基于 Object-Z 多态推理*)

文志诚　缪淮扣　孙军梅
(上海大学计算机学院　上海 200072)

摘　要　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，具有面向对象的特点：类、对象、继承、封装与多态等。用它可以精确描述大型软件需求规格说明，且能够进行严密的逻辑推理与验证。本文主要探讨了它的多态性推理，给出了相应的推理规则与方法，可以推理出 Object-Z 的多态行为，并着重体现推理的重用。

关键词　Object-Z，多态，前置条件，后置条件，推理，重用

Reasoning about Polymorphic Behavior in Object-Z

WEN Zhi-Cheng　MIAO Huai-Kou　SUN Jun-Mei
(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 thus has object-oriented characteristics. It improves the clarity of large specifications through enhanced structuring. Class and its relationship construction technology in Object-Oriented method(OO) are apt to describe large-scale and complicated system with Object-Z, based on mathematics logic and set theory, thus we can reason about its formal specification. One of the most important ideas underlying the object-oriented approach is polymorphism. This paper discusses how to reason about the polymorphic behaviors in Object-Z and presents its inference rule. With our approach, we can reason about the specific behaviors of subclass objects. Moreover, we take into account the reuse of reasoning emphatically.

Keywords　Object-Z, Polymorphism, Precondition, Postcondition, Reasoning, Reuse

1　前言

Object-Z[4]是形式规格说明语言 Z 的面向对象扩展，基于数理逻辑与集合论，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[5,7]，可以推理出规格说明应该有的性质与属性。Object-Z 同时具有面向对象的一些特点：类、对象、继承、封装与多态等。Object-Z 类的实例是抽象的对象；类的继承是同名变量或操作采取合并的原则。

面向对象的一个最重要的概念是多态性。在运行时，多态性允许我们使用一个子类对象来代替其父类的对象，应用到父类对象的操作时会自动调用定义在子类中同名的操作。我们在编写规格说明时，可以按一致的方式来处理它们。这样我们可以提高编程灵活性，父类给出同名操作一致定义的风格，而每个子类操作有其各自的特点。

标准处理多态性是按照行为子类型的方法[1~3]，子类对象保持着父类对象的行为。如果子类的对象代替父类的对象，在运行时，子类的行为不会出现意外，原先经过推理的规格说明不必重新推理。

本文是按照一般 Object-Z 继承的方法来推理它的多态性质。如果知道了多态变量所指的对象属于一个具体的子类时，想知道它的特有的行为，而不是父类操作所具有的普遍行为，因此有必要推理其“特有行为”，即子类操作的具体行为。文[7]研究了在程序级的面向对象的多态推理，给出了推理方法与规则，但没有考虑推理的重用。由于 Object-Z 操作不具备类型及参数传递等，本文基于抽象的规格说明级的多态性推理，对子类“特有行为”提出了推理方法，着重体现了推理的重用。

2　Object-Z 的规格说明及多态

Object-Z 是 Z 的面向对象扩展，类是它的一个重要的特征，有图 1 所示的形式。

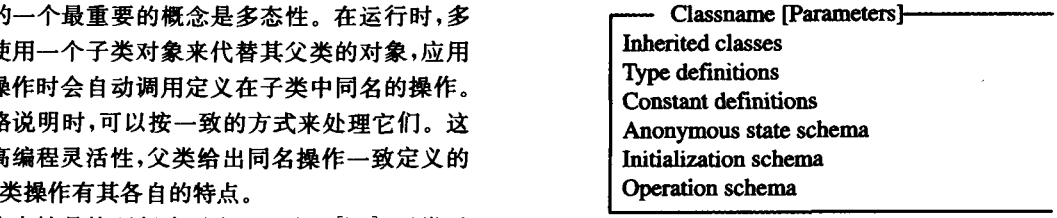

图 1　Object-Z 类模式

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(可有状态不变式对其约束)、无名状态模式、初始状态模式及操

*)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。

作模式。

设一个类的不变式为 Invariant(x)，其中 $x:X$ 是所涉及的状态变量及其类型；一个操作中可能的输入与输出变量为 input:INPUT，output:OUTPUT。根据 Object-Z 前置条件与后置条件的求法，状态不变式含蓄地包含在每一个操作中，因此有：

$$\text{OP. pre} = \exists x': X; \text{output}: \text{OUTPUT} \cdot \text{OP}(x, x', \text{input}, \text{output}) \wedge \text{Invariant}(x) \wedge \text{Invariant}(x') \quad (1)$$

$$\text{OP. post} = \text{OP}(x', \text{output}) \wedge \text{Invariant}(x') \quad (2)$$

其中 OP(x',output)表示操作部分，与后状态变量 x' 与输出变量 output 有关。后置条件只把含有后状态变量的谓词表达式保留，前置条件和后置条件表达式里包含前状态和后状态变量。由求法可知，前状态和后状态变量可能会出现在同一个谓词表达式中。如果一个操作施用于它的一个对象时，那么此对象的当前状态一定要满足它的前置条件。执行完毕，该对象的状态得到修正。

当一个变量被声明为多态时，那么被声明类的对象及子类的对象都可以赋给它，在运行时可以调用适合它的具体操作。在 Object-Z 中，一个变量被声明为多态，有形式：$o: \downarrow$ Ctype，其中 Ctype 是一个父类，类 Ctype 或它的子类的一个实例可以赋给多态变量 o。

3 多态及推理

一个状态变量可以声明为多态，被声明的类及其子类的对象可以赋给它。当动态执行一个操作时，会调用属于它自己的操作。而不一定是父类的操作，父类操作界面虽然一致，而表现的行为不必一致，体现了行为的多态性。本节的目的是，首先在不知道多态变量所指的对象具体类型时，可以推出此变量所指对象共同的属性；其次，如果知道了此变量所指的对象具体类型时，可以推出此对象特有的行为；再之，我们可以考虑推理的重用。

3.1 多态性推理规则

先定义几个符号：dtype. o 表示多态变量 o 所声明的类型，otype. o 表示变量 o 所指的对象实际的类型；currentstate(o)表示变量 o 所指对象现在的状态；oldstate(o)表示变量 o 所指对象在执行操作前的状态；$P(a,b)$表示 a 和 b 满足谓词 P；a_b 是指 a 在 b 上的投影，a 中只与 b 有关的部分。假设 D 是 B 的子类。

如果不知道多态变量 o 所被赋予的对象具体的类型时，那么对变量 o 所进行的推理只能基于它所定义的类型，我们有 Hoare 表达式：

$$\{\text{dtype. } o = \text{B} \wedge \text{OP}_\text{B}. \text{ pre}(\text{currentstate}(o)_\text{B})\}$$
$$o. \text{OPB}() \quad (3)$$
$$\{\text{OP}_\text{B}. \text{ post}(\text{currentstate}(o)_\text{B}, \text{oldstate}(o)_\text{B})\}$$

上面的意思是，如果只知道多态变量 o 所定义的类型是 B，不知道其所指的对象具体的类型时，在执行其操作 OP_B（类 B 中的操作）前，其所指的变量的状态在类 B 上的投影（即此对象只与类 B 有关的内部状态）满足操作 OP_B 的前置条件。执行完毕操作 OP_B，多态变量 o 所指的对象目前状态（执行完毕 OP_B 后的状态）和 o 所指的对象原先状态（执行 OP_B 前的状态），它们分别投影在 B 上，一定满足操作 OP_B 的后置条件。

如果我们知道了多态变量 o 定义的类型是 B，还知道所指的对象具体类型是 D，可以推出执行操作 OP_D 后所表示的具体行为，我们有：

$$\{\text{dtype. } o = \text{B} \wedge \text{otype. } o = \text{D} \wedge \text{OP}_\text{D}. \text{ pre}(\text{currentstate}(o))\}$$
$$o. \text{OP}_\text{D}() \quad (4)$$
$$\{\text{OP}_\text{D}. \text{ post}(\text{currentstate}(o), \text{oldstate}(o))\}$$

其中(3)、(4)式中的前置、后置条件是(1)、(2)式中所定义的。

3.2 推理的重用

面向对象程序设计体现了代码重用，用一个父类可以导出相关的子类，子类不必重写父类相关的代码。形式推理也一样，已经对父类进行了推理，当我们对其子类进行推理时，应尽可能不要废弃父类的推理而重新进行。只有父类的推理不适合于子类的推理时，我们才有必要对其子类重新进行推理，这是面向对象程序设计的推理重用。面向对象的多态行为，是对父类和子类具有同名的操作而言。

对于一个多态变量所指的对象，当知道了其具体的类型时要进行推理，应尽可能地重用先前的推理。这就是本节主要目的，我们下面分几种情况来讨论。

3.2.1 子类不变式同于父类不变式

如果子类不变式同于父类不变式，即子类没有对父类的状态变量重新约束，也没有引入新的状态变量。但子类可继承父类的操作、修改父类的操作、子类引入新的操作或父类的操作被改名后继承下来再在子类中重新定义同名的操作。

(1)对于被继承下来的操作，它们在子类中没有被修改，与父类同名操作是一致的，我们不必对它们进行重新推理，其原先的推理可以重用。

(2)对于被继承下来但在子类中被修改的操作，即在子类中定义了一个可与父类合并的同名操作。D 中操作 OP_D 的前后置条件可改写为：

$$\begin{aligned}&\text{OP}_\text{D}. \text{ pre} = [\exists x': X; \text{output}: \text{OUTPUT} \cdot \text{OP}_\text{D}(x, x', \text{input}, \text{output}) \wedge \text{Invariant}(x) \wedge \text{Invariant}(x')] \\ &\Leftrightarrow [\exists x_\text{B}': X_\text{B}; \text{output}_\text{B}: \text{OUTPUT}_\text{B} \cdot \text{OP}_\text{B}(x_\text{B}, x_\text{B}', \text{input}_\text{B}, \text{output}_\text{B}) \wedge \text{Invariant}(x_\text{B}) \wedge \text{Invariant}(x_\text{B}')] \\ &\wedge [\exists \text{output}_\text{BD}: \text{OUTPUT}_\text{BD} \cdot \text{OP}_\text{BD}(\text{input}_\text{BD}, \text{output}_\text{BD})] \\ &\Leftrightarrow \text{OP}_\text{B}. \text{ pre} \wedge \text{OP}_\text{BD}. \text{ pre}\end{aligned}$$

$$\begin{aligned}&\text{OP}_\text{D}. \text{ post} = [\text{OP}_\text{D}(x', \text{output}) \wedge \text{Invariant}(x')] \\ &\Leftrightarrow [\text{OP}_\text{B}(x_\text{B}', \text{output}_\text{B}) \wedge \text{Invariant}(x_\text{B})] \wedge \text{OP}_\text{BD}(\text{output}_\text{BD}) \\ &\Leftrightarrow \text{OP}_\text{B}. \text{ post} \wedge \text{OP}_\text{BD}. \text{ post}\end{aligned}$$

其中，x_B 表示在类 B 中的状态变量；OP_B 表示在类 B 中的谓词；OP_BD是指定义子类 D 中，且没有和父类 B 同名操作合并之前的部分，即有 $\text{OP}_\text{D} = \text{OP}_\text{B} \wedge \text{OP}_\text{BD}$，其它的表示类似。上式中，在 OP_BD部分没用新引入的状态变量，但可能有新的输入与输出变量。

如果我们知道 o 所指的对象状态满足类 B 中的一个操作 OP_B 的前置条件，那么只要验证它满足投影在 BD 上的前置条件 OP_BD. pre 即可。因此在 OP_B 上的后置条件保持着，不必再推理，而且又满足投影在 BD 上的后置条件 OP_BD. post。把这两个后置条件合取可以得到 OP_D 的后置条件，原先推理过的 OP_B 可以重用。规则可以写成：

$$\begin{aligned}&\{\text{dtype. } o = \text{B} \wedge \text{otype. } o = \text{D} \wedge \text{OP}_\text{B}. \text{ pre}(\text{currentstate}(o)_\text{B}) \wedge \text{OP}_\text{BD}. \text{ pre}(\text{currentstate}(o)_\text{BD})\} \\ &o. \text{OP}_\text{D}() \\ &\{\text{OP}_\text{B}. \text{ post}(\text{currentstate}(o)_\text{B}, \text{oldstate}(o)_\text{B}) \wedge \text{OP}_\text{BD}. \text{ post}(\text{currentstate}(o)_\text{BD}, \text{oldstate}(o)_\text{BD})\}\end{aligned}$$

因此，多态变量 o 所指的对象现状态在 B 上的投影满足 B 中操作 OP_B 的前置条件，只要验证：如果此状态在 BD 上的投影满足 OP_BD部分前置条件，则可以执行。操作 OP_D 的后置条件为在 B 上 OP_B 的后置条件，合取在 BD 上 OP_BD的部分后置条件即可。

(3)对于在继承过程中，父类操作被改名了而在子类中重

新定义了一个同名操作，这时我们只能对子类同名的操作进行重新推理。

3.2.2 子类不变式对父类不变式的加强但不扩充

在这种情况下，子类中没有引入新的状态变量，但对继承下来的状态模式中的状态变量加强了约束，与父类的不变式合取，状态空间变小了，这部分操作推理不便重用。

3.2.3 子类不变式扩充但不加强

在这种情况下，子类引入新的状态变量及其状态不变式（限制条件），但不约束被继承的状态变量，这两部分的状态不变式 $Invariant_B$ 和 $Invariant_{BD}$ 互不相关。

(1)对于被继承没有修改的操作，因子类不变式对父类的扩充，此类操作与新增状态变量及其限制条件无关，不必重新推理，可以重用原先的推理。

(2)对于继承父类的操作，在子类中定义了一个同名的操作，即有 $OP_D = OP_B \wedge OP_{BD}$，但 OP_{BD} 只与新增的变量有关，与继承变量无关，我们有：

$$\begin{aligned}&OP_D.\ pre = [\exists x': X; output: OUTPUT \bullet OP(x, x', input, output) \wedge Invariant(x) \wedge Invariant(x')]\\ &\Leftrightarrow OP_B.\ pre \wedge [\exists x'_{BD}: X_{BD}; output_{BD}: OUTPUT_{BD} \bullet OP_{BD}(x_{BD}, x'_{BD}, input_{BD}, output_{BD}) \wedge Invariant(x_{BD}) \wedge Invariant(x'_{BD})]\\ &\Leftrightarrow OP_B.\ pre \wedge OP_{BD}.\ pre\end{aligned}$$

$$\begin{aligned}&O_{PD}.\ post = OP(x', output) \wedge Invariant(x')\\ &\Leftrightarrow OP_B.\ post \wedge [OP_{BD}(x'_{BD}, output_{BD}) \wedge Invariant(x'_{BD})]\\ &\Leftrightarrow OP_B.\ post \wedge OP_{BD}.\ post\end{aligned}$$

这说明，只要 o 所指的对象目前状态满足 OP_{BD} 的前置条件，可以重用原先的推理，OP_D 后置条件是 OP_B 部分和 OP_{BD} 部分后置条件的合取。

(3)对于继承父类的操作，子类定义了一个同名的操作，但在此部分 OP_{BD} 中既重新约束父类 OP_B 中的变量又引进了新的状态变量，则只有重新加以推理。

(4)对于那些在继承过程中被改名的操作，在子类中重新定义了一个同名操作，我们只有对它重新进行推理，不能重用原先推理。

3.2.4 子类不变式既加强又扩充

在这种情况下，子类状态模式中既引入了新的状态变量，又在子类中对被继承的变量加强约束，因此不能重用原先的推理。

4 例

下面举一个具体例子来说明。设一个队列类 Queue [T]（图 2），其参数为 T，可以指定。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，其不变式 size = # items ∧ size≤max，初始状态队列为空。类中还定义了两个操作：Join 和 Leave。子类 CQueue [T]（图 3）继承父类 Queue [T]，并引入了新的状态变量 count，子类不变式只是对父类的不变式扩充不加强。在子类中定义了一个与父类同名的操作 Join，其中只引入了与新状态变量 count 有关的条件，但没有对父类操作 Join 的变量加以约束。

假设有 o: ↓Queue [T]。现就两个操作进行推理及相关的推理重用：o. Join 和 o. Leave。

对于不变式，有：

$$Invariant_{Queue[T]} \equiv size = \#items \wedge 0 \leqslant size \leqslant max$$
$$Invariant_{CQueue[T]} \equiv size = \#items \wedge 0 \leqslant size \leqslant max \wedge count \geqslant 0$$
$$Invariant_{Queue[T]CQueue[T]} \equiv count \geqslant 0$$

对于前置条件，有：

$$\begin{aligned}&Join.\ pre_{Queue[T]} \equiv \exists items': seqT \bullet items' = items \frown \langle item? \rangle \wedge size < max \wedge Invariant_{Queue[T]}(items, size) \wedge Invariant_{Queue[T]}(items', size')\\ &Leave.\ pre_{Queue[T]} \equiv \exists item!: T; items': seqT \bullet items = \langle item! \rangle \frown items \wedge size > 0 \wedge Invariant_{Queue[T]}(items, size) \wedge Invariant_{Queue[T]}(items', size')\\ &Join.\ pre_{CQueue[T]} \equiv \exists items': seqT; count': \mathbb{N} \bullet items' = items \frown \langle item? \rangle \wedge size < max \wedge count' = count + 1 \wedge Invariant_{CQueue[T]}(items, size, count) \wedge Invariant_{CQueue[T]}(items', size', count')\\ &Leave.\ pre_{CQueue[T]} \equiv Leave.\ pre_{Queue[T]}\\ &Join.\ pre_{Queue[T]CQueue[T]} \equiv \exists count': \mathbb{N} \bullet count' = count + 1 \wedge Invariant_{Queue[T]CQueue[T]}(count) \wedge Invariant_{Queue[T]CQueue[T]}(count')\\ &Leave.\ pre_{Queue[T]CQueue[T]} \equiv TRUE\end{aligned}$$

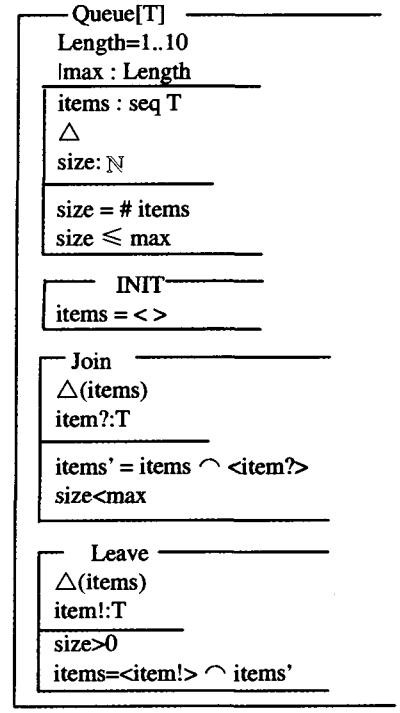

图 2 类 Queue [T]

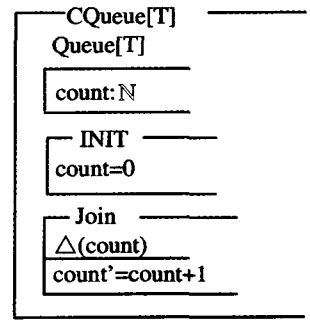

图 3 类 CQueue [T]

对于后置条件，有：

$$\begin{aligned}&Join.\ post_{Queue[T]} \equiv items' = items \frown \langle item? \rangle \wedge Invariant_{Queue[T]}(items', size')\\ &Join.\ post_{CQueue[T]} \equiv Join.\ post_{Queue[T]} \wedge count' = count + 1 \wedge Invariant_{Queue[T]CQueue[T]}(count')\\ &Join.\ post_{Queue[T]CQueue[T]} \equiv count' = count + 1 \wedge Invariant_{Queue[T]CQueue[T]}(count')\\ &Leave.\ post_{Queue[T]} \equiv items = \langle item! \rangle \frown items' \wedge Invariant_{Queue[T]}(items', size')\\ &Leave.\ post_{CQueue[T]} \equiv Leave.\ post_{Queue[T]}\\ &Leave.\ post_{Queue[T]CQueue[T]} \equiv TRUE\end{aligned}$$

(1)对于 o. Join，因为变量 o 被声明为多态，被声明的类为 Queue [T]，它的对象及子类 CQueue [T] 的对象可以赋给多态变量 o，适合情况 3.2.3(2)。

如果我们不知道多态变量 o 所指的对象的具体类型，对此操作推理时，只能用它所被定义的类型 Queue [T]，此对象目前状态在类 Queue [T] 上的投影在分量 items 及 size 上的

（下转第 256 页）

可以将编译生成的复合构件实体添加至构件库中，以便今后的构件复用。该算法的执行可以在复合构件的组装约生成之后或在整个应用(系统)运行之前。具体的执行步骤如图 6 所示。

相关工作及结论 构件组装是构件化软件开发的重点，软件的需求分解、体系结构、构件模型、构件粒度、构件评估标准、市场供应都对构件组装产生一定的影响，但其中最为核心的问题是如何采用有效的组装方法来支持构件化软件的开发。目前，构件的组装方法与技术有多种，根据组装的构件粒度以及应用环境的不同，组装技术主要集中在构件接口之间的静态连线组装、服务构件之间动态流程组装这两个方面。接口连接式构件组装是这两类组装的基本组装技术，而其中的构件组装形式化描述以及组装方案正确性的检验，将直接影响到构件组装的效果。本文主要针对构件接口连接式组装技术，采用形式化方法描述与推导了构件组装的相关问题，为构件组装的形式化分析以及组装正确性的检验打下了基础。用户可以根据应用需求，遵循体系结构的思想，自顶向下定义各层复合构件的组装规约。通过形式化分析并验证通过之后，采用构件组装规约映射算法 CJMA，自动生成源代码并编译产生新的复合构件，自底向上地快速搭建应用系统。通过这种技术可以规范基于构件的软件开发，有利于提高系统的开发效率和质量。

参 考 文 献

1 Clements P C. From Subroutines to Subsystems: Component-Based Software Development. In: Component-Based Software Engineering: Selected Papers from the Software Engineering Institute. IEEE Computer Society Press, 1996. 3～6

2 梅宏，陈锋，等. ABC：基于软件体系结构、面向构件的软件开发方法. 软件学报，2003，14(4)：721～732

3 Department of Computing. The Darwin Language. 3d edition. Imperial College of Science, Technology and Medicine, September 1997

4 Allen R, Garlan D. A formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, 1997, 6(3): 213～249

5 Shaw M, DeLine R, Klein D, et al. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, 1995, 21(4): 314～335

6 张世琨，张文娟，常欣，等. 基于软件体系结构的可复用构件制作和组装. 软件学报，2000，12(9)：1351～1359

7 Adamek J, Plasil F. Component Composition Errors and Update Atomicity: Static Analysis. Accepted for publication in the Journal of Software Maintenance and Evolution: Research and Practice, 2004

8 Visnovsky S. Modeling Software Components Using Behavior Protocols: [Ph D. Thesis]. advisor: Frantisek Plasil, Dec. 2002

9 Tansalarak N, Claypool K T. XCompose: An XML-Based Component Composition Framework. Third International Workshop on Composition Languages, 2003

10 任洪敏，钱乐秋. 构件组件及其开式化推导研究. 软件学报，2003，14(6)：1066～1074

(上接第 232 页)

值，如果它们满足操作 Join 的前置条件，则在后置条件中用 currentstate(o)加到后状态变量前，把后状态的标志"′"取消，用 oldstate(o)加到前状态变量前，用"."分开，我们有：

$$\{\text{dtype. o} = \text{Queue}[T] \wedge \text{Join. pre}_{\text{Queue}[T]}(\text{currentstate}_{(o)\text{queue}[T]})\}$$
$$\text{o. Join}_{\text{Queue}[T]}()$$
$$\{\text{currentstate(o). items} = \text{oldstate(o). items} \frown \langle \text{item?} \rangle$$
$$\wedge \text{currentstate(o). size} = \#\text{currentstate(o). items}$$
$$\wedge 0 \leqslant \text{currentstate(o). size} \leqslant \text{max}\}$$

现在可以考虑推理的重用。如果现在知道了 o 所指的对象的类型是 CQueue [T]，且此对象的状态满足操作 Join 在类 CQueue [T]上的前置条件，我们有：

$$\{\text{dtype. o} = \text{Queue}[T]$$
$$\wedge \text{otype} = \text{CQueue}[T] \wedge \text{Join. pre}_{\text{Queue}[T]}(\text{currentstate(o)})$$
$$\wedge \text{Join. pre}_{\text{Queue}[T]\text{CQueue}[T]}(\text{currentstate(o)})\}$$
$$\text{o. Join}_{\text{CQueue}[T]}()$$
$$\{[\text{currentstate(o). items} = \text{oldstate(o). items} \frown \langle \text{item?} \rangle$$
$$\wedge \text{currentstate(o). size} = \#\text{currentstate(o). items}$$
$$\wedge 0 \leqslant \text{currentstate(o). size} \leqslant \text{max}$$
$$\wedge \text{currentstate(o). count} = \text{oldstate(o). count} + 1$$
$$\wedge 0 \leqslant \text{currentstate(o). count}]$$
$$\Leftrightarrow \text{Join. post}_{\text{Queue}[T]} \wedge \text{Join. post}_{\text{Queue}[T]\text{CQueue}[T]}\}$$

即只要验证对象目前状态满足 Join. $\text{pre}_{\text{Queue}[T]\text{CQueue}[T]}$ 即可，如果它满足，那么它的后状态就满足 Join. $\text{post}_{\text{Queue}[T]}$ 和 Join. $\text{post}_{\text{Queue}[T]\text{CQueue}[T]}$ 部分后置条件，把它们合取就可以得到后置条件 Join. $\text{post}_{\text{CQueue}[T]}$，这就体现推理重用。

(2)对于 o. Leave，它适合情况 3.2.3(1)，按上述操作，我们有：

$$\{\text{dtype. o} = \text{Queue}[T] \wedge \text{Leave. pre}_{\text{Queue}[T]}(\text{currentstate(o)}_{\text{Queue}[T]})\}$$
$$\text{o. Leave}_{\text{Queue}[T]}()$$
$$\{\text{oldstate(o). items} = \langle \text{item!} \rangle \frown \text{currentstate(o). items}$$
$$(\text{currentstate(o). size}$$
$$= \#\text{currentstate(o). items} \wedge 0 \leqslant \text{currentstate(o). size} \leqslant \text{max}\}$$

如果知道了 o 所指的对象的类型是 CQueue [T]，我们有：

$$\{[\text{dtype. o} = \text{Queue}[T] \wedge \text{otype} = \text{CQueue}[T]$$
$$\wedge \text{Leave. pre}_{\text{Queue}[T]}(\text{currentstate(o)})$$
$$\wedge \text{Leave. pre}_{\text{Queue}[T]\text{CQueue}[T]}(\text{currentstate(o)})]$$
$$\Leftrightarrow [\text{dtype. o} = \text{Queue}[T] \wedge \text{otype} = \text{CQueue}[T]$$
$$\wedge \text{Leave. pre}_{\text{Queue}[T]}(\text{currentstate(o)})]\}$$
$$\text{o. Leave}_{\text{CQueue}[T]}()$$
$$\{[\text{oldstate(o). items} = \langle \text{item!} \rangle \frown \text{currentstate(o). items}$$
$$\wedge \text{currentstate(o). size}$$
$$= \#\text{currentstate(o). items} \wedge 0 \leqslant \text{currentstate(o). size} \leqslant \text{max}]$$
$$\Leftrightarrow \text{Join. post}_{\text{Queue}[T]}\}$$

这可以说明，对于子类的不变式扩充但不加强的情况下，子类原本继承父类的操作。虽然子类的不变式扩充了，引进了新的状态变量，但对这类操作是不影响的，我们不必对它进行重新推理。

总结及今后的工作 本文探讨了基于面向对象形式规格说明语言 Object-Z 的多态推理及其重用。根据 Object-Z 不变式、前置条件和后置条件，提出了推理的一般规则，既可以推理一般的行为，也可以推理子类特有的行为。知道了多态变量所指的对象普遍的行为，那么在有些情况下，可以推理它具体所指的对象特有行为时，可以考虑推理的重用。

因为面向对象形式规格说明语言 Object-Z 可以精确地描述大型软件系统，能够对这些形式规格说明进行精确的验证与推理，可以及时地发现需求规格说明中的错误。因此，今后的工作是研究对形式规格说明进行更有效推理的方法。

参 考 文 献

1 Soundarajan N, Fridella S. Behavioral subtyping and behavioral enrichment of multimethods Technology of Object-Oriented Languages and Systems. TOOLS, 2000. 105～114

2 王云峰，李必信，郑国梁. 面向对象 Z 的子类型继承和推理规则. 软件学报，2000，11(4)：481～487

3 Liskov B H. A Behavioral Notion of Subtyping. ACM, 1994, 16(6): 1811～1841

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

5 Sun Jing, Dong Jing Song. Specifying and Reasoning about Generic Architecture in TCOZ, APSEC, 2002

6 Soundarajan N, Fridella S. Inheriting and modifying behavior. Technology of Object-Oriented Languages and Systems, TOOLS, 1997. 148～162

7 Soundarajan N, Fridella S. Reasoning about polymorphic behavior. Technology of Object-Oriented Languages and Systems, TOOLS, 1998. 346～358