Gem #89: Архетипы кода программирования в реальном времени - Часть 1

Автор: Marco Panunzio, University of Padua

Краткое содержание: Gem Ada #89 - В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определённым образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких архетипов, которые значительно облегчают разработку систем реального времени.

Введение.

В этой серии Ada Gems мы предлагаем ряд архетипов кода для разработки систем реального времени. Архетипы кода соответствуют ограничениям Профиля Ravenscar, подмножеству языка Ada, которому в частности удовлетворяют разработки систем реального времени высокой целостности. Профиль Ravenscar был создан, чтобы гарантировать, что программы, записанные в соответствии с ним, поддаются статическому анализу в части измерения времени выполнения. На самом деле профиль исключает все конструкции Ada, которые представлены недетерминированному или неограниченному времени выполнения. В космической отрасли производства программного обеспечения, профиль запрещает использование конструкций, которые неявно выполняют динамическое выделение памяти.

В качестве дополнительного преимущества, системы Ravenscar могут быть реализованы поверх ядер реального времени, которые могут быть очень маленькими по размеру и быстрыми по времени. Ядра реального времени являются привлекательными своими характеристиками для приложений, которые могут позволить себе только небольшие накладные расходы и должны пройти обширную квалификацию/сертификацию.

Существенные элементы ограничений Ravenscar:

(i) статическая модель существования. Система состоит из фиксированного набора задач и защищённых объектов, определённых на уровне библиотеки и со статически присвоенным приоритетом или приоритетом потолка. Никакая задача не завершается, и операторы преждевременного завершения отвергнуты.

(ii) коммуникационная модель. Задачам только позволяют связаться асинхронно через защищённые объекты. Рандеву задачи поэтому отвергнуто.

(iii) детерминированная модель выполнения. Профиль исключает все конструкции, которые представляют недетерминизм:

   a. относительные задержки, поскольку они представляют недетерминизм во время приостановки задач; операторы перерасчёта очереди;

   b. использование Ada.Calendar, поскольку все связанные со временем операции полагаются на высокую точность пакета Ada.Real_Time;

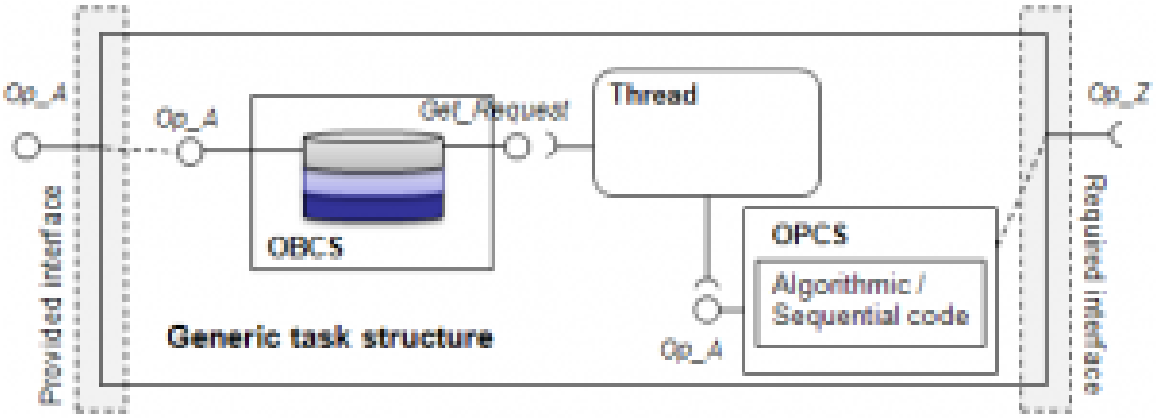
   c. защищённые объекты ограничены наличием самое большее одной записи, на которой только единственная задача может ставить в очередь;

   d. защита записей состоит из простого булева условия, чтобы избежать побочных эффектов и недетерминизма во время оценки.

В этой серии Ada Gems мы иллюстрируем ряд архетипов кода, которые реализуют общие шаблоны программирования, подходящие для разработки систем реального времени, совместимых с Ravenscar и пригодных для разработки Ravenscar-совместимых систем реального времени.

Использование архетипов разрешает факторизацию и таким образом помогает уменьшать размер кода, который реализует параллельные элементы системы.

Важная дополнительная цель этих архетипов состоит в том, чтобы достигнуть полного разделения между алгоритмическим/последовательным кодом системы и кодом, который управляет параллелизмом и аспектами в реальном времени. Это разделение проблем позволяет разрабатывать алгоритмическое содержание системы (т.е. поведение системы) независимо от управления параллельными задачами и проблемами работы задач в реальном времени.



Эта цель достигается путём инкапсуляции последовательного кода в подходящую структуру задачи. На рисунке выше показана общая структура архетипов нашей задачи.

Последовательный код заключён в структуру, которую мы называем структурой оперативного управления (OPCS). Код выполняется потоком, который представляет собой отдельный поток управления системой. Структура задачи может быть дополнительно оснащена структурой управления объектами (OBCS). OBCS представляет собой агент синхронизации для задачи: как мы увидим, мы используем его в основном для спорадических задач. OBCS состоит из защищённого объекта, котором хранятся входящие запросы на службы, выполняемые потоком. Поскольку несколько клиентов могут независимо требовать, чтобы службы выполнялись этим потоком, операции защищены. Отправляются запросы выполнения в OBCS. При каждом получении управления, Thread извлекает один из этих запросов (порядок FIFO по умолчанию), а затем выполняет последовательный код, хранящийся в OPCS, который соответствует запросу.

Как мы проиллюстрируем в третьем Gem этой серии, операции, предоставляемые OBCS, которые образуют предоставленный интерфейс общей сущности, соответствуют сигнатуре последовательных операций OPCS (Op_A на рисунке выше). Благодаря этому вызывающим абонентам не нужно знать, что они фактически отправляют запросы на выполнение в OBCS, тогда как фактическое выполнение будет выполняться Thread.

Последовательный код, встроенный в OPCS, может потребоваться вызывать службы от других программных объектов (операция Op_Z на рисунке выше). В пятом Gem этой серии мы опишем, как эти функциональные потребности могут быть выполнены.

Как вывод, наши сущности инкапсулируют свою внутреннюю структуру и раскрывают внешнему миру только интерфейс, который соответствует сигнатуре операций, встроенных в OPCS. Различные проблемы, с которыми сталкивается каждый такой объект, отдельно распределяются между его внутренними составляющими: последовательное поведение обрабатывается OPCS; задачи и выполнение задач Thread; взаимодействие с параллельными клиентами и обработка запросов на выполнение обрабатываются OBCS.

Структура этого Мини-сериала Ada Gems

1. Введение и циклическая задача
2. Простая Спорадическая Задача

3. Спорадические задачи-Системные типы и типы задач
4. Спорадическая задача-последовательный код и OBCS
5. Межзадачной Связи

Выражение признательности.

Структура задачи, которую мы принимаем, является эволюцией методологии проектирования HRT-HOOD [1], из которой мы также наследуем термины OBCS и OPCS.

Ранняя работа по генерации кода из HRT-HOOD в Ada была описана в [2] и [3].

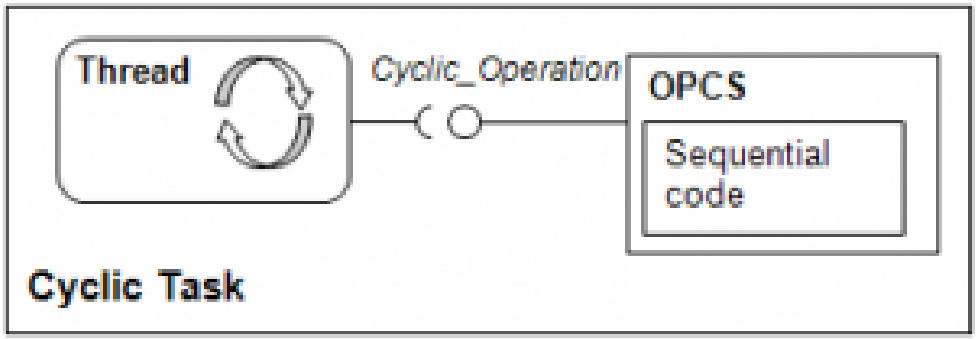
Архетипы кода, которые мы описываем в этом минисериале Ada Gems, использовались для генерации кода в HRT-UML-треке проекта ASSERT, финансируемого EC [4]. В этом проекте Маттео Бордин, затем в Падуанском университете, был главным разработчиком стратегии генерации кода и архетипов кода.

Наконец, мы хотели бы поблагодарить Туллио Варданегу за его предварительный обзор содержания этого минисериала, а Маттео Бордин, теперь с AdaCore, за его обширный обзор содержания и его полезных предложений.

Давайте начнём…

Циклическая Задача

В этом разделе мы проиллюстрируем наш архетип кода для циклических задач. Это позволяет разработчику создать циклическую задачу путём создания экземпляра универсального пакета, передавая операцию, которая должна выполняться периодически.



Архетип довольно прост. Фактически, нам нужно только создать Тип задачи, который циклически выполняет данную операцию с фиксированным периодом. Элемент спецификации архетипа:

```
with System; use System;

generic
   with procedure Cyclic_Operation;
package Cyclic_Task is

   task type Thread_T
      (Thread_Priority : Priority;
       Period : Positive) is
      pragma Priority (Thread_Priority);
   end Thread_T;

end Cyclic_Task;
```

Приведённая выше Спецификация пакета определяет тип задачи для циклического потока. Каждый поток создаётся с статически назначенным приоритетом и периодом, который остаётся фиксированным в течение всего времени существования потока. Тип задачи создаётся внутри универсального пакета, который используется для факторизации архетипа кода и сделать его универсальным для циклической операции. Тело Ada пакета определено следующим образом:

```
with Ada.Real_Time;
with System_Time;

package body Cyclic_Task is

   task body Thread_T is
      use Ada.Real_Time;
      Next_Time : Time := System_Time.System_Start_Time;
   begin
      loop
         delay until Next_Time;
         Cyclic_Operation;
         Next_Time := Next_Time + Milliseconds (Period);
      end loop;
   end Thread_T;

end Cyclic_Task;
```

Тело задачи состоит из бесконечного цикла. Сразу после активации задача входит в цикл и немедленно приостанавливается до общесистемного времени запуска (System_Start_Time). Эта первоначальная приостановка используется для синхронизации всех задач, которые должны выполняться в фазе и позволить им иметь свой первый запуск в то же самое абсолютное время. При возобновлении из приостановки (которая условно совпадает с запуском задачи) задача конкурирует за процессор и выполняет Циклическую операцию, указанную в экземпляре её универсального пакета. Затем он вычисляет следующий раз, когда он должен быть запущен (Next_Time) и в качестве первой инструкции последующего цикла, он выдаёт запрос на абсолютную приостановку до следующего кратного его периода.

Архетип кода прост для понимания, но несколько комментариев в порядке пояснения.

Во-первых, мы должны подчеркнуть, что использование абсолютного времени и, следовательно, конструкции задержки `delay until Next_Time` (в отличие от относительного времени и задержки конструкции) имеет важное значение. В результате мы предотвращаем дрейф фактического времени периодического запуска.

Во-вторых, читатель должен отметить, что `Cyclic_Operation` не имеет параметров. Это не удивительно, так как это согласуется с самой природой циклических операций, которые не запрашиваются явно любым программным клиентом.

Наконец, эта версия циклической задачи предполагает, что все задачи изначально запускаются одновременно (`System_Start_Time`). Поддержка смещения для конкретной задачи (фазы) легко реализовать: нам просто нужно указать дополнительный параметр смещения при создании экземпляра задачи, который затем добавляется в `System_Start_Time`, чтобы определить время первого запуска задачи. Периодическое освобождение задачи затем примет желаемую фазу относительно синхронизированного выпуска задач без смещения. В следующем ADA Gem мы проиллюстрируем простой архетип кода для реализации спорадических задач.

Ссылки на литературу

[1] Alan Burns and Andy J. Wellings: "HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems". Elsevier Science, 1995. ISBN 978-0444821645.

[2] Juan Antonio de la Puente, Alejandro Alonso, and Angel Alvarez: "Mapping HRT-HOOD Designs to Ada 95 Hierarchical Libraries." Reliable Software Technologies - Ada-Europe, Springer Volume LNCS 1088, 1996.

[3] Matteo Bordin and Tullio Vardanega: "Automated Model-Based Generation of Ravenscar-Compliant Source Code". Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2005. ISBN 0-7695-2400-1.

[4] ASSERT project (Automated proof-based System and Software Engineering for Real-Time Systems) http://www.assert-project.net

Связанный со статьёй текст программы

Attached Files отсутствуют

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Сведения об авторе отсутствуют

Last Updated: 10/13/2017
Posted on: 6/22/2010

Обсуждение…

7 ответов на "Gem # 89: Code Archetypes для программирования в реальном времени - часть 1"

1. 23 июня 2010 года в 22:28

Simon Wright (Саймон Райт) сказал:

Несколько провокационных вопросов: Почему спецификация `Cyclic_Task` использует систему? и почему он вообще упоминает `Ada.Real_Time`?

2. 24 июня 2010 года в 2:31 утра

Anh Vo (Ань Во) сказал:

Благодарим вас за GEM. Мне это нравится.

Как выглядит `System_Time`?

3. 24 июня 2010 года в 10:02

Matteo Bordin (Маттео Бордин) сказал:

@Simon

Система необходима для Thread_Priority, которая имеет тип Priority. Ada.Real_Time не требуется, так как он находится внутри тела. Я исправлю код Gem.

@Anh

Примером может служить следующее:

```
with Ada.Real_Time;
package System_Time is
System_Start_Time: constant
Ada.Real_Time.Time: = Ada.Real_Time.Clock;
Task_Activation_Delay: constant
Ada.Real_Time.Time_Span: = Ada.Real_Time.Milliseconds (5_000);
end System_Time;
```

4. 24 июня 2010 года в 14:19

Marco Panunzio (Марко Панунцио) сказал:

Я вижу, что Маттео был быстрее меня. :-)

В примере пакета в комментарии Matteo вы используете `System_Start_Time`, чтобы узнать время активации вашей системы.

`Task_Activation_Delay` определяет установленный период времени после активации системы, при котором все «синхронные» задачи одновременно выпускаются в первый раз (5 секунд в примере).

Используя этот подход, вы можете захотеть использовать в теле архетипа:

```
Next_Time: Time: = System_Time.System_Start_Time +
System_Time.Task_Activation_Delay;
```

Как было предложено в Gem, вы также можете добавить статическое смещение по каждой задаче (w.r.t. синхронная версия), передав требуемое смещение в качестве дополнительного параметра при создании экземпляра родового.

В этом случае вам просто нужно добавить смещение, когда вы определяете момент времени первого выпуска задачи.

```
Next_Time: Time: = System_Time.System_Start_Time +
System_Time.Task_Activation_Delay +
Offset;
```

Это гарантирует, что задача сохраняет свою фазу w.r.t. «синхронные» задачи (с 0 смещением).

5. 24 июня 2010 года в 22:03

Саймон Райт сказал:

@Matteo: Извините; Я получаю «with System», то, что я спрашивал, было «use System». Сказал вам, что это ничто.

6. 26 июня 2010 года в 1:32

Ань Во сказал:

Для повышения эффективности период также должен быть постоянным. Расчет / преобразование выполняется только один раз в этом случае.

```
The_Period:    Ada.Real_Time.Time_Span:    =    Ada.Real_Time.Milliseconds
(Period);
...
...
Next_Time: = Next_Time + The_Period;
...
```

7. 27 июня 2010 года в 7:31

Денк Паджe сказал:

Было бы лучше, если бы период в спецификации `Thread_T` имел тип `Milliseconds` вместо `Positive`? Затем клиенты сразу видят, что задача ожидает период в миллисекундах.