Система параллельных вычислений Templet: спецификация, реализация, применение

С.В. Востокин

Самарский национальный исследовательский университет имени академика С.П.Королёва, 443086, Московское шоссе, 34, Самара, Россия

Аннотация

В статье описывается реализованный прототип системы параллельных вычислений Templet для языка программирования C++. В системе используется новая версия акторной модели выполнения. Дизайн данной акторной модели позволяет определить поведение разрабатываемой параллельной программы математически точно на основе формул темпоральной логики. Это свойство важно, так как даёт разработчику свободу реализации акторных вычислений на произвольной платформе. Представленный вариант акторной модели может быть легко реализован на различном аппаратном обеспечении и на различных языках программирования для вычислений на системах с разделяемой памятью. В статье даётся определение модели акторов Templet с использованием логики TLA, обсуждается дизайн системы программирования и описывается несколько примеров практического использования представленной системы параллельных вычислений.

Ключевые слова: модель акторов; модель выполнения; темпоральная логика; система параллельных вычислений; платформонезависимость

1. Введение

Система Templet – это инструментарий параллельного программирования в терминах модели акторов. В системе имеется компактная библиотека поддержки времени выполнения, которая реализует акторную семантику. Для удобства практического использования этой библиотеки был разработан предметный язык Templet. Препроцессор языка Templet преобразует высокоуровневую спецификацию модели акторов, включая протокол обмена сообщениями, правила соединения акторов и так далее, в код на языке C++. Этот сгенерированный код имеет вид «скелета» программы: некоторого каркаса с точками расширения куда пользователь должен поместить код на языке C++, чтобы получить окончательный вариант программы. Описываемая система программирования включает препроцессор и несколько библиотек времени выполнения, которые можно найти на сайте проекта GitHub по адресу *https://github.com/templet-language.*

Статья имеет следующую структуру. Вначале строится формальная спецификация варианта акторной модели программирования для системы Templet. Для спецификации мы используем темпоральную логику, предложенную Лесли Лампортом [1]. Далее описываем библиотеку времени выполнения, которая реализует данную спецификацию. Затем обсуждается привязка пользовательского кода к данной библиотеке. В завершении статьи обсуждаются примеры разработанных программ и возможности практического использования описываемой системы.

2. Предмет исследования

Имеется большое число систем программирования с поддержкой акторов, которые выглядят более или менее одинаково. Нашей целью исследования является демонстрация того, что: (1) практическая акторная система может быть крайне простой; (2) система не требует специальной поддержки со стороны языка реализации или библиотеки времени выполнения; (3) модель акторов может рассматриваться как простое расширение стандартной многопоточности, а поддержка многопоточности – это обычное свойство практически любой системы программирования.

Имея заявленные преимущества, наш вариант реализации акторов также сохраняет традиционные предпочтительные характеристики модели акторов: отсутствие состояния состязания, отсутствие состояния тупиков, взгляд на параллельную систему как на композицию последовательных систем. Эти особенности делают акторы более удобными, чем потоки для программирования приложений.

3. Методы исследования

Метод, используемый для достижения заявленных целей проектирования, состоит в (а) подборе минимального ядра акторной модели; (б) предложении механизма связывания данного ядра и пользовательского кода, который бы не полагался на особенности языка реализации. Например, такой механизм не должен использовать механизм шаблонов C++ или другие расширенные свойства языка C++.

Первая часть метода достигается на основе математической спецификации варианта акторной модели с использованием темпоральной логики действий – универсальной модели параллельных вычислений, предложенной Лесли Лампортом, что позволяет достичь минимизации ядра акторной модели.

Вторая часть метода заключается в использовании дополнительного предметно-ориентированного языка Templet. Цель применения данного языка – подготовка детализированного описания поведения акторов (типов принимаемых ими сообщений, односторонних коммуникаций, коммуникаций типа запрос-ответ, сложных структур из акторов и так

далее) в начале разработки программы и, затем, автоматическое генерирование C++ кода скелета программы по данному описанию.

Существует большое число реализаций акторной модели, предложенных после первых работ Карла Хьювитта и его коллег [2]. Обсуждаемый в статье подход отличается от следующих реализаций: (а) традиционных библиотек (Akka [3], Intel TBB [4]); (б) полностью новых языков с акторной семантикой (Scala [5], Go [6], Erlang [7]); (в) расширений известных языков (ACT++ [8], Axum [9]).

Мы предлагаем языково-ориентированный [10], основанный на использовании предметного языка инструментарий акторного программирования, который является разновидностью систем скелетного программирования [11]. Поэтому, система естественным образом совместима с существующими компиляторами, библиотеками, интегрированными средами разработки, что значительно упростило её разработку и использование. Наша система имеет много общих черт с подходом «разработка, управляемая моделями» [12].

Было выполнено много исследований с целью адекватной формализации модели акторов [13]. Однако, подход, использующий темпоральную логику действий для представления акторной семантики при помощи переменных и изменения их значений, является новым и перспективным. В работе делается попытка построения спецификации только для варианта модели акторов, используемого в системе Templet. Мы не претендуем на полную формализацию модели акторов.

4. Спецификация акторной модели

Система Templet проектировалась для программ с акторной семантикой выполнения. Используемый метод реализации акторной модели фокусируется на передаче прав доступа к объектам и на передаче активности между акторами, а не на классическом обмене сообщениями. Этот подход позволил нам построить точную спецификацию и компактную реализацию соответствующей библиотеки времени выполнения.

Пусть *Var* – это множество всех переменных программы. Допустим, что имеется функция *F*, которая определяет принадлежность переменных программы некоторому актору или сообщению

$$F : Var \to \{actor, message\} \times N, \tag{1}$$

где число из множества *N* обозначает идентификатор актора или сообщения. В конкретной реализации, а именно, в реализации на языке C++ это соответствие может быть выражено при помощи объявления переменной в классе актора или классе сообщения, а может просто подразумеваться программистом.

Пусть множество

$$\{a[1], a[2], \ldots, a[i], \ldots, b[1], b[2], \ldots, b[j], \ldots, m[1], m[2], \ldots, m[j], \ldots\} \tag{2}$$

обозначает все переменные среды времени выполнения и включает следующие компоненты: (а) активность актора *i* обозначается как $a[i] \in \{0,1\}$, где булево значение 0 обозначает «не активен», а булево значение 1 обозначает, что актор «активен» и обрабатывает сообщение; (б) активность сообщения обозначено как $m[j] \in \{0,1\}$, где булево значение 0 говорит о том, что сообщение поступило в актор, а булево значение 1 означает, что сообщение находится на доставке; (в) в массиве переменных *b*[.] хранятся идентификаторы акторов, которым принадлежат сообщения *j*. Сообщение принадлежит актору, если оно в данный момент доставляется в этот актор или уже доставлено ему.

С использованием введённых обозначений мы можем записать атомарные операции, которые изменяют состояние системы во время вычислений.

$$A_1 \equiv \exists! j : \neg a[i] \wedge m[j] \wedge b[j] = i \wedge a'[i] \wedge \neg m'[j] \wedge b'[j] = i \tag{3}$$

Формула (3) означает, что (а) когда поступает сообщение, оно активирует процедуру обработки; (б) только одно сообщение может обрабатываться актором за один раз[1].

$$A_2 \equiv a[i] \wedge \neg a'[i] \tag{4}$$

Действие, обозначенное формулой (4) исполняется в конце обработки сообщения.

$$A_3 \equiv \exists i : a[i] \wedge b[j] = i \wedge \neg m[j] \wedge m'[j] \tag{5}$$

Формула (5) обозначает, что любое сообщение, которое доступно во время процедуры обработки сообщения, может быть отправлено в любой актор.

[1] Для передачи изменения состояния системы мы используем переменные со штрихами в логических формулах (*var'*). Они обозначают значения переменных в следующем состоянии ($t + 1$), в то время как переменные без штрихов(*var*) обозначают значения в текущем состоянии системы (t). Заметим, что переменные *i* и *j* являются темпоральными константами. Их значения неизвестны, но неизменны во времени и обозначают некоторый актор или сообщение.

Формула (6)

$$A_4 \equiv m[j] \wedge \neg m'[j] \tag{6}$$

обозначает приход сообщения в актор.

Свойство «живучести» (liveness) в описываемой среде выполнения предполагает, что если действия A_2 (4) и A_4 (6) готовы исполняться в течение достаточно долгого времени, то они в конечном счете выполнятся: сообщение будет доставлено, а актор завершит обработку сообщения. Говорят, что формулы (2) и (6) исполняются в условиях слабой справедливости *WF* (weak fairness). Комбинируя формулы (3)-(6) при помощи темпоральных операторов и предполагая начальное состояние системы равным $I \equiv \exists i : a[i] \vee \exists j : m[j]$, состояние актора обозначенным как $f_1 \equiv a[i]$, а состояние сообщение обозначенным как $f_2 \equiv (m[j], b[j])$, мы окончательно получаем формулу

$$S \equiv I \wedge [A_1 \vee A_2]_{f_1} \wedge [A_3 \vee A_4]_{f_2} \wedge WF_{f_1}(A_2) \wedge WF_{f_2}(A_4), \tag{7}$$

которая завершает спецификацию поведения нашей среды времени выполнения[2].

5. Реализация библиотеки времени выполнения

Спецификация (7) используется для определения примитивных операций библиотеки времени выполнения. Библиотека имеет три примитивных операции. Первая операция *recv*() является процедурой обратного вызова. Она запускается в контексте актора *i* для обработки поступившего сообщения *j*:

$$recv_{(call)}(i, j) \equiv \neg a[i] \wedge m[j] \wedge b[j] = i \wedge a'[i] \wedge \neg m'[j] \wedge b'[j] = i, \tag{8}$$

$$recv_{(return)}(i) \equiv a[i] \wedge \neg a'[i]. \tag{9}$$

Действие (8) выполняется, когда процедура *recv*() вызывается из библиотеки времени выполнения. Действие (9) выполняется при возврате из процедуры *recv*().

Вторая примитивная операция *access*() – это логическая функция. Она вызывается программистом из процедуры *recv*(). Функция проверяет, являются ли переменные, ассоциированные с сообщением *j*, доступными из процедуры обработки сообщений актора *i*.

$$access(i, j) \equiv b[j] = i \wedge \neg m[j] \tag{10}$$

Наконец, *send*() – это процедура отправки сообщения. Эта процедура также вызывается программистом из процедуры *recv*(). Она отправляет сообщение *j* актору *i*:

$$send(i, j) \equiv b'[j] = i \wedge m'[j]. \tag{11}$$

Следующий фрагмент кода на языке C++ является частью библиотеки времени выполнения.

```
// engine
struct engine{ std::vector<message*> ready;};
// actor objects
struct actor{ void(*recv)(actor*,message*);};
// message objects
struct message{ actor*a; bool sending;};

inline void send(engine*e, actor*a, message*m){
      if (m->sending) return;
      m->sending = true;
      m->a = a;
      e->ready.push_back(m);
}

inline bool access(actor*a, message*m){
        return m->a == a && !m->sending;
}
inline void run(engine*e){
        size_t rsize;
        while (rsize = e->ready.size()){
                int n = rand() % rsize;
                auto it = e->ready.begin() + n;
                message*m = *it; e->ready.erase(it);
```

[2] Запись $\Box[A]_f$ означает, что для каждой пары состояний системы выражение $A \vee (f = f')$ – истинно. Это говорит о том, что в любой момент времени состояние системы изменяется под действием *A* или остаётся неизменным.

```
                m->sending = false;
                m->a->recv(m->a,m);
        }
}
```

Показанный код библиотеки времени выполнения используется для моделирования конкурентного выполнения на одном процессоре. Вызов функции *rand*() библиотеки языка C++ используется для рандомизации. Показанная библиотека – это отладочная реализация спецификации (7). Библиотечный код расположен в файле *cpp11runtime/lib/dbg/tet.h*. Также в репозитории *cpp11runtime* проекта Templet на сервисе GitHub имеется поддержка эффективного последовательного выполнения *(.../seq/tet.h*), параллельного многопоточного выполнения *(.../par/tet.h*) и предсказания производительности методом дискретно-событийного моделирования *(.../sim/tet.h*). Эти среды выполнения также являются реализациями модели (7).

6. Привязка кода приложения к библиотеке

Непосредственное использование такой упрощенной библиотеки времени выполнения может оказаться затруднительным по следующим соображениям. Отношение, задаваемое формулой (1) между переменной и её управляющим объектом (актором или сообщением), желательно обозначить синтаксически. Каждая операция чтения/записи с переменными сообщений должна быть предварена проверочным вызовом примитива *access*(). Код должен представлять сущности приложения, а не акторной модели. Для решения этих проблем разработан препроцессор системы Templet. Препроцессор выполняет синтез необходимого дополнительного кода.

В текущей реализации препроцессора используются специальные комментарии для разделения трёх частей кода: команд предметно-ориентированного языка, рукописного кода, автоматически генерированного кода. Команды предметного языка используются для определения свойств акторов и сообщений. Акторы и сообщения являются сущностями соответствующих классов C++.

Например, команда **actor.* на предметно-ориентированном языке внутри комментариев

```
/*$TET$templet$!templet!*/
/* *actor. */
/*$TET$*/
```

обозначает, что препроцессор должен генерировать C++ класс *actor* с поведением актора. Фрагмент этого класса показан ниже.

```
class actor:public TEMPLET::actor{
/*$TET$actor$!userdata!*/
       double some_user_defined_variable;
/*$TET$*/
};
```

Генерированный фрагмент имеет точку расширения, куда пользователь может добавить рукописный код. Точка расширения в примере, приведённом выше, размещается между метками */*TETactor$!userdata!*/* и */*$TET$*/*. Автоматически генерированный код также содержит вызовы библиотечных функций *access*() и *send*() внутри перегруженного метода актора *recv*(). Здесь этот код пропущен по причине его сложности. Программист может изменять команды предметного языка и код внутри точек расширения. Препроцессор при каждом изменении команд предметного языка повторно синтезирует код, сохраняя изоморфизм между предметной спецификацией и генерированным C++ кодом.

7. Пример синтаксиса языка Templet

Более реалистичный пример описания акторов и сообщений на предметном языке Templet из папки *cpp11runtime/samples/templet* показан ниже.

```
~Link=
       + Begin ? argSin2 -> Calc | argCos2 -> Calc;
         Calc ! result -> End;
End.


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

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

Данный предметный код определяет программу, которая проверяет корректность основного тригонометрического тождества $\sin^2 x + \cos^2 x = 1$. Генерируемая программа вычисляет выражения $\sin^2 x$ и $\cos^2 x$ одновременно в двух дочерних

акторах типа *Child*, в то время как родительский актор *Parent* их координирует. Класс *Link* управляет порядком передачи сообщений (*argSin2*, *argCos2*, *result*).

После определения предметного кода акторной модели программист должен закодировать точки расширения на языке C++. Фрагмент на предметном языке, показанный выше, подразумевает следующие точки расширения. Определение структур сообщений *argSin2*, *argCos2*, *result*. Определение методов *fork*, *join*, calcCos2 и *calcSin2* классов акторов на C++. Возможны объявления других членов C++ классов *Parent* и *Child*. Наконец, программист создаёт акторы в виде экземпляров классов *Parent* и *Child* и соединяет акторы при помощи экземпляров классов сообщений *Link*. Этот код является простым и для написания не требует знаний в области параллельного программирования. Объяснение синтаксиса предметного языка можно найти в работе [14].

8. Результаты и их применение

Мы видим практические преимущества нашего акторного подхода в областях с высоким «естественным» параллелизмом: при программировании систем с большим числом вычислительных ядер; для организации высокопроизводительных вычислений; программировании систем управления техническими объектами; в приложениях интернета вещей; управлении бизнес процессами и других областях. Доступные примеры использования описываемой системы на сайте GitHub – из области высокопроизводительных вычислений на кластерных системах и управления бизнесс-процессами.

Пакет cpp11runtime предлагает три примера, иллюстрирующих практическое применение системы Templet. Метод Гаусса-Зейделя для решения уравнения Лапласа расположен в каталоге *samples/pipeline*. Этот пример иллюстрирует применение системы в области научных вычислений. Он также показывает, как имитационная среда помогает предсказать производительность без построения явной математической модели или реального параллельного выполнения.

Пример из области линейной алгебры размещается в каталоге *samples/ringmult*. Это иллюстрация алгоритма распределенного умножения матриц. Предложенная реализация акторной модели хорошо приспособлена для вычислений в архитектурах с разделяемой и распределённой памятью.

Пример моделирования бизнес-процесса размещается в каталоге *samples/order*. Система Templet может использоваться для моделирования и анализа параллелизма в различных системах, например, в области моделирования бизнес-процессов. Изучен написанный на разговорном языке бизнес-сценарий, построена спецификация этого сценария на предметном языке системы Templet. Статический анализ типов C++, отладка и тестирование программы были использованы для проверки корректности полученной спецификации. В частности, мы сравнили программно генерированные последовательности событий с ожидаемыми последовательностями событий для изучаемого бизнес-процесса. Этот пример иллюстрирует возможность удобного тестирования и отладки параллельных программ в системе Templet.

9. Заключение

Проведённое исследование показало, что наш подход к акторному программированию, основанный на построении математической модели вычислений, имеет большое практическое значение. Мы получили полностью работоспособную, притом относительно простую, реализацию системы вычислений для параллельного программирования в разделяемой памяти. В будущем планируем расширить систему для распределённого программирования и упростить синтаксис предметного языка. Система развёрнута в сети интернет по адресу http://templet.ssau.ru/app как компонент сервиса научных вычислений.

Благодарности

Работа выполнена при государственной поддержке Министерства образования и науки РФ в рамках реализации мероприятий Программы повышения конкурентоспособности Самарского национального исследовательского университета имени академика С.П. Королева среди ведущих мировых научно-образовательных центров на 2013-2020 годы. Работа частично поддержана грантом РФФИ № 15-08-05934 А.

Литература

[1] Lamport, L. The Temporal Logic of Actions / L. Lamport // ACM Transactions on Programming Languages and Systems. – 1994. – Vol. 16 (3). – P. 872–923.

[2] Hewitt, C. A universal modular actor formalism for artificial intelligence/ C. Hewitt, P. Bishop, R. Steiger // Proceedings of the 3rd international joint conference on Artificial intelligence. – 1973. – P. 235–245.

[3] Allen, J. Effective akka / J. Allen – O'Reilly Media, Inc., 2013.

[4] Wooyoung, K. Multicore desktop programming with Intel threading building blocks / K. Wooyoung, M. Voss // IEEE software. – 2011. Vol. 28 (1). – P 23-31.

[5] Haller, Ph. Actors in Scala / Ph. Haller, F. Sommers – Artima Incorporation, 2012.

[6] Pike, Rob. The Go Programming Language. Talk given at Google Tech Talks Oct.30, 2009. [Electronic resource]. – Access mode:https://www.youtube.com/watch?v=rKnDgT73v8s.

[7] Cesarini, F. Erlang programming / F. Cesarini, S.Thompson – O'Reilly Media, Inc., 2009.
[8] Kafura, D.G. Act++ 2.0: a Class Library for Concurrent Programming in C++ Using Actors / D.G. Kafura, M. Mukherji, G.R. Lavender – Technical Report. Virginia Polytechnic Institute & State University, Blacksburg, VA, USA, 1992.
[9] Gustafsson, N. Axum. A .NET Language for Safe and Scalable Concurrency / N. Gustafsson – Microsoft PDC 9, 2009.
[10] Ward, M.P. Language-oriented programming / M.P. Ward // Software-Concepts and toolkits. – 1994. – Vol. 15(4). – P.147–161.
[11] Gonzalez-Velez, H. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers / H. Gonzalez-Velez, M. Leyton //Software: Practice and Experience. – 2010. – Vol. 40(12). – P.1135–1160.
[12] Atkinson, C. Model-driven development: a metamodeling foundation / C. Atkinson, T. Kuhne // Software, IEEE. – 2003. – Vol. 20(5). – P. 36–41.
[13] Varela, C.A. Programming Distributed Computing Systems: A Foundational Approach / C.A.Varela, Gul Agha – MIT Press, 2013.
[14] Vostokin, S.V. Templet: a markup language for concurrent actor-oriented programming/ S.V. Vostokin // CEUR Workshop Proceedings. – 2016. Vol. 1638. – P. 460-468.