Двухуровневая схема отладки

М.С.Карташев
perez@tercom.ru
Санкт-Петербургский государственный университет
198504, Университетский пр., 28
Санкт-Петербург, Россия

Аннотация

При разработке отладчиков для программ на языках высокого уровня, оттранслированных в машинный код, используются приемы, ставшие почти стандартными. Однако в ряде случаев использование "стандартного" подхода к реализации функций отладчика неэффективно или невозможно; одним из таких случаев является отладка языков со сложными проекциями типов данных и операторов. В статье представлен подход к реализации функций отладчика для таких языков по двухуровневой схеме, т.е. на основе существующего отладчика объектного языка.

Введение

Промышленный процесс разработки программного обеспечения обязательно включает стадию отладки, т.е. поиска и исправления ошибок. В свою очередь, важным этапом при исправлении ошибки в исходном коде является ее локализация — нахождение того места в исходном коде, где она была допущена. Одним из основных инструментов в таких случаях служат отладчики.

В связи с тем, что отладчики стали неотъемлемой частью инструментария разработчика ПО, многие существующие и разрабатываемые [10] процессоры имеют встроенную поддержку некоторых распространенных отладочных функций [2], таких, как пошаговое исполнение программы, установка точки останова, наблюдаемые переменные (т.е. области памяти заданного размера). Такой набор функций представлен практически в любом

современном отладчике (аппаратно реализованный отладочный интерфейс можно также считать отладчиком машинного языка), поэтому наряду с проблемой эмуляции таких функций (если они не реализованы аппаратно) стоит не менее актуальная проблема разработки отладчика исходного языка на основе имеющегося отладчика объектного, который может быть как машинным языком, так и языком высокого уровня.

Некоторые языки хорошо согласованы по типам данных и операторам, т.е. типы данных и операторы одного языка имеют аналоги в другом языке. Примером таких языков могут служить C и язык ассемблера некоторых процессоров. В случае, если исходный и объектный языки хорошо согласованы, способ реализации функций отладчика проработан и представляет лишь техническую сложность. Основываясь на том, что машинная инструкция вызова может появиться в объектном коде только в качестве проекции вызова в исходном коде, реализация команды пошаговой отладки с остановкой в вызванных процедурах может "предсказывать" вызовы и заранее устанавливать точку останова по адресу перехода[1]. Такой способ реализации, помимо прочего, предполагает легкость анализа объектного кода: найти инструкцию вызова в заданных пределах (соответствующих одной строке исходного кода) не представляет никакой сложности.

Однако существуют сочетания языков, не столь хорошо подходящие для создания транслятора с одного языка в другой. Когда типы данных и операторы исходного языка не имеют очевидных аналогов в объектном, проекции получаются сложными и громоздкими. Например, если в исходном языке семантика присваивания переменной, связанной с полем для редактирования в окне приложения, включает немедленное обновление этого поля при изменении связанной с ним переменной, проекция даже простого обнуления целой переменной будет состоять из множества операторов такого языка, как, например, C, и среди этих операторов почти наверняка будут вызовы библиотеки поддержки времени исполнения. Таким образом, проекция большинства операторов представляет собой один или несколько вызовов. В этом случае способ реализации функций отладчика, применимый для языков с простыми проекциями, не работает. Дополни-

[1]Безусловно, кроме "предсказания" вызовов, для корректной работы отладчика необходимо отслеживать "длинные переходы" (функция longjmp() языка C).

тельную трудность в его применении представляет анализ объектного кода, так как даже введение соглашения на именование пользовательских и системных процедур, позволяющее отладчику отличать нужные ему вызовы, не решает проблемы полностью — остается проблема поиска оператора вызова в целевой программе. В случае с ассемблером проблем не возникает, но что делать, если объектный язык — язык высокого уровня? Возможным решением данной проблемы мог бы быть анализ исполняемого (машинного) кода, полученного из объектного. Однако для этого нужно знать проекции конструкций объектного кода в машинный код, а контролировать трансляцию объектного кода в исполняемый, как правило, невозможно, так как это осуществляется другим транслятором.

Еще одним препятствием к применению обычного способа реализации функций отладчика является использование объектного языка, для исполнения которого нужна виртуальная машина. В этом случае использование предоставляемого ею отладочного интерфейса является практически единственным выходом.

Кроме того, при трансляции в языки высокого уровня может оказаться, что для выбранного объектного языка отладочный интерфейс разработан, отлажен и давно применяется на практике, а поэтому его использование в качестве базы для построения отладчика исходного языка не только оправдано, но и предпочтительно.

1 Описание задачи и области применимости подхода

Идея двухуровневой отладки непосредственно следует из двухуровневой схемы трансляции, суть которой заключается в том, что программа на *исходном* языке транслируется в код на некотором другом языке (*объектном*, *целевом*), для которого существуют как компилятор (в *исполняемый* код), так и отладчик. Предполагается, что проекции типов данных и операторов исходного языка сложны и большинство из них включает вызовы процедур библиотеки поддержки времени исполнения.

Кратко задача, решаемая любым отладчиком уровня исходного языка, может быть сформулирована следующим обра-

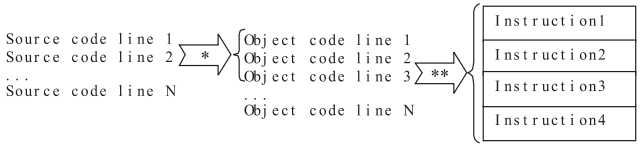

Рис. 1: Проекция исходного кода в исполняемый через объектный

зом: отладчик должен представлять пользователю информацию об отлаживаемой программе в терминах исходного языка, что включает как указание текущего положения в *исходном* коде, так и выполнение команд отладки в этих же терминах. Таким образом, команда пошаговой отладки с остановкой на том же уровне стека вызовов (часто называемая *step over* или просто *step*) должна принимать во внимание изменение уровня стека не объектной программы, а исходной, в то время как стек вызовов исходной программы реально не существует и должен быть каким-то образом реконструирован на основе стека вызовов объектной.

Задача построения отладчика исходного языка может быть решена на основе предлагаемой двухуровневой схемы, в рамках которой предлагается разработка отладчика исходного языка на основе имеющегося в наличии отладчика объектного языка. Безусловно, последний должен предоставлять интерфейс доступа к ряду своих функций, которые будут перечислены ниже. Прежде чем приступить к вопросам реализации отладчика исходного языка, требуется сформулировать некоторые необходимые условия применимости предлагаемой схемы.

1.1 Требования к отладчику объектного языка

Основным требованием к отладчику объектного языка является возможность работы с программой на объектном языке в терминах этого языка. Это вытекает из того, что соответствие объектного кода исполняемому (см. (**) на рис. 1) является “знанием”, доступным лишь второму транслятору, что исключает возможность, например, установить точку останова в исходной программе по адресу машинной инструкции — адрес этой инструкции отладчику исходного кода не известен.

Перечисленные ниже функции составляют полный базис, на основе которого будет строиться отладчик исходного кода:

- запуск и принудительное завершение приложения, а также временная приостановка его работы с возможностью последующего возобновления;
- установка и снятие точки останова по спецификации точки объектного кода (номера строки, сигнатуры функции и т.п.);
- получение информации о работающих потоках в приложении, состоянии этих потоков, стека вызовов каждого потока и текущего положения в объектном коде для рамки каждой процедуры на стеке.

Эти требования являются основными и необходимыми, но практических ограничений на отладчик объектного языка они не накладывают, так как подавляющее большинство отладчиков располагает перечисленным выше набором функций.

Остальные требования, перечисленные ниже, могут существенно облегчить реализацию некоторых функций, но обязательными не являются:

- возможность получать уведомление о выходе из процедуры объектного кода (т.е. об уменьшении стека вызовов);
- выполнение пошаговой отладки с остановкой на следующей исполняемой строке на том же или объемлющем уровне стека вызовов (step over);
- возможность временной отмены установленной точки останова. Наличие такой функции существенно только в том случае, если она выполняется быстрее, чем снятие и установка точки останова.

Кроме перечисленных требований к отладчику объектного языка для применимости описываемого подхода необходимо выполнение некоторых соглашений о генерируемом транслятором объектном коде.

- Необходимо предоставить отладчику возможность отслеживать инициализацию (загрузку) нового модуля объектного кода. Это может быть реализовано путем генерации

транслятором вызова определенной функции библиотеки поддержки времени исполнения в точках входа в модуль, а отладчик может отслеживать такие вызовы, например, путем установки точки останова на упомянутую функцию.

- Желательно иметь легко реализуемую возможность выяснять, является ли тот или иной модуль образом исходного кода (например, путем введения специального именования сгенерированных модулей или предварительной регистрации всех "интересных" пользователю модулей в отладчике).

- Транслятор исходной программы должен генерировать отладочную информацию для отладчика исходного кода, содержащую, как минимум, описание соответствия исходного кода объектному (иногда достаточно соответствия номеров строк).

2 Формализация понятий предметной области

Для предлагаемого в этой статье отладчика отлаживаемая программа состоит из следующих объектов:

- одного или нескольких модулей, являющихся результатом трансляции программы на исходном языке;

- потоков;

- стека вызовов (для каждого потока) и рамок вызванных процедур.

В этот список не входят файлы исходного кода, так как здесь рассматривается отладчик низкого уровня и в его функции не входит работа с исходным кодом. Эта роль отводится отладчику верхнего уровня, непосредственно с которым уже и будет работать пользователь.

Рассмотрим упомянутые выше объекты подробнее.

Понятие модуля в некотором смысле инвариантно относительно языка, в то время как понятие потока и статики процедуры (рамки стека вызовов) в терминах исходного и объектного кода отличаются.

Для удобства определения новых понятий начнем со стека вызовов и рамок вызываемых процедур.

Фрагмент исходного кода	Сгенерированный объектный код

`PROC LOCAL_PROCEDURE(I INTEGER)`
`  IntEditField = I`
`ENDPROC`
 	
`...`
`SET COUNTER := COUNTER + 1`
`LOCAL_PROCEDURE(COUNTER)`
	`public void LocalProcedure(MyInteger I)`
	`{`
	`IntEditField.assignIntValue(I.getIntValue());`
	`}`
	
	`public void run()`
	`{`
	`  ...`
	`  COUNTER.assignValue(COUNTER.add(1));`
	`  LocalProcedure(COUNTER);`
	`}`

Исходный код	Объектный код	Уровень стека вызовов
`Main body`	`MyClass.run()`	1
`LOCAL_PROCEDURE(INTEGER)`	`MyClass.LocalProcedure(MyInteger)`	2
`N/A`	`MyInteger.assignIntValue(int)`	3

Рис. 2: Стек вызовов исходной и объектной программ

2.1 Стек вызовов

Как можно заметить из рисунка 2, не всякой рамке на стеке вызовов объектной программы соответствует рамка стека вызовов исходной программы. Так, рамке № 3, соответствующей функции библиотеки поддержки времени исполнения, не соответствует никакой код в исходной программе, т.е. рамку метода `MyInteger.assignIntValue(int)` не следует включать в стек вызовов исходной программы.

Вообще говоря, результатом трансляции процедуры исходной программы может быть 0 и более процедур объектной программы. В этом случае при вызове такой процедуры для нее образуется 0 или более одной рамки в терминах объектной программы, тогда как в терминах исходной программы рамка одна. Крайние случаи (0 и более одной) могут возникнуть в основном в случае оптимизации программы транслятором (например, выполнена подстановка тела процедуры вместо вызова и реально процедура не образует выделенной статики на стеке вызовов или же некоторые вычисления переведены транслятором в параллельные). Методы отладки оптимизированных приложений обсуждаются в [4, 6, 3, 1]. В этой статье предполагается, что отлаживаемый код не был оптимизирован и процедуре исходного языка соответствует не более одной процедуры объектного.

Теперь можно сформулировать метод выделения рамок про-

цедур исходного языка из стека вызовов объектного, таким образом, чтобы он подходил под нужды пользователя и отладчика: *рамка процедуры* исходного языка имеет прообразом рамку процедуры объектного языка, если код, соответствующий последней, имеет прообразом какой-либо код исходной программы.

На рисунке 2 рамки под номерами 1 и 2 являются рамками процедур исходного языка, а рамка 3 не является, так как процедуры поддержки времени исполнения (в данном случае — `MyInteger.assignIntValue(int)`) не имеют прообраза в исходной программе (т.е. не были получены в результате трансляции исходной программы).

Практические следствия данного определения рамки процедуры исходного языка таковы, что при составлении стека вызовов в терминах исходного языка, отладчику для каждой статики процедуры объектного кода придется определять, существует ли код (процедура) исходной программы, которому соответствует эта статика. Для упрощения этой проверки можно отсекать заведомо не интересующие отладчик случаи, проверяя, находится ли рассматриваемая процедура в модуле, сгенерированном транслятором, или же это модуль библиотеки поддержки времени исполнения.

2.2 Поток

Как и в случае со статиками процедур, пользователя отладчика интересуют не все потоки, существующие при исполнении объектного кода. Соответственно, встает тот же вопрос: каким образом выделить эти "нужные" потоки? Ответ прост — выбирать только те потоки, о которых можно что-то "сказать" в терминах исходного языка. Итак, поток (набор потоков) в терминах объектного языка соответствует *потоку исходного языка*, если стек вызовов этого потока содержит хотя бы одну рамку процедуры, являющейся результатом трансляции (или образом) процедуры исходного языка.

Из этого определения вытекает несколько следствий:

- Потоки исходного языка существуют не дольше (а, как правило, существенно меньше) потоков объектного языка. Поток исходного языка может завершить исполнение, в то время как соответствующий ему поток объектного языка будет

продолжать исполнение и, возможно, снова станет прообразом нового потока исходного языка.

- Для того чтобы определить, является ли какой-либо поток объектного языка потоком исходного, придется проверить весь его стек вызовов, что требует относительно больших затрат. Следовательно, отладчик нужно оптимизировать таким образом, чтобы выполнять эту операцию как можно реже. Кроме того, при возможности внесения изменений в библиотеку поддержки времени исполнения можно организовать, например, специальное именование потоков таким образом, чтобы отсекать потоки, которые являются системными и не могут быть прообразом потока исходной программы.

В общем случае потоку исходного языка может соответствовать не менее одного потока объектного языка. Например, в случае отладки распределенных приложений [8] вызов процедуры на удаленной ЭВМ образует, разумеется, другой поток на этой ЭВМ. Для пользователя же, отлаживающего такое приложение, он является простым вызовом, и ожидается, что он будет выполняться в том же потоке. В этом случае логическому потоку (потоку исходного языка) соответствуют 2 физических потока.

3 Реализация распространенных функций

Целью статьи не является подробное описание требуемой от отладчика функциональности, оно может быть найдено, например, в [5]. Здесь будут рассматриваться команды и функции отладчиков, ставшие стандартом де-факто для современных отладчиков, и предполагается, что читатель знаком с ними.

Итак, среди основных функций современного отладчика можно выделить следующие:

- установка точки останова (breakpoint);
- команда пошаговой отладки с остановкой на следующей исполняемой строке исходного кода, вне зависимости от потока и уровня стека вызовов (step into)[2];

[2]Здесь следует упомянуть, что возможна и другая интерпретация этой коман-

- команда пошаговой отладки с остановкой на следующей исполняемой строке исходного кода на том же (или объемлющем) уровне стека вызовов (step over);
- команда пошаговой отладки с остановкой на следующей исполняемой строке исходного кода в вызвавшей процедуре (step out);
- команда временной приостановки потока управления (pause), которая означает "остановить исполнение в момент достижения следующей исполняемой строки исходного кода хотя бы одним из потоков";
- уведомление о возникновении исключительной ситуации (exception) в отлаживаемой программе.

3.1 Установка точки останова

Эта функция отладчика является базовой потому, что во-первых она обязательно должна быть предоставлена пользователям, а во-вторых, она будет активно использоваться в реализации других команд (см. ниже). В связи с этим, имеет смысл ввести два различных типа точек останова — пользовательские (т.е. установленные пользователем) и системные (используемые отладчиком для своих собственных нужд).

Что же касается реализации самой функции установки точки останова в исходном коде, то ее реализация достаточно проста. Эта функция сводится к аналогичной функции отладчика объектного кода; достаточно пересчитать адрес, на который она установлена с адреса в исходном коде на адрес в объектном, используя отладочную информацию:

```
setBreakpoint(SourceLanguageModule module, int sourceLanguageLine){
 int objectLanguageLine =
   module.getDebugInfo().getObjectLanguageLine(sourceLanguageLine);

 requestObjectLanguageBreakpointEvent(objectLanguageLine);
}
```

ды, учитывающая поток, для которого команда должна быть выполнена. В этом случае шаг может завершиться при любом уровне стека, но следующая исполняемая строка должна быть достигнута именно тем потоком, для которого шаг был заказан. Если же пользователя отладчика необходимо "оградить" от упоминания потоков, то следует применять первую интерпретацию этой команды.

3.2 Step into

Если интерфейс отладчика объектного языка предоставляет возможность пошагового исполнения программы, то можно предложить следующие варианты реализации функции step into.

- Реализация с использованием step into объектного языка. Подробнее: при завершении каждого шага осуществлять проверку текущего положения потока управления и в случае остановки на исполняемой строке исходного кода генерировать событие окончания шага. В противном случае продолжить исполнение, выполнив ту же операцию. И так до тех пор, пока приложение не завершится либо исполнение не остановится на следующей строке исходного кода.

  Изложенный алгоритм может быть кратко представлен в следующей форме:

```
stepInto(){
 requestObjectLanguageStepIntoEvent();
 resumeExecution();
}


/* вызывается отладчиком нижнего уровня при возникновении
 *  события StepEvent
 */
objectLanguageStepIntoHandler(StepEvent e){
 int sourceLine = DebugInfo.getSourceLine(e.getObjectLine());
 if( sourceLine == NOT_FOUND ){
  requestObjectLanguageStepIntoEvent();
  resumeExecution();
 }
 else{
  fireSourceLanguageStepIntoEvent(sourceLine);
 }
}
```

  Реализация, как видно, весьма неэффективная. При такой схеме отладчик приостанавливает исполнение приложения на каждой строке объектного кода, что в несколько раз замедляет его выполнение из-за большого количества вызовов функций библиотеки поддержки времени исполнения.

- Эмулирование step into при помощи точек останова. Для реализации этого подхода отладчику необходимо расставить точки останова на исполняемых (т.е. имеющих образ в объектной программе) строках исходной программы, затем продолжить исполнение приложения и по достижении следующей точки останова генерировать событие, уведомляющее об окончании шага. Этот алгоритм можно записать следующим образом:

```
stepInto(){
 for(module in allModues){
  module.setAllBreakpoints();
 }
 resumeExecution();
}

/* вызывается отладчиком нижнего уровня
 * при возникновении события BreakpointEvent
 */
objectLanguageBreakpointHandler(BreakpointEvent e){
 int sourceLine = DebugInfo.getSourceLine(e.getObjectLine());
 fireSourceLanguageStepIntoEvent(sourceLine);
}
```

Потери, связанные с расстановкой всех точек останова, не столь существенны, так как основным преимуществом является исполнение отлаживаемого приложения на полной скорости, т.е. без остановок, невидимых пользователю отладчика. Здесь удобно применить возможность быстрого включения и отключения точек останова, если она присутствует у отладчика объектного языка. Точки останова могут быть убраны (или отключены) в разные моменты времени — после завершения шага или перед другой операцией, которой точки останова будут мешать, например step over. Последнее является более предпочтительным, так как обычно команды пошагового исполнения используются несколько раз подряд.

3.3 Pause

При реализации этой функции отладчика удобно применить тот же подход, что и для step into, т.е. приостановить исполнение

приложения, установить точки останова на все исполняемые строки исходного кода и продолжить исполнение, ожидая события, сигнализирующего о достижении каким-либо из потоков точки останова.

```
pause(){
 suspendExecution();
 for(module in allModues){
  module.setAllBreakpoints();
 }
 resumeExecution();
}

/* вызывается отладчиком нижнего уровня
 * при возникновении события BreakpointEvent
 */
objectLanguageBreakpointHandler(BreakpointEvent e){
 int sourceLine = DebugInfo.getSourceLine(e.getObjectLine());
 fireSourceLanguagePauseEvent(sourceLine);
}
```

Таким образом, команда pause не означает немедленную остановку выполнения, а представляет собой лишь "заказ" на остановку на первой "подходящей" строке.

3.4 Step over

Так как для выполнения этой команды необходима информация о текущем уровне стека вызовов, который принадлежит некоторому потоку, для ее выполнения поток должен обязательно указываться. Для упрощения работы с отладчиком имеет смысл по умолчанию считать, что step over заказывается для "текущего" потока, т.е. потока, в котором произошло последнее событие, вызвавшее остановку всего приложения.

Как и в реализации step into, здесь возможно несколько подходов.

1. Если используемый отладчик объектного языка предоставляет функцию step over, целесообразно использовать ее и повторять до тех пор, пока приложение не завершится или очередной шаг не закончится на строке объектного кода, имеющей прообразом строку исходного кода:

```
stepOver(){
 requestObjectLanguageStepOverEvent();
 resumeExecution();
}

/* вызывается отладчиком нижнего уровня
 * при возникновении события StepEvent
 */
objectLanguageStepOverHandler(StepEvent e){
 int sourceLine = DebugInfo.getSourceLine(e.getObjectLine());
 if( sourceLine == NOT_FOUND ){
  requestObjectLanguageStepOverEvent();
  resumeExecution();
 }
 else fireSourceLanguageStepIntoEvent(sourceLine);
}
```

2. Если же отладчик объектного языка не предоставляет такой функции, можно поступить следующим образом: расставить точки останова на все исполняемые строки исходного кода (как это проводилось в step into и pause) и игнорировать все нежелательные остановки (т.е. те, что происходят в процедурах, вызванных данной или в других потоках).

Очевидно, последний подход связан с большими потерями в скорости работы отлаживаемого приложения (особенно при большой глубине вызовов), хотя и допускает некоторую оптимизацию при условии наличия у отладчика объектного языка таких функций, как контроль над вызовами и возвратами из процедур. В том случае, когда одной строке исходного кода соответствует порядка 2-3 строк объектного кода, “лишние” остановки (по первой схеме реализации) не играют решающей роли в скорости работы отлаживаемого приложения. При реализации же второй схемы “лишние” остановки на всех строках вызываемых процедур могут существенно замедлить работу, так как к этому времени прибавляется еще время работы собственно кода отлаживаемого приложения и реальная скорость выполнения одного шага будет существенно отличаться от ожидаемой.

3.5 Step out

Эта команда требует обязательного указания потока, так как термин “вызвавшая процедура” определен только в рамках од-

ного потока. Поэтому в ее реализации необходимо учесть, что поток может завершить свою работу раньше, чем достигнет точки останова (в этом случае отладчик должен автоматически отменить заказ step out). Кроме того, обработчик событий о достижении точки останова должен уведомить об окончании шага только в случае остановки в нужном потоке.

Общая идея реализации этой функции такова: при выходе из той процедуры, в которой была получена команда step out (т.е. при уменьшении стека вызовов интересующего потока)[3], нужно (как предлагалось для step into и pause) включить все точки останова и продолжить исполнение до тех пор, пока какая-нибудь из них не сработает или поток не завершит свою работу:

```
stepOut(SourceLanguageThread thread){
 storeCurrentCallStackSize();
 requestObjectLanguageStepOutEvent(
   SourceLanguageThread.getObjectLanguageThread());
 resumeExecution();
}

/* вызывается отладчиком нижнего уровня
 * при возникновении события StepEvent
 */
objectLanguageStepOutHandler(StepEvent e){
 SourceLanguageThread thread =
  constructSourceLanguageThread(e.objectLanguageThread());
  if( thread.getFramesCount() < getStoredCallStackSize() ){
   for(module in allModues){
    module.setAllBreakpoints();
   }
   resumeExecution();
  }
  else{
   requestObjectLanguageStepOutEvent(e.objectLanguageThread());
   resumeExecution();
  }
}
```

[3]Здесь предполагается, что отладчик объектного кода может отслеживать изменение (в данном случае — уменьшение) стека вызовов. В противном случае команда step out может быть реализована так же с помощью точек останова, но это будет весьма неэффективно с точки зрения производительности. Таким образом, при отсутствии упомянутой возможности отладчика объектного языка лучше отказаться от реализации step out в отладчике исходного.

```
/* вызывается отладчиком нижнего уровня
 * при возникновении события BreakpointEvent
 */

objectLanguageBreakpointrHandler(StepEvent e){
 int sourceLine = DebugInfo.getSourLine(e.getObjectLine());
 fireSourceLanguageStepIntoEvent(sourceLine);
}
```

3.6 Обработка исключительных ситуаций

Под этим подразумевается уведомление пользователя отладчика о возникновении в отлаживаемой программе исключительной ситуации, такой, как деление на ноль.

Как подчеркивалось в требованиях к отладчику объектного языка, отладчик исходного языка оперирует с отладчиком объектного только в терминах объектного языка. Следовательно, исключительные ситуации, которые не могут быть обработаны отладчиком объектного языка, не могут быть обработаны и отладчиком исходного. Если же какие-либо исключительные ситуации обрабатываются библиотекой поддержки времени исполнения объектного языка, то, вероятнее всего, к их обработчикам можно получить доступ и отслеживать обращения к ним, например, с помощью установки точки останова. Для пользователя отладчика генерация события достижения такой точки будет означать возникновение исключительной ситуации.

Заключение

В данной статье представлен один из возможных подходов к реализации функций отладчика для языков со сложными проекциями типов данных и операторов по двухуровневой схеме, т.е. на основе существующего отладочного интерфейса объектного языка. Для таких языков (а также в ряде других описанных здесь случаев) применение “стандартного” подхода не эффективно или же вовсе невозможно в основном из-за наличия большого количества вызовов функций библиотеки поддержки времени исполнения в проекциях операторов иходного языка. Основным отличием этого подхода является использование “системных” точек останова на всех исполняемых строках нужных модулей ис-

ходной программы и реализация функций пошаговой отладки с их помощью. Схема не накладывает практических ограничений на используемый отладчик объектного языка и сам объектный код. Кроме того, она может быть расширена на n уровней, так как двухуровневость не является необходимым условием.

Описанный подход был вполне успешно применен при создании отладчика для языка Rules Language [9], транслируемого в Java. В качестве отладчика объектного языка использовался отладочный интерфейс Java-машины JPDA [7].

Список литературы

[1] Brooks G., Hansen G.J., Simmons S. A New Approach to Debugging Optimized Code // ACM SIGPLAN'92 Conference on Programming Language Design and Implementation. — 1992. — P. 1-11.

[2] Cagney A. GDB Internals — http://sourceware.cygnus.com/gdb/onlinedocs/gdbint_toc.html. — 2000.

[3] Coutant D.S., Meloy S., Ruscetta M. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code // ACM SIGPLAN'88 Conference on Programming Language Design and Implementation. — 1988. — P. 125-134.

[4] LaFrance-Linden D.C.P. Challenges in Designing an HPF Debugger. — http://www.digital.com/info/DTJR04/DTJR04HM.HTM. — 1998.

[5] High Performance Debugging Forum — HPDS Version 1 Standard — http://www.ptools.org/hpdf/draft. — 1998.

[6] Hoelzle U., Chambers C., Ungar D. Debugging Optimized Code with Dynamic Deoptimization // ACM SIGPLAN'92 Conference on Programming Language Design and Implementation. — 1992. — P. 32-42.

[7] Java (tm) Platform Debugger Architecture 1.0 Overview and Documentation. — http://java.sun.com/products/jpda/doc. — 1997-1999.

[8] Meier S.M., Miller K.L., Pazel D.P. e.a. — Experiences with Building Distributed Debuggers // SPDT '96 SIGMETRICS Symposium on Parallel and Distributed Tools. — 1996. — P. 70-79.

[9] Rules Language Reference Guide. — BluePhoenix Solutions, 2003.

[10] Sang-Joon Nam, Jun-Hee Lee, Kyong-Gu Kang e.a. Fast Development of Source-level Debugging System Using Hardware Emulation // Asia and South-Pacific Design Automation Conference. — 2000. P. 401-404.