УДК 681.3

К.А. Жереб

Программный инструментарий, основанный на правилах, для автоматизации разработки приложений на платформе *Microsoft .NET*

Рассмотрен программный инструментарий для проектирования и разработки приложений на платформе *Microsoft .NET*. Описано использование данного инструментария для перехода от последовательных к параллельным многопоточным программам. Приведен пример работы инструментария и его преимущества.

A software toolkit for the design and development of applications on the Microsoft .NET platform is considered. The usage of the developed toolkit for the transition from serial to parallel multithreaded applications is described. An example demonstrating the operation of the toolkit and its benefits is given.

Розглянуто програмний інструментарій для проектування та розробки застосувань на платформі *Microsoft .NET*. Описано використання цього інструментарію для переходу від послідовних до паралельних мультипотокових програм. Наведено приклад роботи інструментарію та його переваги.

Введение. Последние достижения в области разработки многоядерных микропроцессоров открывают новые возможности повышения производительности программ путем распараллеливания вычислений. Для распараллеливания программ на многоядерных платформах используется в основном многопоточное программирование [1]. При этом актуальны задачи разработки новых эффективных многопоточных программ и преобразование существующих однопоточных программ в более производительные многопоточные аналоги.

В настоящее время распараллеливание однопоточных программ и оптимизация многопоточных алгоритмов проводится в основном вручную. Это приводит к значительным затратам времени и усилий разработчиков, многие из которых не имеют опыта работы с параллельными многопоточными приложениями. Кроме того, увеличивается количество ошибок в многопоточном коде, которые достаточно трудно обнаруживать и исправлять. В связи с чем особую актуальность приобретает задача автоматизации разработки параллельных многопоточных приложений.

В работах [2, 3] авторами предложен подход к автоматизации преобразований многопоточных программ, основанный на использовании алгебро-алгоритмического инструментария проектирования и синтеза программ «ИПС» [4] и системы переписывающих правил *Termware* [5–7]. В данной статье рассмотрен разработанный программный инструментарий для автоматизации разработки многопоточных приложений, реализующий предложенный подход. Как и в работах [2, 3], для автоматизации преобразований используется система переписывающих правил *Termware*, которая перенесена на платформу *Microsoft .NET*. Однако в отличие от работ [2, 3], где входные данные для правил поступали исключительно из системы ИПС, разработанный в статье инструментарий содержит анализатор (парсер), позволяющий работать непосредственно с кодом программы на языке *C#*.

Для автоматизации задач, связанных с разработкой приложений, используются различные системы, основанные на правилах, такие как *Jess* [8], *Maude* [9], *Stratego* [10], *ASF+SDF* [11] и другие. Так, в работе [12] описано применение *Maude* для проверки корректности определенных классов преобразований программ (рефакторинга). В работе [13] используется *Jess* для поиска и устранения проблем с производительностью разрабатываемой системы. Разработаны также платформы для создания подоб-

ных приложений, такие как *XT* [14] или *DMS* [15]. Отметим, что большинство работ рассматривают системы на языках *Java* или (в меньшей степени) *C*/*C++*. Системы такого рода, основанные на платформе *Microsoft .NET* (язык *C#*), практически отсутствуют, несмотря на то что эта платформа в последнее время приобретает все большую популярность. Поэтому важной особенностью инструментария, описанного в статье, является его ориентированность на платформу *.NET*. Инструментарий реализован на языке *C#*, использует версию *Termware*, перенесенную на платформу *.NET*, а также включает анализатор и генератор кода для языка *C#*, что позволяет применять его при разработке приложений для *.NET*.

Важная особенность разработанного инструментария – наличие пользовательского интерфейса для создания и редактирования правил. Системы, основанные на правилах, являются мощным средством для осуществления преобразований; однако написание правил в текстовом виде требует знания языка записи правил и определенных навыков работы с ним. Для пользователей, не имеющих такого опыта, использование графического редактора позволяет создавать правила, концентрируясь на требуемом преобразовании, а не на особенностях записи правил и реализации некоторых стандартных действий.

Материал статьи организован следующим образом. Описаны основные компоненты разработанного инструментария: анализатор кода, система правил, генератор кода и интерфейс пользователя. Далее приведен пример использования инструментария для перехода от последовательной к параллельной многопоточной программе. Статью завершают выводы и направления дальнейшей работы.

Компоненты инструментария

Разработанный программный инструментарий состоит из следующих основных компонентов:

- анализатор для перевода программного кода с языков высокого уровня (таких как *C#*, *Java*, *C++*) во внутреннее представление;
- система применения переписывающих правил для преобразования программ, записанных во внутреннем представлении;
- генератор для обратного перевода из внутреннего представления в язык программирования;
- интерфейс пользователя.

Взаимодействие между этими компонентами схематически представлено на рис. 1.

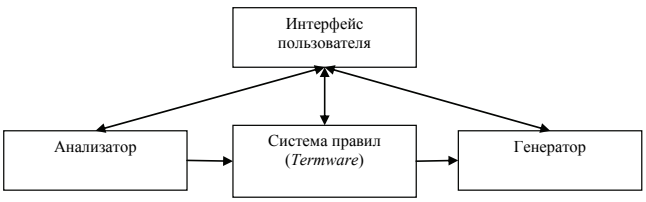

Рис. 1. Основные компоненты системы

Работа системы происходит следующим образом. Вначале пользователь задает исходный код разрабатываемой программы. Анализатор соответствующего языка программирования преобразует исходный код во внутреннее представление программы. Это представление состоит из термов (выражений вида $f(x_1, ..., x_n)$) и эквивалентно абстрактному синтаксическому дереву (*AST*) программы. Внутреннее представление может содержать дополнительные объектные структуры, моделирующие программу (например, таблицы идентификаторов).

Далее к полученному внутреннему представлению применяются переписывающие правила *Termware*. Возможно применение заранее разработанных правил, а также непосредственное создание новых правил в интерфейсе пользователя. При этом пользователь может видеть результаты действия правил в виде новых (преобразованных) термов. К результату действия правил затем применяют генератор, создающий код преобразованной программы.

Возможно также использование системы совместно с алгеброалгоритмическим инструментарием проектирования и синтеза программ (ИПС) [4]. В этом случае вместо исходного кода на вход системе подается представление алгоритма в виде *CAA*-схемы, которое затем преобразовывается в термы такого же вида, как и

в случае использования анализатора. Дальнейшая схема работы остается без изменений: применяются правила, а затем генератор кода.

Общая схема работы пользователя с системой представлена на рис. 2.

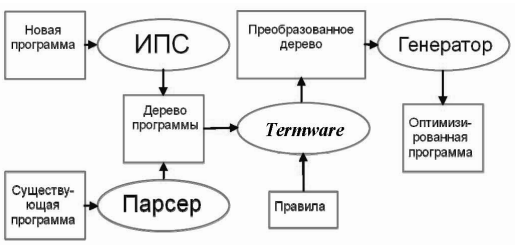

Рис. 2. Последовательность действий при работе с системой

Рассмотрим подробно каждый из компонентов системы.

Анализатор

Существуют два основных подхода к разработке анализаторов:

• Анализаторы создаются вручную, с использованием неформализованных знаний разработчика о спецификации языка. Примером таких анализаторов для языка *C#* могут служить *Metaspec C# parser library* [16] и *C# parser project* [17].

• Анализаторы генерируются автоматически при помощи специальных инструментальных средств, так называемых «компиляторов компиляторов» (*compiler compiler*). При этом разработчик задает формализованное описание языка в виде грамматики. На основании этих данных генерируется код анализатора. К таким инструментальным средствам относятся *yacc*, *ANTLR* [18], *GOLD parser engine* [19] и др.

При разработке системы проанализированы существующие анализаторы языка *C#*, которые могут быть встроены в *.NET*-приложение. Особенность языка *C#* – наличие нескольких версий. Версия 1.0 была первой версией языка, использовалась в *.NET Framework* 1.0 и 1.1. Версия *C#* 2.0 использовалась в *.NET Framework* версий 2.0 и 3.0, она поддерживает все возможности *C#* 1.0 и добавляет такие возможности, как обобщенные классы (*generics*), анонимные методы и др. Последней версией *C#* на момент разработки системы является *C#* 3.0, появившаяся в *.NET Framework* 3.5. Помимо возможностей *C#* 2.0, в ней появилась поддержка лямбда-выражений, *LINQ* и другие возможности.

С учетом разработки анализаторов, появление новых возможностей в языке приводит к усложнению грамматики и повышению затрат на разработку анализатора. Поэтому многие анализаторы не поддерживают последнюю версию языка *C#*. Так, доступные на сайте *ANTLR* [18] грамматики поддерживают только версию 1.0; проект *C# parser* на *Codeplex* [17] поддерживает версию 2.0. Библиотеки *Metaspec C# parser* [16] поддерживают последнюю версию *C#* 3.0; однако этот проект является закрытым и коммерческим, что не позволяет при необходимости модифицировать анализатор.

В связи с этим принято решение не ограничиваться только одним анализатором *C#*, а заложить в архитектуру системы возможность поддержки множества различных анализаторов. Таким образом, появляется возможность поддержки различных версий *C#*. Реальна ситуация, когда какой-то анализатор не поддерживает определенные конструкции языка; в таком случае при работе с программным кодом, использующим эти возможности, можно использовать другой анализатор. Наконец, такая архитектура позволяет легко добавлять анализаторы других языков.

Анализаторы подключаются к системе при помощи интерфейса *IParser*. Этот интерфейс содержит единственный метод

void Parse (*string filePath*, *ParserResults result*).

Каждый анализатор реализует этот метод, получающий на входе доступ к файлу с исходным кодом, обрабатывает этот файл и добавляет результат работы в объект класса *ParserResults*, который содержит термы, представляющие все файлы проекта. Возможно также добавление дополнительной информации, специфической для данного анализатора.

Система содержит информацию обо всех добавленных анализаторах, позволяя пользователю выбрать наиболее подходящий для данной задачи. Также за пределы индивидуальных ана-

лизаторов вынесена такая общая функциональность, как обработка каталогов и выявление файлов с исходным кодом. При этом у анализаторов остается возможность добавить специфическую обработку исходного кода (например, использование скомпилированных библиотек или анализ зависимостей между различными файлами) за счет реализации дополнительных необязательных интерфейсов.

В качестве примера анализатора в систему был включен *C#* parser, доступный на *Codeplex*, предоставляющий программный код в виде объектной модели, где различными классами представлены разные конструкции языка. Полученный в результате работы анализатора объектный граф затем трансформируется в терм; при этом сохраняется и исходная версия.

Система правил

После того, как программный код переведен во внутреннее представление (при помощи анализатора или импортированное из внешней системы), к нему могут быть применены преобразования. Для задания преобразований используются правила *Termware*, т.е. конструкции вида

source [*condition*] → *destination* [*action*],

где *source* – исходный терм (образец для поиска), *condition* – условие применения правила, *destination* – преобразованный терм, *action* – дополнительное действие при срабатывании правила. Каждый из четырех компонентов правила может содержать переменные (которые записываются в виде $*var*), что обеспечивает общность правил. Компоненты *condition* и *action* – необязательны. Они могут исполнять произвольный процедурный код, в частности использовать дополнительные данные о программе.

Применение правила происходит следующим образом: сначала находится подтерм входного терма (дерева программы), который подходит под *source.* Затем проверяется условие применения (если оно присутствует). Если условие выполняется, происходит замена *source* на *destination.* При этом переменные в *destination* заменяются соответствующими значениями из *source*. Также выполняется действие *action* (если оно присутствовало).

Каждое преобразование задается системой правил, т.е. набором правил, последовательно применяющихся к данному терму (дереву программы). Порядок применения правил определяется стратегией. В систему встроены несколько основных стратегий, таких как *TopDown*, *BottomUp*, *FirstTop*. Кроме того, возможно создание дополнительных стратегий.

Кроме использования правил *TermWare*, возможно также написание дополнительных преобразований на *C#*. Они могут напрямую работать как с представлением в виде термов, так и с другими данными о программном коде.

Генератор кода

После применения необходимых преобразований происходит генерация программного кода из (преобразованного) внутреннего представления. При этом из термов восстанавливается соответствующий им исходный код. Возможны два варианта реализации генератора: генерация кода непосредственно из термов или перевод термов в объектную модель анализатора с последующим использованием встроенного генератора. Первый вариант является более простым, если нет необходимости поддерживать термы и объектную модель в синхронизированном состоянии. Если же такая необходимость есть, то по мере проведения преобразований все равно необходимо вносить соответствующие изменения в объектную модель. Поэтому лучше работает второй вариант.

Пользовательский интерфейс

Все возможности системы доступны через графический интерфейс. Он позволяет выбирать код для анализа, проводить преобразования, просматривать представление программы в виде термов, генерировать преобразованный код. Внешний вид интерфейса представлен на рис. 3.

Особенность разработанного графического интерфейса – возможность представления термов в графическом виде (дерева). Даже для не очень больших программ размер полученных

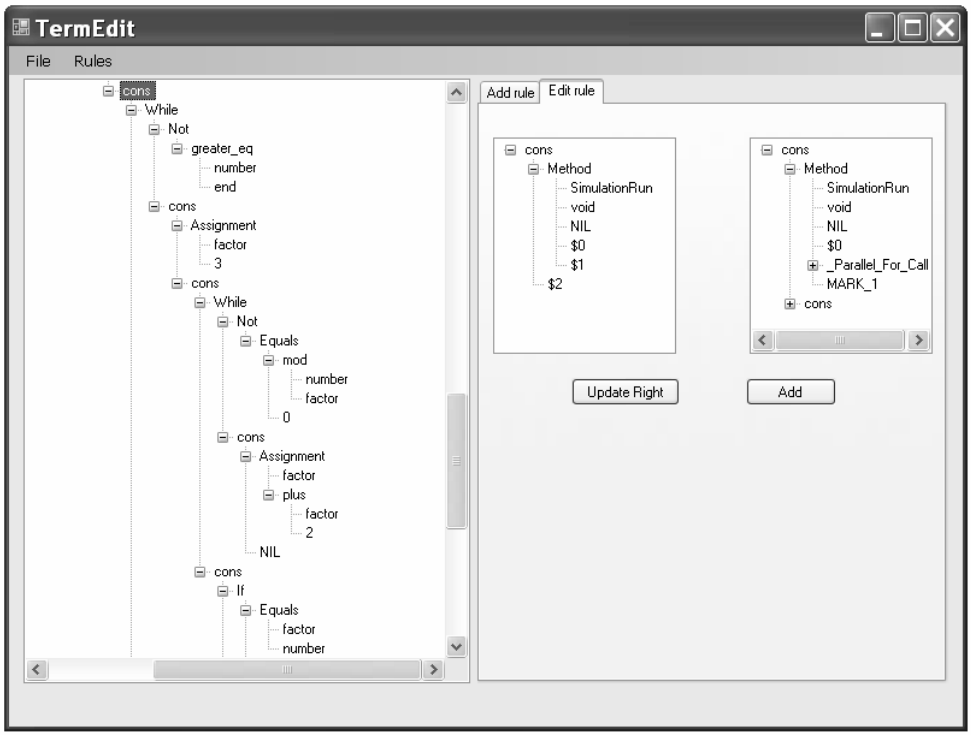

Рис. 3. Графический интерфейс пользователя

термов оказывается достаточно существенным, и манипуляции с ними в текстовом представлении оказываются малоэффективными. Графическое представление позволяет сконцентрировать внимание на нужных частях терма. Кроме того, появляется возможность редактирования термов в древовидной форме. Пользовательский интерфейс позволяет добавлять, удалять и редактировать узлы (термы). При этом возможно комбинированное изменение: когда добавляется не один листовой терм, а целое поддерево, записанное в текстовой форме (например, «*term*1(*term*2,*term*3)»).

Также возможно создание и редактирование правил в графическом представлении. При этом пользователь выбирает подтерм (поддерево) дерева программы, используемого как базис для построения правила, что даст возможность редактирования правила в текстовом виде, а также редактирования левой и правой части в графическом виде. В уменьшенных элементах управления доступны все стандартные возможности основного дерева (добавление/модификация/удаление узлов, копирование и вставка, работа с текстовым представлением подтермов).

Кроме того, доступны две специфические функции: замена переменных и синхронизация изменений. Функция замены переменных предполагает замену некоторого подтерма на переменную *Termware*. По умолчанию используются имена переменных $\$x0$, $\$x1$, …, но при желании пользователь может использовать более осмысленные имена. Особенность данной функции состоит в том, что производится замена не только выбранного подтерма, но и всех вхождений этого подтерма.

Функция синхронизации изменений синхронизирует левую и правую части правила. Это целесообразно в том случае, когда перед описанием изменений пользователю необходимо внести изменения в образец и использовать как в левой, так и в правой частях правила. Примером таких изменений может служить замена переменных.

Кроме создания стандартных заготовок правил, инструментарий предусматривает возможность генерации правил специального вида, например для добавления/перемещения элементов списка, перемещения терма вверх или вниз по дереву и так далее, что позволяет ускорить создание правил определенного вида, а также дает возможность начинающим пользователям изучить на примерах стандартные приемы работы с правилами.

Пример использования

В качестве примера использования разработанного инструментария рассмотрим преобразование последовательной программы в парал-

лельную многопоточную для задачи вычисления простых чисел (*Primes*).

Работа анализатора

Исходная последовательная программа для вычисления простых чисел реализована на языке *C#* и содержит методы для ввода и вывода данных, а также основной метод, вычисляющий простые числа по методу решета Эратосфена. Далее приведен код основного метода:

```
protected override void SimulationRun()
 {
      long number;
      long start;
      long end;
      long stride;
      long factor;
      start = 1;
      end = Number;
      stride = 2;
      if (start == 1)
      {
          start = start + stride;
      };
      number = start;
      while (!(number >= end))
      {
         factor = 3;
          while (!((number % factor) == 0))
          {
            factor = factor + 2;
          };
          if (factor == number)
          {
              Primes[PrimeCount] = number;
              PrimeCount = PrimeCount + 1;
          };
          number = number + stride;
      }
 }
```

Для обработки в системе код загружается и преобразовывается во внутреннее представление (дерево термов). Часть терма, соответствующая данному методу, приведена ниже:

```
Method(SimulationRun,void,NIL,[protected,over
ride],
 [Declaration(number,long),
  Declaration(start,long),
  Declaration(end,long),
  Declaration(stride,long),
  Declaration(factor,long),
  Assignment(start,1),
  Assignment(end,Number),
  Assignment(stride,2),
  If(Equals(start,1), Assignment(start,
start+stride)),
  While(Not(number>=end),[
  Assignment(factor,3),
  While(Not(Equals(number % factor,0)),
[Assignment(factor,factor+2)]),
  If(Equals(factor,number),[
Assignment(ArrayElement(Primes,PrimeCount),
number),
Assignment(PrimeCount, PrimeCount +1)]),
Assignment(number,number+stride)
])])
```

Отметим, что в данной записи терма использованы некоторые конструкции языка *Termware* для сокращения записи. Например, списки имеют вид [*t*1, *t*2, …, *tn*], что эквивалентно терму *cons*(*t*1,*cons*(*t*2,…,*cons*(*tn*,*NIL*))..). Используются также обычная запись арифметических операторов $a + b$, $a >= b$ вместо термов *plus*(*a*,*b*) или *geq*(*a*,*b*), что облегчает работу с термами для пользователя. Однако при выполнении правил система использует именно полные представления в виде термов, а не сокращенные варианты.

Разработка правил

После перехода к представлению кода в виде термов, к полученному терму программы применяются правила. Для распараллеливания основного цикла применялась следующая система правил:

1. *cons*(*Method*(*SimulationRun*,*void*,*NIL*,$*x*1,$*x*2),$*x*0) -> *cons*(*Method*(*SimulationRun*,*void*,*NIL*,$*x*1, *_Parallel_For_Call* (*RunThreadNum*,[*_Index*], *Threads*),*_MARK*_1), *cons*(*Method*(*RunThreadNum*,*void*, [*Parameter*(*ThreadNum*,*int*)], [*protected*,*virtual*],$*x*2),$*x*0))
2. *Assignment*(*start*,1) -> *Assignment*(*start*, *ThreadNum* * *BLOCKSIZE* + 1)
3. *Assignment*(*end*,*Number*) -> *Assignment*(*end*, *ThreadNum* * *BLOCKSIZE* + *BLOCKSIZE*)
4. *If*(*Equals*(*factor*,*number*),$*x*0) -> *If*(*Equals*(*factor*,*number*), *Lock*(*This*,$*x*0))

Эта система действует следующим образом: правило 1 заменяет главный метод *SimulationRun* на вызов нового метода *RunThreadNum* на заданном числе потоков. При этом код метода *RunThreadNum* после действия правила 1 оказывается таким же, как исходный код метода *SimulationRun*. Далее правила 2–4 модифицируют код вычислений с учетом введения многопоточности. Правила 2 и 3 меняют границы цикла, которые теперь зависят от номера потока. Правило 4 добавляет блокировку для защиты операции доступа к общей памяти (переменной *PrimeCount* и массиву *Primes*).

Заметим, что данная система правил разрабатывалась с использованием возможностей

пользовательского интерфейса инструментария. Создание правила начиналось с выбора нужной части терма, подлежащего преобразованию, и генерации заготовки правила на основании выбранного подтерма. Далее правая часть правила заменялась таким образом, чтобы получить корректную многопоточную программу (так разрабатывались правила 2 и 3). При необходимости использовалась операция замены переменных (например, для правила 4, где нужно было переместить достаточно большой подтерм).

В приведенном примере системы правил самое сложное – правило 1, поскольку оно выполняет довольно сложное преобразование – собственно замену однопоточного метода *SimulationRun* на многопоточный вызов *RunThreadNum*. При разработке правила 1 использовались дополнительные возможности разработанного инструментария: создание правил специального вида, в данном случае добавление элемента в список. Для создания такого правила пользователь выбирает из контекстного меню для терма, соответствующего методу *SimulationRun*, команду "*Create Special Rule – List Insert After*". Эта команда, примененная к терму *T*(*x*1,…, *xn*), создает следующую заготовку правила: *cons*(*T*(*x*1,…, *xn*),$*x*0)->*cons*(*T*(*x*1,…,*xn*,_*MARK*_1),*cons*(_*TODO_Add*,$*x*0)). Такое правило вставляет в список, содержащий терм *T*(*x*1,…, *xn*), сразу за ним, терм *_TODO_Add*. Кроме того, терм *T* помечается меткой *_MARK*_1 (т.е. метка добавляется к терму как еще один подтерм). Это делается для того, чтобы избежать зацикливания системы правил: если не добавлять такую метку, то же правило может быть применено повторно. Метки обрабатываются специальным образом: инструментарий поддерживает команду для удаления всех меток, а также они игнорируются при генерации кода (т.е. по терму *Method*(…, *_MARK*_1) будет сгенерирован корректный код метода, а добавленный терм *_MARK*_1 – проигнорирован).

После применения специального правила разработка правила 1 идет аналогично другим правилам: добавляются нужные термы, используется операция замены переменных. Представляет интерес добавленный подтерм *_Parallel _For_Call* (*RunThreadNum*, [*_Index*], *Threads*). Этот терм является примером специальных встроенных термов, расширяющих возможности целевого языка (например, *C#*) путем добавления новых конструкций. При разработке правил такие термы могут использоваться наравне с конструкциями языка, такими как *For* или *Assignment*. Перед генерацией кода запускаются встроенные системы правил, которые заменяют такие добавленные конструкции эквивалентными термами из числа конструкций языка. Например, для *_Parallel_For_Call* (многопоточный вызов метода в цикле) вызывается следующая система правил:

1. *_Parallel_For_Call* ($*MethodName*,$*Parameters*,$*Thread Number*) ->
[*_DeclarationAssignment_Array*(*threads*,*Thread*, $*Thread Number*), *_For_Common* ($*ThreadNumber*,
_Run_Thread(*ArrayElement*(*threads*,i), $*MethodName*,$*Parameters*)),
_For_Common ($*ThreadNumber*,*_Wait_For_Thread*(*ArrayElement*(*threads*,i)))]
2. *_Index*->i

Таким образом, конструкция *_Parallel_For_Call* заменяется на создание массива потоков, вызов в цикле на каждом из них метода с соответствующими параметрами и последующего ожидания завершения всех потоков. При этом используются дополнительные термы: *_DeclarationAssignment_Array*, *_For_Common*, *_RunThread*, *_Wait_For_Thread*. Эти термы в свою очередь преобразуются в конструкции языка с использованием встроенных систем правил. Таким образом, инструментарий облегчает разработку, предоставляя набор конструкций, расширяющих возможности языка программирования и позволяющих сконцентрироваться на особенностях задачи, а не на реализации стандартных участков кода.

Генерация кода

Разработанная система правил применяется к терму последовательной программы, в результате чего получается терм параллельной многопоточной программы. Добавляется новый метод *RunThreadNum*, меняются границы

циклов с учетом номера потока, добавляется блокировка критической области. Приведем для примера терм, соответствующий новому методу *RunThreadNum*:

```
   Method(RunThreadNum,void,[Parameter(Thread
Num,int)],[protected,virtual],
    [Declaration(number,long),
     Declaration(start,long),
     Declaration(end,long),
     Declaration(stride,long),
     Declaration(factor,long),
     Assignment(start, ThreadNum * BLOCKSIZE
+ 1),
     Assignment(end, ThreadNum * BLOCKSIZE +
BLOCKSIZE),
     Assignment(stride,2),
     If(Equals(start,1), Assignment(start,
start+stride)),
     While(Not(number>=end),[
     Assignment(factor,3),
     While(Not(Equals(number % factor,0)),
[Assignment(factor,factor+2)]),
     If(Equals(factor,number),Lock(This,[
   Assignment(ArrayElement(Primes,Prime
Count),number),
   Assignment(PrimeCount, PrimeCount +1)])),
   Assignment(number,number+stride)
   ])])
```

По полученному терму генерируется преобразованный код параллельной программы:

```
      protected virtual void RunThreadNum(int
ThreadNum)
      {
          long number;
          long start;
          long end;
          long stride;
          long factor;
          start = ThreadNum * BLOCKSIZE + 1;
          end = ThreadNum * BLOCKSIZE +
BLOCKSIZE;
          stride = 2;
          if (start == 1)
          {
              start = start + stride;
          };
          number = start;
          while (!(number >= end))
          {
              factor = 3;
              while (!((number % factor) ==
0))
              {
                  factor = factor + 2;
              };
              if (factor == number)
              {
                  lock (this)
                  {
                      Primes[PrimeCount] =
number;
                      PrimeCount = PrimeCount
+ 1;
                  }
              };
              number = number + stride;
          }
      }
      protected override void SimulationRun()
      {
          Thread[] threads = new
Thread[Threads];
          for (int i = 0; i < Threads; i++)
          {
              Thread t = new Thread(new
ParameterizedThreadStart(RunThreadNum));
              threads[i] = t;
              t.Start(i);
          }
          for (int i = 0; i < threads.Length;
i++)
          {
              threads[i].Join();
          }
      }
```

Отметим, что весь код метода *SimulationRun* является новым, добавленным в процессе распараллеливания. При этом весь этот код был сгенерирован из единственного терма

_Parallel_For_Call(*RunThread-Num*,[*_Index*],*Threads*).

Можно сравнить размер терма, введенного пользователем, и терма, соответствующего сгенерированному коду: в первом случае размер (количество узлов дерева) равен 5, тогда как во втором случае – 54. Таким образом, в данном примере использование встроенных правил позволило сократить в 10,8 раз размер терма (что приводит к аналогичному сокращению объема работ программиста и количества потенциальных ошибок). Это демонстрирует преимущества разработанного инструментария, а именно возможность кратко выражать достаточно сложные конструкции.

Заключение. В статье рассмотрен программный инструментарий для проектирования и разработки приложений на платформе *Microsoft .NET*. Инструментарий позволяет автоматизировать преобразования программ на языке *C#*, в частности переход от последовательных к параллельным многопоточным программам. Инструментарий состоит из анализатора (анализатора) для перевода кода на *C#* во внутреннее представление, системы переписывающих правил *Termware*, генератора кода и графиче-

ского интерфейса для редактирования термов и правил. Наличие графического интерфейса, а также набора встроенных правил и шаблонов правил облегчают работу пользователя с системой, позволяют сконцентрироваться на реализации нужных преобразований, а не на синтаксисе языка правил. Применение разработанного инструментария проиллюстрировано на примере перехода от последовательной к параллельной многопоточной программе для вычисления простых чисел.

В рамках дальнейшего развития разработанного инструментария планируется расширение набора встроенных правил, добавление новых возможностей в графический интерфейс пользователя (например, подсветка изменений при применении правил). Также планируется добавить поддержку других языков программирования (*Java*, *C*/*C*++). Для проверки возможностей разработанного инструментария планируется его применение на масштабных промышленных проектах.

Автор выражает благодарность профессору А.Е. Дорошенко за помощь при подготовке статьи.

1. *Эндрюс Г.* Основы многопоточного, параллельного и распределенного программирования. – М.: ИД Вильямс, 2003. – 512 с.
2. *Дорошенко А.Е.*, *Жереб К.А.*, *Яценко Е.А.* Формализованное проектирование эффективных многопоточных программ // Проблемы программирования. – 2007. – № 1. – С. 17–30.
3. *Дорошенко А.Е.*, *Жереб К.А.*, *Яценко Е.А.* Об оценке сложности и координации вычислений в многопоточных программах // Там же. – № 2. – С. 41–55.
4. *Яценко Е.А.*, *Мохница А.С.* Инструментальные средства конструирования синтаксически правильных параллельных алгоритмов и программ // Там же. – 2004. – № 2–3. – С. 444–450.
5. *Doroshenko A.*, *Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications // Fundamenta Informaticae. –2006. – **72**, N 1–3, 2006. – P. 95–108.
6. *TermWare*. – http://www.gradsoft.com.ua/products/termware_rus.html
7. *Дорошенко А.Е.*, *Шевченко Р.С.* Система символьных вычислений для программирования динамических приложений // Проблемы программирования. – 2005. – № 4. – С. 718–727.
8. *Ernest Friedman-Hill*. Jess in Action. Green wech Manning Publ. Co, 2003. – 480 p.
9. *Timothy Winkler*. Programming in OBJ and Maude // Functional Programming, Concurrency, Simulation and Automated Reasoning, Intern. Lecture Series 1991–1992, Springer-Verlag, McMaster University, Hamilton, Ontario, Canada, 1993. – P. 229–277.
10. *Visser E.* Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5 // Rewriting Techniques and Applications (RTA 01) LNCS vol. 2051, Springer, 2001. – P. 357–361.
11. *Compiling* Language Definitions: The {ASF}+{SDF} compiler / M. van den Brand, J. Heering, P. Klint et al. // University of Amsterdam, 1999. – P. 334–368.
12. *Alejandra Garrido*, *Jose Meseguer*. Formal Specification and Verification of Java Refactorings // Proc. of 6th IEEE Intern. Workshop on Source Code Analysis and Manipulation (SCAM'06, 27–29 Sept. 2006, Filadelfia). – P. 165–174.
13. *Jing Xu.* Rule-based Automatic Software Performance Diagnosis and Improvement // Proc. of the 7th Intern. Workshop on Software and performance (WOSP'08), 24–26 June 2008, Princeton, New Jersey, USA. – P. 1–12.
14. *Stratego/XT* 0.16: components for transformation systems / M. Bravenboer, K.T. Kalleberg, R. Vermaas et al. // Proc. of the 2006 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (Charleston, South Carolina, 9–10 Jan. 2006). PEPM '06. ACM, New York, NY. – P. 95–99.
15. *Ira D. Baxter*, *Christopher Pidgeon*, *Michael Mehlich.* DMS: Program Transformations for Practical Scalable Software Evolution // Proc. of 26th Intern. Conf. on Software Engin. (ICSE'04), 2004. – P. 625–634.
16. *Metaspec* C# Library. – http://www.csharpparser.com/csparser.php
17. *C#* Parser on Codeplex.– http://www.codeplex.com/csparser
18. *ANTLR* Parser Generator. – http://www.antlr.org/
19. *GOLD* Parsing System. – http://www.devincook.com/goldparser/

Поступила 30.03.2009
Тел. для справок: (044) 419-6450 (Киев)
E-mail: *zhereb@gmail.com*