プログラムデータベースを利用したオブジェクト指向プログラムのデバック

7M－7

大島 満
日本アイ・ビー・エム (株) 東京基礎研究所

1 はじめに

オブジェクト指向言語のプログラムを効率的にデバッグするには、オブジェクト指向特有の概念を利用することが重要かつ効果的である。例えば、インヘリタンスやメソッド情報に関する情報の利用や、注目するオブジェクトの自動的なクラス判定、クラスやオブジェクトに対するブレイクポイントや表明 [1] などが有効であろう。しかし、これらの機能を実現すためには、クラス、メソッドやインスタンス変数などに関する情報が必要になる。

しかし、C++のようなハイブリッド型でかつ実行効率を重視している言語では、実行時にこのようなメタな情報が扱えない場合が多い。

本稿では、C++のプログラムデータベース [2] をデバッガとともに使用することによって、オブジェクト指向プログラムを効率的にデバッグする利点とその実現について述べる。

2 ＯＯプログラムのデバッグ

オブジェクト指向のプログラムをデバッグするためには、クラスやオブジェクトなど抽象化されたレベルでのデバッグが必要になる。以下では図1のようなC++のプログラムに対していくつか問題になる例をあげてみる。

2.1 クラスに対するデバッグ

`(Base_A::B > 0)` のようなクラスに対する不変表明を侵している場所を知りたい要求があったとしよう。これを発見するために必要な手続きは、

1. **Base_A** のクラスに属するオブジェクトに対する操作を行なっている場所を (この場合はメンバ変数 B に対する操作) 見つけ出す。このとき、操作はサブクラスでおこなわれる可能性があるので、自分のサブクラス内のメソッドやサブクラスに対する操作も含めて調べる。

2. 該当場所を通過した時に表明を判定する処理を呼ぶ設定を行なう。

3. 実行時、表明に違反していればユーザーに報告する。

```
class Base_A {
    int B;
    void bar();
};
class Base_B {
    void foo();
    virtual void vfunc() = 0;
};
class Derived_1 : public Base_A, public Base_B {
    virtual void vfunc();
};
class Derived_2 : public Base_B {
    virtual void vfunc();
};
```

図 1: サンプルプログラム

である。この場合、**Base_A::bar()** だけでなく、**Derived_1::vfunc()**, **Derived_2::vfunc()** を調べる必要がある。public メンバの場合はさらにこのクラスおよびサブクラスを使用している場所を探しだす必要がある。通常の C++ではメンバ変数へアクセスしている場所や、クラス階層などの情報が実行時に保存されていないので、1) の手続きは実質的に人が行なうことになる。

また、クライアントに対するクラス不変表明などは public なメンバ変数およびメンバ関数に設定すれば十分であるが、これらの情報も実行時にはわからない。

2.2 オブジェクトに対するデバッグ

クラスではなくオブジェクトに設定したい場合を考える。この場合、クラスに対する手続きに加え、

- ポインタが指しているオブジェクトが注目しているオブジェクトかどうかを判断する。

という手続きがさらに必要になるが、通常の C++ではオブジェクトが属しているクラスに関する情報がないため、オブジェクト生成時にそのクラスをどこかに記録する必要がある。しかし、C++で多重継承を使用すると、オブジェクトのポインタに対する操作が行なわれるので、単に生成時のポインタを記憶しておくだけでは不十分である。(実際は処理系に依存する) たとえば、**Derived_1** の this で示されているポインタ

Debugging Object-Oriented Programs using a Program Database.
Misturu Oshima.
IBM Research, Tokyo Research Laboratory.

は、**Base_B** のメソッドの this とは、違う場合がある。よって生成時には、関連するすべてのクラスとオフセットを調べ、ポインタとクラスの関係を登録する必要がある。

また型判定が可能になると、C++ではソースコード上で現れる型としてしか見ることの出来ないオブジェクトを、実際のクラスのオブジェクトへ自動的にキャストして表示することが容易になるなど、別の利点もある。

2.3 仮想関数へのブレイクポイント

仮想関数に対してブレイクポイントを設定する方法として、2つのアプローチが考えられる。一つは、仮想関数を再定義しているすべてのクラスを探しだして、そのクラスのメンバ関数に対してブレイクポイントを設定する。または、仮想関数を呼び出している場所をさがし出し、呼び指す前にブレイクポイントを設定する。いずれにせよ、プログラムに関する情報が必要になる。

ここで上げた問題がすべてではないが、プログラムの情報を得ることは、ここまでに述べた手続きの自動化を可能にし、より抽象化されたレベルでの効率的なデバッグが可能になると考えられる。

3 プログラムデータベースの使用

プログラムに関する情報を取り出す方法として、1) 簡単なパースで得られた情報をもとに on demand で情報を検索する方法や、2) ソースコードに専用コードを挿入・コンパイルし、実行時情報を直接取り出す、などが考えられるが、我々はプログラムデータベースを使用するアプローチをとった。

プログラムデータベースにはあらかじめ解析したプログラムの構造をデータベースに保存しておく。デバッグ時にこれを使用することにより、指定したクラスのメンバや使用されている場所、多重継承時のオフセットなどの情報を容易に取り出すことが出来る。

プログラムデータベースの利点は、情報検索のパフォーマンスや保存効率が良い、実行する前に必要な情報が得られる、などがある。逆に現在の欠点として、データベースの作成のオーバーヘッドがあげられるが、これは現在改善されつつある。だたし、いずれにせよデータベースの作成には時間がかかるので、目的を絞って light-weight-parsing など必要最低限の情報だけ取り出すなどの方法も検討する必要があるだろう。

4 統合環境

デバッグの機能を効果的に使用するためには、プログラムの情報を見るためのブラウザや連動して動作するエディタなどを自由に往来でき、各々の情報を相互に参照できる、統合化された開発支援環境も重要である。しかし統合化の方法によっては、拡張性が低い、結局ユーザーとって使いづらいものになる可能性がある。その他、ユーザーが自由に表明や、独自の機能を記述するためのデバッグ言語も必要である。

我々はこれら拡張性および言語機能を考え、Tcl[3] をコアシステムとして使用することにし、デバッグ機能および Program Database にアクセスする機能は、Tcl の拡張機能としてインターフェイスを作成した。そして、前述したクラスやオブジェクトに対するデバッグ操作は Tcl の関数で記述した。これにより、デバッグ機能は特定のブラウザやプログラムデータベースにも依存することなく実現され、拡張のインターフェイスさえ同じであれば、同じ Tcl のプログラムを利用することが出来る。ユーザーは自分でプログラムを記述することにより、自分の好きなエディタからデバッグ機能を使用でき、またユーザー定義の拡張などを導入することが出来る。

5 関連研究と考察

オブジェクト指向プログラムのデバッグに関して [4] は、視覚化が効果的な方法であると報告している。我々は、これらの視覚化技法は、デバッグ機能とスクリプト機能を用い視覚化情報を取り出すことで、容易に実現可能であると考えている。また、この方法を用いると実際のプログラムに手を加えない、再コンパイル不用といった利点もある。

現時点の問題点は、データベース作成時のオーバーヘッドと、データベースの情報とデバッガが利用している情報に重複があることである。現在、Debugger および ProgramDatabase に対するインターフェイス部がほぼ完成しており、アプリケーションとしての debugger の部分を現在開発中である。

参考文献

[1] Bertrand Meyer: オブジェクト指向入門 (Object-Oriented Software Construction)

[2] Tamiya Onodera: Experience with Representing C++ Program Information in an Object-Oriented Database, OOPSLA'94.

[3] John K. Ousterhout: Tcl and the Tk Toolkit, Addison-Wesley Publishing Company (1994).

[4] Chris Laffra, Ashok Malhotra: HotWire – A Visual Debugger for C++, C++ Technical Conference 94.

[5] Margaret A. Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual