维普资讯 http://www.cqvip.com

一种从 Object-Z 到 CSP 规格说明的转化方法*)

文志诚　缪淮扣　许庆国
(上海大学计算机学院　上海 200072)

摘　要　面向对象形式规格说明语言 Object-Z 与进程代数 CSP 相结合是当今的一个热点,它既可以表示复杂的模块化数据与算法,又可以表示系统的行为,但求精与验证对它们结合后的规格说明需要分别进行处理。本文提出了一个方法,把 Object-Z 规格说明转化为 CSP 规格说明,可以方便地处理结合后的规格说明,因此求精与推理对结合后的规格说明可以按 CSP 规则与方法一致来进行处理。此外,转化后的 Object-Z 规格说明可以按照 CSP 方法进行模型检查。

关键词　Object-Z,CSP,形式规格说明,参数化进程,转化

An Approach to Translating Object-Z Specification to CSP Specification

WEN Zhi-Cheng　MIAO Huai-Kou　XU Qing-Guo
(School of Computer Engineering and Science, Shanghai University, Shanghai 200072)

Abstract　Object-Z is an extension to the formal specification language Z, which facilitates specification in an object-oriented style and improves the clarity of large specifications through enhanced structuring. Object-Z is an excellent tool for modeling data and algorithms, but it is difficult to be used to capture the behaviour of concurrent reactive systems. CSP is good at modeling concurrent behaviour, but has little support for modeling the state of a complex system. The blending of Object-Z and CSP is a trend today, but the refinement and verification are taken apart for the blending speicification respectively, which is very inconvenient. This paper introduces an approach to translating the Object-Z specification to CSP specification, which is in favor of refinement and verification for the blending specification because refinement and verification can accord with the method of CSP uniformly. Moreover, conversional Object-Z specification can be model-checked according to that of CSP.

Keywords　Object-Z, CSP, Formal specification, Parameterized process, Conversion

1　前言

Object-Z[1]是形式规格说明语言 Z 的面向对象扩充,基于集合论与数理逻辑,具有严密的逻辑性,适应于精确地描述大型软件系统,并且可以对其规格说明进行推理,一些文献[2]已提出了相应的推理规则与验证方法,可以推理出规格说明应该具有的性质。标准的 Object-Z 是采用历史语义[3],把一个对象的状态演变过程看成是一个历史,由状态及操作事件构成。

CSP (Communicating Sequential Process)是通信顺序进程代数,一个命令式形式语言,由 Hoare[11]在 20 世纪 80 年代发明,广泛应用于描述并发系统。一个 CSP 程序就是一个进程,进程的创建是静态的,每个进程可以分解为许多子进程。CSP 的最大特点是有一对通信原语,进程间通信采用了一对输入/输出原语:"?"和"!",这是进程间交换数据的唯一方式。CSP 避免了一个进程直接向另一个进程的变量赋值,即一个进程不能修改另一个进程的变量。CSP 有顺序组合,选择组合和循环组合三种组合运算。标准 CSP 的语义采用的是发散集与绝境集语义。CSP 推理是采用进程迹来进行的,相应的推理规则及方法已被提出[8,11]。

Object-Z 易于模块化设计复杂的算法与数据来表示面向对象程序设计中的类及操作,但对并发程序设计及表示动作有缺憾。而 CSP 可以广泛地应用于描述并发式系统,但对模块化及复杂的数据与算法处理功能有限。因此,把 Object-Z 及 CSP 相互结合来描述复杂的并发系统是一种可采用的方式,许多文献[4~6]提出了相应的方法,其中文[7]还提出了 Object-Z 与时间 CSP 相结合,并在文[8]给出了相应的推理规则与方法。

求精与形式验证是形式化方法中的两大核心,一些文献[2,4,8]提出了相应的方法。但对于 Object-Z 和 CSP 结合的形式规格说明,验证与求精只能分开来进行处理,这将带来不方便,因为它们有不同的表示形式与语义。本文提出了一个方法,如何把 Object-Z 形式规格说明转化到 CSP 规格说明,这样 Object-Z 规格说明的求精与验证可以按 CSP 的方法来进行处理。特别地,Object-Z 一般不能进行模型检查来验证它,而 CSP 有一个模型检测工具 FDR[9,10],经过转化后的 Object-Z 形式规格说明具有 CSP 形式,因此可以用 FDR 来对它进行模型检查。

Object-Z 类中有状态变量及操作(含初始化操作),经过转化后的 CSP 规格说明一般是参数化进程。本文主要的工作是把 Object-Z 规格说明转化为 CSP 规格说明。

2　Object-Z 简介

Object-Z 是 Z 的面向对象扩展,一个形式规格说明包括

*)国家自然科学基金(60373072)和上海市教委第四期重点学科建设基金资助。文志诚　博士生,讲师,主要研究领域为面向对象技术与形式化方法。

维普资讯 http://www.cqvip.com

一些全局定义和类定义。全局定义可以被系统中所有的类共享，它定义在所有的类的外面，不属于任何一个类。全局定义和在一个类中的其余的部分使用Z的定义方式。类是它的一个重要的特征，它的结构形式如图1。

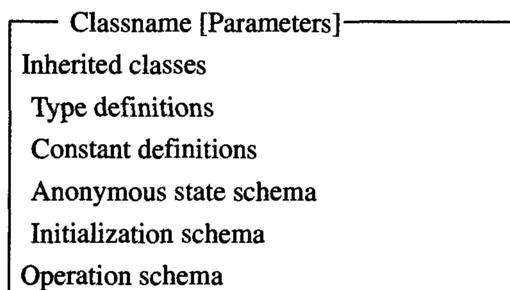

图1　Object-Z的类结构

一个类能够带有参数，可以继承它的父类，与父类同名的变量及操作采取谓词合取的方式。父类的操作在子类继承过程中被改名，可作为另一个操作被继承下来，而子类中可以重新定义一个同名的操作。类中还涉及常量定义、变量定义、无名状态模式、初始状态模式及操作模式。

其中，状态变量定义在无名状态模式中，可以被此类中所有的操作使用，状态变量可以有其约束条件，即状态不变式，它定义了状态空间，对象的每个状态是其中的一个状态。初始化操作模式用来给状态变量赋初值。操作中可以有变量定义部分及谓词部分，变量定义部分可以声明在本操作中要改变的状态变量及可以定义局部的输入与输出变量，这些局部输入与输出变量只能在本操作中起作用，其它的操作不能更改这些局部变量；谓词部分隐式定义了其前置条件及后置条件。全局变量、状态变量及局部输入与输出变量都有其相应的类型。

3　CSP简介

CSP是Hoare于1985年提出的一种进程代数的理论，它主要通过实体之间外部可观察的交互行为描述实体的行为，可以用来描述和分析并行系统通信，具有较为完整的分布计算环境中的代数演算能力，进程与事件是CSP中的重要概念。从通信的角度，进程通过它能执行的通信事件来定义。CSP中定义了两种中止进程：Stop与Skip，其中Stop表示死锁，Skip表示成功中止执行。例如，下述进程P_1先后执行事件x与y之后，处于死锁状态：

$P_1 = x \rightarrow y \rightarrow Stop$

CSP将事件名视为信道，可以规定它所传递的数据类型，例如，下述进程

$P_2 = in?\ x: T \rightarrow out!\ x \rightarrow Stop$

表示只要x为T类型数据，进程P_2就可以执行这个$in?\ x$事件。上式中如果不指明数据类型，则表示只要x为信道in范围内定义的数据类型，进程P_2即可执行该$in?\ x$事件。信道out也具有相应的数据类型x，并且一旦执行了事件$in!\ x$，并给变量x赋值v，则它的后续事件$out!\ v$就可以执行。在该进程中，out信道的数值与in信道输入的数值相同。下述进程

$P_3 = in?\ x: T \rightarrow out!\ x \rightarrow P_3$

是循环进程，由无限多个in和out事件组成。进程也可以通过参数方式表示，例如

$P_4[X] = in?\ x: T \rightarrow out!\ x \rightarrow P_4[X \cup x]$

CSP提供了丰富的不确定和并发运算操作，如不确定选择符，交互运算选择符，或运算选择符，选择进程运算选择符和并发进程运算选择符等，这些都体现了分布式程序的特征。以下是CSP中应用的几种重要的操作符：

(1) $[]$是外部非确定选择操作符，例如：$P_5 = a \rightarrow P_5$；$P_6 = b \rightarrow P_6$；$P_7 = P_5 [] P_6$为外部环境提供了两种选择，如果外部环境选择事件a，则$P_7 = P_5$；如果外部环境选择事件b，则$P_7 = P_6$。

(2) $\sqcap$是内部非确定选择操作符，允许进程间进行非确定性的选择，亦即由进程内部进行选择。例如：$P_8 = P_5 \sqcap P_6$，P_8可以内部选择执行事件a或b。如果外部环境希望执行事件a，而P_8的内部选择是执行事件b，则事件a将被阻塞。CSP中不确定的选择语句要求选择项是带前卫(guard)的语句。

(3) $P|Q$表示外部确定选择操作，可以选择P，也可以选择Q。例如：$P_9 = P_5 | P_6 = a \rightarrow P_5 | b \rightarrow P_6$，如果一开始选择a，则按$P_5$执行；如果一开始选择b，则按$P_6$执行。

(4) $P \| Q$表示通常的并行操作，进程P和Q将在共同的字母表上的事件上同步，即进程P和Q在事件$X \cap Y$上同步。对于非共同事件，可以穿插进行。一个CSP程序就是一个进程，每个进程可以通过并发进程运算符$\|$分解为若干个并发执行的子进程，这种分解可以嵌套至任意深度。

(5) $P|||Q$表示穿插操作。进程P和Q并不在任何事件上同步，它们之间完全相互独立。

(6) $P \backslash E$表示隐藏操作，其中P是进程，E是在P的外部环境中被隐藏的事件集。例如，$P_{10} = P_7 \backslash \{a\}$，如果$P_7$的可见事件为$\{a,b\}$，则$P_{10}$的可见事件为$\{b\}$。

(7) 顺序进程有赋值、条件判断、循环等操作。

4　转化Object-Z类到CSP进程

Object-Z规格说明由一些类组成。一个类是一个模板，可以生成不同的对象。一个对象在状态空间中可以从一个状态转换到另一个状态。整个系统由不同的对象组成，它们相互作用着，可以把一个类看成是一个进程。

把Object-Z中的类转化为CSP规格说明中的一个进程，可以分层进行。一个CSP进程可以由一些其它的进程复合操作复合而成，因此，在转化过程中，我们把进程广义化，即一个类看成是一个主进程，一个类中的操作也看成是一个进程，即操作进程。主进程可以包括操作进程，主进程与操作进程都是参数化进程，相应于Object-Z无名状态模式中定义的状态变量作为参数。这些参数化的进程可以互相递归调用。

相对于CSP的主进程与操作进程的命名分别是相应于Object-Z规格说明中的类及操作加上相应的参数而成。一个参数是一个序偶，第一个元素代表参数名，是一个标识符，不能改变，而第二个元素表示此标识符的值，随着进程执行而改变。

因为Object-Z类中的操作可以由复合算子复合而成，所以，我们把没有经过复合的操作称为简单操作，相应于CSP进程为简单操作进程，经过复合而得到的操作为复合操作，相应的为复合操作进程。

4.1　主进程

在定义主进程之前，我们先定义一个赋值进程Value[variables]，即对状态变量variables进行赋值，它是一个顺序进程，有形式为：

维普资讯 http://www.cqvip.com

$Value[variables]=(variables:=es);Skip$

其中,variables 表示状态变量的集合,es 是相应值的集合。赋值进程只对状态变量进行赋值。

一个类可以看成是一个主进程,其参数为全部状态变量;一个操作进程是由 Object-Z 相应的操作名加上参数而成。因此,主进程可以有如下形式:

$Class = Value[init_variables];Class[init_variables]$

$Class[variables]=(OP_1[variables][]OP_2[variables][]\cdots[] OP_n[variables])$

其中,Value[init_variables]表示对状态变量赋初值,即把相应的 Object-Z 类中初始化模式 Init 里的初值赋给相应的状态变量。符号";"是 CSP 中顺序进程的组合算子,连接两个进程,若 P,Q 是进程,则(P;Q)也是进程,先按 P 动作,当 P 成功终止后,(P;Q)就按 Q 动作。主进程与操作进程有一个区别,即主进程要初始化,才得到相应的参数化主进程,而操作进程不用初始化。我们把 Class[variables]表示参数主进程,里面的参数表示一对数偶,为了方便起见,把数偶省略了,在具体应用时再表示出来。

4.2 简单操作进程

因为 Object-Z 操作中可能有输入/输出变量,所以它相应的操作进程就有相应的输入与输出事件。在操作进程中,把 Object-Z 中相应的操作看成是一个通道,加上符号"?"和"!"分别表示在此通道上的输入或输出。在 Object-Z 中,一个操作是否能执行,要判断当前状态能否满足它的前置条件,操作结束后,其状态变量得到修正。因此可以利用 CSP 中顺序进程中的判断结构"$P \not< b \not> Q$",它表示如果条件 b 成立则按 P 动作,否则按 Q 动作。操作进程可以表示为:

$OP_i[variables]=(OP_i?\ inputs:INPUTS \to OP_i!\ outputs:OUTPUTS \to Class[variables'])\not< OP.pre \not> Class[variables]$

在 Object-Z 的操作中,输入变量表示为"input?",输出变量表示为"output!",它们相应于 CSP 中的输入与输出事件,但在表示 CSP 进程中的输入与输出事件时,把其后缀去掉。Object-Z 的输入与输出变量是有类型的,相应的类型在 CSP 进程中表示在相应的变量后面。

由于 Object-Z 操作中的输入或输出没有次序之分,因此有几个输入与输出变量时,它们顺序是无关的,但有一个原则,任何一个输入总要在每一个输出前面,在 CSP 进程中也是如此。当然我们可以采取外部环境选择算子"|"来表示,把所有可能的形式都表示出来。variable'表示经过此操作进程后改变其参数变量的值,用后状态变量来更新原先的参数,它对应于 Object-Z 操作 OP 中状态变量的后状态。OP. pre 表示为相应的 Object-Z 操作 OP 的前置条件,前置条件的求法为:

设 Object-Z 一个类的不变式为 Invariant(variables),其中 variables: Variables 是所涉及的状态变量及其类型;一个操作 OP 中可能的输入与输出变量为 inputs: Inputs, output: Outputs。根据 Object-Z 操作前置条件的求法,状态不变式含蓄的包含在每一个操作中,因此有:

$$OP.pre = \exists\, variables': Variables;\ outputs: Outputs \bullet OP(variables,\ variables',\ inputs,\ outputs) \wedge Invariant(variables) \wedge Invariant(variables')$$

其中 OP(variables, variables', inputs, outputs)表示操作 OP 部分,与状态变量 variables、后状态变量 variables'、输入变量 inputs、输出变量 outputs 及它们相应的类型有关,它对后状态变量及输出变量用存在性量词约束。

4.3 复合操作进程

在 Object-Z 中,有几个复合操作符:合取(∧)、平行(||)、序列复合(⨾)、选择([])、环境操作符(•),使用它们可以构成一个新的操作 OP。虽然平行操作(||)、选择操作([])与 CSP 中的并发操作(||)、不确定外部选择操作([])的符号表示上是一样的,但在含义上是不全相同的,因为对于 Object-Z 来说是连接两个操作,而对 CSP 来说是连接两个进程。符号";"是 CSP 的顺序进程连接符,而"⨾"是 Object-Z 序列捉拿复合符号,它们是不同的。

(1) $OP=OP_1 \wedge OP_2$,表示操作 OP 是由操作 OP_1 和操作 OP_2 同时发生的,但 OP_1 和 OP_2 执行是相互独立的,完成也是独立的。因此,在 CSP 上,可以把它们表示成穿插形式。

它们相应的参数化进程与简单操作进程有点区别,因为它们独立地更新状态变量的值,状态变量是它们的共享变量,我们用二值互斥信号量来作为共享资源,使它们在更新状态变量时不互相干扰。信号量定义为:

$SEM=P \to V \to SEM$

它可以作为参数化进程的附庸进程,我们把操作进程定义为:

$OP_i[variables]=(mutex_i:SEM\ //\ (OP_i?\ inputs:INPUTS \to OP_i!\ outputs:OUTPUTS \to mutex_i.P \to Value[variables'] \to mutex_i.V \to Skip))\not< OP_i.pre \not> Skip$

赋值进程 Value[variable']表示把此相应于 Object-Z 操作 OP_i 中的后置条件的后状态值赋给相应的状态变量,当然如果状态变量值没改变,赋值也不会改变。一个参数化操作进程赋值时,先要发出信号,赋值结束后也要发出信号。因此有二值互斥信号量,它不会受另一个参数化操作进程干扰。因此,我们有:

$$OP[variables]=(OP_1[variables] ||| OP_2[variables]);\ Class[variables']$$

(2) $OP=OP_1 || OP_2$,表示两个操作互相通讯,一个操作向另一个操作输出值,而这输出值,作为此操作的输入值,要求此输入与输出的变量同名,但可以有不同的后缀"!"和"?"。在 CSP 上,这些输入与输出通道都改为通道 OP,我们可以把它们相应的参数化进程定义为:

$OP_i[variables]=(mutex_i:SEM\ //\ (OP?\ inputs:INPUTS \to OP!\ outputs:OUTPUTS \to mutex_i.P \to Value[variables'] \to mutex_i.P))\not< OP_i.pre \not> Skip$

因此,两个操作在 CSP 中复合成并发,我们有:

$$OP[variables]=(OP_1[variables] || OP_2[variables]);\ Class[variables']$$

(3) $OP=OP_1 ⨾ OP_2$,表示先执行完操作 OP_1,再接着执行操作 OP_2。因为它们有相后次序,不会同时更新状态变量,所以不必要用互斥信号量。可以把这两个相应的参数化操作进程定义在 CSP 上有:

$OP_i[variables]=(OP_i?\ inputs:INPUTS \to OP_i!\ outputs:OUTPUTS \to Value[variables'])\not< OP_i.pre \not> Skip$

维普资讯 http://www.cqvip.com

OP的参数化操作进程定义为：

OP[variables]=(OP$_1$[variables];OP$_2$[variables])；Class[variables']

(4) OP=OP$_1$[]OP$_2$，表示操作 OP$_1$ 和操作 OP$_2$ 引起是非确定性的，只能一个发生，不能同时发生，因此，操作OP相应的参数化进程可以由操作 OP$_1$ 和操作 OP$_2$ 相应的参数化进程由非确定性算子连接起来。因为操作 OP$_1$ 和操作 OP$_2$ 不会同时发生，也不必用互斥信号量。对于操作 OP$_i$，它们相应的参数化进程为：

OP$_i$[variables]=(OP$_i$? inputs: INPUTS→OP$_i$! outputs:
OUTPUTS→Value[variables'])≮OP$_i$. pre≯Skip

OP的参数化操作进程定义为：

OP[variables]=(OP$_1$[variables][] OP$_2$[variables])；Class[variables']

(5) OP=OP$_1$ • OP$_2$，表示操作 OP$_1$ 中定义的局部变量范围扩展到操作 OP$_2$ 中，操作 OP$_1$ 先执行，执行完毕再执行操作 OP$_2$。在CSP中，可以用顺序连接符“;”连接它们

OP$_1$[variables]=(OP$_1$? inputs: INPUTS→OP$_1$! outputs:
OUTPUTS→Value[variables'])≮OP$_1$. pre≯Skip

OP$_{11}$[variables]=(OP$_1$? inputs: INPUTS→OP! inputs:
INPUTS→OP! outputs: OUTPUTS→Skip

OP$_{11}$[variables]表示把所有定义范围内的输入与输出变量输出给操作进程 OP$_2$[variables]使用，但 OP$_{11}$[variables]不会改变量主进程的状态变量的参数值。

OP$_2$[variables]=(OP? inputs: INPUTS→OP$_2$! outputs:
OUTPUTS→Value[variables'])≮OP$_2$. pre≯Skip

OP的参数化操作进程可以定义为：

OP[variables]= OP$_1$[variables];(OP$_{11}$[variables] ||
OP$_2$[variables]);Class[variables']

表示先执行操作进程 OP$_1$[variables]，接着进程 OP$_{11}$[variables]把所有定义范围内的输入与输出变量输出给进程 OP$_2$[variables]，因此它们是并发执行。

5 例

下面举一个例子来说明如何把 Object-Z 形式规格说明转化到CSP规格说明。设有一个队列类 Queue[T]（图2），其参数为T。类中定义了局部类型 Length 和局部常量 max，主要变量 items 和次要变量 size，队列类 Queue[T]的状态不变式 Invariant(items, size) = (size=#items ∧size≤max)，初始状态队列为空。类中还定义了两个操作：进队列操作 Join 和出队列操作 Leave。

在队列类 Queue[T]的状态模式中定义了状态变量为：items: seq T 和 size: ℕ，即 variables={items,size}，它们的初值由初始操作模式 INIT 中定义为：items=⟨⟩和 size=0。虽然 size 初始值没显式给出，但它是次要变量（在状态模式中主要状态变量与次要状态变量用符号“Δ”分开，符号“Δ”下面的是次要状态变量），从属于主要变量 items（因为有不变式：size = # items）。

初始化赋值进程相应于类中的初始化操作，可以对状态变量赋初值。因此，其初始化赋值进程为：

Value[init_variables] = (variables: =es);Skip
= (items, size=⟨⟩, 0);Skip

队列类 Queue[T]中有两个操作：进队列 Join 操作与出队列 Leave 操作。因此其相应的操作进程有形式为：Join[(Items, items), (Size, size)]和 Leave[(Items, items), (Size, size)]。在参数化操作进程 Join[(Items, items), (Size, size)]中，序偶(Items, items)表示，第一个 Items 是状态变量的名字，它不能改变，是一个标识符，标识这个参数；而第二个 items 是其标识符相应的值，它是随进程执行而变动的，对应于 Object-Z 类中的后状态变量的值，但在表示CSP相应的值时，可能对定义在 Object-Z 操作中相应的值要进行函数变换。因此我们可以得到其相应的两个参数化操作进程：

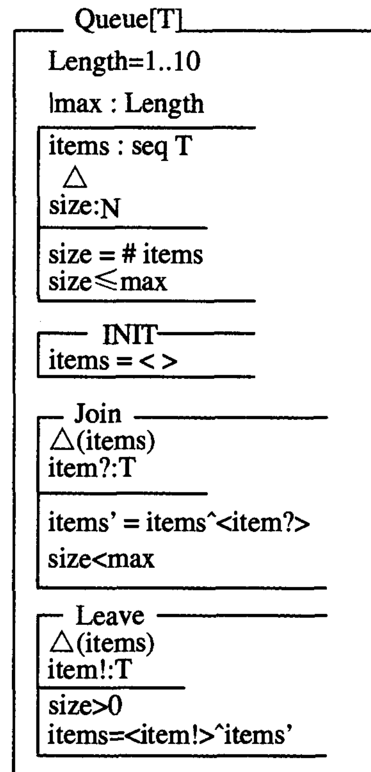

图2 类 Queue[T]

(1) 对于操作 Join，其前置条件为：

Join. pre= ∃variables': Variables; outputs: Outputs •
Join(variables, variables', inputs, outputs)
∧Invariant(variables)∧Invariant(variables')
= ∃items': T •[items: seq T; size: ℕ • items'
=items⁀⟨item?⟩∧ size⟨max∧(size=#items∧size≤max)∧(size'=#items'∧size'≤max)]

使用点规则(one-point rule)把它进行化简，即可得：Join. pre=(size<max)。因此其参数化操作进程为：

Join[(Items, items), (Size, size)]=(Join? item: T
→Queue[(Items, items⁀⟨item?⟩), (Size, size+1)])
≮(size<max)≯Queue[(Items, items), (Size, size)]

(2) 对于操作 Leave，同理可得其前置条件为：

Leave. pre=(size>0)。

操作 Leave 相应的参数化操作进程为：

Leave[(Items, items), (Size, size)]=(leave! item: T
→Queue[(Items, tail(items)), (Size, size−1)])
≮(size>0)≯Queue[(Items, items), (Size, size)]

维普资讯 http://www.cqvip.com

因为在操作 Leave 谓词部分中，有 items = ⟨item!⟩⁀items'，其后状态变量 items'的值要求原序列 items 移去第一个元素以后得到的序列，所以可以用函数 tail 来表示，这就经过了等价变换。

由(1)、(2)，我们可得队列类 Queue[T]相应的参数化主进程为：

Queue = Value[init_variables]；Queue[(Items，⟨ ⟩)，(Size，0)]

Queue[(Items，items)，(Size，size)]= (Join[(Items，items)，(Size，size)][]Leave[(Items，items)，(Size，size)])

结论 Object-Z 是形式规格说明语言 Z 的面向对象扩充，基于严格的集合论与数理逻辑，能够很好地描述大型复杂系统的算法与结构，可以对其规格说明进行推理。进程代数 CSP 能够方便地描述并发系统，可以利用进程迹来进行推理。Object-Z 和 CSP 各有本身的局限性，即 Object-Z 不能描述并发系统及其行为，而 CSP 不能很好描述系统的结构与算法，因此，目前 Object-Z 与 CSP 相结合是一个热点。结合后可以模块化描述并发的大型面向对象系统，它们的语言可以混合在一起，因而求精与验证对它们结合后的规格说明需要分别进行处理，因为 Object-Z 与 CSP 有不同的表示形式与语义，这样将带不方便。本文提出了一个方法，把 Object-Z 规格说明转化为 CSP 规格说明，对结合后的规格说明可以按 CSP 规则与方法一致来进行求精与验证。特别地，Object-Z 一般不能进行模型检查来验证它，而 CSP 有一个模型检测工具 FDR，经过转化后的 Object-Z 形式规格说明可以用 FDR 来进行模型检查。

对转化后的 Object-Z 形式规格说明的模式检查与验证是我们今后的工作。

参考文献

1 Smith G. The Object-Z Specification Language. Advances in Formal Methods，Kluwer Academic Publishers，2000

2 Smith G. Reasoning about Object-Z specification. In：APSEC 1995. 489～497

3 Smith G. A fully abstract semantics of classes for Object-Z. Formal Aspects of Computing，1995，7(3)：289～313

4 Smith G，Derrick J. Specification，Refinement and Verification of Concurrent Systems-An Integration of Object-Z and CSP. Formal Methods in System Design，2001，18(3)：249～284

5 Olderog E R，Wehrheim H. Specification and (property) inheritance in CSP-OZ. Science of Computer Programming，2005，55(1-3)：227～257

6 Hoenicke J，Olderog E R. CSP-OZ-DC：a combination of specification techniques for processes，data and time. Nordic Journal of Computing，2002，9 (4)：301～334

7 Mahony B，Dong Jin Song. Blending Object-Z and Timed CSP：An introduction to TCOZ. In：Proceedings of the 20th international conference on Software engineering，IEEE，1998. 95～104

8 Sun Jing，Dong Jin Song. Specifying and Reasoning about Generic Architecture in TCOZ. In：APSEC'02，IEEE，2002. 405～414

9 Kim Il-Gon，Choi Jin-Young. Formal Verification of PAP and EAP-MD5 Protocols in Wireless Networks：FDR Model Checking. In：Proceedings of the 18th International Conference on Advanced Information Networking and Applications，2004

10 Lowe G，Roscoe B. Using CSP to Detect Errors in the TMN Protocol. IEEE Transactions on Software Engineering，1997，23(10)

11 Hoare C A R. 通信顺序进程. 周巢尘译. 北京大学出版社，1988

(上接第 258 页)

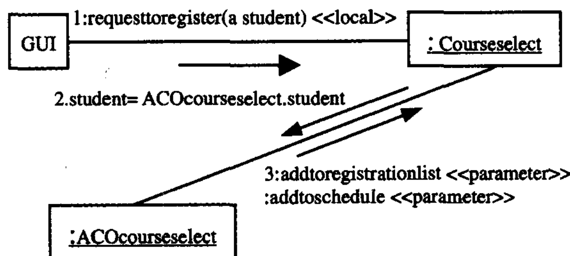

图 6 利用本文提出的方法处理过的“选课业务”协作图

程这两个类也不需要重新或修改类定义。根据我们提供的形式化方法，可以将上述规则表示成：

BR1：⟨*Courseselect*，*xh*=*student. xh*，*student*1⟩

BR2：⟨*Courseselect*，*Course* ⟨ ⟩ *student. selectedcourse*，*Course*1⟩

BR3：⟨*Courseselect*，*registrationlist. size*( )⟨*Course. getminimum*( )，*Course*2⟩

BR4：⟨*Courseselect*，*Course. getprerequisite*( )=*student. haspassedcoures*( )，*Course*3⟩，等等。执行完这些规则集，ACOBRMS 提供一个 ACOcourseselect 对象给应用，包含符合条件的学生表和课程表。

结论 快速灵活地响应需求变化是面向业务规则的信息系统分析方法的目标。本文从需求理解出发，提出了 ACO 这类业务规则，以及此类规则的形式化描述方法，并提供了基于 ACO 规则驱动的面向对象建模方法。实例表明，该方法较好地解决了由于类之间关系变化使得应用程序频繁变化的问题。

参考文献

1 GUIDE International Corporation. Guide Business Rules Project：[Final Report]. 1995

2 Herbst H. Business Rules in Systems Analysis：a Meta2model and Repository System. Information Systems，1996，21(2)：147～166

3 Valatkaite I，Vasilecas O. A Conceptual Graphs Approach for Business Rules Modeling. Computer Science，2003，2798：178～189

4 Valatkaite I，Vasilecas O. Automatic Enforcement of Business Rules as ADBMS Triggers from Conceptual Graphs Model. Information Technology And Control，Kaunas，Technologija，2004，2 (31)：36 ～ 42

5 Valatkaite I，Vasilecas O. On Business Rules Automation：The BR-Centric IS Development Framework. Computer Science，2005，3631：349～364

6 Kardasis P，Loucopoulos P. Expressing and organising business rules. Information and Software Technology，2004，46：701～718

7 Wan-Kadir W M N，Loucopoulos P. Relating evolving business rules to software design. Journal of Systems Architecture，2004，50：367～382

8 张维明. 信息系统建模. 北京：电子工业出版社，2002