УДК 004.4'233
DOI: 10.25559/SITITO.16.202002.389-397

Система отладки программ с различными моделями вычисления

М. В. Аксенов*, **В. А. Сухомлин**
ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова», г. Москва, Россия
119991, Россия, г. Москва, ГСП-1, Ленинские горы, д. 1
*miaks1511@mail.ru

Аннотация

Данная статья посвящена вопросам отладки – поиска ошибок в программе – для современных языков программирования высокого уровня. В настоящее время насчитываются десятки различных языков с различными парадигмами программирования и различными моделями вычисления, и для каждого из них актуален вопрос анализа программы во время ее исполнения. Для большинства языков программирования существуют отладочные инструменты, дающие такую возможность, однако отладка программы, состоящей из частей на нескольких языках, вызывает серьезные трудности.

В статье приводится анализ средств отладки программ для различных языков программирования. Сформулированы минимальные требования к отладчику. Исследованы основные подходы к реализации отладчиков для базовых моделей вычисления – компиляции, интерпретации и динамической компиляции. Выявлены недостатки применения существующих инструментов к отладке программ, совмещающих в себе две различные модели вычисления. Описана система, позволяющая решить выявленную проблему путем объединения возможностей отладчиков каждого из языков. Осуществлена программная реализации описанной системы для отладки программ, комбинирующих языки C# и C++. Протокол взаимодействия отладчиков, лежащий в основе системы, не зависит от реализации ее компонентов и может быть использован для других сочетаний языков.

Ключевые слова: отладка, отладчик, виртуальная машина, модель вычисления.

Для цитирования: Аксенов, М. В. Система отладки программ с различными моделями вычисления / М. В. Аксенов, В. А. Сухомлин. – DOI 10.25559/SITITO.16.202002.389-397 // Современные информационные технологии и ИТ-образование. – 2020. – Т. 16, № 2. – С. 389-397.

Debugging System for Programs with Different Execution Models

M. V. Aksenov*, V. A. Sukhomlin
Lomonosov Moscow State University, Moscow, Russia
1, Leninskie gory, Moscow 119991, Russia
*miaks1511@mail.ru

Abstract

This article is devoted to the issues of debugging programs in modern high-level languages. There are many different languages with different paradigms and different execution models. Each of them needs an instrument to analyze a program during its execution. For most programming languages, there are debugging tools that provide this capability, but debugging a program that consists of several parts in different programming languages is very difficult.
The article contains an analysis of the tools for debugging programs for various programming languages. We provide the minimum requirements for the debugger and describe the main approaches to the implementation of debuggers for basic computation models – compilation, interpretation and dynamic compilation. The drawbacks of using existing tools for debugging programs that combine two different computation models are identified. As a solution, we propose a system that allows to debug such programs by combining the capabilities of the debuggers for each of the languages. The software implementation of the described system for debugging programs combining the C # and C ++ languages has been developed. The debugger interaction protocol underlying the system does not depend on the implementation of its components and can be used for other combinations of languages.

Keywords: debug, debugger, virtual machine, compute model.

For citation: Aksenov M.V., Sukhomlin V.A. Debugging System for Programs with Different Execution Models. *Sovremennye informacionnye tehnologii i IT-obrazovanie* = Modern Information Technologies and IT-Education. 2020; 16(2):389-397. DOI: https://doi.org/10.25559/SITITO.16.202002.389-397

Введение

С середины прошлого столетия можно наблюдать стремительное и непрекращающееся развитие вычислительной техники. Параллельно и с такой же большой скоростью развиваются средства разработки программного обеспечения для этих устройств. В результате написание программ в машинных кодах эволюционировало в огромное разнообразие высокоуровневых языков программирования.

Программа на языке высокого уровня преобразуется в машинный код путем трансляции. Трансляцию осуществляет программное средство, называемое транслятором. Именно транслятор позволяет писать программы на высокоуровневом языке, не задумываясь об особенностях конкретной платформы. В настоящее время существует три основных типа трансляции императивных языков программирования — компиляция, интерпретация и динамическая компиляция[1] [1].У каждой из этих схем есть как сильные, так и слабые стороны. Для объединения преимуществ тех или иных схем создаются как трансляторы, совмещающие в себе несколько из моделей (к примеру, большинство реализаций виртуальной машины Java [2, 3]), так и программы, отдельные части которых написаны на различных языках программирования с отличающимися моделями трансляции и вычисления. В дальнейшем будем называть такие программы смешанными.

Несмотря на удобный инструментарий, предоставляемый языками программирования высокого уровня, при написании на них программ неизбежны ошибки. Поэтому на сегодняшний день цикл разработки программного обеспечения обязательно включает стадию отладки — выявления, локализации и исправления ошибок, сделанных на этапе кодирования [4]. Для этого необходима возможность отслеживать ход исполнения программы и текущие значения переменных, используемых в ней.

В настоящее время существует два основных подхода к отладке ПО. Наиболее интуитивный из них — добавление в исходный код программы вывода информации, интересующей программиста. Далее после запуска необходимо сопоставить полученные промежуточные результаты с исходным кодом, и на основании это сделать вывод о местонахождении ошибки. Такой подход называется журналированием.

Отладка с помощью журналирования имеет несколько недостатков. Основные неудобства связаны с необходимостью менять исходную программу, осуществлять трансляцию и запуск, анализировать результаты (и поскольку заранее неизвестно, какая информация поможет выявить ошибку, повторять эти действия несколько раз). При этом повторная трансляция может занимать существенное время, а полученный отладочный вывод зачастую оказывается очень большим и из-за этого трудным для анализа. Кроме того, подобные изменения в программе в некоторых случаях могут привести к побочным эффектам [5], что усложнит поиск ошибки.

Из-за наличия вышеперечисленных проблем стали разрабатывать инструментальные средства, предназначенные непосредственно для поиска и исправления ошибок в программе путем изучения ее внутреннего состояния во время выполнения. Такие средства получили название отладчиков.

Главной задачей отладчика можно назвать сопоставление внутреннего состояния программы с ее исходным кодом.

Это позволяет программисту анализировать промежуточные состояния программы, чтобы понять, в какой момент была допущена ошибка. Разумеется, реализация отладчика зависит от отлаживаемого языка. Более того, далее будет показано, что наиболее радикальные отличия наблюдаются между отладчиками языков с различными моделями трансляции и вычисления. Таким образом, в описанных ранее смешанных программах достаточно трудно искать ошибки, поскольку отдельные их части требуют различных подходов для отладки, а специализированных отладчиков для таких программ на сегодняшний день нет. В данной статье проводится краткий обзор отладчиков для современных языков программирования высокого уровня, а также описывается программная система, позволяющая осуществлять отладку смешанных программ.

Функционал отладчика

На этапе отладки написанной программы после обнаружения ошибки наиболее трудоемкой частью работы является ее локализация — нахождение одного или нескольких мест в программном коде, приводящих к неправильному результату. Несмотря на то, что активно предпринимаются попытки [6, 7, 8] автоматизировать этот процесс, чаще всего программисту приходится вручную исследовать программу для выявления сбоя. В некоторых случаях этот процесс может занять еще больше времени, чем собственно написание кода, поэтому основной задачей отладчика является предоставление максимально удобного функционала для анализа программы.

В настоящее время этот функционал для большинства отладчиков включает в себя [9, 10, 11]:

- точки останова;
- просмотр переменных;
- вычисление выражений;
- просмотр стека вызовов функций;
- пошаговое исполнение;
- изменение значений переменных.

Требования к реализации отладчика

Для реализации функций, требующихся программисту для отладки, отладчик должен иметь возможность управлять выполнением программы. При этом поиск ошибок с помощью отладчика можно производить только в многозадачной системе, так как программа отладчика должна работать во время исполнения отлаживаемой программы.

Как было сказано во введении, в настоящее время существует три основных модели трансляции и вычисления императивных языков программирования — компиляция, интерпретация и динамическая компиляция. В многозадачной системе при каждой из этих схем исполнение транслируемой программы контролируется другой программой. В случае динамиче-

[1] Alpern, B. Dynamic type checking in jalapeño / B. Alpern, A. Cocchi, D. Grove // Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium. – Vol. 1 (JVM'01). – USENIX Association, USA; 2001. – Pp. 4. – URL: https://dl.acm.org/doi/10.5555/1267847.1267851 (дата обращения: 12.07.2020).

ской компиляции или интерпретации исполнением управляет специализированная виртуальная машина или интерпретатор, тогда как при классической компиляции обязанности контроля за скомпилированной программой возлагаются на операционную систему компьютера.

Поскольку отладчик — отдельная программа, по своей сути ничем не отличающаяся от отлаживаемой, для получения необходимых привилегий он должен уметь взаимодействовать с исполнителем программы, которую должен отлаживать [12]. В свою очередь, исполнитель для обеспечения возможности отладки программ должен предоставлять специализированный интерфейс, через который можно получать информацию о состоянии выполняемой программы, а также влиять на ход ее исполнения. Таким образом, реализация отладчика существенным образом зависит от типа исполнителя и способа взаимодействия с ним [13].

Отладка компилируемых программ

Для компилируемых языков исходный текст программы преобразуется компилятором в исполняемый файл, содержащий машинный код для целевой платформы. При запуске этого файла на исполнение операционная система выделяет программе требуемые ресурсы и время на исполнение, создавая тем самым процесс исполнения [14]. На нее также возлагается обязанность предоставления отладчику интерфейса для контроля над этим процессом [15]. Однако операционная система не имеет представления об исходном коде процесса, он представляется ей лишь как совокупность инструкций и данных. Компиляция программы на языке высокого уровня в машинный код — процесс в общем случае необратимый [16]. Более того, установление соответствия между исходным кодом и исполняемым файлом для просмотра в памяти значений переменных — задача чрезвычайно трудная. Поэтому помимо возможности запросов к операционной системе отладчику нужна дополнительная информация от компилятора [17]. Соответствие нужно установить между строками кода и машинными инструкциями, а также между переменными и ячейками памяти. Информация, как правило, сохраняется в бинарном файле вместе с машинным кодом в специальном формате [18].

Таким образом, характерным требованием для возможности отладки компилируемых программ является предоставление компилятором отладочной информации. Операционная система при этом должна давать возможность читать и модифицировать память отлаживаемого процесса, а также управлять его исполнением.

Отладка интерпретируемых программ

В случае интерпретации исходного кода программа на языке высокого уровня непосредственно исполняется специальной программой, называемой интерпретатором [19]. Таким образом, достигаются простота в распространении программ, отсутствие дополнительного шага в виде компиляции перед запуском, что упрощает разработку. К недостаткам этой схеме относится существенное снижение производительности из-за накладных расходов на исполнение интерпретатора во время исполнения программы [20]. Но задача предоставления интерфейса отладки при этом не является сложной, поскольку для исполнения интерпретатор хранит внутреннее представление исходной программы. Генерация отладочной информации также не требуется, так как интерпретатор исполняет файлы исходного кода, написанные программистом, и при этом непосредственно оперирует переменными, функциями и строками кода.

Отладка динамически компилируемых программ

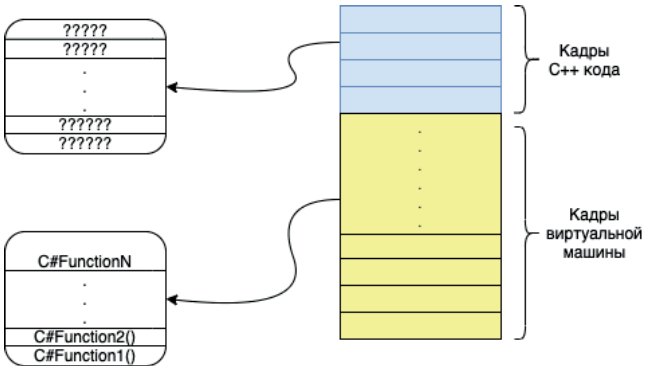
Эта модель вычисления выступает компромиссом между быстродействием компиляции и гибкостью интерпретации. Суть ее состоит в следующем. Сначала исходный код программы компилируется в промежуточный платформонезависимый язык (его часто называют байткодом, так как его инструкции кодируются одним байтом [21]). Это позволяет снизить нагрузку на основной компилятор, работающий во время исполнения программы. Затем полученный код загружается в виртуальную машину, задача которой — компилировать промежуточный язык в машинный код и исполнять [22]. Преимущество перед классической компиляцией в том, что компилируются лишь те фрагменты кода, которые должны быть исполнены (как правило, компилируются функции перед вызовом). При этой схеме трансляции в обязанности виртуальной машины входит, как правило, отслеживание хода выполнения программы, обработка вызовов функции из внешних библиотек, а также освобождение памяти, которую программа больше не использует (так называемая сборка мусора) [23, 24]. Поскольку трансляция программы в этом случае комбинирует в себе две уже рассмотренные схемы, в реализации отладчика несложно увидеть знакомые подходы. Для построения соответствия между исходным кодом и выполняемым процессом нужна отладочная информация, так как при компиляции в байткод теряется информация об исходных номерах строк и именах переменных. В то же время виртуальной машине необходимо хранить внутреннее представление вызываемых функций, чтобы при необходимости компилировать их и запускать на исполнение. Поэтому при отладке ответственность за предоставление стека вызовов возлагается на нее.

Описание выполнения смешанной программы

Возможность вызова скомпилированного кода из виртуальной машины может существенно упростить написание программы, если требуемый функционал уже реализован на компилируемом языке и доступен в виде программной библиотеки. Также довольно популярным является архитектурное решение, в котором обработка данных приложения производится в компилируемой его части для достижения производительности, а пользовательский интерфейс приложения написан на языке, управляемом виртуальной машиной, для получения возможности быстрой пересборки приложения для смены дизайна и управляющих элементов. На уровне синтаксиса вызывающего языка возможность взаимодействия двух языков представлена как вызов внешней функции. Для этого вызова код должен быть загружен в виртуальную машину, поэтому перед вызовом всегда дается указание динамически подключаемой библиотеки. Далее по данному имени находится нужная функ-

ция в библиотеке, и подготавливается ее вызов. Приготовления регламентированы соглашениями о вызовах[2] того языка и архитектуры, для которых скомпилирована функция.

Наибольший интерес здесь представляют подготовка и передача параметров для используемой функции, а также возврат результата. Поскольку эта функция написана на вызываемом языке, она ожидает параметры типов именно из этого языка, между тем как в управляемом виртуальной машиной языке соответствующие типы могут иметь отличное от требуемых представление. Поэтому перед вызовом осуществляется преобразование — так называемый маршалинг — параметров из одного представления в другой [25]. Так как представления соответствующих типов в языках могут быть очень схожими (к примеру, строки в большинстве языков представляются наборов символов, расположенных в памяти подряд), при маршалинге не всегда происходит создание параметров вызываемого языка, соответствующих вызывающему, память может быть переиспользована. Поэтому на время вызова скомпилированного кода виртуальная машина должна перейти в специальный режим работы, чтобы параметры не были уничтожены сборщиком мусора. Для возврата значения из скомпилированного кода выполняется обратное преобразование этого значения в тип, известный виртуальной машине.

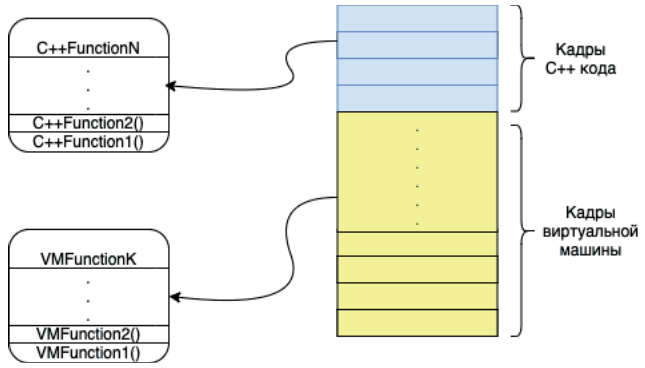


Р и с. 1. Стек вызовов смешанной программы на языках C# и C++, доступный с помощью C# отладчика

F i g. 1. Mixed program call stack in C# and C++, accessible via the C# debugger

Трудности отладки смешанной программы

Ранее были описаны подходы к отладке программ с различными моделями вычисления. С помощью этого описания нетрудно понять, что существующие отладчики не дают возможности отследить выполнение смешанной программы на всех ее этапах. В самом деле, сейчас можно использовать отладчик, предназначенный лишь для одной части программы. Если мы берем отладчик управляемого кода, то можем контролировать исключительно ту часть программы, которую исполняет виртуальная машина, поскольку отладчик целиком зависит от ее интерфейса (рис. 1). В нашей программе задача виртуальной машины — лишь подготовить параметры для вызова кода и дождаться результата. И сам отладчик не имеет доступа к интерфейсу операционной системы, который мог бы помочь в отладке, поскольку предназначен для отладки кроссплатформенных приложений. Таким образом, с помощью этого отладчика мы не сможем отследить, какие параметры реально попали в скомпилированный код, и не сможем узнать причину, по которой полученный результат вызванного кода отличается от ожидаемого.



Р и с. 2. Стек вызовов смешанной программы на языках C# и C++, доступный с помощью C++ отладчика

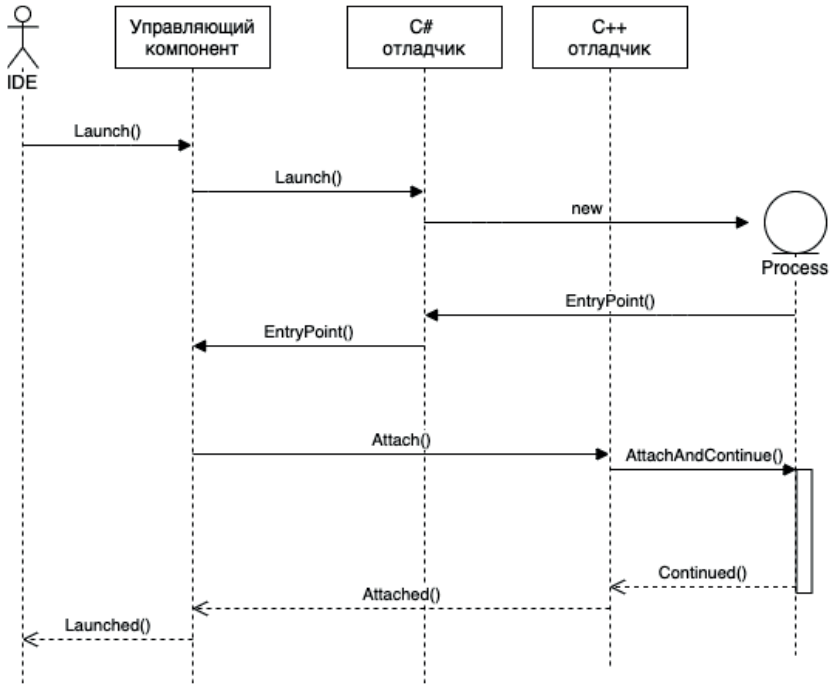
F i g. 2. Mixed program call stack in C# and C++, accessible via the C++ debugger

При применении отладчика скомпилированного кода получаем иную картину. Этот отладчик, имея доступ к интерфейсу операционной системы, дает нам контроль над всей виртуальной машиной, поскольку она является самостоятельным процессом. Но тем не менее, это не дает возможности отлаживать управляемый ей код, поскольку его внутреннее представление в процессе может оказаться очень трудным для понимания программисту, который не знаком с устройством машины, а лишь пользуется ей (а при условии достаточной популярности языка таковых большинство). То есть в большей части осуществляется отладка кода виртуальной машины, а не кода программиста, предназначенного для решения поставленной задачи. Тем не менее, можно отловить участок выполнения процесса, в рамках которого выполняется вызываемый скомпилированный код, и отладить его (рис. 2).

Описание разработанной системы для отладки смешанных программ

Для решения описанных проблем предлагается комбинированная система отладки смешанных программ на компилируемом (C++) и динамически компилируемом (C#) языках программирования. Система реализует описанный выше интерфейс отладчика и позволяет исследовать выполнение смешанной программы, дистанцировавшись от различий в моделях выполнения ее отдельных частей. Основополагающая идея — совместить в одном процессе отладчики соответствующих частей программы, синхронизировать их работу и разделить между ними ответственность за предоставление единого интерфейса для пользователя. Таким образом, в системе можно выделить три главных компонента: отладчик C#, отладчик C++ и управляющий компонент.

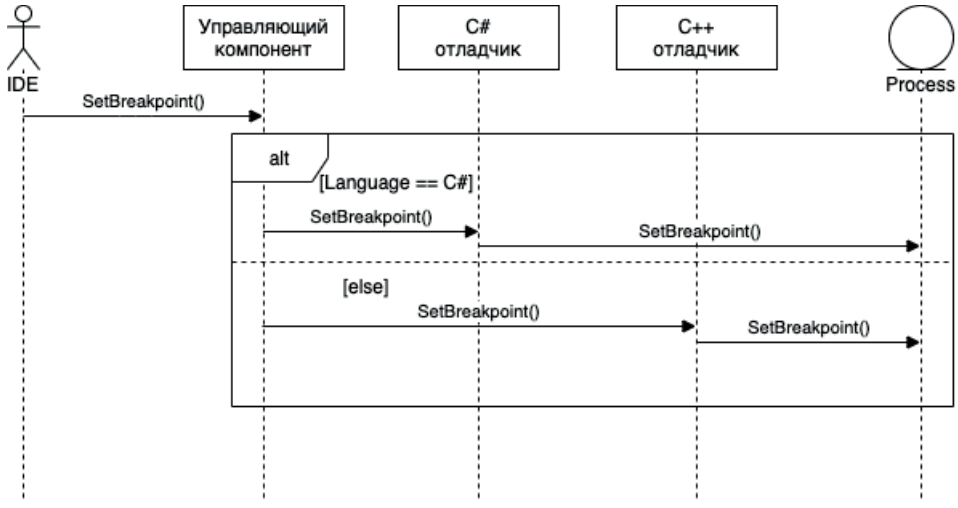
[2] Fog, A. Calling conventions for different C++ compilers and operating systems / A. Fog. – Technical University of Denmark, 2019. – URL: https://www.agner.org/optimize/calling_conventions.pdf (дата обращения: 12.07.2020).



Р и с. 3. Инициализация системы отладки
F i g. 3. Debug system initialization

Последовательно рассмотрим процесс отладки с помощью этой системы и предоставление интерфейса. Для запуска смешанной программы на отладку используется следующая схема (рис. 3). Управляющий компонент получает от пользователя (в качестве пользователя чаще всего выступает среда разработки программ, или IDE) запрос на запуск определенной программы. После этого он дает команду отладчику языка C# запустить виртуальную машину с последующей остановкой перед выполнением кода. После остановки к процессу подключается отладчик C++ и возобновляет выполнение процесса. После этого управляющий компонент уведомляет IDE об успешном запуске программы.

Для выставления точек останова (рис. 4) управляющий компонент, получив соответствующий запрос, определяет, к какой части программы относится запрашиваемое место остановки. Поскольку исходный код программы четко подразделяется на части из компилируемого и управляемого языка, такое определение тривиально. Далее команда на установку передается соответствующему отладчику. При достижении выполнением программы указанного места этот отладчик получит соответствующий сигнал и передаст его управляющему компоненту, а тот в свою очередь уведомит об этом пользователя.

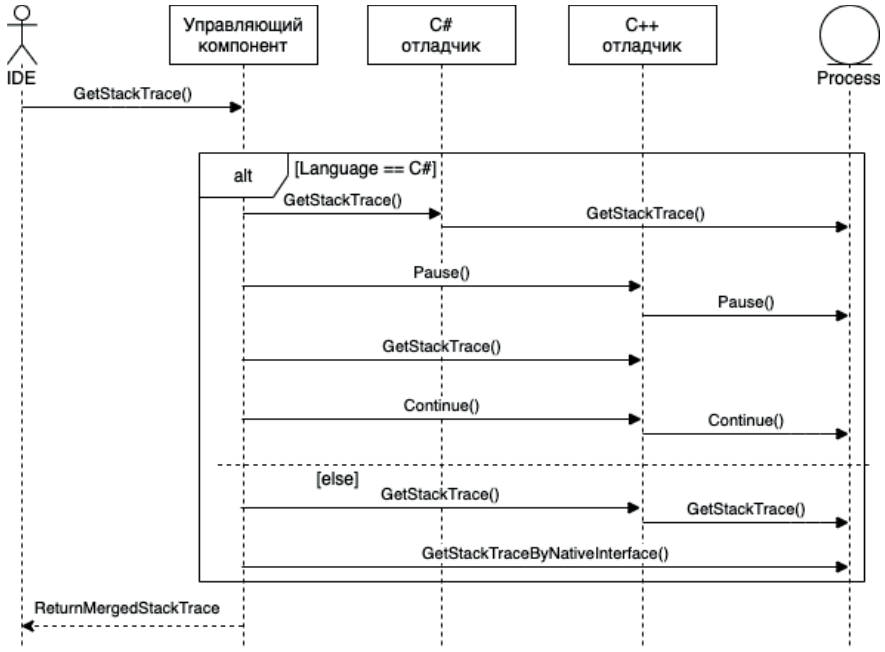


Р и с. 4. Схема добавления точек останова
F i g. 4. Scheme for adding breakpoints

Реализация получения стека вызовов представлена на рис. 5. Ранее было описано, что отладчик каждого из языков имеет доступ лишь к определенному участку стека. Поэтому для получения полной картины требуется объединить данные, полученные с обоих отладчиков. Однако здесь важное значение приобретает еще и то место, где был остановлен процесс до запроса стека. Если остановка имела место в C# коде, управляющий компонент запрашивает стек у отладчика C#, но перед запросом к отладчику C++ он должен дать ему команду остановить процесс, поскольку виртуальная машина в этот момент все еще работает, несмотря на то, что не исполняет код.

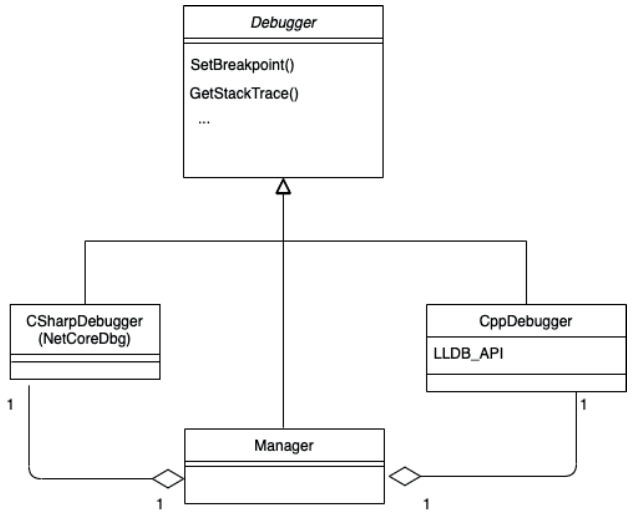
Если же процесс остановился в C++ коде, виртуальная машина также остановлена в этот момент. Это значит, что нельзя выполнять запросы к C# отладчику, поскольку отладочный интерфейс виртуальной машины в данный момент не работает, и отладчик «бессилен». Для преодоления этой проблемы разработчики языка C# предоставили дополнительный интерфейс, который позволяет получить стек вызовов остановленного процесса исключительно по состоянию его виртуальной памяти. Разработанная система использует этот интерфейс в данном случае.

Таким образом, управляющий компонент получает два варианта стека вызовов одного и того же процесса. Ему остается лишь объединить их по имеющимся адресам стековых кадров, заполнив тем самым имеющиеся пробелы в каждом из вариантов. Объединенный стек возвращается пользователю системы.



Р и с. 5. Схема получения стека вызовов

F i g. 5. Call stack retrieval scheme



Р и с. 6. Архитектура полученного решения

F i g. 6. Architecture of the resulting solution

Просмотр и изменение переменных, а также вычисление выражений реализованы в системе сходным образом. В зависимости от того, в какой части программы остановлен процесс, запрос передается соответствующему отладчику. Системой поддерживаются запросы подобного рода только для текущей части программы, взаимодействие с другим отладчиком не требуется. Аналогично реализован и запрос на продолжение процесса после остановки.

Пошаговое исполнение реализовано в системе похожим образом. Однако у такой реализации есть существенный недостаток: она не позволяет выполнять шаги из одной части программы в другую. У виртуальной машины C# не было найдено возможности отловить событие вызова внешнего кода, поскольку синтаксически он идентичен вызову внутренней функции. Таким образом, в настоящее время у системы отсутствует возможность перехода из одной части смешанной программы в другую посредством шага. Однако эта возможность доступна с помощью точки останова в требуемой части, и система позволяет осуществлять шаги в рамках каждой из частей, исключая лишь пограничные случаи.

Реализация

Реализация описанной программной системы была выполнена на языке C++. В качестве составных компонентов были взяты отладчики NetCoreDbg (как класс CSharpDebugger) и LLDB[3] (CppDebugger) [26], описанные в предыдущих главах. Архитектура решения представлена на рис. 6. Все три компонента реализуют интерфейс отладчика и взаимодействуют в рамках одного процесса.

В ходе тестирования системы была проверена работа каждой из составляющих интерфейса отладчика как в C#, так и в C++ коде. Результаты тестирования выявили способность полученной системы к одновременной отладке двух частей программы на различных языках, что позволяет сделать вывод о практической применимости разработанного протокола к задаче отладки смешанных программ.

Заключение

В статье было проведено исследование существующих подходов к отладке программного обеспечения на высокоуровневых языках программирования. Для устранения выявленных недостатков при отладке смешанных программ разработан протокол многоязыковой отладки. Полученный протокол основан на интерфейсе взаимодействующих отладчиков и не привязан к внутренней реализации, что позволяет применять его для различных сочетаний языков.

References

[1] Watson D. A Practical Approach to Compiler Construction. *Undergraduate Topics in Computer Science*. Springer, Cham; 2017. (In Eng.) DOI: https://doi.org/10.1007/978-3-319-52789-5

[2] Dimpsey R., Arora R., Kuiper K. Java server performance: a case study of building efficient, scalable Jvms. *IBM Systems Journal*. 2000; 39(1):151-174. (In Eng.) DOI: https://doi.org/10.1147/sj.391.0151

[3] Kotzmann T., Wimmer C., Mössenböck H., Rodriguez T., Russell K., Cox D. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*. 2008; 5(1):1-32. (In Eng.) DOI: https://doi.org/10.1145/1369396.1370017

[4] Everett G.D., McLeod Jr.R. Software Testing: Testing Across the Entire Software Development Life Cycle. John Wiley & Sons; 2007. (In Eng.)

[5] Veeraraghavan K., Lee D., Wester B., Ouyang J., Chen P.M., Flinn J., Narayanasamy S. DoublePlay: Parallelizing Sequential Logging and Replay. *ACM Transactions on Computer Systems*. 2012; 30(1):1-24. (In Eng.) DOI: https://doi.org/10.1145/2110356.2110359

[6] Elsaka E. Fault Localization Using Hybrid Static/Dynamic Analysis. *Advances in Computers*. 2017; 105:79-114. (In Eng.) DOI: https://doi.org/10.1016/bs.adcom.2016.12.004

[7] Huang T.-Y., Chou P.-C., Tsai C.-H., Chen H.-A. Automated fault localization with statistically suspicious program states. In: *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '07)*. Association for Computing Machinery, New York, NY, USA; 2007. p. 11-20. (In Eng.) DOI: https://doi.org/10.1145/1254766.1254769

[8] Debroy V., Wong W.E. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software.* 2014; 90:45-60. (In Eng.) DOI: https://doi.org/10.1016/j.jss.2013.10.042

[9] Ungar D., Lieberman H., Fry C. Debugging and the experience of immediacy. *Communications of the ACM*. 1997; 40(4):38-43. (In Eng.) DOI: https://doi.org/10.1145/248448.248457

[10] Lawrance J., Bogart C., Burnett M., Bellamy R., Rector K., Fleming S.D. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*. 2010; 39(2):197-215. (In Eng.) DOI: https://doi.org/10.1109/TSE.2010.111

[11] Spinellis D. Modern debugging: the art of finding a needle in a haystack. *Communications of the ACM*. 2018; 61(11): 124-134. (In Eng.) DOI: https://doi.org/10.1145/3186278

[12] Layman L., Diep M., Nagappan M., Singer J., Deline R., Venolia G. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Baltimore, MD; 2013. p. 383-392. (In Eng.) DOI: https://doi.org/10.1109/ESEM.2013.43

[13] Parson D., Murray D.J., Chen Y. Object-oriented design patterns for debugging heterogeneous languages and virtual machines. *Software: Practice and Experience*. 2005; 35(3):255-279. (In Eng.) DOI: https://doi.org/10.1002/spe.634

[14] Tanenbaum A.S., Bos H. Modern Operating Systems. 4th Edition, Pearson; 2015. (In Eng.)

[15] Padala P. Playing with ptrace. Part I. *Linux Journal*. 2002; 2002(103):5. (In Eng.)

[16] Cifuentes C., Gough K.J. Decompilation of binary programs. *Software – Practice & Experience*. 1995; 25(7):811-829. (In Eng.) DOI: https://doi.org/10.1002/spe.4380250706

[3] The LLDB Debugger [Электронный ресурс]. – URL: https://lldb.llvm.org (дата обращения: 12.07.2020).

[17] Li Y., Ding Sh., Zhang Q., Italiano D. Debug information validation for optimized code. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020).* Association for Computing Machinery, New York, NY, USA; 2020. p. 1052-1065. (In Eng.) DOI: https://doi.org/10.1145/3385412.3386020

[18] Eager M.J., Consulting E. Introduction to the DWARF Debugging Format. Group; 2007. (In Eng.)

[19] Pagan F.G. On Interpreter-Oriented Definitions of Programming Languages. *The Computer Journal*. 1976; 19(2):151-155. (In Eng.) DOI: https://doi.org/10.1093/comjnl/19.2.151

[20] Romer T.H., Lee D., Voelker G.M., Wolman A., Wong W.A., Baer J.-L., Bershad B.N., Levy H.M. The structure and performance of interpreters. *ACM SIGPLAN Notices*. 1996; 31(9):150-159. (In Eng.) DOI: https://doi.org/10.1145/248209.237175

[21] Chow F. Intermediate Representation: The increasing significance of intermediate representations in compilers. *Queue*. 2013; 11(10):30-37. (In Eng.) DOI: https://doi.org/10.1145/2542661.2544374

[22] Aycock J. A brief history of just-in-time. *ACM Computing Surveys*. 2003; 35(2):97-113. (In Eng.) DOI: https://doi.org/10.1145/857076.857077

[23] Kulkarni P.A. JIT compilation policy for modern machines. In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11).* Association for Computing Machinery, New York, NY, USA; 2011. p. 773-788. (In Eng.) DOI: https://doi.org/10.1145/2048066.2048126

[24] Rogov S.V., Kirillin V.A., Sidelnikov V.V. Optimization of Java Virtual Machine with Safe-Point Garbage Collection. In: *2006 IEEE International Symposium on Consumer Electronics*. St. Petersburg; 2006. p. 1-5. (In Eng.) DOI: https://doi.org/10.1109/ISCE.2006.1689453

[25] Bishop J., Horspool R.N., Worrall B. Experience in integrating Java with C# and .NET. *Concurrency and Computation: Practice & Experience. Special Issue: 2002 ACM Java Grande–ISCOPE Conference Part I.* 2005; 17(5-6):663-680. (In Eng.) DOI: https://doi.org/10.1002/cpe.858

[26] Soldatov A., Balykov G., Zhukov A., Shapovalova E., Pavlov E. .NET Runtime and Tools for Tizen Operating System. In: Ivanov V., Kruglov A., Masyagin S., Sillitti A., Succi G. (ed.) Open Source Systems. OSS 2020. IFIP *Advances in Information and Communication Technology*. 2020; 582: 190-195. Springer, Cham. (In Eng.) DOI: https://doi.org/10.1007/978-3-030-47240-5_19

Поступила 12.07.2020; принята к публикации 25.08.2020;
опубликована онлайн 30.09.2020.
Submitted 12.07.2020; revised 25.08.2020;
published online 30.09.2020.

Об авторах:

Аксенов Михаил Владимирович, магистр факультета вычислительной математики и кибернетики, ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова» (119991, Россия, г. Москва, ГСП-1, Ленинские горы, д. 1), ORCID: https://orcid.org/0000-0002-2486-4191, miaks1511@mail.ru

Сухомлин Владимир Александрович, заведующий лабораторией открытых информационных технологий, факультет вычислительной математики и кибернетики, ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова» (119991, Россия, г. Москва, ГСП-1, Ленинские горы, д. 1), доктор технических наук, профессор, ORCID: https://orcid.org/0000-0001-9468-7138, sukhomlin@mail.ru

Все авторы прочитали и одобрили окончательный вариант рукописи.

About the authors:

Mikhail V. Aksenov, graduate of the Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1, Leninskie gory, Moscow 119991, Russia), ORCID: https://orcid.org/0000-0002-2486-4191, miaks1511@mail.ru

Vladimir A. Sukhomlin, Head of Open Information Technologies Lab, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1, Leninskie gory, Moscow 119991, Russia), Dr.Sci. (Technology), Professor, ORCID: http://orcid.org/0000-0001-9468-7138, sukhomlin@mail.ru

All authors have read and approved the final manuscript.