Игорь Егоров
Руслан Богатырев
Дмитрий Петровичев

Еще один подход к реализации механизма обработки исключений в языке Modula-2

Источник: Structured Programming, 1993, #14, pp. 23—36.
Перевод с англ. Р. Богатырев

Аннотация

Настоящая работа посвящена изложению предлагаемого авторами подхода к механизму обработки исключений, важной особенностью которого является его реализация в мультипроцессорной среде в рамках языка Modula-2. В статье дается сравнительный анализ различных механизмов обработки исключений, реализованных в существующих языках программирования, таких как Ada, Modula-3, Eiffel и CLU. На основе этого анализа раскрываются суть и особенности реализации предлагаемого подхода.

Терминология

Обработка исключений, параллельные процессы, ресурсы, финализация, Modula-2, Ada, Modula-3, Eiffel, CLU.

1. Введение

Механизмы обработки исключений (EHM — exception handling mechanism) в значительной мере способствуют разработке надежного программного обеспечения. И вряд ли стоит говорить о том, что построенные с их помощью системы гораздо проще тестировать и верифицировать.

Поэтому не может не удручать тот факт, что такой язык как Modula-2 [Wir85] лишен подобных средств. Правда, к моменту написания статьи к своему завершению подходили работы над ISO-стандартом языка Modula-2, которые велись группой SC22/WG13. Этот стандарт предусматривает включение в язык соответствующего механизма. Тем не менее, нам хочется ознакомить читателя с теми результатами, которые нам удалось достичь в решении проблемы обработки исключений в рамках языка Modula-2. Для нас, как для авторов, гораздо важнее было изложить суть своего подхода, нежели дать четкий и скрупулезный обзор того, что реализовано в других языках. Именно этой посылкой и продиктованы характер и структура статьи. Сначала в ней обсуждается трактовка понятия исключения. Затем формулируются основные требования к предлагаемому механизму обработки исключений, поскольку они и заложили базу наших исследований. Далее приводится краткая историческая справка о других языках, имеющих встроенный EHM.

После этого идет основная часть статьи, в которой представлены ключевые аспекты EHM. Она включает анализ достоинств и недостатков нашего механизма EHM. Статья завершается кратким пояснением разработанной авторами концепции охранника ресурса, которая тесно связана с механизмом EHM. Наконец, в качестве иллюстрации изложенных идей в приложениях приведены исходные тексты ряда модулей нашей библиотеки, анализ которых позволит читателю не только лучше разобраться в нашей реализации, но и найти ответы на те вопросы, которые остались за рамками статьи.

2. Понятие исключения

Понятие исключения играет важную роль в программировании. Появление этого понятия было вызвано потребностью в таких средствах, которые позволили бы в ходе выполнения программы распознавать и корректировать различные ошибки (нестандартные ситуации) вне зависимости от их природы. Это могут быть ошибки оборудования, некорректные исходные данные или же ошибки программирования. Кроме того, из-за ограничения ресурсов компьютерной системы могут возникать всевозможные нестандартные ситуации, такие как выход за границы представления (ограничение разрядной сетки) и нехватка оперативной памяти или же свободного пространства на внешних устройствах.

Сложность задачи состоит в том, что исключения могут возникать в любое время в любой точке выполнения программы и количество их может быть достаточно велико. Нестандартность возникающих ситуаций возлагает особую ответственность на программиста, который должен предусмотреть их появление и возможные последствия. В связи с этим одно из главных требований к механизму обработки исключительных ситуаций — это простота в его понимании и использовании.

Принцип действия механизма ЕНМ состоит в следующем. Сначала (еще до выполнения программы) определяется список возможных исключений. Он может быть линейным или каким-то образом структурированным, но это не меняет сути дела. После этого по телу программы расставляются всевозможные сети, ловушки и капканы, которые отвечают за обработку исключений. Они некоторым образом (в разных языках это делается по-разному) увязываются с зоной возникновения соответствующих исключений. Затем, уже после запуска программы, когда возникает исключение (самостоятельно или же с помощью специальных средств возбуждения), происходит "замораживание" выполнения программы и идет принудительная передача управления в область обработки исключения, связанную с зоной "конфликта". В этой области, которая синтаксически четко отделена от других частей программы, происходит распознавание исключения, предусматривающее соответствующую реакцию. Реакция может быть либо немедленной, либо отложенной. В последнем случае исключение распространяется "наверх" в объемлющий программный блок, где и решается его судьба.

Изложенный принцип остается практически неизменным для разных механизмов обработки исключений. И в нем не столь уж сложно заметить одну интересную особенность. Поскольку при возникновении исключения прерывается нормальное выполнение программы, которое затем может быть возобновлено, то исключение можно рассматривать как программное прерывание - некий аналог аппаратного прерывания со всеми вытекающими отсюда последствиями. Наш подход опирается именно на такую, более широкую по сравнению с общепринятой трактовку понятия исключения. При таком рассмотрении использование исключений может носить не только пассивный, но и активный характер. Другими словами, мы делаем акцент не только на обработку исключений, но и на их возбуждение.

3. Основные требования к механизму ЕНМ

Одним из основных побудительных мотивов разработки собственного механизма ЕНМ было отсутствие подобных средств в языке Modula-2. И дело тут не только в нашей привязанности к этому языку, но и в том, что полностью удовлетворяющего нас механизма в существующих языках найти не удалось.

Какие же требования мы предъявляем к подобному механизму?

Во-первых, он должен быть очень простым и понятным. Во-вторых, обеспечивать эффективную реализацию в рамках языка Modula-2, не требующую внесения в язык никаких изменений. И, наконец, в-третьих, предоставлять удобные средства для обработки исключений в среде параллельных и квазипараллельных процессов. Еще одно немаловажное требование связано с обеспечением переносимости этого механизма.

4. Исторические замечания

В данный обзор включены четыре наиболее интересных на наш взгляд языка, каждый из которых обладает встроенным механизмом ЕНМ: это языки Ada, Modula-3, Eiffel и CLU.

Язык Ada [Geh84] был разработан по заказу Министерства Обороны США в 1980 году и обрел свой современный вид в 1983 году после принятия соответствующего ANSI-стандарта. Основное назначение языка - разработка больших встроенных систем.

Язык Modula-3 [Car89] представляет собой плод совместных усилий Центра системных исследований фирмы DEC в Пало-Альто (DEC Systems Research Center, Palo Alto, California) и Исследовательского центра фирмы Olivetti в Менло-Парк (Olivetti Research Center, Menlo Park, California). Этот язык расширяет возможности языка Modula-2 средствами объектно-ориентированного программирования, обработки исключительных ситуаций, а также средствами надежного программирования.

Язык Modula-3 разрабатывался на базе экспериментального языка Modula-2+ [RLW85], который зародился в Западной исследовательской лаборатории DEC (DEC Western Research Lab - WRL). Первое официальное описание языка Modula-3 увидело свет в августе 1988 года; язык не застыл, а постоянно развивается. Одно из последних его описаний содержится в книге [Har92].

Язык Eiffel [Mey89] принадлежит перу Бертранда Мейера (Bertrand Meyer). Основные идеи этого языка были заложены в Калифорнийском университете в Санта-Барбара (University of California, Santa Barbara). Первое описание языка появилось в 1987 году. Этот язык интересен прежде всего тем, что в нем наряду с объектно-ориентированным подходом и обработкой исключений предпринята попытка создать язык для надежного программирования, который ориентирован на формальное доказательство правильности программ.

Язык CLU [LiG86], который обязан своим появлением на свет Барбаре Лисков (Barbara Liskov), создавался с расчетом на преподавание современных методов программирования. Свое название язык CLU получил благодаря предложенной в нем концепции кластера (cluster). Идеи этого языка были заложены в ходе работ на проектом MAC, которые велись в Массачусетском технологическом институте (Massachusetts Institute of Technology).

5. Ключевые аспекты механизма ЕНМ

Чтобы немного облегчить задачу сравнения механизмов обработки исключений в разных языках, мы выделим следующие ключевые аспекты.

Идентификация исключения.

Возможности обработки исключения напрямую зависят от выбранной формы обозначения исключительной ситуации.

Возбуждение исключения.

Исключение может возбуждаться как явно (с помощью соответствующего оператора возбуждения), так и неявно (в результате выполнения встроенных операций с фиксированным набором исключений).

Распознавание и обработка исключения.

Распознавание и обработка тесно связаны друг с другом: обычно рядом со списком распознаваемых исключений указывается соответствующий обработчик.

Возобновление вычислительного процесса.

Механизмы ЕНМ различаются по способу возобновления вычислений после возбуждения исключения.

Распространение исключений.

Все рассматриваемые языки поддерживают иерархию обработки исключений, но способы достижения этой цели от языка к языку сильно разнятся.

ЕНМ в среде параллельных процессов.

Поведение механизмов ЕНМ в параллельной среде ведет к появлению ряда новых проблем.

Этот список далеко не полон. Так, в частности, в нашей работе никак не затронут вопрос об обработке исключений в объектно-ориентированной среде. Сделано это умышленно, поскольку его освещение значительно увеличило бы размеры нашей статьи.

Из-за отсутствия четко устоявшейся терминологии в области обработки исключений мы будем пользоваться своей. Одно из важнейших понятий — это синхронные и асинхронные исключения. Синхронные (внутренние) — это исключения, возникающие по инициативе процедур или встроенных в язык операторов и операций, которые выполняются процессом. Эти исключения являются следствием выполнения вычислительного процесса при конкретных начальных данных. Асинхронные (внешние) — это исключения, возникающие по инициативе оборудования или других процессов. Исключения оборудования, как правило, обрабатываются специальными системнозависимыми процессами и в обычных процессах не возникают.

Важное отличие синхронного исключения от асинхронного состоит в том, что первое легко повторить, поскольку оно не зависит от внешних факторов, таких как время и оборудование.

Еще одно немаловажное понятие — это уровень исключений. Уровень исключений представляет собой связанную пару, состоящую из двух блоков: блока контроля и блока обработки исключений. Блок контроля — это синтаксически выделенный программный фрагмент, в котором контролируется возникновение исключений. Блок обработки — это синтаксически выделенный программный фрагмент, в котором происходит обработка исключений, возбужденных в соответствующем блоке контроля.

В нашей реализации основные средства, предназначенные для обработки исключений, сконцентрированы в библиотечном модуле с именем "Ex" (см. Приложение 1).

5.1. Идентификация исключений

В нашем подходе исключение обозначается парой идентификаторов. Первый из них отвечает за номер программного модуля, а другой — за номер ошибки, которая может возникать внутри этого модуля. Идентификаторы модуля — это константы типа CARDINAL, которые описаны в специальных интерфейсных модулях. Каждый такой модуль предназначен для описания одного слоя библиотеки и в нем разрешается задавать идентификаторы модулей этого слоя только в пределах определенного диапазона, причем такие диапазоны между собой не перекрываются. Идентификатор ошибки — это константа типа CARDINAL, которая определяется для каждого исключения, порождаемого процедурами данного модуля. Другими словами, идентификатор (абсолютный) исключения состоит из номера соответствующего модуля и идентификатора (относительного) исключения внутри этого модуля. Для параметризации исключений с каждым из них может связываться строка сообщения (для целей отладки) и некоторый аргумент типа LONGCARD. Такая схема с двумя уровнями идентификации (модуль и ошибка) предоставляет более удобные средства для обработки исключений.

К сожалению, по нашей схеме статический анализ средствами компилятора невозможен. Мы просто добавляем библиотечные модули run-time поддержки, что улучшает использование языка, но не расширяет сам язык.

В языке Ada используется зарезервированный тип "exception".

Существует ряд предопределенных исключений: CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAMM_ERROR, STORAGE_ERROR, TASKING_ERROR.

Исключения в языке Modula-3 описываются не просто как элементы некоего зарезервированного типа данных; их можно параметризовать некоторым типом. Так, если "id" является идентификатором исключения, а тип "T" - это некий тип, отличный от открытого массива, то запись

```
EXCEPTION id(T)
```

описывает "id" как исключение с аргументом типа "T". Если "(T)" опущено, то исключение не имеет аргументов. Описания исключений допустимы только на верхнем уровне интерфейсов и модулей. Modula-3 не имеет зарезервированных исключений.

В языке Eiffel для работы с исключениями выделен специальный класс EXCEPTIONS. Каждое исключение представляется целочисленной константой. Существуют зарезервированные исключения, такие как Overflow, No_more_memory и др., которые описаны в классе EXCEPTIONS как символьные константы (константные атрибуты - в терминологии языка Eiffel).

В языке CLU исключения рассматриваются как необязательные атрибуты процедур и поэтому описываются при спецификации заголовка процедуры. Для этого используется ключевое слово "signals".

4.2. Возбуждение исключений

Помимо зарезервированных исключений, генерируемых run-time системой языка Modula-2 (идентификатор модуля у них равен нулю), явное возбуждение исключений может осуществляться с помощью процедур (своеобразных операторов):

```
Raise (mod,err: CARDINAL; msg: ARRAY OF CHAR);
RaiseArg (arg: LONGCARD; mod,err: CARDINAL; msg: ARRAY OF CHAR);
RaisePos (pos: ADDRESS;  mod,err: CARDINAL; msg: ARRAY OF CHAR);
```

Процедура "Raise" возбуждает исключение с идентификатором "mod,err" и со строкой сообщения "msg". Процедура "RaiseArg" — это разновидность процедуры "Raise", которая позволяет указывать дополнительный параметр возникшего исключения (arg: LONGCARD). Процедура "RaisePos" - тоже разновидность "Raise". Она предназначена для возбуждения исключения с указанием точки его возникновения (т.е. места, где оно якобы возникло).

Возбуждение исключений в языке Ada производится с помощью оператора возбуждения, который имеет следующий вид:

```
raise [имя_исключения]
```

При выполнении оператора возбуждения с именем исключения возбуждается указанное исключение. Оператор возбуждения без имени исключения может появиться лишь в обработчике исключений, если только обработчик не расположен во вложенной подпрограмме, пакете или задаче. Этот оператор вторично возбуждает то же исключение, которое вызвало переход на обработчик исключения с данным оператором возбуждения.

Явное возбуждение исключений в языке Modula-3 осуществляется с помощью специального оператора RAISE, который может использовать имя исключения как с аргументом, так и без него.

Для явного возбуждения исключений в языке Eiffel используется процедура "raise", заданная в классе EXCEPTIONS. В качестве входного параметра в эту процедуру передается код исключения.

Для явного возбуждения исключений в языке CLU используется оператор "signal". Оператор "signal" разрешается применять только в том случае, когда имя соответствующего исключения указано в заголовке процедуры. Этот оператор может быть расположен в любом месте тела программы.

4.3. Распознавание и обработка исключений

Структура уровня исключений в нашей реализации синтаксически выглядит следующим образом:

```
CASE Ex.Case (ex) OF
  Ex.TRY: (* блок контроля *)
    тело
| Ex.INTERNAL: (* блок обработки внутренних исключений *)
    IF Ex.Name (mod1,err1) THEN обработчик_I1 END
    . . .
    IF Ex.Name (modN,errN) THEN обработчик_IN END
| Ex.EXTERNAL: (* блок обработки внешних исключений *)
    IF Ex.Name (m1,e1) THEN обработчик_E1 END
    . . .
    IF Ex.Name (mM,eM) THEN обработчик_EM END
END; Ex.EndCase (ex);
```

Как видно из этой схемы, уровень исключений обрамляется двумя процедурами: "Case" и "EndCase", причем каждый уровень должен иметь свой собственный заголовок (переменная "ex" типа "Ex.Ptr").

Функция "Case" в качестве выходного параметра возвращает проинициализированный заголовок уровня исключений, а в качестве своего результата — идентификатор блока уровня исключений.

Процедура "EndCase" управляет последовательностью выполнения блоков уровня исключений и после завершения уровня высвобождает его заголовок. Разрыв пары "Case-EndCase" такими операторами, как RETURN и EXIT, недопустим. Это соглашение должно строго соблюдаться программистом.

Распознавание исключений производится внутри соответствующих блоков обработки. Для распознавания можно использовать следующие три процедуры-функции:

```
Name (mod,err: CARDINAL): BOOLEAN;
NameMod (mod: CARDINAL): BOOLEAN;
Others (): BOOLEAN;
```

Все эти процедуры-функции нужно использовать с оператором IF без ELSE-части. Процедура "Name" сопоставляет идентификатор возникшего исключения с идентификатором, который определяется ее фактическими параметрами. В случае их совпадения исключение помечается как "распознанное". Если исключение до этого уже было распознано, то "Name" всегда будет возвращать FALSE. Процедура "NameMod" — это разновидность процедуры "Name". Она полезна в тех случаях, когда нужно распознать, что возникло какое-то исключение, возбуждаемое процедурами конкретного модуля. Процедура "Others" — еще одна разновидность процедуры "Name". Она помечает как распознанное любое(!) возникшее исключение. Пользоваться этой процедурой рекомендуется только в очень редких случаях!

Обработка исключений задается в THEN-части соответствующего оператора IF. Если THEN-часть пуста, то происходит поглощение исключения. Кроме того, здесь разрешается использовать все процедуры возбуждения исключений и описывать новые вложенные уровни исключений. Модуль "Ex" предоставляет также процедуру Arg (): LONGCARD; которая возвращает значение аргумента возникшего исключения. Если аргумента не было, то, в свою очередь, возбуждается исключение "Ex.NotArg".

Семантика конструкции уровня исключений состоит в следующем. Сначала выполняется блок контроля. Если во время его выполнения не возникло ни одного исключения, то блоки обработки игнорируются. Впротивном случае нормальное выполнение приостанавливается и управление передается в соответствующий блок обработки (INTERNAL или EXTERNAL, в зависимости от природы исключения). В нем и происходит распознавание исключения. Если возникло одно из тех исключений, идентификатор которого определяется фактическими параметрами процедуры "Ex.Name", то выполняется соответствующий обработчик. Если же возникшее исключение так и не будет обработано, то автоматически возбуждается исключение Ex.FatalExcept. Если в блоке обработки до распознавания обрабатываемого исключения возникает еще одно исключение, то оно трансформируется в Ex.FatalExcept и распространяется "наверх", при этом оба исключения теряются.

Распознавание исключений в языке Ada осуществляется с помощью оператора "when". Оператор "when" определяет обработчик, соответствующий указанному исключению. Область обработчиков синтаксически отделена от обычных операторов:

```
begin
  тело
  exception
    when id1 => обработчик1
    . . .
    when idN => обработчикN
    when others => обработчик0
end
```

Для распознавания и обработки исключений в языке Modula-3 используется оператор TRY-EXCEPT, который имеет вид:

```
TRY
  тело
EXCEPT
id1 (v1) => обработчик1
| . . .
| idN (vN) => обработчикN
ELSE  обработчик0
END;
```

Здесь "тело" и каждый "обработчик" представляют собой последовательность операторов; "id" — идентификаторы исключений, а "v" — их аргументы. Части "(vi)" и "ELSE обработчик0" являются необязательными. Если один и тот же идентификатор исключения присутствует в списке более одного раза, то это считается статической ошибкой (которая обнаруживается на этапе компиляции). Семантика оператора TRY-EXCEPT состоит в следующем. Сначала выполняется "тело". Если во время его выполнения не возникло ни одного исключения, то часть EXCEPT игнорируется. В противном случае нормальное выполнение прекращается и управление передается в часть EXCEPT, где и происходит распознавание исключения. Если возникло одно из тех исключений, которые указаны в списке "id1...idN", то выполняется соответствующий обработчик. Если же возникает исключение, отсутствующее в списке, то выполняется часть ELSE ("обработчик0"). Если же и эта часть отсутствует, то исключение распространяется наверх. Каждая запись "(vi)" объявляет переменную, тип которой совпадает с типом аргумента исключения "idi" и областью видимости которой является обработчикi.

Когда возникает исключение "idi" с аргументом "x", "vi" инициализируется значением "x" еще до того, как начинает выполняться обработчикi. Если "vi" присутствует, а исключение "idi" не имеет аргумента, то это считается статической ошибкой. Для сохранения общности операторы EXIT (выход из LOOP, WHILE, REPEAT или FOR) и RETURN (выход из процедуры и процедуры-функции) рассматриваются как соответствующие исключения. Кроме того, в языке Modula-3 есть тесно связанный с обработкой исключений механизм финализации. Оператор финализации выглядит следующим образом:

```
TRY s1 FINALLY s2 END;
```

Семантика его состоит в том, что сначала выполняется последовательность операторов s1, а затем и s2 вне зависимости от того, возникло ли в s1 какое-либо исключение. Идея этого оператора состоит в том, чтобы можно было выполнить некоторые заключительные действия, связанные с возвратом захваченных ресурсов. Правда, здесь возникает один нюанс. Что будет, если исключение возникает не только в s1, но также и в s2? В этом случае информация о первоначальном исключении (в s1) теряется.

В языке Eiffel для выделения в конкретной процедуре (routine) области, в которой осуществляется распознавание и обработка исключений, используется ключевое слово "rescue". Таким образом, в каждой процедуре может быть не более одной области обработки исключений. Для распознавания исключений в этой rescue-части можно использовать оператор "if".

Для распознавания обработки исключений в языке CLU используется оператор "except". Структура этого оператора выглядит следующим образом:

```
тело
except
  when id1: обработчик1
  . . .
  when idN: обработчикN
  others: обработчик0
end
```

Рассмотрим подробнее семантику этого оператора. Пусть "тело" — оператор, с которым связаны обработчики исключений. Каждый "when"-обработчик связан с одним или несколькими именами исключений.

Тело его выполняется в том случае, когда обращение к части "тело" порождает исключение с одним из этих имен. Все имена, указанные в списке, должны отличаться друг от друга. Для обработки исключений, не указанных в части "when", используется "others"-обработчик. В роли "тело" может выступать любой оператор, включая и оператор "except". Если в ходе выполнения возникло исключение, которое не удалось распознать в части "when", а часть "others" отсутствует, то автоматически возбуждается зарезервированное исключение "failure". В соответствии с идеологией языка CLU должны обрабатываться все исключительные ситуации, кроме тех, которые не могут возникнуть. Поэтому "failure" обычно означает, что произошла программная или системная ошибка, которую должен анализировать программист.

4.4. Возобновление вычислительного процесса

В книге [You82] выделяются три возможных типа действий, связанных с продолжением вычислительного процесса. Они называются соответственно "уйти", "отметить" и "разобраться". Принцип "уйти" состоит в том, что после завершения обработки возникшего исключения управление передается оператору, непосредственно следующему за данным уровнем исключений. Принцип "отметить" подразумевает, что после завершения обработки управление возвращается в точку, которая следует сразу за тем оператором, где было возбуждено исключение. Наконец, принцип "разобраться" объединяет вышеописанные принципы; при этом в блоке обработки может быть задействован оператор "resume", выполнение которого аналогично действию принципа "отметить". В [You82] убедительно доказано, что принцип "уйти" является наиболее приемлемым. Наш подход (как, впрочем, и языки Ada, Modula-3, CLU) следует этому принципу.

В языке Eiffel принцип "уйти" расширяется механизмом возобновления ("resumption"), идея которого состоит в том, чтобы попытаться зафиксировать причины, приведшие к возникновению исключений, и запустить выполнение процедуры с самого начала. Для этой цели в конце части "rescue" используется оператор "retry", который приводит к перезапуску процедуры.

4.5. Распространение исключений

Распространение исключений подразумевает вложенность уровней исключений. В нашем подходе заведение новых уровней возможно как в блоке контроля, так и в блоке обработки исключений. Под распространением исключений мы понимаем механизмы возбуждения исключения на ближайшем объемлющем уровне исключений. Это достигается одним из трех способов.

1. Явным возбуждением исключения в теле обработчика.
2. Перевозбуждением исключения в теле обработчика (с помощью специальной процедуры "RaiseNext").
3. Неявным возбуждением исключения "Ex.FatalExcept" (если возникшее исключение не идентифицировано в блоке обработки).

Распространение исключений в языке Ada также может быть как явным, так и неявным. Явное распространение достигается за счет использования соответствующих операторов возбуждения в области обработчиков исключений. Механизм неявного распространения исключений включается тогда, когда на данном уровне исключений отсутствует область обработчиков, либо когда исключения возникают внутри обработчика. Действия в подобной ситуации зависят от природы окружения. Одна из особенностей языка Ada состоит в том, что исключение может возникать не только в программном блоке, но и при упреждающем выполнении раздела описаний данного окружения.

В языке Modula-3 исключения распространяются наверх через вложенные динамические контексты до тех пор, пока не будет найден соответствующий обработчик. Причем это происходит и в том случае, когда в части EXCEPT оператора TRY-EXCEPT нет явного распознавания возникшего исключения и в этом же операторе отсутствует часть ELSE.

Распространение исключений в языке Eiffel производится в соответствии с принципом "организованной паники" ("organized panic"). Суть его кроется в том, чтобы при возникновении исключительной ситуации привести все затронутые объекты в нормальное (coherent) состояние и сообщить о факте возникновения исключения. Важная особенность механизма исключений языка Eiffel состоит в том, что сразу по завершении обработки в "rescue"-части в точке вызова данной процедуры будет возбуждено исключение. Если в процедуре отсутствует "rescue"-часть, то считается, что она имеет пустую "rescue"-часть.

Распространение исключений в языке CLU может достигаться одним из следующих способов. Во-первых, путем явного возбуждения исключения в теле обработчика. Во-вторых, неявно при отсутствии соответствующего обработчика (при этом оно преобразуется в исключение "failure"). И, наконец, с помощью оператора "resignal", назначение которого состоит в том, чтобы распространять исключение под тем же именем.

4.6. Работа в среде параллельных процессов

Взаимодействие параллельных процессов (аналогов сопрограмм языка Modula-2) у нас производится с помощью специального механизма промежуточных областей данных (IDA), схожих по своему назначению с IDA в методологии Mascot-3 [Bat86]. Поэтому непосредственное взаимодействие процессов с помощью возбуждения исключений используется в крайних случаях, таких как уничтожение "взбесившихся" процессов, поддержание иерархии процессов и тому подобное. Как уже отмечалось ранее, все исключения в нашем подходе разбиты на два типа: внешние и внутренние. Возбуждение внешнего исключения достигается с помощью процедуры "Raise" из модуля "LowProcess" (см. Приложение 1). Внешнее исключение может возникнуть в любой момент. В связи с этим процесс может находится в одном из двух режимов - защищенном (protected), когда возбуждение внешних исключений блокируется (они задерживаются до выхода процесса из защищенного режима) и незащищенном (unprotected), когда допустимо возникновение в процессе внешних исключений. Наличие этих режимов не влияет на возникновение синхронных (внутренних) исключений. Явное управление этими режимами осуществляется с помощью процедур "Protect" и "Unprotect" из модуля "LowProcess". Неявное управление происходит при переходе из блока контроля в блок обработки и по завершению этого блока обработки. При этом гарантируется, что весь блок обработки находится в режиме protected. Когда осуществляется выход из блока обработки, прежний режим восстанавливается. Здесь важно отметить то, что внутренние исключения могут распознаваться и внутри защищенной области.

В языке Ada можно выделить два момента: исключения, возникающие при активизации задач, и исключения, появляющиеся в ходе их выполнения. Активизация различных задач (как и их выполнение) производится параллельно. Если задачный объект объявлен непосредственно в разделе описаний, то его активизация начинается после упреждающего выполнения раздела описаний. Если при активизации одной из таких задач возбуждается исключение, то эта задача завершается и в этом случае сразу после завершения активизации (успешного или неудачного) остальных задач возбуждается исключение "TASKING_ERROR". Это исключение возбуждается лишь один раз, даже если во время активизации таким образом завершились сразу несколько задач. Исключение может распространяться на взаимодействие задач или на попытку начать взаимодействие одной задачи с другой. Исключение может также распространяться на вызывающую задачу, если оно было возбуждено в ходе рандеву.

Работа с параллельными процессами (в терминологии Modula-3 они называются "thread") вынесена за пределы языка в модуль "Thread". Причем этот модуль в зависимости от реализации может видоизменяться. Для поддержки асинхронных прерываний стандартно в нем заведено единственное исключение с именем "Alerted". Для того чтобы возбудить это исключение в другом процессе используется процедура Alert (thread:T); при ее выполнении процесс "thread" помечается как "потревоженный" (alerted). Исключение "Alerted" не ведет себя как обычное исключение. Модуль предоставляет специальную процедуру

```
TestAlert (): BOOLEAN;
```

которая возвращает TRUE, если текущий процесс помечен как "потревоженный". Из этого следует, что процессу необходимо периодически вызывать "TestAlert". Некоторые расширения модуля "Thread", как это описано в [Car89], позволяют процессу пользоваться асинхронными прерываниями. Распространение и обработка исключений остаются локальными по отношению к каждому процессу. Другими словами, возбужденное исключение всегда обрабатывается в текущем процессе. Оно не может пересекать границы другого.

В языках Eiffel и CLU отсутствует понятие параллельного процесса, а потому нет и никаких проблем, связанных с обработкой исключительных ситуаций в условиях параллелизма.

5. Сравнение языков

Как видно из предыдущего раздела, наиболее сбалансированный механизм обработки исключений представлен в языке Modula-3. К достоинствам его можно отнести и довольно мощные средства идентификации исключений, и возможность описания в теле процедуры более одной области обработки исключений (как и в языке CLU), и более богатые по сравнению с другими языками средства обработки исключений в среде параллельных процессов (здесь уже исключение выступает в роли прерывания).

Одна из интересных особенностей языка Modula-3 — механизм финализации. С его введением сделана попытка облегчить обработку исключений за счет сокращения явных действий по приведению в нормальное состояние используемых ресурсов. Однако, реализацию механизма финализации сложно отнести к достоинствам языка. Дело в том, что при возникновении исключений в самой части финализации первоначальное исключение теряется. Таким образом, трудно судить, выполнялась ли финализация при нормальном ходе вычислительного процесса или же ее выполнение было инициировано возбуждением исключения. Что же касается среды параллельных процессов, то в ней такой подход вряд ли может считаться удачным, как, впрочем, и сама идея связывать процедуру восстановления захваченных ресурсов не напрямую с ними, а с областями возможного возникновения исключительных ситуаций.

На другом полюсе сравнения находится язык Ada. В нем сложно выделить какие-то интересные моменты решения механизма ЕНМ. Здесь и жесткая привязка области обработки исключений ко всему телу процедуры, и крайне запутанная (в силу большого количества программных единиц и наличия предописаний) логика обработки исключений, и просто примитивное решение для среды параллельных процессов.

Языки CLU и Eiffel, на наш взгляд, располагаются где-то в середине. CLU интересен своими четкими прозрачными решениями и продуманностью всего механизма в целом. Eiffel предлагает механизм возобновления (RETRY-механизм) и трудно спорить с тем, что при его использовании наглядность программ значительно возрастает. Однако, возможность возникновения зацикливания — довольно серьезный аргумент против наличия такого механизма.

6. Достоинства и недостатки предлагаемого механизма

Практика использования нашего механизма ЕНМ позволяет выделить следующие его достоинства.

1. Удобная и наглядная структура уровня исключений. Привлечение в схему операторов CASE и IF дает возможность легко разобраться в такой структуре.

2. Наш подход не требует внесения изменений в язык Modula-2. Теперь при программировании требуется лишь аккуратно соблюдать определенные соглашения.

3. Он дал возможность распространить сферу действия механизма ЕНМ и на область параллельных процессов, позволив тем самым получить очень мощный механизм асинхронных прерываний.

Кроме того, он вобрал в себя самое лучшее, что есть в языках Ada, Modula-3, CLU и Eiffel. После тщательного изучения всех плюсов и минусов мы намеренно отказались от таких решений, как финализация и RETRY-механизм. Финализация была заменена охранниками ресурсов, а RETRY-механизм можно имитировать.

Естественно, что основные недостатки кроются в самой идее реализации столь сложного механизма за пределами языка. Здесь сразу теряется вся прелесть статического контроля на этапе компиляции. Идентификация исключений также далеко не самое лучшее, что есть в нашем механизме. Любое изменение номера модуля требует перекомпиляции всех модулей соответствующего библиотечного слоя. Да и параметризация исключений весьма ограничена.

7. Понятие охранника ресурса

Как уже было замечено выше, решение проблемы финализации нами было вынесено на другой уровень. С этой целью было введено понятие ресурса. Под ресурсом мы понимаем набор компонент, которые состоят из данных и связанных с ними процедур. Эти процедуры позволяют выполнять три функции: создание, манипулирование и уничтожение ресурса. Все эти функции изначально ориентированы на работу в нормальных условиях (без исключительных ситуаций). Для работы в случае исключений введено понятие охранника ресурса. Он представляет собой дополнительный набор данных и специальную процедуру, отвечающую за приведение ресурса в устойчивое состояние. Для работы с охранниками ресурсов используется модуль "Guard". Он имеет 4 процедуры (см. Приложение 1):

```
New (life: Global.Life; size: CARDINAL; proc: Proc): Object;
Del (VAR obj: Object);
Exist (obj: Object): BOOLEAN;
My (obj: Object): BOOLEAN;
```

Процедура "New" предназначена для заведения охранника ресурса. В качестве входных параметров указываются соответственно область существования охранника, размер дескриптора охранника в байтах (нужно для неявной автоматической утилизации) и функция охранника. Процедура "Del" используется для уничтожения охранника. Процедура "Exist" может использоваться для проверки того, что охранник не разрушен. Процедура "My" требуется в тех случаях, когда надо определить, принадлежит ли охранник текущему процессу.

С каждым ресурсом связывается такое понятие, как область его существования (параметр "life" процедуры "New"). Мы выделяем два вида областей: tmp и process, которые тесно связаны с обработкой исключительных ситуаций. Tmp-ресурс предназначен для заведения временных данных, которые будут неявно утилизироваться при завершении блока контроля исключений. Причем непосредственно до начала утилизации запускается функция охранника этого ресурса. Process-ресурс отличается от tmp-ресурса в двух аспектах. Во-первых, при нормальном завершении блока контроля уровня исключений, которому принадлежит этот ресурс, последний не утилизируется, а приписывается охватывающему уровню исключений. Во-вторых, при завершении процесса все его process-ресурсы утилизируются.

К сожалению, рамки статьи не дают возможности более подробно описать механизм ресурсов, но даже здесь можно заметить, что он позволяет придать идее финализации более четкую форму. В нашем варианте восстановительные действия для конкретного ресурса собраны в одном месте (функция охранника), тогда как в механизме TRY-FINALLY эти действия рассредоточены.

8. Пояснения к приложениям

В конце статьи приведены исходные тексты модулей нашей библиотеки. Они тесно связаны с обработкой исключений. Наша реализация проводилась в среде MS-DOS (JPI TopSpeed V2.0). При этом потребовалось модифицировать библиотеку поддержки TopSpeed. Приложение 1 включает в себя интерфейсы следующих модулей:

```
Ex         (обработка исключений);
LowProcess (низкоуровневое управление процессами);
Guard      (управление ресурсами).
```

В Приложении 2 приводится пример использования EHM (модуль D_Ex). Это программа для вычисления факториала числа. В нем используются две процедуры "ExecCard" и "ExecReal". "ExecCard" вычисляет факториал с помощью целочисленной арифметики, а "ExecReal" вычисляет факториал с помощью арифметики плавающей точки. Переключение с одной процедуры на другую выполняется в процедуре "Exec".

```
PROCEDURE Exec (card: CARDINAL);
  VAR ex1,ex2: Ex.Ptr;
      result : ARRAY [0..29] OF CHAR; r: REAL;
BEGIN
  CASE Ex.Case(ex1) OF Ex.TRY:
    CASE Ex.Case(ex2) OF Ex.TRY:
      card := ExecCard (card);
      StrNum.CardToStr (card,result);
    | Ex.INTERNAL:
        IF Ex.Name(ExRTS.MODNUM,ExRTS.MathOverflow) THEN
          real := ExecReal(FLOAT(card));
          StrNum.RealToStr(real,result);
        END;
        IF Ex.Name(ExRTS.MODNUM,ExRTS.StackOverflow) THEN
          Ex.RaiseNext
        END;
    | Ex.EXTERNAL:
    END; Ex.EndCase(ex2);
    TTIO.WrStr(" N! : "); TTIO.WrStr(result);
  | Ex.INTERNAL:
    IF Ex.Name(ExRTS.MODNUM,ExRTS.StackOverflow) THEN
      TTIO.WrStr (" Переполнение стека");
    END;
  | Ex.EXTERNAL:
  END; Ex.EndCase(ex1);
END Exec;
```

Эта процедура использует два уровня исключений. Первый уровень ("ex1") обрабатывает исключения, которые возникают в случае переполнения стека. Второй уровень исключений ("ex2") размещается в TRY-части первого уровня. Сначала в TRY-части второго уровня делается попытка вычислить факториал с помощью процедуры "ExecCard". Если это не удается (из-за переполнения CARDINAL), то run-time система возбуждает исключение (ExRTS.MODNUM, ExRTS.MathOverflow), которое обрабатывается в INTERNAL-части второго уровня исключений. Здесь вычисление факториала осуществляет уже процедура ExecReal. В этом примере обработка исключения, связанного с потерей точности (Floating Loss of Precision), не производится. Если такое исключение возникает, то из-за отсутствия соответствующего обработчика в INTERNAL-части первого уровня будет возбуждаться исключение Ex.FatalExcept.

9. Заключение

Теоретические изыскания и практика использования нашего механизма обработки исключений показали, что он в некоторых моментах превосходит хорошо известные механизмы. Главная его привлекательность состоит в том, что он придает свежие краски языку Modula-2, начинающему блекнуть на фоне своих молодых конкурентов — языков Oberon-2 и Modula-3. Наш механизм не является частью языка и может быть реализован в рамках других языков. В ходе экспериментов нам удалось выявить, что наш механизм ЕНМ позволяет унифицировать реакцию на ошибки в библиотечных модулей, решить задачу финализации, и осуществить поддержку механизма автоматической сборки мусора. Важно отметить также возможность его полноценного использования в (возможно, распределенной) среде асинхронных процессов.

К сожалению, ISO-стандарт языка Modula-2 все еще не принят. Поэтому мы и не включили язык Modula-2 в сравнительный обзор языков со встроенным механизмом ЕНМ.

Благодарности

Мы благодарим Гюнтера Дотцеля (Guenter Dotzel) из фирмы ModulaWare GmbH, оказавшего помощь в подготовке предварительной версии этой статьи (где были представлены все исходные тексты реализации ЕНМ) и публикации его в своем журнале The ModulaTor. Мы также признательны всем анонимным рецензентам за их помощь в улучшении качества статьи, а также Роберту Баху (Robert Bach) за его неустанную поддержку наших усилий.

Литература

- [Bat86] Bate G. (1986) "Mascot-3: An Informal Introductory Tutorial"// Software Engineering Journal, Vol.1, No.3-4, pp.95-102.
- [Car89] Cardelli L., et al. (1989) "Modula-3 Report (revised)" //DEC SRC Report #52.
- [Geh84]. Gehani N. (1984) "Ada: An Advanced Introduction Including. Reference Manual for the Ada Programming Language" //Prentice-Hall. Н.Джехани "Язык Ада".- М.: Мир, 1988.
- [Har92] Harbison S.P. (1992) "Modula-3" // Prentice-Hall.
- [LiG86] Liskov B., Guttag J. (1986) "Abstraction and Specification in Program Development" //The MIT Press, Massachusetts. Б.Лисков, Дж.Гатэг "Использование абстракций и спецификаций при разработке программ".- М.: Мир, 1989.
- [Mey89] Meyer B. (1989) "From Structured Programming to Object-Oriented Design: The Road to Eiffel" // Structured Programming, Vol.10, No.1, pp.19-39.
- [RLW85] Rovner P., Levin R., Wick J. (1985) "On Extending Modula-2 for Building Large, Integrated Systems", DEC SRC Report #3.
- [You82] Young S.J. (1982) "Real Time Languages: Design and Development" //Ellis Horwood Ltd. С.Янг "Алгоритмические языки реального времени. Конструирование и разработка".- М.: Мир, 1985.
- [Wir85] Wirth N. (1985) "Programming in Modula-2" // Springer-Verlag. Н.Вирт "Программирование на языке Модула-2".- М.: Мир, 1988.