼ばれるものである。

```
void *philo(void *arg)
{
/* ... */
while (1) {
```

```
read(pipes[i%NUM].fd[0], &c, 1);
printf("%d takes his left fork...\n", i);
read(pipes[(i+1)%NUM].fd[0], &c, 1);
printf("%d takes his right fork...\n", i);
printf("%d waiting...\n",  da->id);
sleep(1);
write(pipes[i%NUM].fd[1], &c, 1);
printf("%d leaves his left fork...\n", i);
write(pipes[(i+1)%NUM].fd[1], &c, 1);
printf("%d leaves his right fork...\n", i);
sleep(1);
}
return 0;
}
```

このコードが実行された時に生成されるプロセスにおいて、注目される内部状態は次のものに限定することが出来る。

- 各プロセスが実行されているコード上の位置
- 全プロセスで共有されているパイプの中身の情報

これらの情報は、両者ともスナップショット内のメモリイメージから直接取得することが出来る。上の例では、Dining Philosopher Problem に特化させた状態抽出用のプログラムを SBUML とは別途に作成し、その中で、我々が必要とするこれらの情報を取り出せるようにした。すなわち、この状態抽出用プログラムにより、我々が注目すべき情報を明示的に指定したことになる。この技術を汎用化させて、UML カーネルが持つ各 UML プロセスの情報を自在に取り出したり、task_struct 構造体内にあるメモリマッピング情報を取得することにより、UML プロセスのアドレス空間を参照し、プロセスのユーザアドレス空間の情報を得たりすることが可能になると考えられる。

4.3 UML プロセスのステップ実行

以上に挙げた二つの機能を用いて、我々は、UML 内部の一つのプロセスのみをステップ実行させる機能を実現した。ここで言う「ステップ実行」とは、デバッガにおけるステップ実行とは意味が異なり、「一つのプロセスの注目すべき状態が変化するまでの間実行を続ける」という意味である。これを実現することが出来ると、あるプロセスの状態遷移を表現するグラフを記述することが可能となる。

実現方法は以下の通りである。

1. 始めに、現在の UML の状態をスナップショットとして保存しておく。
2. 指定されたプロセスに関して UML スケジューラ操作機能を用い優先的に実行する。この実行は、次回の UML のタイマー割込み発生まで行われる。
3. 実行後の状態と最初に保存したスナップショットから静的情報抽出を行い、異なる情報が得られるまでスケジューラ操作を繰り返す。

これによって、対象プロセスの注目する状態が遷移するまでの 1 ステップのみの実行を実現することが可能となった。

5 シェルスクリプトによる統合の応用例：SBUML 実環境モデル検査システム

以上に示したような様々な機能を実装した我々は、SBUML によるスナップショットプログラミングの一例として、また、状態保存システムの新たな可能性を示す応用例として、SBUML 実環境モデル検査システムの実装に取り組んだ。

モデル検査 [10] は、有限状態システムが安全性などの性質を満たすかどうかを、主に状態空間の網羅的探索を用いて自動検証する技術である。一般的なモデル検査では、検証対象のシステムは抽象的なモデルとして専用の言語を用いて記述され、そのモデルに対して検証が行われる。そのため、モデル検査は設計段階における誤りの早期発見に関して成功を収めてきた。

しかし近年、特にソフトウェア検証において、設計段階だけではなく実装段階にもモデル検査技術を応用しようという動きが広まってきている。すなわち、汎用のプログラミング言語のコードやバイトコードといったものを検証対象のシステムの記述として扱うのである。例えば、CMC(C Model Checker) と呼ばれるシステム [2] においては、C 言語のソースコードから実際にコンパイルされたバイナリを実行し、想定される状態を網羅することによりモデル検査を行っている。

今回我々が目指したシステムも、実装段階におけるモデル検査を目的とした、実環境を用いたモデル

検査システムである。すなわち、実際にコンパイルされたバイナリを、仮想オペレーティングシステム上の実際の実行環境で動作させることにより、モデル検査を行うのである。前述の CMC においては、状態遷移の道筋を記憶して再実行することにより、それまでに現れた状態の再現を行っているが、我々の SBUML においては、UML のいかなる瞬間の実行状態も保存、再現することが出来るため、再実行による時間的コストを削減することができ、より高速なモデル検査を行うことが出来る可能性が高い。

以下に挙げるコードは、我々が SBUML コマンドラインインターフェースを用いて記述した、SBUML モデル検査システムの中心となる部分のシェルスクリプトのコードの概略である。モデル検査システムのようなシステムを、SBUML コマンドラインインターフェースを用いて簡潔に記述出来ることを表している。

```
#!/bin/sh
#
# sbumlenumeratestate
#

mname=$1
hashfile=$2
parentss=${3:-0}
pname=${4:-0}

list=`sbuml--choices $mname`
[ -z "$list" ] && exit

ntime=`date +%s`
sbumlsave $mname $ntime -c -f

OLDIFS=$IFS
IFS=,
for i in $list ; do
sleep 1
sbuml--donext-unless-changing-state \
$mname $i
hash=`cat sshash.tmp`
rm -f sshash.tmp
echo "$parentss", "$pname" > ssinfo

if grep $hash $hashfile > /dev/null
then
exit 0
else
echo $hash >> $hashfile
sbumlenumeratestate \
$mname $hashfile $ntime $i
fi
sbumlrestore $mname $ntime -c -f
done
IFS=$OLDIFS
```

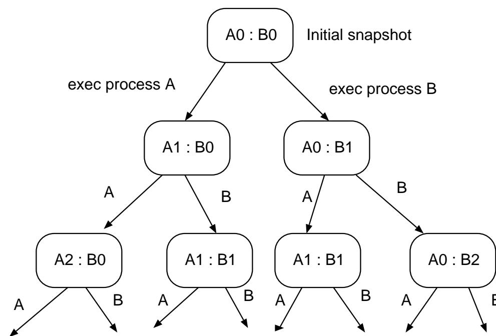

図 5: スナップショット木の生成による状態列挙

内容を簡潔に説明すると、引数として与えられたスナップショットを基準として、実行可能な各プロセスに対して状態遷移が起きるまでステップ実行し、実行後のスナップショットを保存し、それを引数に再帰的にスクリプトを呼び出す、というものである。例えば、我々が注目する実行可能なプロセスが A, B の 2 つのみ存在したとする。初期状態のスナップショットから A の 1 ステップ目を実行したスナップショット、B の 1 ステップ目を実行したスナップショットの 2 つを生成する。前者のスナップショットからは、A の 2 ステップ目、B の 1 ステップ目を実行したスナップショット、後者からは A の 1 ステップ目、B の 2 ステップ目を実行したスナップショットをそれぞれ作ることが出来る。これを繰り返すと、一般には無限のサイズの状態をノードとする木になるが、スナップショット内の状態のごく一部に着目して、着目した状態に関して以前現れたスナップショットと同一のものが現れた時点で枝刈りを行うことで、有限サイズの木を構築できる (図 5)。

現在までに我々は、4.2 節で紹介した Dining Philosopher のコードを SBUML 内で実際に実行して状態を網羅することにより、スナップショットの状態遷移に基づいた木を構築することに成功した。またそのスナップショットの木から、デッドロック状態が起こりうるスケジューリングを検出することにも成功した。

6 今後の課題

本研究においては、UML の実行状態を自在に保存、再実行できるシステム SBUML の設計と実装を

行った。さらに、スケジューリング操作、静的状態抽出の機能とともにコマンドラインインターフェースを実装し、それらを組み合わせたスナップショットプログラミングの例を示した。

我々には今後 2 つの大きな課題が残されている。

1. SBUML の最適化
現在の SBUML の実装はプロトタイプという意味合いが強い。SBUML を操作するインターフェースのほとんどはシェルスクリプトで記述され、速度の面から考えても最適とは言い難いものである。また、SBUML 自体の実装もまだまだ最適化の余地は多い。その中の一つが、スナップショットのファイルサイズの問題である。スナップショットイメージは全体で 30MB 以上にもなるもので、何百個以上もスナップショットを保存したりする状況を考えるとファイルサイズの問題は無視できないものとなる。差分スナップショットの生成、スナップショットの圧縮などの機能を実装しなければならない。

2. SBUML 実環境モデル検査システムの実装
SBUML の最適化と同時に、現在我々は、SBUML を用いた実環境モデル検査システムのさらなる実装を続けている。本論文ではスナップショットプログラミングの一例として挙げたに過ぎなかったが、今後の目標として、この SBUML モデル検査システムを実用的なサーバプログラムに対して適用し、具体的なバグを発見することを考えている。そのためには、今回実現した静的状態抽出の手法を更に発展させて、サーバの内部変数なども取得出来るように拡張を加える必要がまず考えられる。また、着目するサーバの内部状態、状態遷移の 1 ステップの定義など、様々な課題が残っている。

7 関連研究

我々以外にも、UML の実行状態保存を行うアイデアは存在する。例を挙げると、Linux カーネルのソフトウェアサスペンド拡張 (swsusp)[6] を UML に移植したり、UML の内部からではなく、CRAK[8] に代表されるような既存のアプリケーション状態保存システムを用いて外部から checkpointing を行ったりするものなどである。しかし、これらのアイデアは未だに完成段階には至ってはいない。その理由として最も大きなものと考えられるのは、User-mode Linux, また Linux カーネル自体のバージョンの進行があまりにも速く、追随が困難であるという事だろう。

また、我々と類似の研究として挙げられるものとしては、ネットワーク上を渡り歩けるコンピュータ (Network Transferable Computer)[7] が挙げられる。この研究においては、VMware, Linux, swsusp を組み合わせて用い、VMware 内部で動く Linux の実行状態をネットワーク上で転送できるようなシステムを構築している。また、SoftwarePot[9] というシステムでは、ソフトウェアと仮想的なファイルシステムの情報をアーカイブし、仮想環境の保存、移動を実現している。Computation Scrapbook の研究 [3, 4, 5] においては、Emacs Lisp の実行状態を保存、再実行するシステムを構築し、コードの可視化、動的コード生成などのプログラムデバッギングへの実用例を示している。

8 おわりに

本研究において、User-mode Linux の実行状態をスナップショットとして保存、再実行できるシステム SBUML を実装し、SBUML を用いたスナップショットプログラミング環境の実現を行った。

SBUML は現在、いくつかの動作安定化、高速化のための修正が加えられている。近日中に公開することができるであろう。

謝 辞

本研究は、科学技術振興機構　戦略的創造研究推進事業　個人型研究（さきがけタイプ）の研究助成を受けて実施されたものです。また、本論文に対して有益なコメントを頂いた筑波大学の加藤和彦先生、新城靖先生には感謝します。

参考文献

[1] The User-mode Linux Kernel Home Page [http://user-mode-linux.sourceforge.net/]

[2] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. “CMC: A Pragmatic Approach to Model Checking Real Code”, Usenix Association, OSDI 2002.

[3] Richard Potter, “Computation Scrapbook of Emacs Lisp Runtime State”, Symposia on Human-Centric Computing Languages and Environments, September 2001.

[4] Richard Potter and Masami Hagiya, “Computation Scrapbooks for Software Evolution”, Fifth International Workshop on Principles of Software Evolution, IWPSE 2002, 143–147, May 2002.

[5] Richard Potter, “Computation Scrapbooks”, 第4回プログラミングおよび応用のシステムに関するワークショップ SPA 2001, March 2001.

[6] Software Suspend for Linux [http://swsusp.sourceforge.net/]

[7] Network Transferable Computer [http://staff.aist.go.jp/k.suzaki/NTC/]

[8] Hua Zhong and Jason Nieh, “CRAK: Linux Checkpoint/Restart As a Kernel Module”, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001.

[9] Kazuhiko Kato and Yoshihiro Oyama, “SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation”, Software Security — Theories and Systems, Volume 2609 of Lecture Notes in Computer Science, Springer-Verlag, February 2003.

[10] Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled, “Model Checking”, The MIT Press, 1999.

[11] VMware [http://www.vmware.com/]

[12] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood & Andy Hopper, “Virtual Network Computing”, IEEE Internet Computing, Vol.2 No.1, Jan/Feb 1998 pp33–38.