Вестн. Сам. гос. техн. ун-та. Сер. Физ.-мат. науки. 2014. № 3 (36). С. 169–182
ISSN: 2310-7081 (online), 1991-8615 (print) doi: http://dx.doi.org/10.14498/vsgtu1334

Информатика

УДК 519.682

ПРЕПРОЦЕССОР ЯЗЫКА TEMPLET: ИНСТРУМЕНТ ПРОГРАММИРОВАНИЯ В ТЕРМИНАХ МОДЕЛИ «ПРОЦЕСС–СООБЩЕНИЕ»*

С. В. Востокин

Самарский государственный аэрокосмический университет им. ак. С. П. Королёва (национальный исследовательский университет),
443086, Россия, Самара, Московское ш., 34.

Аннотация

Мотивация: Для большого числа прикладных задач целесообразно представление кода решаемой задачи в виде совокупности процессов, обменивающихся сообщениями. Традиционное применение модели «процесс–сообщение» в средстве программирования состоит в разработке специального языка или библиотеки времени исполнения для имеющегося языка. Недостаток первого подхода — это сложность разработки, а второго — сложность применения. Предлагается новый метод описания модели «процесс–сообщение», использующий язык программирования с процедурной семантикой и препроцессор, устраняющий указанные недостатки. **Метод:** Код программы в терминах модели «процесс–сообщение» делится на блоки: библиотека времени исполнения; связывающий код, объединяющий библиотеку времени исполнения и код конкретной задачи; типы данных для сообщений и процедуры их обработки. Границы блоков обозначаются комментариями. Для описания структуры кода в целом предлагается предметно-ориентированный язык, названный `Templet`. Метод позволяет проводить контроль соответствия структуры кода модели «процесс–сообщение» автоматически перед компиляцией. **Описание каналов:** Канал описывает протокол обмена сообщениями между парой процессов. Приведён синтаксис каналов с использованием расширенной нотации Бекуса—Наура (EBNF). Информационная структура каналов, используемая для генерации кода, описана с применением диаграммы «сущность–связь» (ER). **Описание процессов:** Процесс определяет алгоритм обработки сообщений, поступающих по каналам

Образец для цитирования: Востокин С. В. Препроцессор языка Templet: инструмент программирования в терминах модели «процесс–сообщение» // *Вестн. Сам. гос. техн. ун-та. Сер. Физ.-мат. науки*, 2014. № 3 (36). С. 169–182. doi: 10.14498/vsgtu1334.

Сведения об авторе: *Сергей Владимирович Востокин* (д.т.н., доц.; easts@mail.ru), профессор, каф. информационных систем и технологий.

*Настоящая статья представляет собой расширенный вариант доклада [1], сделанного автором на Международной научно-технической конференции «Перспективные информационные технологии» (Самара, СГАУ, 30 июня – 2 июля 2014).

от других процессов, и ответы на сообщения. Показана информационная структура каналов во взаимосвязи с синтаксисом. В описании также использованы метамодели EBNF и ER. Описание синтаксиса сопровождается примером процесса «разветвление–слияние». **Схема работы препроцессора:** Рассматривается укрупнённый алгоритм работы и архитектура препроцессора языка `Templet`. Препроцессор состоит из подсистем синтаксического анализатора, анализатора блоков, базы данных, семантического анализатора, механизма вывода и генератора кода. Описывается алгоритм работы препроцессора. На основе анализа количества строк кода в контрольном примере показано сокращение объёма ручного кодирования примерно в 20 раз. **Применение и сравнение с аналогами:** Препроцессор применяется в составе web-сервиса для автоматизации параллельных вычислений. Он используется для подготовки скелетов программ, которые пользователи дополняют проблемно-ориентированным кодом. Описаны преимущества, общие черты и различия предложенного подхода в сравнении с технологиями разметки последовательного кода, генерирующими макропроцессорами, специализированными параллельными языками, метапрограммированием и разработкой, управляемой моделями.

Ключевые слова: препроцессор, предметно-ориентированный язык, процесс, сообщение, параллельное программирование.

doi: http://dx.doi.org/10.14498/vsgtu1334

Мотивация. Для большого числа прикладных задач целесообразно представление кода решаемой задачи 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 разработанного универсального препроцессора вместе с предметно-ориентированным языком позволяет автоматизировать кодирование модели «процесс–сообщение» средствами традиционного языка программирования.

В работе рассмотрены архитектура и принцип работы препроцессора. Описан синтаксис предметно-ориентированного языка (DSL) для модели «процесс–сообщение» на основе расширенной нотации Бекуса—Наура (EBNF). Семантика языка показана с использованием диаграмм «сущность–связь» (ER) и вербального описания. Приведён пример описания системы процессов. Рас-

смотрено применение предлагаемого метода, выполнено сравнение с известными аналогами.

Метод. 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 препроцессору 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 учётом описанных допущений контроль соответствия структуры кода модели «процесс–сообщение» может быть проведён 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.

Как компактно представить структуру кода на DSL-языке модели «процесс–сообщение» и выполнить генерацию кода на целевом языке программирования, показано ниже. Для этого рассматриваются два типа объектов модели — канал и процесс.

Описание каналов. Канал описывает протокол взаимодействия при помощи сообщений от двух участников — клиента и сервера. Модель каналов представлена на рис. 1. Модель описывает множество типов каналов (сущностей

CHANNEL). Каждый тип канала имеет имя (атрибут `channel`), однозначно идентифицирующее его. Канал может состоять из нескольких состояний (сущностей STATE). Вышеизложенное описывается правилом EBNF:

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

Нетерминал `ident` соответствует имени типа канала `channel` на ER-диаграмме рис. 1. Состояния STATE описывают последовательность взаимодействия между двумя участниками — клиентом и сервером. Состояния имеют имя (атрибут `state`); делятся на два множества — состояние клиента и состояние сервера (атрибут `cli_or_srv`); одно из состояний определяется как начальное (атрибут `is_initial`). Если состояние является состоянием клиента, то сообщение передаёт клиент, а принимает сервер. Иначе — передаёт сервер, а принимает клиент. Правило EBNF для состояния имеет вид:

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

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

Наконец, правило, представленное сущностью RULE, описывает, в какое состояние `new_state` будет выполняться переход при отправке сообщения с именем `message` из текущего состояния. Сущность RULE задаётся правилом

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

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

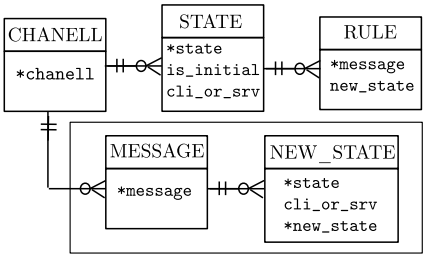

Рис. 1. ER-модель каналов [Figure 1. ER model of the channels]

Пример синтаксиса каналов. Пусть требуется проверить тождество $\sin^2 x + \cos^2 x = 1$. Для этого выполним вычисление квадратов синуса и косинуса одновременно (параллельно) в двух процессах. Взаимодействие между управляющим процессом и процессом, вычисляющим функцию, представим каналом

```
~Link=
    +Begin ? agrSin2 -> Calc | argCos2 -> Calc;
     Calc  ! result  -> End;
     End.
```

Это описание имеет следующую интерпретацию. В начальном состоянии `Begin` управляющий процесс (в роли клиента) посылает сообщение `argSin2` (которое заставляет вычислить квадрат синуса от переданного в сообщении значения) или `argCos2` (для вычисления квадрата косинуса). Исполняющий процесс в состоянии `Calc` (в роли сервера) выполняет вычисление и возвращает результат в сообщении `result`. После этого обработка заканчивается, так как из состояния `End` нет исходящих сообщений.

Для генерации кода сущности, полученные при синтаксическом анализе, дополняются вычисленными сущностями MESSAGE и NEW_STATE. Сущность MESSAGE объединяет все упомянутые в канале сообщения. Сущность NEW_STATE показывает, в каком состоянии `state` может передаваться сообщение и в какое состояние `new_state` затем переходит канал. Вычисленные сущности выделены на рис. 1 рамкой. Генерация кода выполняется путём обхода дополненного дерева ER-модели каналов.

Описание процессов. Процесс определяет алгоритм обработки сообщений, поступающих по каналам от других процессов, и ответы на сообщения. Модель процессов показана на рис. 2. Модель описывает множество типов процессов (сущностей PROCESS). Каждый тип процесса имеет имя (атрибут `process`), однозначно идентифицирующее его. Процесс может состоять из нескольких портов (сущностей PORT) и нескольких действий (сущностей ACTION). Вышеизложенное представлено правилом EBNF:

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

Порт описывает точку подключения канала к процессу. Он характеризуется идентификатором `port`; типом подключаемого канала `channel`; ролью процесса во взаимодействии через данный порт `cli_or_srv` (клиент или сервер); а также действием `alt_action`, которое запускается при поступлении сообщения в данный порт, если не выполнены (или не указаны) связанные с портом правила RULE. 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

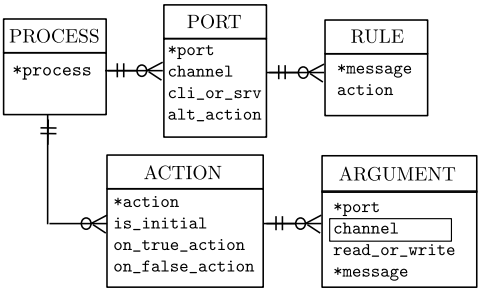

Рис. 2. ER-модель процессов [Figure 2. ER model of the processes]

'!' 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 message, поступающего в канал, а справа — имя действия action, 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 action. Одно из действий может помечаться как начальное is_initial (оно запустится при запуске программы). Действие может иметь связанное действие on_true_action, запускаемое при успешном выполнении. Также может указываться связанное действие on_false_action, запускаемое при неуспешном 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 action; идентификатор, записанный после знака '->', задаёт действие on_true_action; идентификатор, записанный после знака '|', задаёт действие on_false_action. Сущность ARGUMENT описывает аргумент действия — сообщение message на заданном порте port, которое может считываться или записываться read_or_write, а затем отправляться получателю. Правило EBNF для аргумента имеет вид

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

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

Генерация кода, аналогично каналам, выполняется обходом дерева ER-модели процессов. Для генерации кода необходимо вычислить аргумент channel в сущности ARGUMENT (в рамке на рис. 2). 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 on_true_action, то переход к связанному действию происходит, если действие было запущено и вернуло признак успешного завершения. Иначе, если указан атрибут on_false_action, выполняется переход к данному связанному действию.

Поясним использование описаний процессов на примере проверки тождества $\sin^2 x + \cos^2 x = 1$. Управляющий процесс Parent имеет следующий вид:

```
*Parent=
     p1 : Link ! result -> join;
     p2 : Link ! result -> join;
    +fork(p1!argCos2,p2!argSin2);
     join(p1?result,p2?result).
```

Порты `p1` и `p2` связывают процесс `Parent` с процессами, вычисляющими квадрат тригонометрических функций. В действии `fork` выполняется формирование и рассылка сообщений со значением переменной x. В действии `join` выполняется сложение результатов и сравнение суммы с 1. Заметим, что процедура действия `join` запустится только тогда, когда будут доставлены оба результата (согласно правилу A, приведённому выше).

Вычисляющий квадрат синуса или косинуса процесс `Child` определяется так:

```
*Child=
    p : Link ? argCos2 -> calcCos2 | argSin2 -> calcSin2;
    calcCos2(p?argCos2,p!result);
    calcSin2(p?argSin2,p!result).
```

В определении показано, какое действие (вычисление квадрата косинуса `calcCos2` или квадрата синуса `calcSin2`) нужно запустить при получении сообщений `argCos2` или `argSin2`.

Схема работы препроцессора. Рассмотрим работу препроцессора (рис. 3). На этапе 1 из кода извлекаются размеченные блоки. На этапе 2 блок с описанием структуры кода на рассмотренном языке подвергается синтаксическому анализу. Результат анализа (в форме рис. 1 и рис. 2) поступает во внутреннюю базу данных на этапе 3. На этапе 4 выполняется семантический контроль. Контроль семантики канала заключается в проверке уникальности идентификаторов и проверке ссылочной целостности согласно модели рис. 1. Дополнительно выполняется контроль достижимости состояний из начального состояния. Семантический контроль процесса заключается в проверке уникальности идентификаторов согласно модели рис. 2, проверке ссылочной целостности и выполнимости каждого действия. Вывод на этапе 5 дополняет модели рис. 1 и рис. 2 производными сущностями и атрибутами. На последних этапах 6 и 7 из извлеченных блоков и дополненной базы данных генерируется новый код, который замещает собой исходный.

Количество строк кода в программе проверки тождества $\sin^2 x + \cos^2 x = 1$ показано в таблице. Из таблицы видно, что спецификация и остальной пользовательский код занимают незначительную часть общего объема кода. Размер системного кода также невелик (140 строк), то есть механизм исполнения прост для понимания. Количество строк связывающего кода (255 или $\sim 50\,\%$) даёт представление о степени автоматизации программирования препроцессором. Без препроцессора этот код пришлось бы писать вручную. Размер связывающего кода соотносится с размером спецификации как 255/13, то есть объем кодирования сокращается примерно в 20 раз. Связывающий код прост для понимания, так как строится обходом деревьев моделей рис. 1 и рис. 2 без других преобразований.

Применение и сравнение с аналогами. Рассмотренный в настоящей работе препроцессор `Templet` применяется в составе веб-сервиса автоматиза-

Размер фрагментов кода в примере $\sin^2 x + \cos^2 x = 1$ [Size of a code fragments in the sample $\sin^2 x + \cos^2 x = 1$]

Пользователь / система [User code / System code]	Назначение фрагмента кода [The function of the code fragment]	Кол-во строк [Number of lines]	Доля строк в % от общего кол-ва строк [Percentage of lines from the total size]
Код пользователя [User code]	Спецификация [Code specification]	13	2.63
	Пользовательские блоки [User blocks]	35	7.07
	Всего [Total]	48	9.70
Системный код [System code]	Библиотека [Run-time library]	140	28.28
	Разметка блоков [Block mark-up]	52	10.51
	Связывающий код [Binding code]	255	51.51
	Всего [Total]	447	90.30
Всего [Total]	Весь код примера [Entire sample code]	495	100

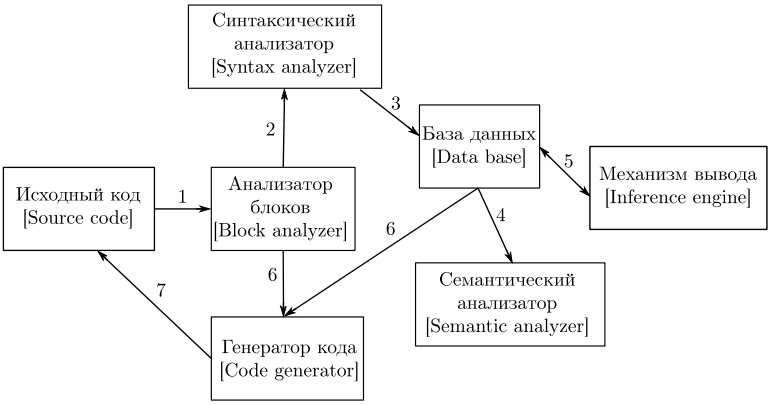

Рис. 3. Схема взаимодействия подсистем препроцессора
[Figure 3. Subsystems of the preprocessor and their interaction]

ции параллельных вычислений на суперкомпьютере «Сергей Королёв» Самарского государственного аэрокосмического университета, развернутого по адресу `http://templet.ssau.ru` [2]. 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`. В текущей реализации также выполняется трансформация кода для исполнения в операционных системах `Windows` с использованием `Windows API`. Применение препроцессора `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` [3]. Сообщение о препроцессоре впервые представлено в работе [1], основные алгоритмы препроцессора и синтаксис языка разметки зарегистрированы в Роспатенте.[1]

Концепции, используемые в дизайне языка, в основном соответствуют стилю языково-ориентированного программирования [4, 5]. Применён близкий к алгебраическим конструкциям способ описания процессов в стиле `CSP` [6]. Идея компактного дизайна, включающего только базовые механизмы абстракции, взята из языка `Oberon` [7].

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` [8], российской системе `DVM` [9, 10]; в системе `Cilk` [11], в её российском аналоге `T++` [12, 13] и других. Но, в отличие от них, в системе `Templet` контроль структуры ко-

[1]Эталонная реализация языка `Templet`: свидетельство о гос. регистрации прогр. для ЭВМ № 2014613169 Российская Федерация / С. В. Востокин; правообладатель С.В. Востокин. – Зарегистрировано в Реестре программ для ЭВМ 19.03.2014; опубл. 20.04.2014, ОБПБТ № 4 (90).

да реализуется автоматически за счёт генерации кода, а не вручную. Генерирующие макропроцессоры (`m4` [14] или современный вариант `T4` [15]) используются для автоматизации программирования. Описанная система также относится к классу так называемых активных генераторов, однако является проблемно-ориентированной. Модель системы `Templet` — это разновидность модели акторов [16–18], но в отличие от специальных языков, например `Erlang` [19], мы реализуем акторную семантику средствами процедурного языка. Это позволяет использовать системы программирования существующих языков. Система `Templet` реализует концепцию разработки, управляемой моделями (model-driven development) [20–23]. 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, причём семантика модели представлена на обычном языке. Роль предметного языка в системе `Templet` — краткое описание каркаса программы. Препроцессор `Templet` может использоваться в режиме метапрограммирования [21, 24]: исходная программа преобразуется в другую программу, выполняющую более сложные преобразования, чем позволяет препроцессор. Например, так могут быть реализованы глубокий семантический анализ и оптимизация кода.

Благодарности. Автор выражает благодарность Самарскому государственному аэрокосмическому университету за поддержку этого исследования. Библиотека времени выполнения препроцессора была реализована для использования на суперкомпьютере «Сергей Королёв» СГАУ при постоянной помощи со стороны обслуживающего персонала.

Работа выполнена при государственной поддержке Министерства образования и науки РФ в рамках реализации мероприятий Программы повышения конкурентоспособности СГАУ среди ведущих мировых научно-образовательных центров на 2013–2020 годы.

ORCID
Sergey Vostokin: http://orcid.org/0000-0001-8106-6893

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Востокин С. В. Базовый синтаксис языка разметки Templet для представления модели «процесс-сообщение» / *Перспективные информационные технологии (ПИТ 2014)*: Труды Международной научно-технической конференции; ред. С. А. Прохоров. Самара: Изд-во Самарского научного центра РАН, 2014. С. 317–323, http://templet.ssau.ru/wiki/lib/exe/fetch.php?media=pit2014:templetlang.pdf.
2. Артамонов Ю. С., Востокин С. В., Назаров Ю. П. Templet – Сервис непрерывной интеграции для разработки высокопроизводительных приложений / *Высокопроизводительные параллельные вычисления на кластерных системах*: Материалы XII всероссийской конференции. Нижний Новгород: Изд-во НГУ, 2012. С. 82.
3. Востокин С. В. `Templet` – метод процессно-ориентированного моделирования параллелизма // *Программные продукты и системы*, 2012. №3. С. 11–14.
4. Ward M. P. Language-oriented programming // *Software-Concepts and Tools*, 1994. vol. 15, no. 4. pp. 147–161.
5. Dmitriev S. *Language oriented programming: The next programming paradigm*: JetBrains onBoard, 2004. 13 pp., http://www.onboard.jetbrains.com/articles/04/10/lop/
6. Hoare C. A. R. Communicating sequential processes / *The origin of concurrent programming*. New York: Springer, 2002. pp. 413–443. doi: 10.1007/978-1-4757-3472-0_16.
7. Wirth N. The programming language Oberon // *Software: Practice and Experience*, 1988. vol. 18, no. 7. pp. 671–690. doi: 10.1002/spe.4380180707.
8. Chandra R., Menon R., Dagum L., Kohr D., Maydan D., McDonald J. *Parallel Programming in OpenMP*. San Francisco: Morgan Kaufmann, 2000. 230+xvi pp.

9. Коновалов Н. А., Крюков В. А., Михайлов С. Н., Погребцов А. А. Fortran-DVM — язык разработки мобильных параллельных программ // *Программирование*, 1995. № 1. С. 49–54.
10. 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. Т. 441, № 6. С. 734–736.
11. Blumofe R. D., Joerg C. F., Kuszmaul B. C., Leiserson C. E., Randall K. H., Zhou Y. Cilk: An efficient multithreaded runtime system // *Journal of parallel and distributed computing*, 1996. vol. 37, no. 1. pp. 55–69. doi: 10.1006/jpdc.1996.0107.
12. Moskovsky A., Roganov V., Abramov S. Parallelism Granules Aggregation with the T-System / *Parallel Computing Technologies* / Lecture Notes in Computer Science, 4671. Berlin, Heidelberg: Springer, 2007. pp. 293–302. doi: 10.1007/978-3-540-73940-1_30.
13. Moskovsky A., Roganov V., Abramov S., Kuznetsov A. Variable Reassignment in the T++ Parallel Programming Language / *Parallel Computing Technologies* / Lecture Notes in Computer Science, 4671. Berlin, Heidelberg: Springer, 2007. pp. 579–588. doi: 10.1007/978-3-540-73940-1_58.
14. Seindal R. *GNU m4 (version 1.4)*: Technical report: Free Software Foundation, 1997.
15. Cook S., Jones G., Kent S., Wills A. K. *Domain-Specific Development with Visual Studio DSL Tools*. Boston: Pearson Education, 2007. 576 pp.
16. Hewitt C. *Actor Model of Computation: Scalable Robust Information Systems*, 2010. 48 pp., arXiv: 1008.1459 [cs.PL].
17. Hewitt C. Viewing control structures as patterns of passing messages // *Artificial Intelligence*, 1977. vol. 8, no. 3. pp. 323–364. doi: 10.1016/0004-3702(77)90033-9.
18. Athas W. C., Boden N. J. Cantor: an actor programming system for scientific computing // *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 1989. vol. 24, no. 4. pp. 66–68. doi: 10.1145/67386.67402.
19. Larson J. Erlang for concurrent programming // *Communications of the ACM*, 2009. vol. 52, no. 3. pp. 48–56. doi: 10.1145/1467247.1467263.
20. Selic B. The pragmatics of model-driven development // *IEEE Software*, 2003. vol. 20, no. 5. pp. 19–25. doi: 10.1109/MS.2003.1231146.
21. Atkinson C., Kühne T. Model-driven development: A metamodeling foundation // *IEEE Software*, 2003. vol. 20, no. 5. pp. 36–41. doi: 10.1109/MS.2003.1231149.
22. Hailpern B., Tarr P. Model-driven development: The good, the bad, and the ugly // *IBM Systems Journal*, 2006. vol. 45, no. 3. pp. 451–461. doi: 10.1147/sj.453.0451.
23. Mukerji J., Miller J. *Overview and guide to OMG's architecture*: IBM Whitepaper, 2001. 62 pp.
24. Hazzard K., Bock J. *Metaprogramming in .NET*. Shelter Island: Manning Publ., 2013. 334+xxiv pp.

Поступила в редакцию 18/VII/2014;
в окончательном варианте — 21/VIII/2014;
принята в печать — 10/IX/2014.

Vestn. Samar. Gos. Techn. Un-ta. Ser. Fiz.-mat. nauki. 2014. Issue 3 (36). Pp. 169–182
[J. Samara State Tech. Univ., Ser. Phys. & Math. Sci. 2014. Issue 3 (36). Pp. 169–182]

ISSN: 2310-7081 (online), 1991-8615 (print) **doi:** http://dx.doi.org/10.14498/vsgtu1334

MSC: 68N15, 68N19

THE TEMPLET LANGUAGE PREPROCESSOR: A PROGRAMMING TOOL FOR PROCESS-PER-MESSAGE MODELING*

S. V. Vostokin

S. P. Korolyov Samara State Aerospace University (National Research University),
34, Moskovskoe sh., Samara, 443086, Russian Federation.

Abstract

Motivation: A large number of applications can be described as a set of processes that exchange messages. Traditionally the process-per-message model is used in the form of a specialized language or a run-time library for general purpose language. The first approach lacks implementation simplicity, while the second approach is difficult in use. We propose a new method that comprises domain-specific language called Templet for code markup, a general purpose language, and the preprocessor. Our approach is free from disadvantages mentioned above. **Method:** A code of a program is divided into blocks. Block boundaries are indicated by comments. The entire code structure is defined in Templet language so it can be checked out automatically before compilation. **Description of Channels:** A channel defines a message exchange protocol between two interconnected processes. We provide channel syntax in the form of Extended Backus-Naur Formalism (EBNF). The informational structure of the channel is described with Entity-Relation diagram (ER). **Description of Processes:** A process defines the algorithm for message processing. Information structure of the process is shown in conjunction with the syntax. EBNF and ER models are also used in the process specification. Syntax rules are illustrated with the fork-join code sample. **Preprocessor structure and work scheme:** We present the algorithm and the structure of the preprocessor. Subsystems discussed are: syntax analyzer; semantics analyzer; internal database; inference mechanism; and code generator. The method for estimation of workload of manual coding is presented. It shows the diminishment of workload in 20 times comparing with manual coding. **Discussion:** The preprocessor is used for skeleton programming as a part of web-service for automated parallel programming. Its advantages and features are discussed in comparison with parallelization with markup technique; general-purpose macro processor; parallel programming language, metaprogramming; and model-driven development.

Keywords: preprocessor, domain-specific language, process, message, parallel programming.

How to cite Reference: Vostokin S. V. The Templet Language Preprocessor: A Programming Tool for Process-per-Message Modeling, *Vestn. Samar. Gos. Tekhn. Univ., Ser. Fiz.-Mat. Nauki* [J. Samara State Tech. Univ., Ser. Phys. & Math. Sci.], 2014, no. 3 (36), pp. 169–182. doi: 10.14498/vsgtu1334. (In Russian)

Author Details: *Sergey V. Vostokin* (Dr. Techn. Sci.; easts@mail.ru), Professor, Dept. of Information Systems and Technology.

*This paper is an extended version of a PIT 2014 paper [1].

doi: http://dx.doi.org/10.14498/vsgtu1334

Acknowledgments. My thanks go to Samara State Aerospace University (SSAU) for its continuous support to this research. The preprocessor run-time library was tested in the university supercomputer "Sergey Korolyov" with ongoing assistance of maintenance team.

This work was supported by the Ministry of Education and Science of the Russian Federation in the framework of the implementation of the Program of increasing the competitiveness of SSAU among the world's leading scientific and educational centers over the period from 2013 till 2020.

ORCID
Sergey Vostokin: http://orcid.org/0000-0001-8106-6893

REFERENCES

1. Vostokin S. V. The basic syntax of `Templet` markup language for representing process-per-message model, *Perspektivnye informatsionnye tekhnologii (PIT 2014)* [Advanced information technologies (PIT 2014)], Proceedings of the Technical Conference; ed. S. A. Prokhorov. Samara, Samara Research Center, 2014, pp. 317–323 (In Russian), http://templet.ssau.ru/wiki/lib/exe/fetch.php?media=pit2014:templetlang.pdf.
2. Artamonov Yu. S., Vostokin S. V., Nazarov Yu. P. Templet – A service of continuous integration to the development of high-performance applications, *Vysokoproizvoditel'nye parallel'nye vychisleniia na klasternykh sistemakh* [High-performance parallel computations on cluster systems], Proceedings of XII All-Russian Conference. Nizhni Novgorod, Nizhni Novgorod State Univ. Publ., 2012, pp. 82 (In Russian).
3. Vostokin S. V. Templet – A method of process oriented simulation of parallelism, *Programmnye produkty i sistemy*, 2012, no. 3, pp. 11–14 (In Russian).
4. Ward M. P. Language-oriented programming, *Software-Concepts and Tools*, 1994, vol. 15, no. 4, pp. 147–161.
5. Dmitriev S. *Language oriented programming: The next programming paradigm*, JetBrains onBoard, 2004, 13 pp., http://www.onboard.jetbrains.com/articles/04/10/lop/
6. Hoare C. A. R. Communicating sequential processes, *The origin of concurrent programming*. New York, Springer, 2002, pp. 413–443. doi: 10.1007/978-1-4757-3472-0_16.
7. Wirth N. The programming language Oberon, *Software: Practice and Experience*, 1988, vol. 18, no. 7, pp. 671–690. doi: 10.1002/spe.4380180707.
8. Chandra R., Menon R., Dagum L., Kohr D., Maydan D., McDonald J. *Parallel Programming in OpenMP*. San Francisco, Morgan Kaufmann, 2000, 230+xvi pp.
9. Konovalov N. A., Kryukov V. A., Mikhailov S. N., Pogrebtsov A. A. FORTRAN-DVM: The Language for Developing Portable Parallel Programs, *Programmirovanie*, 1995, no. 1, pp. 49–54 (In Russian).
10. Bakhtin V. A., Kryukov V. A., Chetverushkin B. N., Shilnikov E. V. Extension of the DVM parallel programming model for clusters with heterogeneous nodes, *Doklady Mathematics*, 2011, vol. 84, no. 3, pp. 879–881. doi: 10.1134/S1064562411060408.
11. Blumofe R. D., Joerg C. F., Kuszmaul B. C., Leiserson C. E., Randall K. H., Zhou Y. Cilk: An efficient multithreaded runtime system, *Journal of parallel and distributed computing*, 1996, vol. 37, no. 1, pp. 55–69. doi: 10.1006/jpdc.1996.0107.
12. Moskovsky A., Roganov V., Abramov S. Parallelism Granules Aggregation with the T-System, *Parallel Computing Technologies*, Lecture Notes in Computer Science, 4671. Berlin, Heidelberg, Springer, 2007, pp. 293–302. doi: 10.1007/978-3-540-73940-1_30.
13. Moskovsky A., Roganov V., Abramov S., Kuznetsov A. Variable Reassignment in the T++ Parallel Programming Language, *Parallel Computing Technologies*, Lecture Notes in Computer Science, 4671. Berlin, Heidelberg, Springer, 2007, pp. 579–588. doi: 10.1007/978-3-540-73940-1_58.
14. Seindal R. *GNU m4 (version 1.4)*, Technical report, Free Software Foundation, 1997.

15. Cook S., Jones G., Kent S., Wills A. K. *Domain-Specific Development with Visual Studio DSL Tools*. Boston, Pearson Education, 2007, 576 pp.
16. Hewitt C. *Actor Model of Computation: Scalable Robust Information Systems*, 2010, 48 pp., arXiv: 1008.1459 [cs.PL].
17. Hewitt C. Viewing control structures as patterns of passing messages, *Artificial Intelligence*, 1977, vol. 8, no. 3, pp. 323–364. doi: 10.1016/0004-3702(77)90033-9.
18. Athas W. C., Boden N. J. Cantor: an actor programming system for scientific computing, *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 1989, vol. 24, no. 4, pp. 66–68. doi: 10.1145/67386.67402.
19. Larson J. Erlang for concurrent programming, *Communications of the ACM*, 2009, vol. 52, no. 3, pp. 48–56. doi: 10.1145/1467247.1467263.
20. Selic B. The pragmatics of model-driven development, *IEEE Software*, 2003, vol. 20, no. 5, pp. 19–25. doi: 10.1109/MS.2003.1231146.
21. Atkinson C., Kühne T. Model-driven development: A metamodeling foundation, *IEEE Software*, 2003, vol. 20, no. 5, pp. 36–41. doi: 10.1109/MS.2003.1231149.
22. Hailpern B., Tarr P. Model-driven development: The good, the bad, and the ugly, *IBM Systems Journal*, 2006, vol. 45, no. 3, pp. 451–461. doi: 10.1147/sj.453.0451.
23. Mukerji J., Miller J. *Overview and guide to OMG's architecture*, IBM Whitepaper, 2001, 62 pp.
24. Hazzard K., Bock J. *Metaprogramming in .NET*. Shelter Island, Manning Publ., 2013, 334+xxiv pp.

Received 18/VII/2014;
received in revised form 21/VIII/2014;
accepted 10/IX/2014.