С.В. Востокин

БАЗОВЫЙ СИНТАКСИС ЯЗЫКА РАЗМЕТКИ TEMPLET ДЛЯ ПРЕДСТАВЛЕНИЯ МОДЕЛИ «ПРОЦЕСС-СООБЩЕНИЕ»

(Самарский государственный аэрокосмический университет)

Введение. Для большого числа прикладных задач целесообразно представление кода в виде совокупности процессов, обменивающихся сообщениями. Например, в области высокопроизводительных вычислений такое разбиение служит для эффективного использования ядер и процессоров. В распределенных вычислениях важен протокол взаимодействия объектов при помощи сообщений. В интеллектуальных многоагентных системах необходимы средства описания процедур обработки сообщений в программных агентах. Системы имитационного дискретно-событийного моделирования обычно в качестве событий рассматривают поступление сообщений в процессы и их обработку.

Традиционное применение модели «процесс-сообщение» в средстве программирования состоит в разработке специального языка программирования или библиотеки времени исполнения для имеющегося языка. Очевидным недостатком первого подхода является сложность разработки, а второго — сложность применения.

Автором предлагается новый подход, основанный на том, что модель «процесс-сообщение» может быть описана на любом языке программирования с пользовательскими типами данных и процедурной семантикой. Для этого необходимо следовать специальным соглашениям на структуру кода. Использование предметно-ориентированного языка разметки кода позволяет автоматизировать построение модели «процесс-сообщение» средствами традиционного языка программирования.

Структура кода. Код программы (например, на языке C++) в терминах модели «процесс-сообщение» можно структурировать следующим образом. Во-первых, это библиотека времени исполнения с базовыми классами «процесс», «сообщение», «диспетчер сообщений» и так далее, для передачи структуры потока управления. Во-вторых, — «связывающий» код, объединяющий библиотеку времени исполнения и код конкретной задачи. В-третьих, – собственно типы данных для сообщений и процедуры их обработки.

Таким образом, код в терминах модели «процесс-сообщение» будет иметь блочную структуру. Поэтому для его анализа не требуется знать синтаксис

языка, а достаточно уметь определять границы блоков. Очевидным способом определения границ блоков является разметка при помощи комментариев. Например, в фрагменте кода

```
bool Parent::hello()
{
/*$TET$Parent$hello*/
     cout<<"Hello world!";
     return true;
/*$TET$*/
}
```

тело метода Parent::hello() может быть легко извлечено из кода, если известны сигнатуры комментариев. В примере bool Parent::hello(){...} — связывающий код, а cout<<"Hello world!"; return true; — код конкретной задачи или пользовательский блок.

Структуру связывающего кода и блоков пользователя можно описать в компактной форме на языке модели «процесс-сообщение». При этом само описание может быть также встроено в код в форме комментария:

```
/*$TET$templet$!templet!*/
/* *Parent=+hello(). */
/*$TET$*/.
```

С учетом описанных допущений контроль соответствия структуры кода модели «процесс-сообщение» может быть проведен перед компиляцией. Для этого требуется выполнить следующую трансформацию кода программы. Необходимо извлечь пользовательские блоки и описание структуры кода из программы, а затем (если структура изменилась) генерировать код программы в соответствии с новым описанием структуры, помещая в код извлеченные ранее пользовательские блоки.

Описание каналов. В модели системы используются два типа объектов - канал и процесс. Канал описывает протокол взаимодействия при помощи сообщений от двух участников: клиента и сервера. Каждый тип канала имеет имя однозначно идентифицирующее его. Канал может состоять из нескольких состояний. Это описывается правилом EBNF:

```
channel = '~' ident ['=' state {';' state}] '.'.
```

Нетерминал ident соответствует имени типа канала. Состояния описывают последовательность взаимодействия между двумя участниками: клиентом и

сервером. Состояния имеют имя; делятся на два множества: состояние клиента и состояние сервера; одно из состояний определяется как начальное. Если состояние является состоянием клиента, то сообщение передает клиент, а принимает сервер. Иначе — передает сервер, а принимает клиент. Правило EBNF для состояния имеет вид:

```
state = ['+'] ident [ ('?'|'!')  [rules] ].
```

Здесь начальное состояние обозначено знаком '+'; состояние клиента обозначено знаком '?' (он задает вопрос); состояние сервера обозначено знаком '!' (он отвечает на вопрос клиента). Нетерминал ident обозначает имя состояния.

Наконец, правило, описывающее, в какое состояние будет выполняться переход при отправке сообщения из текущего состояния, задается в виде

```
rules = ident '->' ident {'|' ident '->' ident}.
```

Идентификатор перед знаком '->' обозначает сообщение, идентификатор после знака '->' обозначает новое состояние.

Описание процессов. Процесс определяет алгоритм обработки сообщений, поступающих по каналам от других процессов, и ответы на сообщения. Каждый тип процесса имеет имя, однозначно идентифицирующее его. Процесс может состоять из нескольких портов и нескольких действий. Это представлено правилом EBNF:

```
 process = '*' ident ['=' ((ports [';' actions]) | actions) ] '.'.
```

Порт описывает точку подключения канала к процессу. Он характеризуется идентификатором; типом подключаемого канала; ролью процесса во взаимодействии через данный порт (клиент или сервер); а также действием, которое запускается при поступлении сообщения в данный порт, если не выполнены (или не указаны) связанные с портом правила. Правила EBNF для порта имеют вид:

```
ports = port {';' port}.
port  = ident ':' ident('?'|'!')[(rules ['|' '->' ident])|( '->' ident)].
```

В правилах комбинация имя-тип порта обозначена как ident ':' ident. Знак '?' показывает, что сообщения в правилах — вопросы от клиента. Знак '!' показывает, что поступающие в данный порт сообщения — ответы сервера.

Синтаксис правил для порта аналогичен правилам для каналов. Правила для канала имеют следующую интерпретацию. Слева от знака '->' записывается

имя сообщения, поступающего в канал, а справа — имя действия, которое нужно запустить при поступлении данного сообщения.

Действия описывают процедуры обработки поступающих в процесс сообщений и выдачу ответных сообщений. Действия имеют уникальные в пределах процесса имена. Одно из действий может помечаться как начальное (оно запустится при запуске программы). Действие может иметь связанное действие, запускаемое при успешном выполнении. Также может указываться связанное действие, запускаемое при не успешном выполнении. Связанные действия образуют цепочку неделимых (атомарных) операций обработки поступившего сообщения (или начального действия при запуске программы). Правила EBNF для действий имеют вид:

```
actions = action {';' action}.
action  = ['+'] ident '(' [args] ')'
              ['->' ([ident] '|' ident) | ident].
```

Здесь первый идентификатор задает имя действия; идентификатор, записанный после знака '->', задает действие при успешном выполнении текущего действия; идентификатор, записанный после знака '|', задает действие при неуспешном выполнении текущего действия.

Аргумент действия — сообщение на заданном порту, которое может считываться или записываться, а затем отправляться получателю. Правило EBNF для аргументов имеет вид:

```
args = ident ('?'|'!') ident
              {',' ident ('?'|'!') ident}.
```

Первый идентификатор обозначает порт; знак '?' обозначает (в данном контексте) чтение сообщения; знак '!' обозначает запись и отправку сообщения. Имя сообщения указывается после знаков '?' или '!'.

При генерации кода для действия должны выполняться следующие правила.

А. Действие запускается, если в текущем состоянии подключенных к процессу каналов возможно считать или отправить сообщения, перечисленные как аргумент действия.

Б. Сообщения отправляются, если действие было запущено и вернуло признак успешного завершения.

В. Если указан атрибут в позиции → X | , то переход к связанному действию происходит, если действие было запущено и вернуло признак успешного завершения. Иначе, если указан атрибут в позиции → | X , выполняется переход к данному связанному действию.

Применение и сравнение с аналогами. Рассмотренный язык разметки TEMPLET используется в препроцессоре, входящем в состав веб-сервиса автоматизации параллельных вычислений на суперкомпьютере «Сергей

Королев» Самарского государственного аэрокосмического университета, развернутого по адресу http://templet.ssau.ru. Он позволяет разработать и отладить каркас приложения как последовательную программу на локальной машине, а затем автоматически запустить код на исполнение на суперкомпьютере в параллельном режиме с использованием API POSIX Threads. Полная спецификация языка разметки Templet приведена в эталонной реализации [1].

В текущей реализации также выполняется трансформация кода для исполнения в операционных системах Windows с использованием Windows API. Применение языка разметки Templet не требует от пользователя знаний методов параллельного программирования в Unix или Windows и позволяет сосредоточиться на решении прикладной задачи. Другие варианты генерации кода позволяют добавлять отладочную информацию для работы со специально ориентированным на данную модель отладчиком. Разрабатывается средство визуализации кода на основе пакетов OpenOffice/LibreOffice с использованием графической нотации Templet [2].

В дизайне препроцессора Templet используются несколько подходов к автоматизации программирования. Один из них — метод разметки последовательного кода препроцессорными инструкциями для распараллеливания, который широко распространен, в частности, в стандарте OpenMP [3], российской системе DVM [4]; в системе Cilk [5], в её российском аналоге T++ [6] и других. Но, в отличие от них, в системе TEMPLET контроль структуры кода реализуется автоматически за счет генерации кода, а не вручную. Генерирующие макропроцессоры (m4 [7] или современный вариант T4 [8]) также используются для автоматизации программирования. Описанная система относится к классу так называемых активных генераторов, однако, является проблемно-ориентированной. Модель языка Templet – это разновидность модели акторов [9], но в отличие от специальных языков, например Erlang [10], мы реализуем акторную семантику средствами процедурного языка. Это позволяет использовать системы программирования существующих языков. Система Templet реализует концепцию разработки, управляемой моделями (model-driven development) [11]. В области параллельного программирования это важно при автоматическом преобразовании кода для распараллеливания. Особенность нашей модели в том, что она является комбинацией обычного и предметного языков. Причем семантика модели представлена на обычном языке. Роль предметного языка в системе Templet — краткое описание каркаса программы. Препроцессор Templet может использоваться в режиме метапрограммирования [12]: исходная

программа преобразуется в другую программу, выполняющую более сложные преобразования, чем позволяет препроцессор. Например, так могут быть реализованы глубокий семантический анализ и оптимизация кода.

Работа выполнена в рамках проекта «Разработка комплекса технологий использования ресурсов суперкомпьютера «Сергей Королёв» в целях развития инновационной и научно-образовательной среды университета» Самарского государственного аэрокосмического университета имени академика С.П. Королева (национального исследовательского университета).

ЛИТЕРАТУРА

1. Востокин С.В. Эталонная реализация языка TEMPLET. Свидетельство о государственной регистрации программы для ЭВМ №2014613169 от 19.03.2014.
2. Востокин С.В. TEMPLET – метод процессно-ориентированного моделирования параллелизма // Программные продукты и системы, 2012. №3. С. 9-12.
3. OpenMP application program interface, version 3.0. OpenMP specification, May 2008.
4. Бахтин В.А., Крюков В.А., Четверушкин Б.Н., Шильников Е.В. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами //Доклады Академии Наук, 2011, том 441, № 6, С. 734–736.
5. Intel Corporation. Intel® Cilk™ Plus Language Specification, 2011.
6. Абрамов С.М., Кузнецов А.А., Роганов В.А.. "Кроссплатформенная версия T-системы с открытой архитектурой" //Вычислительные методы и программирование. - 2007. - Т. 8. - No 1. - Раздел 2. стр. 175-180.
7. Free Software Foundation Inc. GNU M4, version 1.4.16, February 2011.
8. Microsoft Developer Network. Code Generation and T4 Text Templates. http://msdn.microsoft.com/en-us/library/vstudio/bb126445.aspx.
9. Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems. Proceedings of Inconsistency Robustness. 2011. (http://arxiv.org/abs/1008.1459)
10. Larson J. Erlang for Concurrent Programming // Communications of the ACM, 2007, Vol. 52 No. 3, Pages 48-56.
11. Jishnu Mukerji, Joaquin Miller. Overview and guide to OMG's architecture. OMG, June 2003.
12. Чистяков В. R# - метапрограммирование в .NET. RSDN Magazine №5, 2004.