Подход к реализации переносимого TTCN-3 отладчика

П.Н. Яковенко, А.В. Сапожников

1. Введение

TTCN-3 [1, 16] является языком программирования, предназначенным для тестирования программных и аппаратных систем по принципу «черного ящика» [4]. Основу TTCN-3 тестов составляют сценарии вида «запрос-ответ». Тестовая программа стимулирует целевую систему, посылая ей запросы. Целевая система реагирует на них, возвращая ответ. На основе оценки реакции целевой системы на стимулы выносится вердикт: тест прошел или нет.

Взаимодействие с тестируемой системой производится через точки доступа (порты), для которых в TTCN-3 определен ряд синхронных и асинхронных операций, таких как посылка сообщения и вызов процедуры. Особенность TTCN-3 состоит в том, что реализация коммуникационных операций, используемых в тестовом сценарии, возлагается на разработчика этого сценария [14, 15]. Для этого он должен реализовать операции коммуникационного интерфейса времени выполнения, определенного в стандарте языка TTCN-3. Другими словами разработчик должен реализовать набор функций с заданным прототипом на C, Java или каком-либо другом языке. Помимо коммуникационного интерфейса в стандарте TTCN-3 определяется ряд других интерфейсов времени выполнения, позволяющих адаптировать платформо-независимые тесты к специфике конкретной целевой системы [2,3].

Традиционно TTCN-3 применяется в области телекоммуникаций для тестирования реализации протоколов в коммутационном оборудовании [12]. Пакеты тестов для некоторых протоколов, в частности, SIP и IPv6 находятся в свободном доступе. В последние годы наблюдается тенденция применения языка TTCN-3 для тестирования встраиваемых систем [8], в частности, электронных блоков управления в автомобилях [7, 11]. Применение TTCN-3 не ограничивается тестированием аппаратных устройств, подключенных к некоторому каналу связи. TTCN-3 также используется для тестирования CORBA-интерфейсов[9] и веб-сервисов [10].

Начиная с редакции 3.1.1, в языке TTCN-3 появился интерфейс регистрации событий **TCI-TL** (TTCN-3 Control Interface – Test Logging) [3]. Далее в тексте для краткости мы будем называть его просто **TL**. При помощи этого интерфейса разработчик может получать извещения о событиях, возникающих в тестовом сценарии в ходе его выполнения, и обрабатывать их. Набор событий покрывает практически все операции языка TTCN-3, включая операции взаимодействия с целевой системой, вызов функций, операцию присваивания и многие другие.

Тестовый сценарий на языке TTCN-3 представляет собой сравнительно сложную программу. Поэтому тестовый сценарий сам по себе должен быть протестирован и отлажен. Заманчивой является возможность реализации отладчика для языка TTCN-3, использующего для реализации операций отладки только стандартные интерфейсы времени выполнения. Такой отладчик не привязан к какой-либо конкретной среде разработки языка TTCN-3. Для него не важно, компилируется или интерпретируется код TTCN-3, в какой целевой язык он компилируется, и на какой платформе исполняется тестовый сценарий.

Мы задаемся вопросом, можно ли, пользуясь только операциями TL-интерфейса, реализовать типовые операции отладки, а именно:

- расстановку контрольных точек и приостановку выполнения в них;
- просмотр стека вызовов функций;
- пошаговую трассировку;
- просмотр значений переменных.

Если операций TL-интерфейса недостаточно, то в чем заключается его ограниченность? Каким образом следует расширить интерфейс для устранения найденных ограничений? В этой статье мы даем ответы на поставленные вопросы.

Материал статьи организован следующим образом. В разделе 2 дается краткое описание интерфейса TCI-TL. В разделе 3 рассматриваются подходы к реализации отладчика: времени выполнения и «посмертный». В разделе 4 описываются особенности реализации операций отладки на базе TL-интерфейса. В разделе 5 рассматривается проблема обработки запущенных таймеров при остановке в контрольной точке. В разделе 6 подводятся итоги работы.

2. Обзор интерфейса TCI-TL

Основное предназначение TL-интерфейса состоит в реализации разнообразных механизмов регистрации событий, представляющих трассу событий тестового сценария в пригодном для чтения текстовом виде, в формате XML [5], в виде диаграмм MSC [6], в графическом виде [13] и т.д. TL-интерфейс состоит из более чем ста операций, обеспечивающих

исчерпывающую информацию о том, что происходит в тестовом сценарии во время его выполнения.

Каждая операция интерфейса связана с определенным событием в тестовой системе. Если классифицировать события в соответствии с их взаимосвязью с операторами языка, то можно выделить несколько классов, среди которых:

- события, связанные с запуском и остановкой тестовых сценариев;
- события, связанные с запуском и остановкой параллельных компонентов в тестовом сценарии;
- события, связанные с конфигурированием портов (точек взаимодействия) в тестовом сценарии;
- события, связанные с передачей данных – обменом сообщениями, вызовом удаленных процедур и обработкой исключений, возникших во время выполнения удаленной процедуры;
- события, связанные с управлением таймерами в тестовом сценарии.

Для каждого события определен фиксированный набор атрибутов, посредством которых предоставляется дополнительная информация о событии. Часть атрибутов является общей для всех событий. Например, для каждого события сообщается позиция в исходном файле, на которой было сформировано событие. Помимо общих атрибутов, событие, как правило, обладает набором специфичных атрибутов.

С точки зрения разработчика операция TL-интерфейса представляет собой функцию обратного вызова. При возникновении события система поддержки выполнения программ вызывает соответствующую функцию, передавая ей значения атрибутов события в качестве фактических параметров. Таким образом, разработчик должен реализовать более ста функций и предоставить их тела на этапе сборки исполняемого модуля. В простейшем случае тела всех функций могут быть пустыми.

Все операции TL интерфейса имеют прототип вида

```
void tli<имя события>(<общие атрибуты>, <специфичные атрибуты>);
```

Например, после выполнения операции присваивания формируется событие, сообщающее о том, что переменная получила новое значение. При этом вызывается функция *tliVar*, имеющая следующий прототип:

```
void tliVar (String         additionalMessage,
             long int       timeStamp,
             String         sourceFile,
             long int       lineNumber,
             TriComponentId componentId,
             String         variableName,
             TciValue       variableValue);
```


Здесь *additionalMessage* – дополнительное текстовое сообщение, *timeStamp* – время возникновения события, *sourceFile* – имя исходного файла, *lineNumber* – позиция в исходном файле, *componentId* – идентификатор компонента, сформировавшего событие, *variableName* – имя переменной, *variableValue* – новое значение переменной.

Помимо события *tliVar*, мы также будем часто использовать события *tliSEnter* и *tliSLeave*. Это события генерируются соответственно при входе в новую область видимости и выходе из нее. Атрибуты событий указывают тип области видимости, ее имя, список значений фактических параметров и т.д. Имя и параметры области видимости определены только для функций, «альтстепов» (специальные функции в TTCN-3) и непосредственно тестовых сценариев, которые с точки зрения синтаксиса языка TTCN-3 являются разновидностью функций. Для составного оператора (нескольких операторов, обрамленных фигурными скобками), также представляющего собой область видимости, значение этих атрибутов не определено.

В большинстве случаев типы формальных параметров TL-функций соответствуют определенным типам данных целевого языка, такого как Java или C. Например, тип String соответствует типу char* языка C. Однако некоторые типы, такие как TciValue, являются абстрактными типами данных, и доступ к ним осуществляется только посредством операций, определенных в стандарте.

3. Подходы к реализации отладчика

Многие современные отладчики, такие как WinDbg [17], gdb [18], dbx [19], являются отладчиками времени выполнения. Такой подход предполагает, что отлаживаемая программа выполняется под контролем отладчика. Процесс отладки происходит параллельно с выполнением программы.

Альтернативным подходом является «посмертная» отладка, при которой отладка производится после завершения программы. Как правило, под «посмертными» понимают отладчики, использующие файл с записанным в нем содержимым памяти процесса после его аварийного останова [20]. Такие отладчики могут лишь исследовать состояние программы на момент ее останова и не могут применяться для трассировки программы. К «посмертным» можно также отнести отладчики, основывающиеся на трассе событий, собранной в ходе выполнения программы [21]. Трасса формируется путем прогона инструментированной версии программы, т.е. программы, в определенные точки которой вставлены обращения к служебным функциям, добавляющим в трассу информацию о состоянии программы в точке вызова. В дальнейшем под «посмертной» отладкой мы будем понимать полноценную отладку программы на базе трассы событий.

Отладчики времени выполнения обладают рядом несомненных преимуществ: в любой момент времени им доступны все объекты программы, они могут

изменять значения переменных и переопределять адрес следующей выполняемой команды. Пользователю не требуется прогонять программу для сбора трассы, притом, что время выполнения программы может быть значительным, а сама трасса крайне объемной. «Посмертные» отладчики в основном используются в тех случаях, когда отладка времени выполнения затруднительна, или вмешательство отладчика может повлиять на поведение программы, приводя к некорректным результатам, например, при отладке параллельных программ и систем реального времени. Заметим, что «посмертная» отладка не ограничена последовательным движением по трассе событий от начала к концу. Отладчик может двигаться по трассе в любую сторону, заглядывать при необходимости вперед; перескакивать с одного фрагмента трассы на другой.

В чем принципиальная важность подхода «посмертной» отладки с точки зрения реализации отладчика на базе TL-интерфейса? Дело в том, что некоторые ограничения TL-интерфейса, о которых будет говориться ниже, не позволяют корректно реализовать все отладочные операции в отладчике времени выполнения. Однако в случае «посмертной» отладки эти ограничения можно преодолеть путем просмотра по трассе вперед, что не представляется возможным при отладке времени выполнения.

4. Типовые операции отладчика

Этот раздел посвящен реализации типовых операций отладчика на базе TL интерфейса. Рассматривается реализация механизма контрольных точек, пошаговой трассировки, просмотра стека вызовов функций и просмотра значения переменной по имени.

4.1. Установка контрольных точек

Приостановка выполнения тестового сценария в контрольной точке может быть реализована путем отслеживания значений двух атрибутов во всех поступающих TL-событиях. Эти атрибуты определяют позицию события через имя исходного TTCN-3 файла и номер строки в файле. Позиция события сравнивается с позициями определенных пользователем контрольных точек. В случае нахождения совпадения выполнение тестового сценария блокируется путем невозврата из TL-функции.

Одной из особенностей TL-интерфейса является отсутствие события, информирующего о смене текущей позиции в исходном файле. Текущая позиция может быть определена только на основании последнего сгенерированного события. Событие всегда генерируется после выполнения соответствующего оператора TTCN-3. Узнать во время выполнения позицию следующего оператора не представляется возможным.

В силу этой особенности семантика контрольных точек, реализуемых посредством TL-интерфейса, отличается от традиционного подхода.

Традиционно остановка в контрольной точке подразумевает, что поток управления остановлен непосредственно перед выполнением оператора, расположенного на данной позиции. В нашем случае остановка в контрольной точке происходит непосредственно после выполнения оператора на заданной позиции. Реализовать привычную семантику контрольных точек можно лишь при «посмертной» отладке, когда позицию следующего оператора можно легко узнать путем просмотра следующего события в трассе.

Каким образом можно устранить указанное ограничение? Предположим, что в TL-интерфейс добавлено событие, сигнализирующее о смене текущей позиции в исходном файле TTCN-3. Назовем это событие *tliSChange*. Правило его генерации события состоит в следующем. Если компилятор программы TTCN-3 генерирует код для некоторой строки файла, то непосредственно перед выполнением сгенерированного кода формируется событие *tliSChange*. Очевидно, что это событие информирует о движении потока управления в ходе выполнения тестового сценария. Обрабатывая это событие, мы можем реализовать механизм контрольных точек с привычной семантикой.

Событие *tliSChange* позволяет устранить еще одно ограничение TL интерфейса. Дело в том, что события TL-интерфейса не покрывают все множество операторов языка TTCN-3. Например, оператор *while* не генерирует событий. События также не генерируются ни для объявления переменной (если только ей не присваивается начальное значение), ни в точке вызова функции. Это значит, что мы не можем установить контрольную точку на заголовке цикла *while* и в ряде других случаев. При пошаговой трассировке мы будем «перепрыгивать» через такие операторы. Событие *tliSChange* устраняет это ограничение.

4.2. Пошаговая трассировка

Реализация пошаговой трассировки во многом схожа с реализацией механизма контрольных точек. Действительно, выполнение каждого шага можно рассматривать как установку одноразовой контрольной точки на определенной позиции. Позиция эта динамическая и не известна в момент начала трассировки, а вычисляется по мере выполнения тестового сценария и обработки поступающих событий. Позиция окончания трассировки определяется ее типом:

- при трассировке «внутрь» (StepIn) остановка происходит на следующем поступившем событии;
- при трассировке «через» (StepOver) остановка происходит на следующем событии, если только это не событие вызова функции; в противном случае остановка происходит сразу после возврата из вызываемой функции;
- при трассировке «наружу» (StepOut) остановка происходит сразу после возврата из текущей функции в вызывающую функцию.

Для реализации трассировок «через» и «наружу» необходимо отслеживать вызов и возврат из функции. В TL-интерфейсе есть операции *tliSEnter* и *tliSLeave*, соответствующие событиям входа в новую область видимости и выхода из нее. Они могут быть использованы для решения этой задачи. При этом мы должны исключить из рассмотрения события *tliSEnter* и *tliSLeave*, соответствующие области видимости составного оператора, если он не представляет собой тело функции.

Очевидно, что при трассировке «наружу» остановка всегда будет происходить на следующем событии, которое поступает после события *tliSLeave*, характеризующего возврат в вызываемую функцию. Рассмотрим теперь нетривиальный случай трассировки «через», т.е. ситуацию, когда следующий выполняемый оператор в тестовом сценарии является вызовом функции. Признаком такой ситуация является то, что первое полученное событие после начала трассировки – это событие *tliSEnter*. В этом случае мы должны учитывать, что вызываемая функция может обращаться к другим функциям; вызов также сформирует события *tliSEnter* и *tliSLeave*. Остановка должна произойти после получения того события *tliSLeave*, которое соответствует возврату из первой вызванной функции. Для его нахождения достаточно поддерживать счетчик пар событий *tliSEnter–tliSLeave*, получаемых в ходе выполнения шага трассировки.

4.3. Просмотр стека вызовов функций

Для реализации просмотра стека вызовов функций мы отслеживаем событие входа в новую область видимости *tliSEnter*. При этом мы игнорируем те события, которые соответствуют входу в область видимости составного оператора. Наименование вызванной функции содержится в одном из атрибутов этого события. Среди других атрибутов есть также значения фактических параметров.

Каждый элемент стека вызовов, помимо названия функции, должен хранить информацию о позиции текущего оператора в этом фрейме. Очевидно, что для всех фреймов, кроме вершины стека, этим оператором является обращение к функции, соответствующей следующему фрейму.

Позиция текущего оператора во внутренних фреймах в общем случае корректно определена быть не может. Дело в том, что в TL-интерфейсе отсутствует событие «вызов функции». Событие *tliSEnter* генерируется, когда поток управления входит в новую область видимости; поэтому позиция, сообщаемая в этом событии, расположена внутри вызываемой функции. Это может быть, например, позиция фигурной скобки, открывающей тело функции.

Предложенное в разделе 4.1 расширение TL-интерфейса в виде события *tliSChange* позволяет скомпенсировать отсутствие события «вызов функции» и устранить ограничения, связанные с показом текущей позиции во внутреннем фрейме.

4.4. Просмотр значений переменных

Система поддержки выполнения программы извещает об изменениях значений переменных посредством события *tliVar*. Для параметров модуля имеется отдельное событие *tliModulePar*. Оба события имеют два специфичных для них атрибута: имя объекта (переменная, константа, параметр модуля) и его новое значение. Значения фактических параметров функции сообщаются в момент ее вызова посредством атрибутов события *tliSEnter*.

Основной проблемой в реализации просмотра значения переменной, запрашиваемой у отладчика по имени, является определение принадлежности переменной, новое значение которой было получено в событии *tliVar*, к той или иной области видимости. Дело в том, что TL-интерфейс не содержит события, информирующего об объявлении переменной; следовательно, эту информацию нужно косвенным образом извлекать из TL-событий. Только обладая этой информацией, мы можем установить момент, когда поток управления покидает область видимости переменной, а это, в свою очередь, необходимо для корректного отображения значения переменной.

Заметим, что отсутствие события объявления переменной также не позволяет нам различить две ситуации, когда переменная не инициализирована и когда она вообще не определена.

4.4.1. Компонентные переменные

Переменная в языке TTCN-3 может быть локальной или объявленной на уровне компонента. Во втором случае время существования переменной совпадает со временем жизни компонента. Если компонент является главным тестовым компонентом, неявно создаваемым при запуске тестового сценария, то время жизни компонентной переменной совпадает со временем выполнения тестового сценария. Область видимости компонентной переменной распространяется на все функции и тестовые сценарии, явно объявленные при помощи оператора “*runs on*” как выполняющиеся на данной компоненте. В отличие от переменных, константы могут быть объявлены и на уровне модуля – самом верхнем уровне иерархий областей видимости в языке TTCN-3. Область видимости таких констант распространяется на все тестовые сценарии и функции, объявленные в данном модуле. Если какой-либо другой модуль импортирует эту константу, то она также видима во всех функциях и тестовых сценариях импортирующего модуля. Параметры модуля могут быть объявлены только на уровне модуля.

В языке TTCN-3 не допускается перекрытие имен. Если на уровне модуля объявлена константа *myConst*, то нигде больше в модуле не должно присутствовать объявление объектов с таким же именем. При этом наличие одинаково именованных объектов допускается в не вложенных областях видимости. Например, в языке TTCN-3 допустимо следующее объявление функции *f*:

```
function f(in integer a) runs on MyComponent
{
  if (a > 0)
  {
    var integer v;
  }
  if (a < 0)
  {
    var integer v;
  }
}
```

Если же в этом примере заменить *v* на *a*, то имя переменной, объявленной в теле условного оператора, будет совпадать с именем формального параметра функции *f*, и компилятор выдаст ошибку.

Несмотря на недопустимость перекрытия имен, отладчик должен уметь отличать объекты уровня модуля от «компонентных» объектов, а эти объекты, в свою очередь, нужно отличать от локальных объектов. Рассмотрим пример, показывающий важность знания области видимости переменной:

```
type component MyComponent
{
}

function f() runs on MyComponent
{
  var integer v;
  v := 2; //здесь генерируется событие tliVar (v == 2)
}

testcase tc() runs on MyComponent
{
  var integer v;
  v := 1; //здесь генерируется событие tliVar (v == 1)
  f();
  log(v); // v == 1  или v == 2?
}
```

В этом примере в тестовом сценарии *tc* объявляется переменная *v*, ей присваивается единица (при этом генерируется событие *tliVar* для переменной с именем “v”) и вызывается функция *f*. В функции *f* также объявляется переменная *v*, и ей присваивается двойка (при этом снова генерируется событие *tliVar* для переменной с именем “v”). После возврата из функции *f* тестовый сценарий *tc* выводит на экран значение *v*.

Оператор *log(v)* распечатает, разумеется, единицу, однако в рамках TL-интерфейса нельзя сказать, что значение *v* в этой точке равно единице. Действительно, при выполнении тестового сценария *tc* сгенерируется ряд событий, из которых нам интересна часть, соответствующая вызову функции *f*:

```
tliVar("v", 1) // v = 1
tliSEnter("f") // вызов функции f
tliVar("v", 2) // v = 2
tliSLeave("f") // возврат из функции f
```

Получив такую последовательность событий, мы можем неверно предположить, что оба события *tliVar* относятся к одной и той же переменной, и *v* – компонентная переменная. Тогда просмотр значения *v* на позиции оператора *log(v)* в отладчике ошибочно покажет двойку, в то время как оператор *log(v)* выведет на экран единицу. Если же мы будем считать, что *v* – локальная переменная, и события относятся к двум разным одноименным переменным (как оно и есть в нашем примере), объявленным в тестовом сценария *tc* и функции *f*, то просмотр значения *v* на позиции оператора *log(v)* в отладчике покажет единицу. Следовательно, в момент получения события *tliVar* для переменной *v* необходимо уметь определять, является ли она компонентной или локальной.

В общем случае не представляется возможным выделить среди всех событий те из них, которые относятся к компонентным переменным, . Тем не менее, если придерживаться простого соглашения по написанию программ TTCN-3, то в большинстве случаях можно корректно определять характер переменной.

Предположим, что переменная *v* объявлена в компоненте *MyComponent*, и для нее указано начальное значение, например, следующим образом:

```
type component MyComponent
{
  var integer v := 1;
}
```

Тогда система поддержки выполнения программы сгенерирует событие *tliVar* в момент инициализации переменной. Инициализация происходит во время создания компонента. Для главного тестового компонента это произойдет в интервале между командой на запуск тестового сценария (в этот момент генерируется событие *tliTcExecute*) и началом выполнения операторов в теле тестового сценария (в этот момент генерируется событие *tliTcStarted*). Таким образом, все события *tliVar*, полученные в период между событиями *tliTcExecute* и *tliTcStarted,* относятся к инициализации компонентных переменных и констант.

Дополнительные параллельные компоненты в языке TTCN-3 должны явно порождаться при помощи оператора *create*. При этом генерируется событие

tliCCreate. Оператор *create* только создает новый экземпляр компонента заданного типа. Для запуска функции во вновь созданном компоненте необходимо выполнить оператор *start*, указав имя функции и значения формальных параметров. При этом генерируется событие *tliCStart*. Такое разделение сделано для того, чтобы иметь возможность должным образом подключить порты нового компонента до начала выполнения в нем функции. Для параллельных компонентов все события *tliVar*, полученные в период между событиями *tliCCreate* и *tliCStart,* относятся к инициализации компонентных переменных и констант.

Таким образом, если в TTCN-3 программе все компонентные переменные инициализируются начальными значениями, то, применяя вышеизложенный метод, в большинстве случаев можно отделять события *tliVar*, относящиеся к компонентным переменным, от всех остальных.

Отметим особенность языка TTCN-3, из-за которой изложенный выше алгоритм в ряде случаев будет работать неверно. В TTCN-3 функция или тестовый сценарий, явно объявленные как выполняющиеся в некотором компоненте, могут вызывать обычную функцию (без объявления "*runs on*"), допускающую вызов из любой точки программы. Рассмотрим следующий пример:

```
type component MyComponent
{
 var integer v := 1;//здесь генерируется событие tliVar (v == 1)
}

function f() // _НЕ_ объявлена на компоненте
{
  var integer v;
  v := 2; //здесь генерируется событие tliVar (v == 2)
}

testcase tc() runs on MyComponent
{
  f();
  log(v); // v == 1, но отладчик ошибочно выдаст v == 2
}
```

Отличие этого примера от предыдущего состоит в том, что объявление переменной *v* перенесено из тестового сценария *tc* в компонент *MyComponent*. При этом из декларации функции *f* удалено объявление "*runs on*". В противном случае область видимости локальной переменной *v* в функции *f* перекрывалась бы с областью видимости компонентной переменной *v* в типе *MyComponent*, что недопустимо в языке TTCN-3.

Функция *f* не связана с каким-либо определенным типом компонента, поэтому переменные, объявленные в компоненте *MyComponent*, не видны в ней, и объявление локальной переменной *v* допустимо. При применении предложенного выше алгоритма просмотр значения переменной *v* на позиции оператора *log(v)* в отладчике выдаст двойку, что не является корректным. Скорректировать алгоритм в соответствии с этой особенностью не представляется возможным, потому что TL-интерфейс не информирует о том, каким образом объявлена вызываемая функция – с "*runs on*" или без этой спецификации.

Можно предложить следующее универсальное решение задачи распознавания характера переменной, простое в реализации для разработчиков TTCN-3 компилятора, но, тем не менее, требующее внесения изменений в стандарт TTCN-3: в событии *tliVar* передавать не просто имя переменной, а дополнять его префиксом для объектов уровня модуля и «компонентных» объектов. Объекты уровня модуля дополняются именем модуля, отделенным точкой, компонентные объекты дополняются именем типа компонента, отделенным двоеточием. Именование локальных объектов оставляется без изменений, т.е. без префикса. Например, *MyModule.object* – объект уровня модуля, *MyComponent:object* – объект уровня компоненты, *object* – локальный объект. Такой подход к именованию объектов согласуется с синтаксисом языка TTCN-3.

В рамках стандартного интерфейса можно предложить пользователю придерживаться определенного стиля именования объектов. Например, можно начинать все имена объектов уровня модуля с символов *g_*, компонентные объекты – с символов *ct_*. Тогда механизм распознавания областей видимости в отладчике может опираться на подобный стиль именования объектов. Тем не менее, такое решение не является стандартным, а, значит, отладчик теряет универсальность.

Что касается констант, объявленных на уровне модуля, то событие *tliVar* для них генерируется до начала выполнения тестового сценария. Поэтому все события *tliVar*, которые происходят до события *tliTcExecute*, информирующего о запуске тестового сценария, относятся к инициализации объектов уровня модуля.

4.4.2. Блочные переменные

Язык TTCN-3 позволяет объявлять локальные переменные в теле составного оператора. Значение таких переменных становится неопределенным, когда поток управления покидает тело составного оператора. В следующем примере значение переменной *v* не определено на позиции оператора *log(v)*.

```
if (...)
{
 var integer v := 1; //здесь генерируется событие tliVar (v == 1)
}
log(v); //ошибка, значение v не определено
```

Запрос у отладчика значения переменной *v* в точке *log(v)* должен выдать, что переменная *v* не определена, однако реализация такой функциональности не представляется возможной, несмотря на то, что при входе и выходе из составного блока генерируются события *tliSEnter* и *tliSLeave* соответственно. Дело в том, что в TL-интерфейсе отсутствует событие, информирующее об объявлении переменной. Это значит, что локальная переменная, для которой мы получили событие *tliVar*, может быть объявлена не только в текущем блоке, но и в любом блоке, который его объемлет, вплоть до блока, представляющего собой тело функции.

Решить эту проблему в рамках стандартного TL-интерфейса не представляется возможным. Разработчик отладчика может следовать одной из двух стратегий: либо предполагать, что каждая переменная инициализируется начальным значением, и первое событие *tliVar* определяет точку определения переменной, либо предполагать, что блочные переменные в TTCN-3 программе отсутствуют. Другими словами, при этом подходе всегда следует предполагать, что локальные переменные объявляются только на уровне тела функции.

Последнее ограничение для целей отладки не является существенным, и оно было принято в нашей реализации. Действительно, в TTCN-3 перекрытие имен не разрешается, одноименные переменные не могут быть объявлены в разных блоках, если один из них вложен в другой. Поэтому единственным неудобством для пользователя является, что значение «блочной» переменной будет видно в отладчике после выхода из блока вплоть до момента возврата из функции.

4.4.3. Формальные параметры функций и тестовых сценариев

Среди множества объектов, значения которых могут быть запрошены пользователем во время сеанса отладки, выделяются фактические параметры функции и тестового сценария. Значения всех фактических параметров передаются посредством атрибутов события *tliSEnter*.

К сожалению, список значений фактических параметров не содержит имен этих параметров, как они определены в TTCN-3 программе, и получить имена параметров каким-либо другим образом нельзя. В качестве одного из вариантов, отладчик может поддерживать псевдонимы для формальных параметров, например, присваивая им имена вида *@<порядковый_номер>*. *@1* для первого параметра, *@2* – для второго и т.д. Разумеется, такое решение является нестандартным.

В качестве универсального решения мы предлагаем расширить TCI-интерфейс, добавив в него две функции: *tciGetTestCaseParametersNames* и *tciGetFunctionParametersNames*, позволяющие запросить список имен формальных параметров тестового сценария и функции по соответствующему имени. Наличие этих функций позволяет снять отмеченное ограничение.

5. Обработка таймеров в отладчике времени выполнения

В языке TTCN-3 имеется встроенная поддержка таймеров. Для таймеров определен набор операций, в число которых входят операция запуска и остановки таймера, проверка того, истекло ли время таймера и т.д. Понятие времени в TTCN-3 не является жестко закрепленным и определяется пользователем путем реализации соответствующих операций в интерфейсе времени выполнения TRI. Как правило, понятие времени в тестовом сценарии соответствует понятию реального времени, но, вообще говоря, время может быть ускоренным или замедленным вариантом реального времени, событийным или каким-либо еще.

В отладчике времени выполнения остановка тестового сценария в контрольной точке, если к этому моменту активен хотя бы один таймер, может привести к некорректному поведению тестового сценария после возобновления выполнения. Таймеры, активные в момент выхода на контрольную точку, необходимо отключать на время простоя в этой контрольной точке. Действительно, рассмотрим пример:

```
timer T;
T.start(3.0); //запустить таймер на 3 секунды
P.send(Request);
alt
{
  []P.receive {setverdict(pass)} // тест прошел
  []T.timeout {setverdict(fail)} // тест _НЕ_ прошел
}
```

В этом примере таймер *T* устанавливается на три секунды, и затем через порт *P* отправляется сообщение *Request*. После этого тестовый сценарий ожидает прихода одного из двух событий: либо в порт *P* приходит какой-то ответ, и тогда вердикт теста устанавливается в *pass* (т.е. «тест прошел»), либо истекает время таймера *T*, и вердикт устанавливается в *fail* (т.е. «тест не прошел»).

Таймер *T* предназначен для проверки того, что обмен данными через порт *P* (запрос-ответ) укладывается в три секунды. Предположим, что посылка и прием сообщения в совокупности занимают две секунды. В этом случае тест пройдет успешно. Если приостановить выполнение тестового сценария на десять секунд после запуска таймера, но перед отправкой сообщения, то в

операторе *alt* сработает вторая ветка, и тестовый сценарий закончится с вердиктом *fail*.

Решение этой проблемы выходит за рамки данной работы. Отметим лишь, что отладчик может поддерживать список активных таймеров и информировать пользователя о тех случаях, когда выполнение тестового сценария приостанавливается при наличии запущенных таймеров.

6. Заключение

В этой статье предложен подход к реализации отладчика для языка TTCN-3 на базе стандартного интерфейса регистрации событий TCI-TL. Показано, что TL-интерфейс позволяет реализовать типовые операции отладки, такие как установка контрольных точек, пошаговая трассировка, просмотр стека вызовов функций и значений объектов (переменных, констант, параметров модуля). Предложенный подход лег в основу реализации отладчика в среде Telelogic Tester.

Ряд ограничений TL-интерфейса не позволяет реализовать операции отладки с той семантикой, которую они имеют в таких широко используемых отладчиках, как WinDbg, gdb, dbx и др. Для устранения наиболее критичных ограничений достаточно внести три изменения в интерфейсы времени выполнения языка TTCN-3:

- добавить в TL интерфейс операцию *tliSChange*, которая информирует об изменении текущей позиции (в исходном TTCN-3 файле) потока управления во время выполнения тестового сценария;
- в операции *tliVar*, информирующей об изменении значения переменной, сообщать имя переменной вместе с префиксом, по которому можно определить характер переменной; для переменных и констант, объявленных на компоненте, таким префиксом является имя типа компоненты, для констант, объявленных на уровне модуля, таким префиксом является имя модуля;
- добавить в TCI интерфейс две операции, позволяющие запросить имена формальных параметров функции (например, *tciGetFunctionParametersNames*) и тестового сценария (например, *tciGetTestCaseParametersNames*) по соответствующему имени.

Отметим, что отладчик, реализованный на базе стандартного TL-интерфейса, не зависит от конкретной среды разработки для языка TTCN-3. Более того, для отладчика не существенно, компилируется или интерпретируется TTCN-3 код, в какой целевой язык он компилируется и на какой целевой платформе исполняется.

В дальнейшем планируется провести исследование и реализовать в отладчике обработку таймеров в контрольных точках и возможность изменения значений переменных.

Литература

[1] ETSI. Methods for Testing and Specification (MTS). The Testing and Test Control Notation version 3; Part 1 TTCN-3 Core Language; ETSI ES 201 873-1, V3.2.1, 2007

[2] ETSI. Methods for Testing and Specification (MTS) .The Testing and Test Control Notation version 3; Part 5 TTCN-3 Runtime Interface (TRI); ETSI ES 201 873-5, V3.2.1, 2007

[3] ETSI. Methods for Testing and Specification (MTS) .The Testing and Test Control Notation version 3; Part 6 TTCN-3 Control Interface (TCI); ETSI ES 201 873-6, V3.2.1, 2007

[4] B. Beizer. Software Testing Techniques, 2nd Edition. New York. Van Nostrand Reinhold. 1990

[5] Extensible Markup Language (XML) 1.0, W3C Recommendation, 1998

[6] Message Sequence Chart (MSC), ITU-T Recommendation Z.120, 1999

[7] S. Burton, A. Baresel, I. Schieferdecker. “Automated testing of automotive telematics systems using TTCN-3”. Proceedings of 3rd Workshop on Systems Testing and Validation. Paris, France. 2004

[8] I. Schieferdecker, J. Grossmann. “Testing of Embedded Control Systems with Continuous Signals”. Proceedings of 2nd Workshop on Modeling of Embedded Systems. Dagstuhl. Germany. 2006

[9] M. Li, A. Rennoch, I. Schieferdecker, D. Witaszek, O. Halabi, A. Vouffou, A. Yin, “Experience Report on Conformance Tests for CORBA ORBs”. Proceedings of Second Asia-Pacific Conference on Quality Software. Hong Kong. 2001

[10] I. Schieferdecker, G. Din, D. Apostolidis. “Distributed functional and load tests for Web services”. International Journal on Software Tools for Technology Transfer. Volume 7. Number 4. 2005

[11] I. Schieferdecker, A. Rennoch, E. Höfig. “TTCN-3 – A Test Technology for the Automotive Domain” Proceedings of Simulation und Test in der Funktions und Softwareentwicklung für die Automobilelektronik. Berlin. Germany. 2005

[12] I. Schieferdecker, A. Rennoch. “Industrial use of TTCN-3 – Scope and Limits”, International Conference on Software Tests. Düsseldorf. Germany. 2005

[13] G. Din, J. Zander, S. Pietsch. “Test exectution logging and visualization techniques”. Proceedings of Software and Systems Engineering and their Applications. Paris. France. 2004

[14] S. Schulz, T. Vassiliou-Gioles. “Implementation of TTCN-3 Test Systems using the TRI”, Proceedings of 14th International Conference on Testing Communicating Systems. Berlin. Germany. 2002

[15] I. Schieferdecker, T. Vassiliou-Gioles. “Realizing distributed TTCN-3 test systems with TCI”.
Proceedings of 15th International Conference on Testing Communicating Systems. Cannes. France. 2003

[16] C. Willcock, T. Deiß, S. Tobies, S. Schulz, S. Keil, F. Engler. “An Introduction to TTCN-3”. Wiley. 2005

[17] J. Robbins. “Debugging Applications for Microsoft® .NET and Microsoft Windows®”. Microsoft Press. 2003

[18] R. M. Stallman, R. H. Pesch, S. Shebs. “Debugging with GDB“. GNU Press. 2003

[19] B. Tuthill, K.J. Dunlap, "Debugging with dbx". UNIX Programmer's Supplementary Documents, vol. 1. Computer Systems Research Group. Dept. of Electrical Eng. and Computer Science. Univ. of California. Berkeley. 1986

[20] M. Hof. “Post Mortem Debugger for Oberon”. Procedings of Joint Modular Languages Conference, Ulm. Germany. 1994
[21] В.А.Крюков, Р.В.Удовиченко. "Отладка DVM- программ". Программирование . 2001. № 3