摘 要

并行程序的开发是并行计算中一个公认的难题。本文所研究的对象是静态并行程序的开发，即程序规范以程序的初始状态和终止状态之间的关系来描述程序的功能。这类并行程序的开发可以分成两个阶段：设计阶段和实现阶段。其中设计阶段从问题的规范出发，得到求解问题的并行设计，其困难源于应用问题的多样性、问题并行求解所需的创造性思维和计算机处理能力的机械性之间的矛盾。实现阶段从并行设计出发，得到可在具体结构的并行机上运行的并行程序，其困难主要源于并行计算机系统结构的多样性，以及由于缺乏统一的并行计算机模型而导致并行程序的复杂性、低效性、不可移植性等问题。

当前大多数并行程序设计模型本质上是对并行程序编码的研究，它们为并行程序的设计编码提供一个抽象的平台，但缺乏指导程序员在其所提供的平台上进行并行程序开发、推导和验证的方法，因而不能成为并行程序的开发模型。本文的创新性工作在于全面研究了并行程序开发两个阶段的内容与困难，提出了一种基于设计模式的并行程序开发方法和模型 DPaPD(Design-pattern-based Parallel Program Development)，该模型支持从问题规范到并行程序的整个程序开发过程。在设计阶段，本文引入了设计模式的思想，在此基础上针对 DPaPD 模型的思想和方法对设计模式的概念进行了重新的定义。通过设计模式的使用，不仅为问题的并行求解提供了算法设计方法和策略上的指导，而且提供了并行算法实现的抽象框架，使程序员在不具备并行领域知识的情况下，就可以获得求解问题的并行设计。在实现阶段，本文使用了一个抽象的并行模型 arb[51]作为并行设计转换为并行程序的中间模型，先将抽象的并行设计转换为 arb 抽象并行程序，然后转换为具体的共享内存或分布内存系统结构下的并行程序，这一模型转换的过程可以自动地实现。因此，DPaPD 模型不仅是一个并行程序设计的抽象平台，而更是一种系统的并行程序开发的方法和支撑环境。

本文的主要内容围绕 DPaPD 模型的提出，探讨了模型、方法、语言、系统、实现等有关问题。本文的具体贡献包括：(1) 提出了 DPaPD 模型的思想与方法，(2) 扩展了设计模式的概念，并定义了两类用于问题并行求解的设计模式，(3) 将 Z 语言并行扩展为 PaZ，并研究了从 PaZ 所描述的并行设计到 arb 抽象程序的精化与转换，(4) 对 DPaPD 系统进行了初步实现。本文的研究为并行程序的开发提供了一种简单、实用的方法及相应的系统支持，对解决并行程序开发的困难提供了一种可供借鉴的方法和途径。

关键词： 设计模式，并行程序开发，问题求解，规范描述语言，程序精化

ABSTRACT

Development of parallel programs is concerned as an acknowledged difficulty in parallel computing. The subject studied in this thesis is the development of static parallel programs whose specifications describe the relationship between initial and final states. Generally, development of parallel programs of this kind can be partitioned into two phases of designing and programming. In the first phase, parallel designs are gained from specifications. The main difficulty in this phase comes from the multiplicity of problems and the contradiction between the creativity required for solving problems and the mechanism of computers. In the second phase, parallel programs are gained from parallel designs. The main difficulty in this phase comes from the multiplicity of parallel computer architectures, and complexity, low-efficiency, transplantation of parallel programs caused by lack of a unified parallel computer model.

Most parallel programming models at present are studying on parallel programming basically. They offer abstract platforms for parallel programming and coding, but lack of method that can guide programmers to develop, derive and verify parallel programs, so they don't act as parallel program development models. This thesis creatively studies on the contents and difficulties of both phases of parallel program development, and proposes a design-pattern-based parallel program development method and model DpaPD (Design-pattern-based Parallel Program Development), which supports the whole process of parallel program development, from problem specifications to parallel programs. In the designing phase, the idea of design pattern is introduced, and its concept is redefined based on the ideas for DPaPD model. Through applying design patterns, it not only acts as a guide of algorithm design method and strategy for parallel problem solving, but also offers an abstract frame for the implementation of parallel algorithms, which means programmers can gain parallel design for the problem without any knowledge in parallel field. In the programming phase, an abstract model arb is used as a mid-model for transformation from parallel designs to parallel programs. An abstract parallel design can be first transformed into an abstract arb parallel program, and then transformed into a parallel program that runs on shared-memory or distributed-memory parallel computers. The transformation carries on automatically. Therefore, DPaPD model is not only an abstract platform for parallel programming, but also a method and supporting environment for systematic parallel program development.

This thesis discusses related subjects of model, method, language, system and

implementation centered on the proposal of DPaPD model. It contributes in: (1) proposing the idea and method of DpaPD model, (2) expanding the concept of design pattern and defining two design patterns of parallel problem solving, (3) raising a parallel extension of Z to get PaZ, and studying the refinement and transformation from PaZ to arb, (4) implementing DPaPD system tentatively. This thesis offers a simple but practical parallel program developing method and supporting system, and it can be a referential way for lightening the difficulty in parallel computing.

KEY WORDS: design pattern, parallel program development, problem solving, specification language, program refinement

第一章　　绪　论

1.1.引言

并行计算是当今计算机科学中的一个重要研究领域，而并行软件危机是阻碍并行计算发展与应用的重要原因，并行程序开发的复杂性、可靠性、可移植性等问题至今没有得到一个很好的解决，因此，如何开发高性能、可移植的并行程序成为并行计算领域中的一个关键问题。

尽管为了解决这一阻碍高性能并行计算发展的重要问题，计算机工作者们作了多方面的研究与探讨。然而，目前仍没有一种最好的方法，包括从显式的并行程序设计研究到自动并行的编译器研究。在并行程序开发中，最困难的莫过于那些需要与环境进行交互的并行程序的开发。然而即使是可以以初始状态和终止状态表示其规范的并行程序，其开发也是困难重重的。本文所探索的就是针对后者的开发方法。

我们关于并行程序开发方法的观点可扼要叙述如下，本文的研究是以这些基本观点为基础的。

程序的自动并行化是一个理想的目标，但其中存在难以克服的困难，因此是不切实际的。显式的并行程序开发是解决并行程序开发困难的切实途径。

并行程序开发的困难主要在于问题的并行求解，而不是并行程序设计语言。因此我们认为，开发并行程序的重点要放在问题求解和算法设计的方法上，而不是那些并行实现的细节上。正如[49]中所说的那样，“从事并行程序设计实践的人往往把精力耗费在为变量分配内存、为循环体寻求并行上，却忽略对问题本身的分析。其实能否并行的决定因素是应用问题本身。”

充分利用顺序程序开发的经验、方法和工具，特别是顺序领域中的问题求解、算法设计方法，这是简化并行程序开发的重要手段。并行程序的开发并不是一个完全独立的课题，它在很大部分与顺序程序开发是相同的，特别是与语言、体系结构无关的部分。因此，并行程序开发要避免一切从头做起，才能有效降低其复杂性。

在并行算法的设计阶段最大限度地开发出问题本身固有的并行性才是提高计算效率的根本手段。只有粗粒度的并行，才能具有高的计算通信比，而粗粒度的并行只能在算法设计阶段开发出来。

我们的研究旨在简化并行程序开发的过程，提高并行程序开发的高效性，探索一种切实可行的并行程序开发方法。

1.2.前期研究工作

我们的前期研究工作是围绕国家自然科学基金资助项目“若干新的算法程序设计和证明方法研究”“实用的软件形式化方法及其开发工具研究”、国家 863 计划项目“探索系统的算法程序设计和证明方法”、国防科工委军用共性软件预研项目“具有容错功能的 ADA 可重用部件库”等进行的。本文的研究是前期研究工作向并行计算领域的扩展。

1.2.1 顺序程序的开发方法研究

顺序程序的开发是从问题的规范描述出发，通过选取使用适当的算法设计方法进行算法设计，得到解决问题的算法，再通过一定的转换、编译过程得到可执行的程序或代码。如图 1.1 所示：

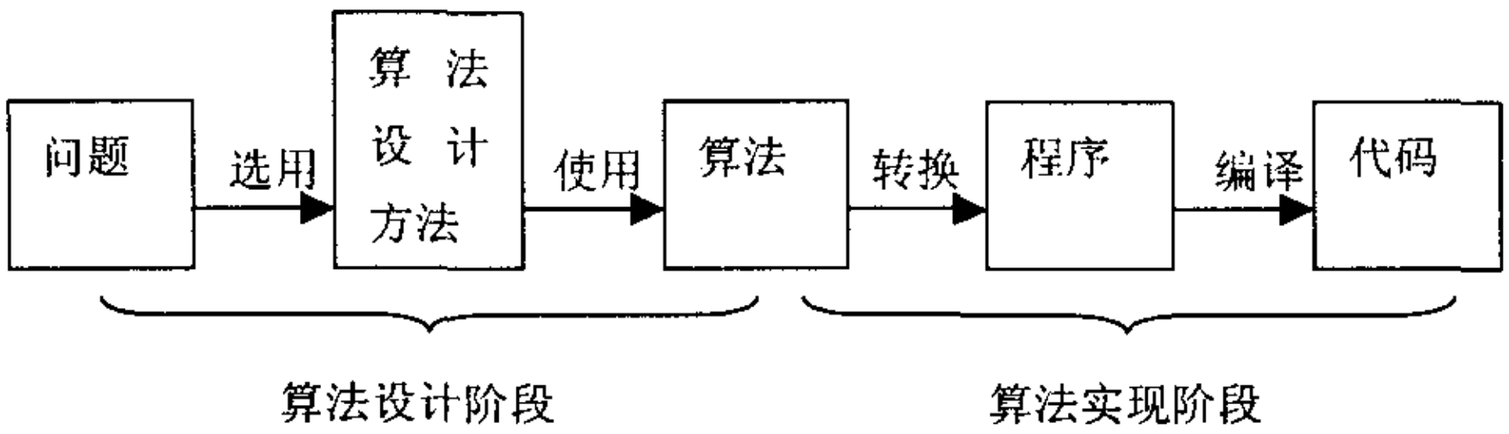


图 1.1　　顺序程序的开发过程

Fig. 1.1 Development of Sequential Programs

该过程可以分成两个阶段：算法设计阶段和算法实现阶段。算法的实现阶段是一个从算法语言描述的算法到程序语言描述的程序的转换过程，这一过程可以通过转换系统完全自动地实现。我们的前期研究工作实现了从一个抽象的算法描述语言到各种程序设计语言的转换系统。而算法设计阶段是从问题到算法的问题求解过程，即从“做什么”到“怎么做”。这一过程需要创造性的思维，不能完全地自动化。一般情况下，算法设计阶段的工作缺乏系统的支持，主要由程序开发人员承担。我们的前期研究工作通过长期以来对问题求解规律和算法设计方法的研究，使用软件重用的手段来简化其中人的工作，提高这一阶段的自动化程度。

1.2.2 软件重用的研究

在软件开发过程中，存在着大量的重复劳动。据有关专家统计，在１９８３年编制的所有代码中，只有少于１５％的部分是针对特定应用的，而８５％以上的代码是共同的。有关研究结果表明，在软件的开发过程中，在存在着生产和需求的巨大矛盾的同时，大量重复劳动的存在是不能忽视的。人们希望在新的软件开发环境中能利用以前开发的工作，减少重复开发的劳动量。为了提高软件生产效率，减少重复劳动，降低软件开发代价，就必须利用原有的软件设计和功能模块。

代码重用是软件重用的最普遍形式。然而，随着计算机应用领域的不断增大，应用问题越来越趋于多样化，仅通过代码重用不能达到软件重用在范围和程度上的需求。更为抽象的设计重用、结构重用成为必然趋势。我们对软件重用的研究重点在于设计重用的研究。在长期的程序设计实践中，人们根据程序设计中的经验总结，得到了一些传统的算法设计技术：贪心法、回溯法、分治法、分枝限界法、动态规划法等。这些算法设计技术各有其自身的特点，分别适用于解决不同的问题。算法设计的经验表明，尽管现有的算法多种多样，数量越来越多，但大多数算法都可以由常见算法设计技术产生。我们的前期研究工作通过分析由各种算法设计技术设计的各种算法的共同特征，以这些共同特征为基础，抽象出每一种算法设计技术的本质，形成可重用的算法生成器部件。对于给定的问题，用户给出适合于算法生成器所要求的输入，算法生成器即可生成解决问题的算法。

1.3. 并行程序开发方法研究概述

并行计算的研究已有二十多年的历史，计算机科学工作者对并行程序开发方法作了大量的研究，然而，并行应用程序的开发仍然是一件困难、易错的工作。从程序的开发过程所经历的不同阶段出发，可以把并行程序开发方法的研究分成三个领域：面向问题的方法、面向算法的方法和面向语言的方法[47]。

并行程序的开发可以分别有三个出发点，即可以从最初始的问题规范出发，可以从顺序的算法出发，也可以从顺序程序出发，分别研究与开发其中的并行性，从而得到可在并行计算机上执行的并行程序。

1.3.1 面向问题的并行程序开发

面向问题的方法是从问题的规范描述出发，开发并行程序的方法。它与顺序程序开发的出发点是相同的，只是在开发过程中要考虑问题潜在的并行性开发、并行求解和如何实现其并行性的问题。因此，概括而言，与顺序程序的开发类似，一种途径是采用形式化的程序综合与推导的方法，使用一系列的语义转换规则，将规范描述语言所描述的问题逐步求精，从而得到程序设计语言所表示的可执行程序。并行程序的形

式化开发，与顺序程序相比要复杂得多，现有的广为人知的并行程序的验证和开发系统模型有 CSP[40] ，UNITY[33,34]，action system [36]，shared-state models [37,38,39]等；另一种途经是采用并行算法的设计和实现技术，通过在某种并行计算模型上设计出解决特定问题的并行算法，然后实现该并行算法，将算法映射为可在特定并行计算机上执行的并行程序[54]。

1.3.2 面向算法的并行程序开发

面向算法的方法并不是指顺序算法的并行实现，也不是单一地考虑某一个问题的算法设计，而是研究问题的求解规律，即算法设计过程的规律性，根据现实世界算法问题中所固有的行为上的相似性对算法问题进行分类，使得同一类中的所有算法问题的求解符合某种特定的设计模式。这样，通过对设计模式的并行性开发，就能解决一类问题的并行程序开发(如[43,44,47]等)。

1.3.3 面向语言的并行程序开发

面向语言的方法可分为两类。一类是显式并行性开发方法，即研究并行的语言或模型，通过在语言或模型中提供一些并行性定义的机制，程序设计人员使用这些机制来刻画并实现算法问题中的并行性。对并行语言与并行模型的研究，包括传统的PRAM, PVM, MPI 等曾在并行计算领域占主体地位的低层次模型，Haskell、UNITY 等高抽象程度的模型，BSP、LOGP 等集中了大量研究工作的中等抽象程序的模型等(详细可见综述性文章[35,57,64])。另一类是隐式并行性的开发，即通过在语言的编译器中加入识别程序中所存在的可并行单位的功能，使得经编译后产生的代码为并行的代码(如[45]等)。

1.3.4 三类研究的比较

在对并行程序开发方法的研究中，大量的研究集中于面向问题的方法和面向语言的方法，而对面向算法的方法研究较少。到目前为止，尽管各个领域都取得了一些成果，但尚未有一种公认的切实可行的开发易编写、可移植、高性能的并行程序的方法。

通过形式化方法推导程序一直是程序设计的理想，国际国内都非常重视这方面的研究，尽管这一方法在理论上已日益成熟，但真正应用起来往往仅限于开发一些小程序，而且其方法难以为一般的用户所运用，而并行程序的复杂性又增加了形式化开发的难度，因此，并行程序的形式化开发离实际的应用仍有一段距离。

并行算法的设计方法的研究一直伴随着并行计算研究的历史，尽管现已开发了许多问题的并行算法，但由于并行算法设计的复杂性，至今尚无一套普遍适用的系统的

设计方法。并行算法的设计不仅要考虑问题的特性，还要考虑其将运行于的并行计算机的特性，且在特定并行模型下设计的并行算法往往只能实现于某一类特定结构的并行计算机，算法的通用性差，随着新的并行计算机体系结构的日益出现，对同一问题需要不断设计新的相应的并行算法。

代码的并行化编译一般要经过相关性分析、优化和代码生成三个阶段。它的主要思想是通过对源代码中数据流和控制流的分析，寻找出一些不具有相关性的片段，将它们并行化。这一方法从源代码出发，一开始就将变量等程序实现方面的细节考虑进去，这些细节往往掩盖了问题本身具有的并行性，因此必然带来更高的复杂性，而且，并行化编译开发的是细粒度的并行，程序效率难以提高，程序并行化的程度也受到原顺序程序的局限，一个好的并行程序往往并非来源于一个好的顺序程序。

显式并行化是近年来研究较多的方法，它通过在语言或模型中提供并行性描述和实现的机制，而将并行性开发的任务完全交给了程序员。由于缺乏系统的开发方法，从问题的求解到并行实现的整个过程对程序员而言是一个复杂与困难的工作。而且，由于程序员水平与风格的差异，软件的质量与效率往往难以得到保证。

面向算法的并行程序开发是一个相对研究较少的领域，从某种意义而言，它也是一种隐式并行化的工作，但与并行化编译有本质不同，它将并行化的工作从语言一级提高到算法一级，减少了琐碎的实现细节的影响，而更关注的是算法问题类本身固有的并行性，同时，通过对算法问题的分类与抽象，可以利用顺序算法的设计经验，构造特定的算法结构模式，预先开发出其中的并行性，这样一来，使程序设计人员可以不必显式地进行并行开发，从而简化了并行程序的开发过程，降低了并行程序开发的复杂性，是一种切实可行的并行程序开发方法的研究途径。

1.4.与本文相关的研究领域

1.4.1 顺序程序的自动并行化

并行化编译是一种以并行程序的自动化开发为目标的方法。从顺序程序代码入手，通过自动并行性检测和转换、编译，将顺序程序转换成并行程序或直接编译成并行代码，就是并行化编译。它是并行程序开发的理想，并行化编译器将现有的顺序程序直接编译成并行代码，这样程序员就完全不需任何附加的劳动。然而实践表明，顺序程序的自动并行化是不切实际的。其原因除了效率问题外，更主要在于顺序的程序代码往往隐藏了一些问题本身固有的并行性，编译器即使通过复杂的分析技术去识别、探测，仍然很难将这些并行性完全发掘出来，另外，由于对算法的不同实现可得到不同的程序代码，从而使得一个问题对应多个不同的并行解，这说明自动并行化所得的并行解并不能真正反映出算法的本质，况且，顺序代码的并行化编译本身还存在许多尚未解决的难题。

我们的方法是从问题的算法设计阶段就开始入手，通过问题求解和进一步的转换得到并行程序。由于从一个更高的抽象层面考虑这一问题，因而有利于开发粗粒度的并行，从而提高并行性能；有利于及早发现并行，避免并行性的隐藏；有利于利用算

法设计的规律，使一个并行化方法适用于一类问题；有利于任务并行性的开发等。需要指出的是，我们的方法也不同于一般的显式并行程序设计。我们采用的是一种更为通用、一般的方式，而不是针对特定的应用问题研究其算法程序的开发，其次，我们并行程序开发的重点在于问题的求解和算法的设计，而不是程序编制。

Nothing Explicit, Parallelism Implicit
Dynamic Structure
Higher-order Functional-Haskell
Concurrent Rewriting– OBJ, Maude
Interleaving – Unitv
Implicit Logic Languages –PPP, AND/OR, REDUCE/OR, Opera, Palm, com\ncurrent Constraint languages
Static Structure
Algorithmic Skelo\etons –P3L, Cole, Darlington
Static and Communication - Limited Structure
Homomorphic Skeletons – Bird-Meerterns Formalism
Cellular Processing Languages –Cellang, Carpet, CDL, Ceprol
Crystal

Parallelism Explicit, Decomposition Implicit
Dynamic Structure
Dataflow – Sisal, Id
Explicit Logic Languages –Concurrent Prolog, PARLOG, GHC, Delta-Prolog, Strand
Multilisp
Static Structure
Data Parallelism Using Loops – ?Fortran Variants, Modula 3*
Data Parallelism on Types – Psetl, parallel Sets, match and move, Gamma, PEI, APL, MOA, Nial and AT
Static and Communication-Limited Structure
Data-Specific Skeletons – scan, multiprefix, Paralations, dataparallel C,NESL, CamlFlight

Decomposition Explicit, Mapping Implicit
Dynamic Structure
Static Structure
BSP,LogP
Static and Communication - Limited Structure.

Mapping Explicit, Communication Implicit
Dynamic Structure
Coordination Languages – Linda, SDL
Non-message Communication Languages – ALMS, PCN, Compositional C++
Virtual Shared Memory
Annotated Functional Languages – Paralf
RPC–DP, Cedar, Concurrent CLU, DP
Static structure
Graphical Languages – Enterprise, Parsec, Code
Contextual Coordination Languages–Ease, ISETL-Linda, Opus
Static and Communication-Limited Structure
Communication Skeletons

Communication Explicit, Synchronization Implicit
Dynamic Structure
Process Networks – Actors, Concurrent Aggregates, ActorSpace, Carwin
External OO – ABCL/1, ABCL/R, POOL-T, EPL, Emerald, Concurrent Smalltalk
Objects and Processes – Argus, Presto, Nexus
Active Messages – Movie
Static Structure
Process Networks – Static dataflow
Internall OO – Mentat
Static and Communication - Limited Structure
Systolic Arrays – Alpha

Everything Explicit
Dynamic Structure
Message Passing – PVM, MPI
Shared Memory –FORK, Java, thread Packages
Rendezvous – Ada, SR, Concurrent C
Static Structure
Occam
PRAM

图 1.2　并行计算模型比较 [35]

Fig. 1.2　Parallel Computing Models [35]

1.4.2 并行程序开发模型

并行程序开发与顺序程序开发的最根本区别，也是其复杂性的根源在于并行计算机结构的多样性。串行计算机体系结构可以使用一种统一的模型“冯.诺依曼模型”来表示，而并行计算机的体系结构尚没有统一的模型。因此，对并行程序开发模型的研究一直是并行计算领域的一个研究热点。一个程序设计模型简单地说就是一个抽象机，它向上面的程序设计级提供某些运算、结构，同时向下在各种体系结构上将之进行实现。 目前国际上存在许多流行的并行模型，但尚未有一种公认理想的并行模型。并行模型有不同的抽象层次，[35]中将并行模型的抽象级别分成六级，并将现有的并行模型依据其抽象级别和一些其他的属性进行了详细的分类。如图 1.2 所示。

最好的并行计算模型莫过于程序员完全不需要了解并行细节的模型。换而言之，这类模型必须具有较高的抽象程度，才能隐藏所有的并行细节。因此，我们的兴趣是抽象的并行模型。在抽象并行模型中，一般有两类方法。一类是动态结构的模型，如高阶函授数模型 Haskell，转换系统模型 Unity，逻辑式程序模型 AND/OR 等。另一类是基于预定义模块结构的模型，这些模块结构嵌入在编译器中，从而可以以实现的模块逐个代替抽象程序中相应的运算或结构，如基于算法骨架(Algorithm Skeleton)的 P3L,基于同态骨架(Homomorphic Skeleton)的 BMF 模型[83]等。本文侧重于对后者的研究，但不局限于此，因为我们的研究内容不仅仅在于并行程序开发的模型，而更在于一种系统的并行程序开发的方法。

1.4.3 并行领域中的设计模式

我们的研究是以设计模式的思想为基础的，事实上，在许多计算方法中都运用了设计模式的思想。在面向对象的设计方法学领域中，设计模式用于描述一些系统地、普遍地重复出现的设计问题的解决方案，在并行计算领域中，设计模式则描述重复出现的并行计算模式及其解决方案。例如，流水线并行、分而治之方法、各种拓扑下的数据并行等。尽管设计模式的思想最先是用于面向对象软件工程领域，然而现在，这个概念已广泛应用于对软件设计公共策略的形式说明。其中，面向对象的可重用设计模式更是一种获取广泛的软件结构重用的有前景的技术，它刻画部件静态和动态的结构及其之间的合作关系，成功地解决商业数据处理、电子通讯、图形用户界面、数据库、分布式通信软件等软件构造中的问题[52]。这些模式在比源代码更高的级别上通过表示各部件的结构及其合作关系来辅助可重用部件及框架的开发。E. Gamma 对设计模式的定义为:"description of communicating objects and classes that are customized to solve a general design problem in a particular context"。设计者可通过在程序中使用模式来解决他们程序设计结构中的一些普遍性问题，就如同在程序中使用算法和数据结构来解决特别的计算问题一样[55]。

在并行程序设计领域中，以设计模式的思想为基础的并行程序设计方法研究是并行程序模型研究的一个重要分支。其中主要有分别以算法骨架，并行程序设计典型(parallel programming archetype)，结构骨架(architecture skeleton)等概念为基础的研究工作。

算法骨架是一种结构化并行程序设计的方法，系统通过提供骨架使用户不需显式地处理如通讯、同步等低层的并行特性，同时从并行程序的死锁、非确定性等问题中

解脱出来。一个骨架是一个算法的抽象，是对重复出现的算法和通信模式的精确严格的定义,它对一系列的应用而言是公共的，并且可以并行地实现。最初提出算法骨架并完整地实现了一个基于骨架的系统是 1989 年的 Murry Cole[50]。德国 Passau 大学与意大利 Pisa 大学一直继续着这方面的工作，他们以骨架的思想为基础，开发了 P3L 系统。系统中定义了一些基本的骨架(如 farm, pipe, map, reduce, loop 等)和一些转换规则，是一个典型的基于骨架的系统。

典型是 Chandy 提出的概念[28,51]，一个并行程序设计典型是一个刻画一类具有相似计算与通信结构的问题的共同特点的抽象，它通过提供抽象的设计方法和具体的代码库来帮助并行应用的开发。它的主要思想是将问题进行分类，使用抽象来描述一类问题共同的计算结构和通信结构，并将一个问题类的计算结构和并行化策略结合起来，产生一个数据流模式及通信结构模式。这些成分组合起来，就构成一个并行程序设计典型。由于典型刻画的是一类问题的共同点，因此可以为其开发相应的转换规则，使之可以将属于该典型的所有程序并行化。目前，对并行程序设计典型及其系统的研究主要在加利福尼亚大学和加利福尼亚技术研究所，他们构造了基于典型的并行程序开发和转换系统。该系统的主要功能是将顺序程序、或由系统所定义的语言描述的程序，通过使用由并行程序设计典型所指导的一系列保持语义的转换，得到功能上等价的并行程序。

另一研究并行程序中的设计模式的机构是加拿大 University of Waterloo。他们定义设计模式(design pattern)为描述重复出现的并行程序设计问题及其可重用的解[58,44]，设计模式实现为一个可重用的代码骨架，用以支持并行应用的快速可靠的开发。最近，在设计模式与 DPnDP 系统的基础上，他们又提出了结构骨架的概念[56]。

此外，相关的研究工作还包括 Basel 并行程序设计方法、设计模式的识别与探测方法、模式选取方法、并行程序分类方法等。

本文的工作建立在我们对设计模式和并行程序开发方法的理解基础上，与上述研究工作一样，本文中的设计模式也有其特定的定义、使用与实现。所不同的是，本文的设计模式更为抽象，这一点一方面体现在其内容上，为问题并行求解的方法，完全不涉及并行实现的细节；另一方面体现在其实现与使用上，我们仅在问题的求解阶段使用设计模式，因而设计模式自身不需预先实现为对应的并行代码，而是用于产生特定问题的并行设计，然后再通过两个阶段的精化、转换得到最后的并行程序实现。

1.4.4 软件重用

当今软件领域的一个巨大挑战是对大规模、复杂与高可维性软件系统的快速有效的开发，而基于构件的软件开发方法是面对这一挑战的一个研究热点与趋势。基于构件的软件开发方法的基本思想就是通过对已有软件中的可重用构件的汇集来构造一个新的软件系统，这样可以大大降低软件开发的复杂性，并利用了已有的软件资源，减少软件开发的时间与花费。然而，这一方法只在一些易于理解的应用领域得到了高效的发挥，它所存在的最严重问题之一就是在构件中缺乏设计的信息[48]。而设计模式将构件实现与软件构造中隐藏的设计思想与求解策略发掘并描述出来，它们将包含领域知识的软件工程专家的思想封装在些概念化的构造块中，在此基础上可以构造出更复杂、更灵活的软件设计。因此，设计模式的方法将软件工程的重点从基于构件的

实现转移到基于构件的问题求解上来。

直接在低层次的并行程序开发工具如 MPI 上开发并行程序需要付出大量的时间和努力，而且，使用这些低层次的工具直接开发并行程序必然导致程序中充满了同步、通信等专用代码，这增加了并行程序的复杂性。而当应用程序要用于不同的并行体系结构上时，所有的程序设计工作都要重新进行，而这样的工作正在被其他的并行程序工作者重复地进行着。因此，我们的研究旨在体现在并行程序设计的过程中的设计重用思想，从而提高并行程序设计与开发的效率与速度，达到降低并行软件开发成本的目的。

1.4.5 并行算法设计

我们研究并行程序的开发方法，是以并行算法的设计和问题的并行求解方法为切入点和重点的，但这并不表明我们的研究内容就是并行算法设计方法。一般意义上的并行算法设计是指在特定结构的并行计算机上为特定的应用问题设计高效的算法。这个过程一般可以分成四个阶段：划分(Partitioning)、通信(Communication)、聚集(Agglomeration)、映射(Mapping)。其中划分阶段将特定问题中的计算和数据划分成小的任务，不考虑处理器数目，主要任务是发掘并行性；通信阶段为任务定义适当的通信结构和算法；聚集阶段对以上所得的任务与通信结构进行评估，根据结果对任务进行必要的合并，以减少开发代价，提高性能；映射阶段将任务分配到处理器上。这种并行算法设计的一般方法称为 PCAM 方法。

PCAM 方法是一个一般性的并行算法的设计方法，然而，使用这一方法进行并行算法设计，设计人员必须面对和处理所有的问题，包括从应用问题的规范到并行体系结构，从问题求解到性能评估。其原因在于这一方法没有运用已有的顺序算法设计经验，而是从零开始。我们的研究是建立在对已有的问题求解和算法设计经验的基础之上，通过对其进行并行化，来帮助和简化程序设计人员的工作，而对并行算法设计方法的研究只是本文系统的并行程序开发方法研究中的一个组成部分。

1.4.6 形式化的软件开发方法

用形式化方法开发软件，被认为是提高软件可靠性和生产效率的革命性途径，是实现软件自动化的关键。无论过去和现在，这一方向都吸引了国内外大批计算机科学工作者，包括象 Dijkstra, Hoare, Milner, Gries, C. Jones 等著名计算机科学家。大量的人力、物力都倾注在这一研究方向上，出现了多种多样的软件形式化开发方法，如程序计算、程序变换以及以此为基础产生的 Z，VDM，RAISE 等方法和基于有限状态自动机、主要用于硬件系统正确性验证的 Model Checking 方法。但是这些技术还远没有广泛地被软件产业界接受[76,42]，事实上有些方法以及以这些方法为基础而建立的软件开发工具和自动生成系统只能处理和生成一些玩具式程序(toy-style program)[77]。针对这种情况，Z 方法的主要发明者、法国学者 J-R Abrial 提出了 B 方法[78]；面向对象 BOOCH 方法的发明者 G. Booch 提出了用于 OO 分折和设计的统一模型语言 UML[79]。

与顺序领域一样，并行程序和并行软件系统的形式化开发也是人们所关注的热点。抽象并行模型的研究往往伴随着相应的形式化开发方法的研究。

我们的研究通过算法程序的设计、精化与证明，以提高算法程序的正确性和软件的可靠性为出发点和目标。我们虽然研究的是并行程序的开发，但其开发思想和过程借鉴了自动程序设计的软件开发过程，即通过计算机辅助的方式对问题的功能规范进行逐步的求精和变换，最后得到具体的实现。

1.5 其他方法和模型

本文工作的主要目标是为并行程序的开发与验证提供一种通用、系统的方法和模型。在此我们简单介绍一些流行的、且与我们的研究有相似之处的一些研究工作。

Darlington 的骨架模型[50,59]的基本思想是采用高阶函数来描述并行程序的行为而非仅仅描述并行程序的语义。该模型建议针对不同体系结构提出相应的可高效实现的骨架组，而程序员根据目标体系结构不同相应选用适当的骨架描述；程序的移植性通过不同骨架间的代数变换予以保证。该模型实际上是一个面向体系结构的程序设计思想，这在一定程度上限制了基于此方法程序的移植性和可伸缩性。类似的系统还有 Skillicorn 基于 BMF 原语的数据并行系统[41,32]。

Unity 系统类似于 GAMMA 模型的基本思想，但是该系统的形式化和可编程性较差，仅提供了一个基于逐步求精的并行程序设计思想。

BSP 模型是一个计算机语言和体系结构之间的桥梁[4,31]。它以下列三个参数描述的分布存储的多计算机模型：①处理器/存储模块；②处理单元间的点对点信息传输网络；③以时间间隔 L 为周期的同步控制器。所以 BSP 模型将并行机的特征抽象为三个定量的参数 p,g,l，分别对应于处理器数、通讯网络吞吐率、全局同步的时间间隔。该模型中计算由一系列用全局同步分开的周期为 L 的超步所组成。在每个超步中每个处理单元均执行局部计算，并通过网络接收和发送信息；然后作一个全局检查，以确定是否所有处理器均完成各自的计算；若是，则前进到下一超步，否则下一个 L 被分配给未曾完成的超步。

类似的模型还有 LogP 和 CCC 模型，这些模型均具有类似的超步结构；区别在于对通讯和网络拥挤等问题处理有所不同。这些模型设计的主要目标是研究更符合实际的性能评价模型，确切地说这些模型仅是执行模型而非并行程序设计开发模型。这些模型对于研究并行计算机的体系结构和依据算法的并行复杂性进行并行程序优化设计有指导性的意义。缺点是缺乏指导程序员如何进行并行程序设计、开发和验证的能力。

PRAM 模型[30]最初作为一种理想的并行计算模型，是假定一个容量无限大的共享处理器，同时有有限台功能相同的处理器，且每台处理器都有简单的算数运算和逻辑判断功能，在任何时候各处理器均可通过共享存储器单元交换数据。早期的研究基本上是基于该模型。PVM[63]、MPI 是基于消息传递的并行程序设计模型，该模型将异构的计算机用做并发计算资源，通过函数库的形式提供并行编程所需的远程任务创建和进程通信等原语。这些模型的抽象级别很低，因此程序员必需考虑所有并行实现的细节。

1.6 本文的结构

本文主要从方法、模型、语言、实现等几个方面讨论了基于设计模式的并行程序开发方法所涉及的问题。

在第二章中，本文提出了一个基于设计模式的并行程序开发方法的思想和模型DPaPD(Design-pattern-based Parallel Program Development)。通过将设计模式的概念引入并行程序开发领域，并对这一概念进行了扩展，使之可用于表示并行化的算法设计方法。通过使用特定的规范描述语言对应用问题进行形式化的描述，使用设计模式帮助问题求解和并行算法设计，再通过一定的精化和转换得到抽象的并行程序，经过进一步的转换得到具体并行体系结构下的并行程序。本章对这一模型的思想、基本概念、所涉及的问题进行了详细的阐述。

在第三章中，本文对基于设计模式的并行程序开发模型 DPaPD 中的规范描述语言 PaZ 的语法和语义进行了详细的说明。该语言是 Z 语言的一个并行扩充版本，用于在 DPaPD 模型下描述应用问题的形式化规范、设计模式和并行设计。

第四章是对设计模式概念和有关问题的讨论。包括我们对设计模式的新的定义、设计模式的描述方法研究、设计模式的选择方法研究、在我们模型下的设计模式的具体内容、设计模式的使用方法、设计模式的一些实例等。

在第五章中，我们介绍了 DPaPD 模型中从 PaZ 语言到抽象并行模型 arb 的一系列精化规则，这些规则为 DPaPD 系统中 PaZ-arb 转换器部件的实现基础。

第六章给出了 DPaPD 模型下一些完整的具体实例的并行程序开发。

第七章讨论了 DPaPD 模型的实现情况和测试结果。

最后在第八章对本文的工作作出了总结并讨论了进一步的研究方向。

第二章　基于设计模式的并行程序开发模型 DPaPD 概述

2.1 引言

并行计算机是适应海量计算而产生的计算机系统。典型的并行计算有以下几种：基于向量和阵列机上的向量计算，主要用于科学计算和工程设计；基于共享存储器的多处理机上的计算，可用作服务器并同时进行并行处理；基于分布式存储器的大规模并行机上的大规模并行计算；工作站机群上的并行计算。随着工作站性能的迅速提高和价格的日益下降，以及高速网络的发展，利用工作站群作为高速并行计算的平台已日益受到广泛的重视和欢迎，同时也为并行计算、高性能计算的应用提供了巨大的潜力和应用的空间。于是，如何充分利用这些已广泛分布于社会各个角落的并行计算的硬件资源，设计使用于并行计算机系统的各类软件就成为一个重要的课题。

并行程序的开发是并行计算中的一个公认的难题。其根本的原因在于并行计算机系统结构的复杂性和多样性，由于缺乏一个如同串行计算机系统的“冯• 诺依曼模型”那样的统一的计算模型，并行程序的设计开发无法在一个统一的标准下进行，从而导致了其复杂性、低效性、不可移植性等问题[46]。而并行程序开发难的另一个重要原因，在于缺乏系统的开发并行程序的方法和有关的辅助工具。现有的并行计算模型基本上是语言级的，它们为并行程序设计提供一个抽象的平台，但缺乏指导程序员在其所提供的平台上进行并行程序开发、推导和验证的方法。如何在一个给定的抽象并行模型的平台上设计出符合问题要求的并行程序完全取决于程序员的技巧和经验，程序的正确性、开发效率都难以得到保证。

本章基于上一章中我们对并行程序开发设计的基本观点，基于我们在顺序程序开发方法的研究工作，提出了一种系统的并行程序开发方法和模型 DPaPD。在下面的小节中，我们将对 DPaPD 的思想方法、基本概念和相关的问题进行一个概述。

2.2 程序开发的三级结构

什么叫做程序？程序是在指定计算机环境下实现一特定问题求解的机械过程的陈述。所谓特定问题求解，就是关于该问题求解的数据结构与算法。

程序设计的实质是关于问题求解机械化的数学思维活动，正如 Hoare 所说，计算机就是数学机，程序设计是数学思维活动，而从另一方面看，计算机软件的生产方式又是工程的，是一种大工业专业化分工的有组织的集体劳动。在程序与程序设计的特

点中，充满了各种矛盾，从而导致了程序设计的种种困难。其中一个尖锐的矛盾就是问题求解的创造性与计算机能力的机械性的矛盾，解决该矛盾的办法是使问题求解数学化，也就是说要从方法论的高度来加以解决，才能解决更广泛更复杂的问题。因此，以程序设计方法学的观点来从事程序设计，才能简化程序设计的困难。从程序设计方法学的观点来看，程序设计是从问题的规范描述开始，直到得到某种计算机语言描述的求解该问题的程序的过程。这个过程有两个内容，一是得到求解问题的算法。这一过程是面向问题的，通常算法的描述可以采用一种抽象的描述语言，甚至可以是规范描述语言，这样，在考虑问题求解时，就可以不必考虑到程序设计语言复杂的数据结构和具体的描述方法，而是将注意力放在如何解决问题上，即“怎么做”；第二个内容是如何使用指定的计算机语言来实现这一解决问题的算法。计算机语言相对于算法描述语言更为具体，低级，更接近于计算机，通过完全自动的编译，可以转换成机器语言，因此这一阶段的注意力放在使用计算机语言中提供的语言结构来实现比较抽象的算法。例如，使用链表或数组(具体的数据结构)来实现集合(比较抽象的结构)。这就是通常所说算法设计阶段和算法实现阶段。然而，在实际的程序设计中，人们往往将这两个阶段混在一起，从一开始就考虑如何设置变量，分配内存等细节问题，从而忽略了对应用问题本身的分析。如果说在顺序程序的设计中，这样做无伤大局的话，那么在并行程序设计中，由于底层并行计算机体系结构，并行程序设计语言本身机制的多样化与复杂性，如果以这种习惯进行并行程序的设计开发，其难度是可想而知的。因此，我们的观点是，只有严格按照程序开发的三级结构来进行程序设计，即规范级，设计级，实现级，才能有效地解决程序开发的困难。这种方法又称为程序设计的两阶段方法，其中，规范级到设计级为算法设计阶段，设计级到实现级为算法实现阶段。

2.2.1 规范级

程序规范是对所欲求解的问题的描述，即程序需要“做什么”。本文中的程序规范仅指程序功能，不包括处理速度、执行时间、响应周期等与时间有关的性能指标。程序规范通常是软件工程第一阶段-------需求分析的结果。

2.2.1.1 规范与正确性

程序的正确性是程序设计的基本条件。遗憾的是，目前开发的软件系统中，真正能保证正确性的寥寥无几，而没有问题的精确描述，正确性就等于是空谈，形式规范是软件工程中对问题的精确的规范描述的定义。然而，只有精确性还是不够的，形式规范作为程序员与用户之间的通信手段，还必须是清晰的，简洁的。

2.2.1.2 规范是一个契约

形式规范是相应于正确性软件开发的方法而出现的。其最重要的特性就是它们在一般意义上规定了总体的设计目标。规范是以最精确的方法描述用户需要什么，而不管如何完成这一任务。也就是说，它是一个描述软件、系统的功能的文本。它回答了该系统要“做什么”而不是“怎么做”。因此，规范是开发者与用户之间的一个契约。

2.2.1.3 规范是一个设计目标

对开发者而言，一个精确的规范使它免于需求的不正确性，同时，也提供了可验证的设计过程的可能。正如 Morgan[65]中的观点，尽管以软件正确性为先导的构造方法的实用性尚待提高，但以这种方式开发软件是完全可能的。这就是一种经典的程序开发方法-----逐步精化方法。

2.2.1.4 规范的书写

在工程上，大多是以自然语言辅以数学表示来书写程序规范的。由于自然语言具有二义性，这种表示常常不准确。而不准确的规范将很可能导致程序的错误，甚至失败。如何使用清晰、无二义的语言来写规范是需要一定的经验和技能的，这也是软件工程的研究内容之一。

2.2.1.5 规范是抽象

软件工程有几种抽象层次。事实上，软件生命周期的每个阶段都有一个基于不同抽象的软件模型。规范是描述用户需求的抽象计算模型，其中不涉及计算任务的实现细节，也不包含与用户需求无关的细节。

2.2.1.6 规范方法

规范方法可以分为两类，一类是面向模型的方法，也称基于状态的方法。其基本思想是利用一些已知特性的数学抽象来为目标软件系统的状态特征和行为特征构造模型。这些数学抽象包括域、元组、集合、序列、包、映射等。Z、VDM 等都是面向模型的方法。另一类方法为代数方法，该方法为目标软件系统的规格说明提供了一些特殊的机制，以允许对目标软件系统描述的结构化，并支持目标软件系统中公共元素的重用。代数方法仅使用带有等词的一阶逻辑的表示，而不引入通常的数学对象。常见的代数方法有 Clear[66]、ACTONE[67]、OBJ[69]和 Larch[72]等。

2.2.2 设计级

设计级是指对求解问题所得的算法的描述。算法描述语言是一个抽象的操作语义模型。Dijkstra 的卫式命令语言是一种最典型的算法描述语言。该语言中具备描述算法的基本结构和命令，是构成程序设计语言的最核心成分。程序设计语言更接近于机器的实现，而算法描述语言更接近于数学的结构和表示。例如，对于一组数据，在算法描述语言中可能描述为一个集合，而在程序设计语言中则可能描述为一个数组。

从规范到算法，是一个问题求解的过程。形式化软件开发方法中，通过精化的方法(包括数据精化和运算精化)，将规范转换成设计，尽管 Hoare[40]中指出，从指称语义到操作语义模型可以进行转换，然而就指称语义模型所描述的具体问题的规范而言，却很难通过完全形式化的转换来得到操作语义模型下求解该问题的一个设计。因为问题求解的过程需要创造性的劳动，而形式化的转换则是一个机械的过程。

从程序正确性的角度而言，只有首先保证设计级所得的设计满足上一级中问题规范的要求，才能保证最终程序的正确性。而设计是否满足规范是可以进行检测和证明的。

对于确定性的语言而言，若一个运算 op 的规范表示为：

op

chg　x:X

pre　　pre(x)

post　　post($\overleftarrow{x}$,x)

那么，测试一个输入输出的序对(i,o)是否满足 op 的规范定义为：

pre(i)⇒post(i,o)

而 op 的设计是否满足其规范则定义为：$\forall i \in X \bullet pre(i) \Rightarrow post(i, f(i))$

其中 f 的定义为：

M:Program→(State→State)

f=M[A]　(A 是一个设计)

则 A sat op 定义为 if $\forall i \in State \bullet pre(i) \Rightarrow post(i, M[A](i))$

对于非确定性语言而言，

M:Program→(State→P(State))

或表示为 R:Program→P(State→State)

此时，A sat op 定义为：

$$\forall i \in State \bullet pre(i) \Rightarrow (\exists o \in State \bullet (i,o) \in R[A] \wedge (\forall o \in State \bullet (i,o) \in R[A] \Rightarrow post(i,o))$$

2.2.3 实现级

实现级是指使用指定的程序设计语言来实现设计级所得的结果。算法描述语言和

程序设计语言都是操作语义模型，它们只在抽象程度上存在着差别。从算法到程序只是对同一事物采用了不同的描述方法，因此，从设计级到实现级可以通过转换来完成。转换规则的正确性保证了转换所得的程序与上一级中的设计是等价的。当然，其中需要考虑实现效率的问题，例如集合可以使用程序设计语言中的数组结构来实现，也可以使用链表结构实现，如何选用适合问题需要的、效率较高的结构来实现算法中的抽象结构和运算是这一阶段的重要内容。

2.2.4 三级结构的优点

在程序开发的过程中按照三级结构来进行，通过将设计阶段与实现阶段的严格分离，可以在设计的阶段不考虑程序实现的细节，从而降低设计的难度；另外，设计和实现阶段的分离，可以使两个阶段各有其考虑的重点，设计阶段强调设计的正确性，而实现阶段更多地考虑效率问题；同时，设计和实现的分离还可以增加设计的通用性和可重用性，因为对于同一个问题，设计可以是相同的，但可以使用不同语言在不同的软硬件环境下实现，这一点对于并行程序尤其重要。由于设计是抽象的，独立于底层的运行环境，因此具有很好的可移植性。

2.3 DPaPD 模型

根据上节中所阐述的程序开发的三级结构，程序开发的问题可以分成两部分来考虑：一是设计的问题，二是实现的问题。传统意义上的并行程序设计模型(parallel programming model)是对并行计算机基本特征的抽象，是并行机硬件与软件系统设计之间的一个界面，它向上为程序设计层提供抽象的运算和结构，向下在各类并行计算机的体系结构上实现这些运算和结构。其主要作用在于一是作为并行算法实现的基础，二是为并行算法的设计与分析提供一个简单、方便的框架，三是使设计的算法更具有通用性和可移植性。因此，这些并行程序设计模型严格意义上说是一个算法设计和程序编码的抽象平台，而没有指导在该平台上进行设计的机制，不能提供支持并行程序开发整个过程的方法和辅助工具。这正是我们要解决的问题。

DPaPD 模型不是传统意义上的并行程序设计模型，而是一种支持整个并行程序开发过程的方法。因此，它不仅是作为一个并行程序设计的模型而提出的，更是作为一个并行算法设计的模型。而且，该模型将并行程序开发的两个基本方面统一在一个抽象框架之下，为从问题规范出发，获得并行程序提供了一种系统的方法。DPaPD 模型的基本结构如图 2.1。

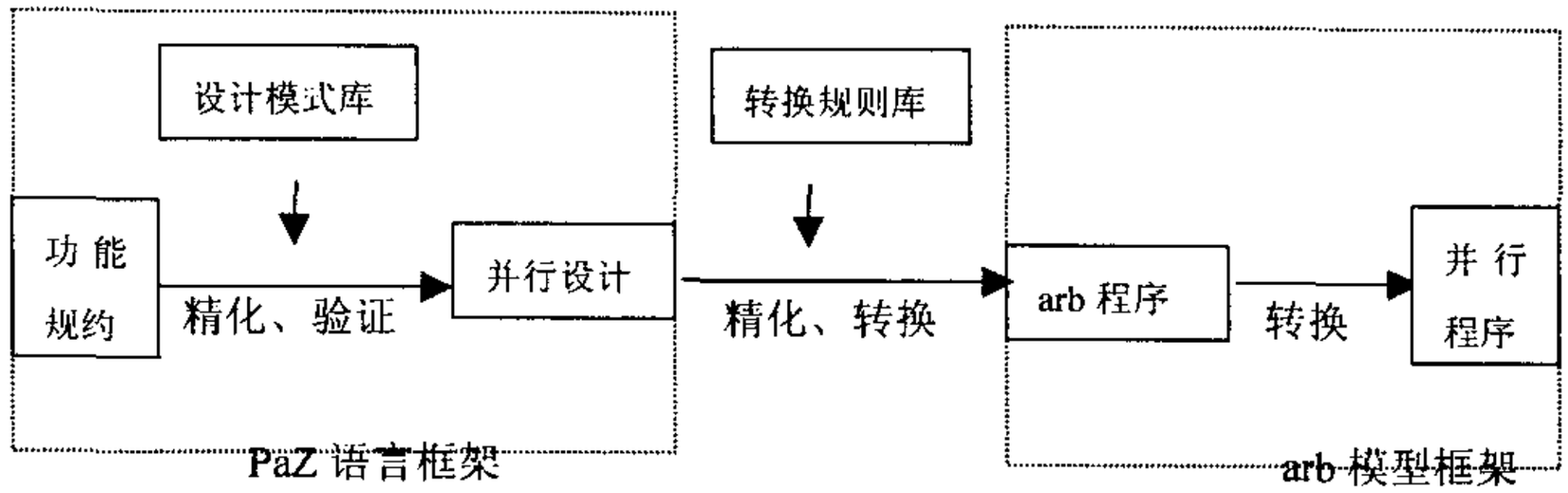


图 2.1　　DPaPD 并行程序开发模型

Fig. 2.1 DPaPD Parallel Program Development Model

下面我们将对 DPaPD 模型中的各个成分和相关的概念进行详细的介绍。

2.3.1PaZ 语言

PaZ 语言是 Z 语言[61,73]的一个并行扩充，它在 Z 语言的基础上扩充了一些进行并行设计描述所需的语言成分。在 DPaPD 模型中，PaZ 语言作为描述问题规范、并行设计和设计模式的语言。

2.3.1.1 为什么选用 Z

Z 语言是软件工程中一种常用的规范描述语言。Z 使用一个用于表示数学文本的习语(conventions)的集合来描述计算系统。它是一种比较成熟的表示方法。其思想启蒙于七十年代末，在八十年代通过牛津大学和其工业伙伴(包括 IBM, Inmos 等)的合作项目得到了迅速的发展。Z 的第一个使用手册出现于己于 1989 年。其时 ANSI、BSI、ISO 等著名的标准化组织考虑确立 Z 标准(包括形式化语义)。在面向对象的程序设计中，Z 也始终是其他表示方法的基础。

Z 是一种基于模型的表示。使用 Z 语言表示的系统通常由一组状态变量和一些可以改变其状态变量的操作组成。这种模型风格与命令式、过程式语言非常吻合，同时，Z 也很自然地适合面向对象的程序设计。然而，Z 并不局限于这些风格的系统，它还可以应用于函数式风格的系统及其它的系统。

Z 是一种表示，不是一种方法。然而 Z 表示可以支持许多不同的方法。一个 Z 文本的意义可以由其作者来决定，它可以理解为系统行为的模型，即一个抽象的形式化规范，或者，Z 文本中的元素也可以理解为以代码表示的结构，如模块、数据类型、函数、类、对象等。在这种情况下，Z 模型就是一个详细的设计。

Z 不是一个可执行的表示。它不能被直接翻译或编译成可运行的程序。Z 不是一个程序设计语言，Z 文本不是高级语言编写的程序。

Z 是为人设计的，而不是为机器设计。因此，Z 语言易学、易用、易理解。

2.3.1.2 对 Z 的扩充

由于 Z 语言具有上述的良好性质，为了描述特定的系统，Z 语言有各种扩充版本。例如 Z 语言的面向对象的扩充 Object-Z，描述并发实时系统的 TCOZ[25]等。在 DPaPD 模型中，Z 语言可用于描述问题规范，而在设计一级，由于我们需要体现问题的并行求解方案，因此，其描述语言必须具有描述基本的抽象并行运算的能力。我们通过对 Z 语言扩充描述并行设计所需的语言成分，得到 Z 的一个并行扩充版本 PaZ。与 TCOZ 等 Z 的其他并行扩充不同，PaZ 由于描述的是相对抽象的并行设计，而不是一个具体的系统，因此保持了规范描述语言的抽象性特点，而 TCOZ 等描述并发系统，指的是对系统的形式化规范描述，既描述系统中的各个部件、部件之间的并发交互，消息传递等行为，因而在 Z 语言中增加了大量用于描述多线程并发控制、时间、通信等原语，同时由于其结合了两种不同的表示方法，因而语言结构、语法、语义上都比较复杂。

2.3.2 Arb 模型

Arb 模型[51]是一个并行程序的操作模型。它所考虑的程序是那些并行组合与顺序组合在语义上等价的程序。这一性质称为 arb-兼容的。Arb 模型对程序的描述也是状态-转换系统的形式，其中对程序、计算、顺序组合、并行组合和程序精化等基本概念进行了形式化的定义。

定义 1 程序(program)

程序 P 定义为六元组(V,L,InitL,A,PV,PA)，其中

V 是一个类型变量的有限集合。V 定义了状态-转换系统的状态空间，也就是说，状态就是 V 中变量的取值。在该语义系统中，不同的变量表示不同的原子数据对象，别名是不允许的。

$L \subseteq V$ 表示 P 的局部变量。这些变量以两种方式区别于 P 中的其他变量：(1)P 的初始状态由他们的取值给定；(2)他们在 P 之外是不可见的，例如，他们不能在 P 的规范中出现，也不能被其他与 P 有组合关系的程序所访问。

InitL 是对 L 中变量的一个取值，表示他们的初始值。

A 是程序动作(program actions)的有限集合。一个程序动作是一个表示其输入变量的状态和其输出变量的状态之间的一个关系。一个程序动作由元组(I_a,O_a,R_a)表示，其中，$I_a \subseteq V$ 表示 A 的输入变量，$O_a \subseteq V$ 表示 A 的输出变量，R_a 为 I_a 和 O_a 之间的关系。

$PV \subseteq V$ 为只能被协议动作(protocol actions)所修改的协议变量(protocol variables)。即如果 v 是一个协议变量，$a=(I_a,O_a,R_a)$是一个协议动作，则 $v \in O_a$。协议变量和协议动作的定义是同步机制定义的基础。

$PA \subseteq A$ 为协议动作。只有协议动作可以修改协议变量。

定义 2 计算(computation)

若 P=(V,L,InitL,A,PV,PA)，P 的一个计算定义为

$$C = (s_0, \langle j : 1 \le j \le N : (a_j, s_j) \rangle)$$

其中，s_0 为 P 的初始状态；

$\langle j : 1 \le j \le N : (a_j, s_j) \rangle$ 为一个序对的序列，其中每个 a_j 为 P 的一个程序动作，且对所有的 j，$s_{j-1} \xrightarrow{a_j} s_j$，我们称这些序对为 C 的状态转换，且动作 a_j 的序列为 C 的动作。N 为非负的整数或为∞，对于前一种情况，我们说 C 为一个长度为 N+1，终止状态为 s_N 的有限可终止计算，后者我们称 C 为无限或不可终止计算。

定义 3 程序的可组合性(composability of programs)

程序 $P_1,P_2,\ldots\ldots,P_N$ 称为可组合的仅当

出现在多个程序中的任何变量必须具有相同的类型；

出现在多个程序中的动作必须有一致的定义；

不同的程序没有公共的局部变量。

定义 4 顺序组合(sequential composition)

若程序 $P_1,P_2,\ldots\ldots,P_N$，$P_j=(V_j,L_j,InitL_j,A_j,PV_j,PA_j)$，为可组合的，则其顺序组合 $(P_1;P_2;\ldots\ldots;P_N)=(V,L,InitL,A,PV,PA)$为：

$V=V_1 \cup V_2 \cup \ldots\ldots \cup V_N \cup L$

$L= L_1 \cup L_2 \cup \ldots\ldots \cup L_N \cup \{En_P,En_1,En_2,\ldots\ldots,En_N\}$，其中 $En_P,En_1,En_2,\ldots\ldots,En_N$ 为不在 V 中的布尔变量：En_P 在顺序组合的初始状态中为真，此后为假，对于所有的 j，En_j 只在 P_j 执行的相应计算时为真。

InitL 定义为：En_P 的初始值为 true，对所有的 j，En_j 的初始值均为 false，L_j 中变量的初始值为 $InitL_j$ 中给定的值。

A 由以下动作组成：

• 动作 A_j 中的相应动作，对于所有属于 A_j 中的动作 a,定义等价的动作 a'，所不同仅在于 a'只在 En_j=true 时为可行的；

• 完成组合的各成分间转换的动作：初始动作 a_{T0} 从任何 En_P=true 的状态 s 到等

价的状态 s'，除了 En_P=false 和 En_1=true。则 s'为 P_1 的初始状态。对任意 $1 \le j < N$，动作 a_{T0} 从任何 En_j=true 的 P_j 的终止状态 s 到等价的状态 s'，除了 En_j=false 和 En_{j+1}=true。则 s'为 P_{j+1} 的初始状态。终止动作 a_{TN} 从任何 En_N=true 的 p_N 的终止状态 s 到等价的状态 s'，除了 En_N=false。则 s'为顺序组合的终止状态。

- $PV=PV_1 \cup PV_2 \cup \ldots \ldots \cup PV_N$
- PA 包含那些从 $PA_1 \cup PA_2 \cup \ldots \ldots \cup PA_N$ 的动作 a 推导出的动作 a'。

定义 5 并行组合(parallel composition)

若程序 $P_1,P_2,\ldots\ldots,P_N$，$P_j=(V_j,L_j,InitL_j,A_j,PV_j,PA_j)$，为可组合的，则其并行组合 $(P_1\|P_2\|\ldots\ldots\|P_N)=(V,L,InitL,A,PV,PA)$为：

$V=V_1 \cup V_2 \cup \ldots \ldots \cup V_N \cup L$

$L= L_1 \cup L_2 \cup \ldots \ldots \cup L_N \cup \{En_P,En_1,En_2,\ldots\ldots,En_N\}$，其中 $En_P,En_1,En_2,\ldots\ldots,En_N$ 为不在 V 中的布尔变量：En_P 在并行组合的初始状态中为真，此后为假，对于所有的 j，En_j 为真直到 P_j 在组合中的相应部分执行终止。

InitL 定义为：En_P 的初始值为 true，对所有的 j，En_j 的初始值均为 false，L_j 中变量的初始值为 $InitL_j$ 中给定的值。

A 由以下动作组成：

• 动作 A_j 中的相应动作，对于所有属于 A_j 中的动作 a,定义等价的动作 a'，所不同仅在于 a'只在 En_j=true 时为可行的；

• 对应于组合中成分的初始和终止的动作：初始动作 a_{T0} 从任何 En_P=true 的状态 s 到等价的状态 s'，除了对所有的 j,En_j=false。则 s'为所有 P_j 的初始状态。对任意 $1 \le j < N$，动作 a_{Tj} 从任何 En_j=true 的 P_j 的终止状态 s 到等价的状态 s'，除了 En_j=false。P 的一个可终止的计算包括每个 a_{Tj} 的执行，在所有 a_{Tj} 的执行之后，结果状态 s'为并行组合的终止状态。

- $PV=PV_1 \cup PV_2 \cup \ldots \ldots \cup PV_N$
- PA 包含那些从 $PA_1 \cup PA_2 \cup \ldots \ldots \cup PA_N$ 的动作 a 推导出的动作 a'。

定义 6 动作的互斥(commutativity of actions)

程序 P 的两个动作 a 和 b 为互斥的，当以下条件成立：

1.b 的执行不影响 a 的是否可行，反之亦然；

2.有可能经过先执行 a 再执行 b 从 s_1 到 s_2，也有可能经过先执行 b 再执行 a 从 s_1 到 s_2。

定义 7 arb-可兼容(arb-compatible)

程序 P_1，P_2，… …，P_N 为 arb-可兼容当且仅当它们可组合且一个程序中的任何动作与其他程序中的任何动作为互斥的。

2.3.3 并行程序

DPaPD 模型中最后所得的并行程序可以是两种类型的并行程序。一种是以提供有路障同步和并行组合结构的语言所实现的并行程序，例如 Fortran X3H5[74]；另一种是以多地址空间、消息传递结构的语言所实现的并行程序，如 Fortran M[75]、PVM[63]等。

2.3.4 从功能规范到并行设计

面向模型的形式化方法中，程序的开发过程是一个设计与验证的多步精化演算的过程，它将一个抽象模型 M_0 通过逐步的数据精化和运算精化：$M_0,M_1,\ldots\ldots M_n$，得到模型的具体程序实现 M_n，其中 M_{i+1} 是 M_i 的一个精化模型，它比 M_i 更具体，更接近于最后的具体实现模型。然而我们知道，从规范到设计的过程是一个“发明”的过程，这种“发明”必须以开发人员对应用问题的理解、程序设计的经验和对软件开发基本原则的把握为基础的。因此，所谓多步精化的程序开发方法只是提供给开发者的一种思维工具，并不能具体地辅助开发人员进行程序的开发。

在程序开发的整个过程中，从规范到设计这部分是最困难的。因为问题求解的创造性思维主要包含在这一环节中。然而，通过长期程序开发的实践和经验我们知道，尽管应用问题多种多样，但问题的求解并不是完全没有规律可循的。我们把一类问题求解策略的本质抽象出来，构成设计模式，那么，设计模式库就是各类问题求解策略的集合，其功能相当于一个有经验的程序开发人员，它能够为其他的程序开发者提供问题求解方法的指导。因此，DPaPD 模型通过设计模式的定义与使用，为从功能规范到并行设计这一创造性思维的过程提供了系统的支持，虽然系统并不能完全实现这一过程，但通过提供问题并行求解的策略和基本框架，大大简化了其中程序开发人员的工作，并为其提供明确的指导。设计模式库涉及的相关问题包括应用问题的描述与分类[19，47]、设计模式的表示方法[7,9,10,14,17,18,20]、设计模式的分类与选择[6,11,13,15,16,22,23,55]等。我们将在后续的章节中对设计模式概念和设计模式库的有关问题进行具体的讨论。

2.3.5 从 PaZ 到 arb 程序

DPaPD 模型的第二个环节是从并行设计到 arb 程序的精化与转换。PaZ 是对规范描述语言 Z 的一个并行扩充，arb 模型是一个并行程序的操作语义模型。与 arb 模型相比， PaZ 语言更为抽象，因此，这一过程是一个从抽象的模型到一个更具体模型的精化过程。概括来讲，精化包括数据精化和运算精化。数据精化是为一个抽象的状态空间寻找具体的数据结构表示，运算精化是指抽象运算的具体化。例如，PaZ 语言

中的集合表示一个抽象的状态空间，可以将它精化为 arb 程序中的数组结构。同样的，一些集合的运算可以相应地具体化为数组的操作。这些精化与转换独立于应用问题的求解，是一个纯粹的模型转换。因此，DPaPD 模型中的转换规则库由从 PaZ 语言的基本成分到 arb 模型的相应结构的转换规则组成，使用这些规则可以将 PaZ 所描述的并行设计精化、转换为等价的 arb 程序。

2.3.6 从 arb 程序到并行程序

Arb 模型是一个抽象的并行操作语义模型。Arb 程序不是一个可直接执行的并行程序，因此需要将其进一步转换成可在具体结构的并行计算机上执行的并行程序(如 2.3.3 节所述)。其转换的基本过程如图 2.2 所示。

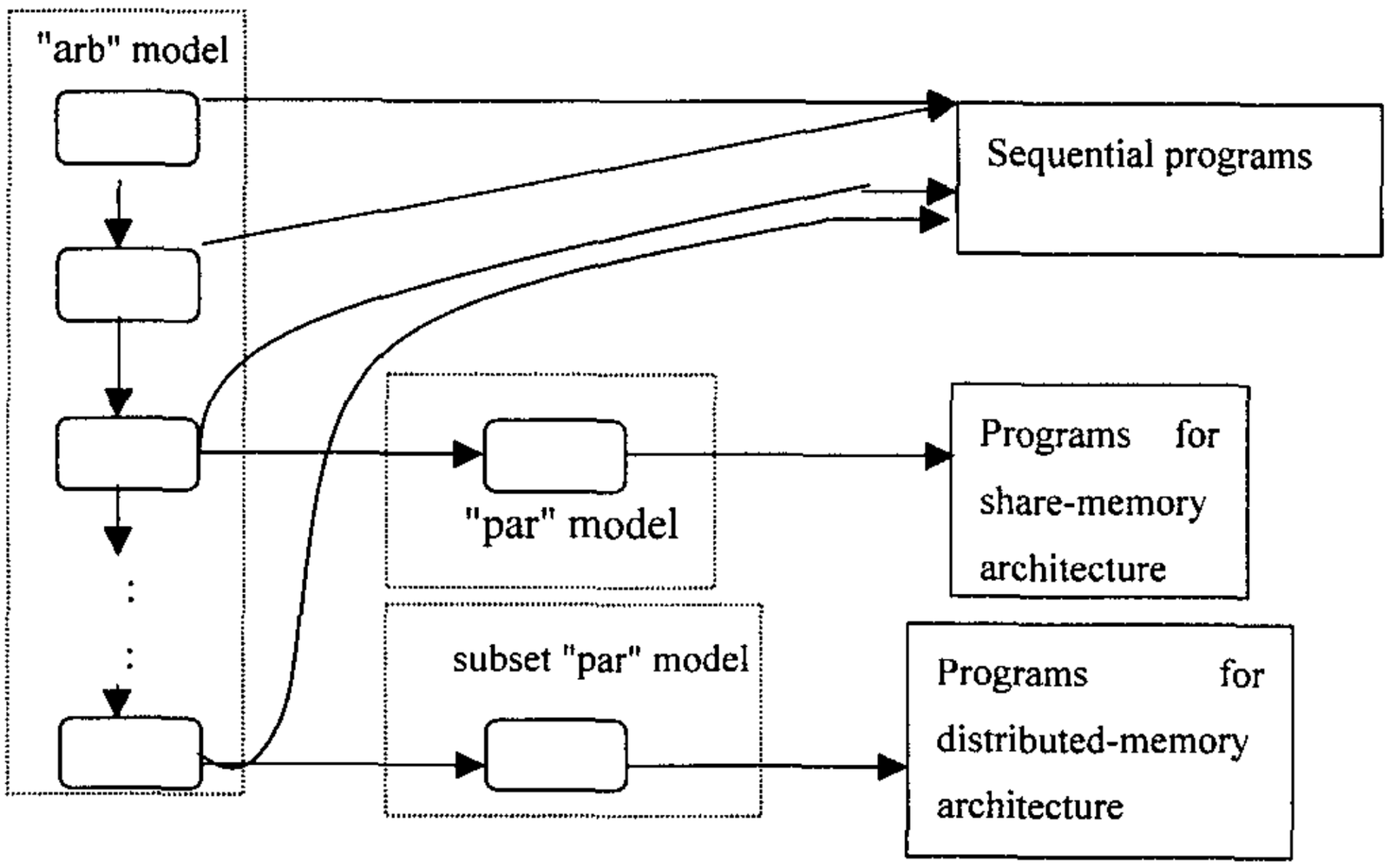


图 2.2　arb 程序到并行程序的转换过程[51]

Fig. 2.2 Transformation from arb Programs to Parallel Programs[51]

Arb 程序可以分别转换成在共享内存结构下的并行程序和分布式内存结构下的并行程序。这部分工作是[51]的主要内容。然而[51]中没有说明如何从一个应用问题的规范来得到 arb 程序，即 arb 并行程序的设计问题，而这部分正是我们的工作重点。因此，DPaPD 模型中我们所做的工作部分如图 2.3 所示。

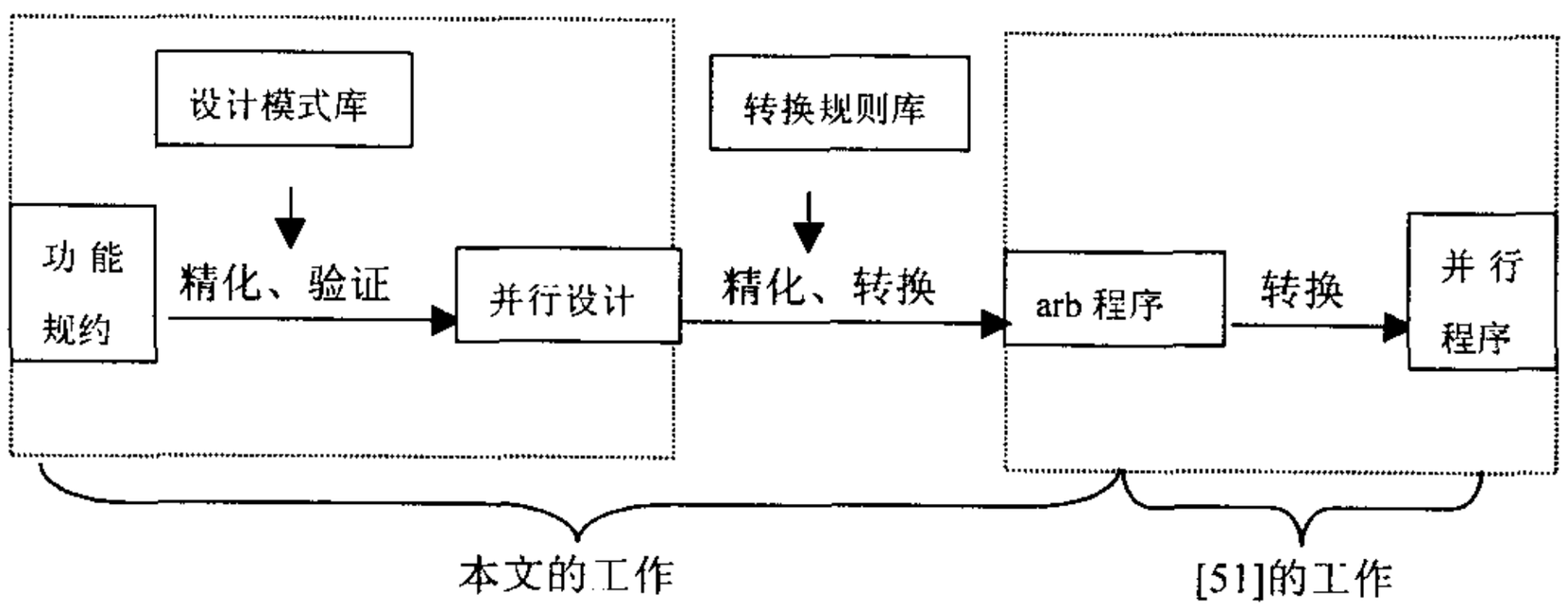


图 2.3　本文的工作示意图

Fig. 2.3　Part of Our work

2.4 小结

在以上的小节中，我们对 DPaPD 模型的思想基础和模型中的各个组成部分进行了介绍。总之，DPaPD 模型下进行并行程序的开发可概述为这样一个过程：首先，它从问题的功能规范出发，通过使用设计模式库中所描述的问题并行求解的策略来对问题进行并行求解，经过精化、验证，得到解决该应用问题的并行设计，由于问题的功能规范、并行设计以及设计模式都使用 PaZ 语言进行描述，因此这个阶段的工作统一在 PaZ 语言的框架之下，从而使程序开发人员这一阶段可以重点考虑问题求解的问题，而忽略语言的因素，另一方面，由于设计模式库为程序开发人员提供了一个问题求解策略的指导，从而为并行设计的获得提供了方法上和系统上的支持。然后，通过使用转换规则库中的转换规则将所得的 PaZ 语言描述的并行设计进一步精化、转换为 arb 模型下的抽象程序，最后再使用[51]中的转换将所得的 arb 程序转换成共享内存结构或分布式内存结构下的具体并行程序。

在后面的章节中，我们将就 DPaPD 模型各部分的设计情况进行具体的讨论。

第三章　PaZ 语言

3.1 Z 语言的基本结构

一个 Z 语言的形式规范说明由若干个段落(paragraph)所组成。段落是 Z 语言表示的核心。一个段落可以是基本类型定义(Basic type definitions)、公理描述(Axiomatic descriptions)、约束(Constraints)、模式定义(Schema definitions)、全局常量定义(Abbreviation definitions)、类属模式(Generic schemas)、类属常量定义(Generic constants)、自由类型定义(free types)等。

3.1.1 基本类型定义

Paragrgh::=[类型标识符,... ...,类型标识符]

这些类型标识符不能在前面已定义过，其有效范围从定义处起直至规范的末尾，且成为全局基本类型的一部分。

例如定义类型 NAME 和 DATE，表示为：

[NAME，DATE]

3.1.2 公理描述

Paragragh::= 声明部分 | 谓词部分

公理描述用于引入一个或多个全程变量和关于他们的谓词，这些谓词给出了进一步的信息。这些变量不能为前面定义过的变量，其作用域为其说明处至整个规格说明的结束处，谓词也具有全程的性质。公理描述中的声明部分和谓词部分都可以不出现，如谓词部分不出现，缺省的谓词为 true。在任何时刻，都可以为已经引入的全程变量增加谓词来对其进行限制。

例如下面的公理描述定义了函数 square：

square: N→N

$\forall n : N \bullet square(n) = n * n$

3.1.3 约束

paragragh::=谓词

谓词单独作为一个段落，用于对一个已经引入的全局变量进行约束(constraint)。其功能相当于公理描述中的谓词。

例如下面的约束表示变量 n_disks 的值必须小于 5：

n_diska<5

3.1.4 模式定义

模式是 Z 中最主要的语言结构。一个模式由一些变量的声明和限制这些变量的值的谓词两部分组成。模式的定义有水平和垂直两种形式。

水平形式：Schema_Name $\triangleq$ Schema_Exp

垂直形式：

Schema_name
Declarations
Predicate;… …;Predicate

这两种形式引入了一个模式名 Schema_name。模式名出现在定义符号的左边或垂直方式模式框的顶端线中。在垂直方式的模式中，中间的横线以上为声明部分，横线以下为谓词部分。声明部分不可缺省，谓词部分若缺省，则为 true。

在水平方式中，定义符号右边是一个模式表达式。新的模式可以通过使用模式运算符将若干个已定义的模式组合在一起而得到。

垂直形式的模式

S
$D_1;\ldots\ldots;D_m$
$P_1;\ldots\ldots;P_n$

等价于水平形式的模式

$$S \triangleq [\, D_1;\ldots\ldots;D_m \mid P_1;\ldots\ldots;P_n]$$

例如下面的模式定义了一个班级：

Class
Enrolled,tested: P Student
#enrolled $\leq$ size
tested $\subseteq$ enrolled

3.1.5 全局常量定义

Paragragh::=Ident==Expression

该段落定义一个新的全局常量。常量名为左部的标识符，其值为右部的表达式，其类型为表达式的类型。

例如以下定义了名字 DATABASE 为从 ADDR 映射到 PAGE 的函数的集合：

DATABASE==ADDR→PAGE

3.1.6 类属式定义

某些结构是独立于其元素的，例如集合，可以是整数的集合，也可以是字符的集合。不管其元素的类型是什么，关于该结构的操作是相同的，如集合的运算：交、并、差等不会受到集合元素的影响。这样的结构可以定义为类属式。

Z 中的类属式有两种：类属常量和类属模式。类属常量可用于定义类属的关系、函数、序列等以及其上的运算。其形式为：

```
[iden,…,iden]
声明部分
谓词部分
```

其中，顶端中括号中的部分为类属形式参数，声明部分声明了所定义的类属常量，谓词部分为对类属常量的约束。当使用一个通用常量时，必须对类属形式参数给定具体的值。例如下面定义的是一个类属的函数 first：

```
[X,Y]
first:X × Y→X
∀x : X; y : Y • first(x, y) = x
```

类属模式的定义为以下形式：

```
S[x1,… …,x2]
声明部分
谓词部分
```

其中 x_1 为类属形式参数，它可以作为类型在声明部分使用，声明部分和谓词部分与一般的模式定义一样。当使用一个通用模式时，必须对类属形式参数给定具体的值。例如以下为类属模式 pool 的定义：

Pool[RESOURCE]

Owner:RESOURCE→USER

Free: P RESOURCE

(dom owner)∪ free=RESOURCE

(dom owner)∩ free=ϕ

3.1.7 自由类型定义

paragraph ::= Ident ::= Branch|… … |Branch

Branch ::= Ident [<<Expression>>]

该段落定义一个新的基本类型 Ident。Ident 为类型名，::=右部为类型构造表达式，它可以是常量，也可以通过构造子构造。

3.2 Z 的基本数学库

Z 的基本成分除了上述的基本结构外，还包括一个重要的组成部分，即一个数学定义的标准库。用户可以直接在 Z 规格说明中使用库中的定义和定律，而不必再进行声明。数学库中除了最基本的一些数学类型及其上的运算，如整数、正整数、非负整数等，还定义了一些更复杂的数学类型及有关的运算，包括：集合、关系、函数、序列和包等。具体的内容可参见[61]。

3.3 PaZ 语言的语法

PaZ 语言在 Z 的基础上扩充了用于描述抽象并行运算的运算符。我们没有对 Z 语言的基本类型进行扩充，而是在现有类型的基础上扩充了一些新的运算。并行运算主要可以施加于一些描述行为的语言成分，例如函数、操作模式等。因此我们在 PaZ 中定义了以下一些运算符：函数的并行兼容运算符_⇳_(parallel-compatible operator of functions)，函数的并行组合运算符_||_ (parallel-composition operator of functions)，操作模式的并行兼容运算符_⇳_ (parallel-compatible operator of schemas) ，操作模式的并行组合运算符_||_ (parallel-composition operator of schemas)，并行函数作用运算符_|(_)| (parallel-function-application operator) 等。

扩充的语法成分如图 3.1 所示：

Op-Name ::= _In-Sym Decoration

| pre-sym Decoration

| _post-sym Decoration

| _(|_|) Decoration

| _|(_)| Decoration

| _ Decoration

In-Sym ::= In-Fun | In-Gen | In-Rel

In-Fun ::= ⇳ | ||| | ∘ | ⊕ | … …

… …

Schema-Exp ::= ¬ Schema-Exp

| … …

| Schema-Exp ⇳ Schema-Exp

| Schema-Exp ‖ Schema-Exp

图 3.1　PaZ 的扩充语法部分

Fig. 3.1 Extension Part of PaZ Syntax

PaZ 语言的完整语法定义可参阅附录。

3.4 PaZ 语言扩充运算符的语义定义

下面我们以 Z 语言数学库中的运算符定义形式对扩充的算子进行数学的定义。

3.4.1 函数并行兼容运算符

名字(name)

⇳

定义(definition)

$[x_1,x_2,y_1,y_2]$

$\Updownarrow : p((x_1 \to y_1) \times (x_2 \to y_2))$

$\forall op_1 : x_1 \to y_1; op_2 : x_2 \to y_2 \bullet$

$op_1 \Updownarrow op_2 =$

$\forall s_1, s_2 : (s_1, s_2) \in op_2 \bullet s_1 \in x_1 = s_2 \in x_1$

$\wedge \forall s_1, s_2 : (s_1, s_2) \in op_1 \bullet s_1 \in x_2 = s_2 \in x_2$

$\wedge \forall s_1 : s_1 \in x_1 \wedge s_1 \in x_2 \bullet$

$(\forall s_2, s_3 : (s_1, s_2) \in op_1 \wedge (s_2, s_3) \in op_2 \bullet (\exists s_2' \bullet (s_1, s_2') \in op_2 \wedge (s_2', s_3) \in op_1)$

$\wedge \forall s_2, s_3 : (s_1, s_2) \in op_2 \wedge (s_2, s_3) \in op_1 \bullet (\exists s_2' \bullet (s_1, s_2') \in op_1 \wedge (s_2', s_3) \in op_2))$

解释(description)

该运算符表示两个函数的并行兼容运算。它描述两个函数是否可以进行并行组合运算。若函数 op_1 和函数 op_2 可以并行执行，则 $op_1 \Updownarrow op_2$=true, 否则 $op_1 \Updownarrow op_2$=false。 实际上，该算子是 DPaPD 模型对函数并行执行的一个约束条件。

例子(examples)

若函数 Op_1 为： $Op_1(x)=x+1$

设有 $b=a+1$ $d=c+1$ 为 Op_1 的两个取值

函数 Op_2 为： $Op_2(x)=x+2$

$c=a+2$ $d=b+2$ 为 Op_2 的取值

则根据定义有：$op_1 \Updownarrow op_2$=true

3.4.2 函数并行组合运算符

名字

‖

定义

$[x_1,x_2,y_1,y_2]$

$\| : p((x_1 \to y_1) \times (x_2 \to y_2)) \to (x_1 \cup x_2) \to (y_1 \cup y_2)$

$\forall op_1 : x_1 \to y_1; op_2 : x_2 \to y_2 \bullet$

$op_1 \| op_2 = op_1 \oplus op_2 \vee op_1 \| op_2 = op_2 \oplus op_1$

解释

该运算符定义了两个函数的并行组合运算，它表示这两个函数为并行执行的。该算子为非确定性的。

例子

函数定义为：

$Op_1(x)=x+1$　设有取值　b=a+1　　d=c+1

$Op_2(x)=x+2$　设有取值　c=a+2　　d=b+2

因为 $op_1 \Updownarrow op_2$=true，所以可以进行并行组合运算，依定义，结果为

$op_1 \| op_2=\{(a,c),(c,d),(b,d)\} \vee \{(a,b),(b,d),(c,d)\}$

规则(laws)

op ‖ op = op

op1‖(op2‖op3) = (op1‖op2)‖op3 = op1‖op2‖op3 = $\forall i : 1 \le i \le 3 :\| opi$

dom(op1‖op2) = dom(op1) ∪ dom(op2)

dom op1 ∩ dom op2 = ϕ ⇒ op1‖op2 = op1 ∪ op2

3.4.3 模式的并行兼容运算符

名字

⇳

定义

S_1 和 S_2 为操作模式

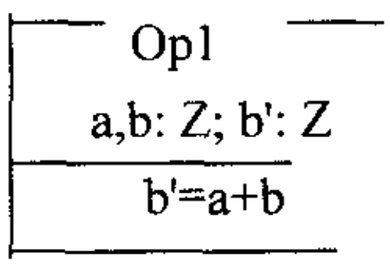
$$S_1 \Updownarrow S_2 = \forall state \bullet ((\exists S_1 \bullet \theta state_1 = \theta state) \wedge (\exists S_2 \bullet \theta state_2 = \theta state) \Rightarrow \theta state_1' \notin S_1 \wedge \theta state_2' \notin S_2$$

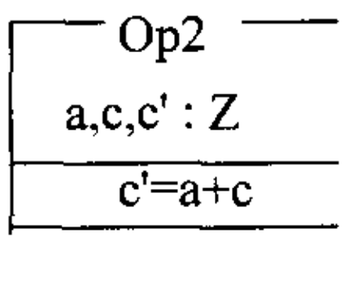
解释

S1 和 S2 只共享只读变量。若变量同时出现在 S1 和 S2 中，则 S1 和 S2 都不能改变其值。这个约束条件比不相交并行(disjoint parallel)的条件要弱，但比函数的并行兼容条件强。

例子

定义模式 Op1 和 Op2 为：





则依定义得 $op_1 \Updownarrow op_2$=true

若定义模式 Op1 和 Op2 为：

Op1

a,b: Z; a',b': Z
a'=a+1
b'=a+b

Op2

a,c,c' : Z
c'=a+c

则 $op_1 \Updownarrow op_2$=false

3.4.4 模式的并行组合运算符

名字

‖

定义

S‖T = S ∧ T

解释

该运算符描述两个操作模式的并行组合运算。

例子

定义 Op1 和 Op2 为：

Op1

a,b: Z; b': Z
b'=a+b

Op2

a,c,c' : Z
c'=a+c

因为 $op_1 \Updownarrow op_2$=true，则依定义，有：

op1‖op2 =

a,b,c:Z b',c':Z
b'=a+b
c'=a+c

3.4.5 并行函数作用符

名字

$$\_|(\_)|$$

定义

[x,y]

$$\_|(\_)|: (x \to y) \times F\ x \to F\ y$$

$$\forall f : x \to y; A : Fx : A \subseteq dom(f) \bullet f|(A)| = \forall x \in A \bullet \| \{x\} \lhd f$$

解释

该算子定义了一个函数并行地作用于集合类型变量中的每个元素的运算。由于集合中不存在相同的元素，因此$\{x\} \lhd f$对于集合 A 中的所有 x 是并行兼容的，则 $f|(A)|$表示函数 f 并行地作用于集合 A 中的每个元素。

例子

定义函数 f : Z→Z　　f(x)=x+1

集合 A : F Z　A={1,2,3}

则 f|(A)|=f(1)||f(2)||f(3)，结果为{2,3,4}。

第四章　设计模式

DPaPD 模型中的一个重要的特点就是在功能规范到并行设计的环节中引入了设计模式的概念来帮助问题的并行求解和得到并行的设计。然而，人们对一般的设计模式概念的理解为面向对象软件工程领域的设计模式。本章将对设计模式的思想、概念进行阐述，并指出我们是如何引用并扩展这一概念，提出 DPaPD 模型下的设计模式概念的。本章还对与设计模式相关的一些问题进行了讨论，并列举了系统中设计模式的一些实例。

4.1 设计模式的概念

4.1.1 设计模式的思想

最初引入设计模式的概念是为了进行面向对象的软件设计。在软件设计过程中，内行的设计者知道：不是任何问题都必须从头做起的。他们更愿意重用以前使用过的解决方案。当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验是他们成为内行的部分原因。因此，在许多面向对象的系统中可以看到类和互相通信的对象的重复模式。这些模式解决特定的设计问题，使面向对象设计更灵活、优雅，最终重用性更好。他们帮助设计者将新的设计建立在以往工作的基础上，重用以往成功的设计方案。一些熟悉这些模式的设计者不需再去发现它们，而能够立即将它们应用于设计问题中。

如果将面向对象软件设计的经验作为模式记录下来，就可以便于软件的设计。每一个这样的设计模式系统地命名、解释和评价了面向对象系统中一个重要的和重复出现的设计。设计模式的目标就是将设计经验以人们能够有效利用的形式记录下来。

设计模式使人们可以简单方便地重用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式可以帮助开发者做出有利于系统重用的选择，避免设计损害系统重用性。通过提供一个显式类和对象作用关系以及它们之间潜在联系的说明规范，设计模式甚至能够提高已有系统的文档管理和系统维护的有效性。简而言之，设计模式可以帮助设计者更好地完成设计工作。

4.1.2 Alexander 的设计模式概念

设计模式概念最初是一位建筑设计师 Chistopher Alexander 提出的，指的是关于城市和建筑模式。他提出："每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动。"他提出的设计模式概念成为其后用于面向对象的设计模式的基础，也为不同领域中设计模式的概念提供了基本的框架。

4.1.3 Gamma 的设计模式概念

将设计模式概念引入面向对象软件工程领域的标志是 Gamma 等人著作的[53]一书。其中对设计模式的定义为："对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述"。一般而言，一个模式有四个基本要素：

1. 模式名称(pattern name) 是一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了设计词汇。设计模式允许我们在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与他人交流设计思想及设计结果。寻找合适的模式名也是设计模式编目工作的难点之一。

2. 问题(problem)描述了应该在何时使用模式。他解释了设计模式问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

3. 解决方案(solution)描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合(类或对象组合)来解决这个问题。

4. 效果(consequences)描述了模式应用的效果及使用模式应权衡的问题。尽管描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为重用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

一个设计模式命名、抽象和确定了一个通用设计结构的主要方面，这些设计结构能被用于构造可重用的面向对象设计。设计模式确定了所包含的类和实例，它们的角色、协作方式以及职责分配。每一个设计模式都集中于一个特定的面向对象设计问题或设计要点，描述了什么时候使用它，在另一些约束条件下是否还能使用，以及使用的效果和如何取舍。

4.1.4 我们的设计模式概念

目前，设计模式的应用主要存在于面向对象的软件工程领域，其概念一直沿用Gamma 的设计模式定义。然而，设计模式的原始概念(Alexanda 的定义)出自于建筑学的范畴，因此，这一概念的使用并不局限于面向对象的软件构造，而是可以根据领域的需要对其进行扩展和应用。尽管不象面向对象领域中对设计模式的研究那么成熟，其他领域在这方面的相关研究也是一直在进行着。在并行程序设计的领域，对设计模式各个方面的研究工作有[1,2,3,8,21,24,58]等。在并行计算中使用设计模式的系统有早期的 Code (Browne et al.,1989)、Frameworks (Singh et al.,1991)，近期的Enterprise (Schaeffer et al., 1993)、Code2 (Browne et al., 1995)、HeNCE (Browne et al., 1995)、Tracs (Bartoli et al., 1995)、DPnDP (Siu and Singh, 1997) 等[3]。

一般地，并行计算中，设计模式用于描述重复出现的具有相似结构与通信同步行为的并行计算问题及其解决方案。Framworks 是最早在工作站机群环境下通过对已有的顺序程序进行重构来开发并行性的系统之一。该系统中的模式称为模板(templates)，应用程序由以类似于远过程调用的形式进行通信、交互的模块组成。模块的通信界面由一个输入模板、一个输出模板和一个体模板构成。开发者通过选取适当的模板和应用过程来进行模块的构造。Enterprise 是对 Frameworks 的一个改进的版本，其中的模式更为抽象。它将 Frameworks 中的三部分模板结合在一起，称为资源(assets)。系统通过提供一个预先定义的资源的固定集合，开发者使用其中的一些资源并将其结合在一起形成一个资源图来表示并行程序的结构。Code、Code2 和 HeNCE 都是基于可视化程序设计技术的系统，开发者通过节点和边来分别表示并行结构的计算和通信，由此构成并行程序的结构图。开发过程分为两步，首先进行顺序部件的设计，然后将顺序部件组合成并行的结构。Tracs 也是一个图形化的开发系统，但其开发过程是由定义阶段和配置阶段构成。在定义阶段，开发者对应用程序的三个基本部件：消息模型、任务模型和结构模型进行定义；然后使用定义好的部件配置出所需的软件。与面向对象领域中的设计模式概念不同的是，面向对象中设计模式的抽象程度是设计一级的，而上述并行程序开发系统中的设计模式并不只停留在设计一级，而更重要的是，它提供了预先的实现。

在并行程序设计系统中，对设计模式概念的使用存在一个矛盾，就是设计模式的抽象性和并行实现的多样性之间的矛盾。由于设计模式要求通用的特点，决定了其必须具备相当的抽象程度，而为了使其能够描述并行计算问题的详细解决方案，又不得不涉及到计算、通信、同步等具体的问题。因此在上述的系统中，设计模式跨越了设计和实现两级，这样一来，损害了设计模式的抽象性特点，使其难以达到通用的目的，而且由于设计模式的概念不明确，从而难以对开发者提供明确的指导。因此，我们的观点是，根据并行计算的特点，可以将并行计算中的设计模式分为两个抽象级别分别进行定义，设计级的设计模式用于描述问题求解和算法设计的方法，而涉及实现的体

系结构级的设计模式我们则将其定义为结构模式。

定义 4.1 设计模式是对可用于解决一类问题，并可并行实现的一般问题求解和算法设计方法的抽象。

在长期的算法设计研究中，人们总结了一些行之有效的算法设计方法，它们可分别用于解决一类算法问题，也可结合起来进行更为复杂的问题的求解，如分而治之、回溯法、贪心法、分枝限界、动态规划等，这些算法设计方法都具有并行化的可能，如果将这些算法设计方法并行化，则可以应用于一类问题的并行求解，因此我们将这些算法设计方法，包括其并行性的抽象，描述为设计模式。

定义 4.2 结构模式是对并行计算中重复出现的计算模式及其实现策略的抽象。

在并行计算中常见的并行计算模式有数据并行、流水线并行、进程农庄、工作池等，将这些并行计算模式的结构与行为进行抽象描述，即为结构模式。

例如，分而治之是一种典型的问题求解策略，划分所得的子问题可以并行地求解，因此，我们可以将子问题并行执行的分而治之定义为一个设计模式，但并不描述子问题如何并行实现。另一方面，结构模式不考虑问题求解策略，而只是考虑并行实现的模式，例如对几个可并行执行的对象，可以将其实现为进程农庄的结构，也可以实现为数据并行的结构，这就是结构模式。这样，假如对一个应用问题，我们选用分而治之设计模式、进程农庄结构模式进行组合对其求解，所得的就是该问题的以进程农庄结构实现的分而治之算法，而假如我们选用分而治之设计模式、数据并行结构模式进行组合对其求解，所得的就是该问题的以数据并行结构实现的分而治之算法。

通过这样将设计模式分成两个抽象级别的定义，可以避免在问题求解的一开始就把并行结构的因素考虑进来，从而掩盖问题的本质，影响问题的求解，这一点对许多复杂的问题而言，由于其求解本身就很困难，因而显得尤其重要。因此，我们在算法设计阶段主要完成对给定的问题确定其并行求解的策略的功能，而不涉及并行性的实现，对于使用某种特定算法设计方法进行求解的问题，这一阶段的实施则通过使用设计模式支持，而设计模式所描述的算法设计方法本身是具有并行性的。然后再使用特定的并行计算模式(结构模式)将抽象的问题并行求解的描述(并行性包含在算法模式中)具体实现为并行算法。

当然，在 DPaPD 模型中，由于本文工作的目标为 arb 抽象并行程序，而不是具体的并行程序实现，因而其中不会涉及到具体的并行计算模式。因此，本文将不对结构模式的内容作进一步的讨论，有关内容可参见论文[w1]。

4.2 设计模式的有关问题

4.2.1 设计模式的描述

设计模式刻画了软件设计解决策略中的一般性原理及其应用方法、应用结果、应用效果等。要对设计模式进行广泛应用，其中一个关键问题就是如何对设计模式进行概括、精确、完整的描述。

Gamma 提出的设计模式描述的统一格式采用的是对设计模式的各方面内容分别进行描述的方法。对一个设计模式，它必须描述以下方面的内容：

模式名和分类：模式名简洁地描述了模式的本质，其分类表示该设计模式属于什么类别的模式。

意图：意图回答了设计模式是做什么的、它的基本原理和意图是什么、它解决的是什么样的特定设计问题等。

别名：设计模式的其他名称。

动机：用以说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定场景。该情景可以帮助用户理解随后对模式更抽象的描述。

适用性：描述什么情况下使用该设计模式、该设计模式可用来改进哪些不良设计、怎样识别这些情况等。

结构：采用基于对象建模技术(OMT)的表示法对模式中的类进行图形描述。使用交互图说明对象之间的请求序列和协作关系。

参与者：指设计模式中的类和对象以及它们各自的职责。

协作：模式的参与者怎样协作以实现它们的职责。

效果：模式怎样支持它的目标，使用模式的效果和所需做的权衡，系统结构的哪些方面可以独立改变等。

实现：实现模式时需要知道的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题。

代码示例：用来说明怎样用特定语言实现该模式的代码片段。

已知应用：实际系统中发现的使用模式的例子。

相关模式：与该模式紧密相关的模式，其间重要的不同之处，该模式一般和哪些模式一起使用等。

从形式上看，Gamma 的设计模式描述方法是采用自然语言叙述、程序设计语言实现的设计模式具体样本、OMT 图相结合的方法。然而，由于这些方法有些具有二义性(如自然语言叙述)，有些缺乏完整性(如程序样本、对象表示图)，因此，均不能很好地支持设计模式的重用，而且，对设计模式的描述也显得十分冗长而不够精确。

近年来，随着对设计模式研究的逐步深入，国际上陆续出现了一些形式化的设计模式规范描述方法[7,9,10,14,17,18,20]。这些方法可以分成两类。一类是图示化方法，如流行的 UML[79]等。这类方法由于不具备充分的表达能力，因此常使用自然语言进行补充，这就导致了规范描述的不精确和二义性。另一类方法是形式化的数学表示，如 Z、VDM 等。这类方法对精确的规范提供了有力的理论基础，但由于其表示方法的抽象性，在实际应用中很难被一般的设计人员所接受。目前的研究趋势是这两类方

法逐步靠拢，图示化方法选用比较抽象的语言作为补充，如 UML 使用 OCL 作为补充说明语言[80]，而数学表示方法发展出等价的图形表示，如 LePUS[18]等。尽管这些方法与传统设计模式描述方法相比有了很大进展，但目前为止，设计模式的重用仍处于手工阶段，基本上依赖于用户自身的知识和经验[29]。我们对设计模式描述方法的有关研究可参见论文[w5]。

DPaPD 模型中，根据我们对设计模式的定义与 DPaPD 模型中设计模式的使用特点，在形式上，我们使用 PaZ 语言对设计模式进行形式化的描述，并使用自然语言进行辅助说明；在内容上，由于设计模式刻画问题并行求解和算法设计方法的并行抽象，因而对设计模式的描述主要由名字、结构、意图和效果四个部分构成。其中名字为设计模式在系统中的标识，结构为使用 PaZ 语言对模式的内容进行的形式化定义，意图与效果使用自然语言描述，它对该设计模式所刻画的问题并行求解的方法进行解释，并就其使用的方法、效果及其它相关的问题进行说明。

4.2.2 设计模式的分类

由于设计模式在粒度和抽象层次上各不相同，因此需要用一种方法将其组织起来。最初 Gamma 提出了 23 个设计模式，根据目的准则将其分为三类：创建型(Creational)、结构型(Structural)和行为型(Behavioral)，而根据范围准则可分为类模式和对象模式两类。创建型类模式将对象的部分创建工作延迟到子类，而创建型对象模式则将它延迟到另一个对象中。结构型类模式使用继承机制来组合类，而结构型对象模式则描述了对象的组装方式。行为类模式使用继承描述算法和控制流，而行为型对象模式则描述一组对象怎样协作完成单个对象所无法完成的任务。

Buschmann 按照抽象级别将设计模式分类为结构模式、设计模式和短语。结构模式提供整个软件的顶层结构，设计模式为中间层上精化的部件，短语为低级的特定语言的代码序列。

随着设计模式研究的不断深入和应用的不断扩展，设计模式的数量也在不断增加，目前面向对象领域的设计模式已达一百多个。[22]中对现有的设计模式进行了详细的分类。它是按照模式所解决的问题对设计模式进行分类为：

分划(decoupling)：将软件系统划分为独立的部分，每个部分可以独立地创建、修改、替代、重用。

变量管理(variant management)：通过指出变量的共同点来统一管理不同的变量。

状态处理(state handling)：对象状态的通用操作。

控制(control)：执行和方法选择的控制。

虚拟机(virtual machine)：模拟处理器。

便利模式(convenience patterns)：简化的编码。

复合模式(compound patterns)：由其他模式组合而成的模式。

并发(concurrency)：控制并行并发执行。

分布式(distribution)：与分布式系统有关的问题。

关于设计模式分类的研究还可参阅文献[15,22,55]等。

DPaPD 系统是以我们所提出的设计模式概念为基础，目前，本系统的开发处于原形阶段，其中所开发和实现的设计模式的数量很少，因而系统中暂时没有考虑设计模式的分类问题，然而随着系统的成熟与完善和更多设计模式的开发，设计模式的分类将成为一个不可忽略的问题。

4.2.3 设计模式的选择

当面对一个特定的设计问题时，如何从现有的设计模式中选择适当的设计模式？[53]中给出了一些方法来帮助开发者发现适合手中问题的设计模式：

考虑设计模式是怎样解决设计问题的：设计模式可以帮助开发者找到合适的对象、决定对象的粒度、指定对象的接口等。

浏览模式的意图部分：通读模式的意图部分，可以找出与问题相关的一个或多个模式，可以使用分类方法来缩小搜索的范围。

研究模式怎样互相关联：研究模式之间的关系可以指导开发者获得合适的模式或模式组。

研究目的相似的模式：根据对模式的介绍、比较和对照，分析具有相似目的的模式之间的共同点和不同点。

检查重新设计的原因：检查可能引起重新设计的各种原因，看问题是否与之相关，找出哪些模式可以帮助避免这些会导致重新设计的因素。

考虑设计中哪些是可变的：该方法与关注引起重新设计的原因正好相反。它不是考虑什么会迫使设计改变，而是考虑想要什么变化却不引起重新设计。

以上的方法是对开发者进行设计模式选择的一个思想上的指导，而不是通过系统提供设计模式的获取方法。事实上主要还是用户凭借对设计模式功能的了解和自身的设计经验，来选取适当的设计模式进行软件开发，而唯一对用户提供支持的是设计模式的类别(catalog)，类别信息对设计模式的获取具有一定的辅助作用。因此，到目前为止设计模式的获取和使用基本上仍是手工方式，依赖于开发者自身的知识和经验[29]，然而，随着对设计模式研究的广泛开展，越来越多的设计模式被开发出来，与最初的 23 个设计模式相比[53]，数量上已大大增加。显然，这种纯人工的方法已不能满足现实的需要。为了协助用户对设计模式的理解、选取和使用，近年来开始出现对设计模式形式化描述方法的研究(如 4.2.1 节所述)，这些研究通过对设计模式进行比较精确、完整、一致的描述，为用户进行设计模式的获取提供了比较准确的依据，但并没有从本质上改变获取过程由人工实施的现状。规范匹配是一种软部件获取和重用方法，这种获取方法由于携带了部件的语义信息，因而准确率比传统的关键字获取、

侧面分类法等高得多，是目前研究较多的一种软部件获取方法。通过研究我们发现，将设计模式通过一定的方法进行形式化描述，与规范匹配方法相结合，并对这一框架进行一定的扩充，可以实现设计模式的规范匹配，从而提高设计模式获取和使用的自动化程度。这部分研究工作可参见论文[w6]。

当然，DPaPD 系统的实现中，设计模式的选择主要还是通过用户对设计模式中结构和意图内容的理解和自身的经验来进行的，这要求用户具有算法程序设计的有关知识，但不要求并行知识。在系统中实现设计模式的自动获取还是一个比较长远的目标。

4.2.4 设计模式的使用

面向对象软件构造时有效地使用设计模式，是这样一个过程：

大致浏览一遍模式：特别注意其适用性部分和效果部分，确定它适合应用问题。

回头研究结构部分、参与者部分和协作部分：确保对该模式的类和对象以及它们的关联的正确理解。

看代码示例部分：以帮助模式的实现。

选择模式参与者的名字：设计模式参与者的名字通常过于抽象而不会直接出现在应用中。然而，将参与者的名字和应用中出现的名字合并起来是有用的。这会帮助在实现中更显式地将模式体现出来。

定义类：声明它们的接口，建立它们的继承关系，定义代表数据和对象引用的实例变量。

定义模式中专用于应用的操作名称。

实现执行模式中责任和协作的操作。

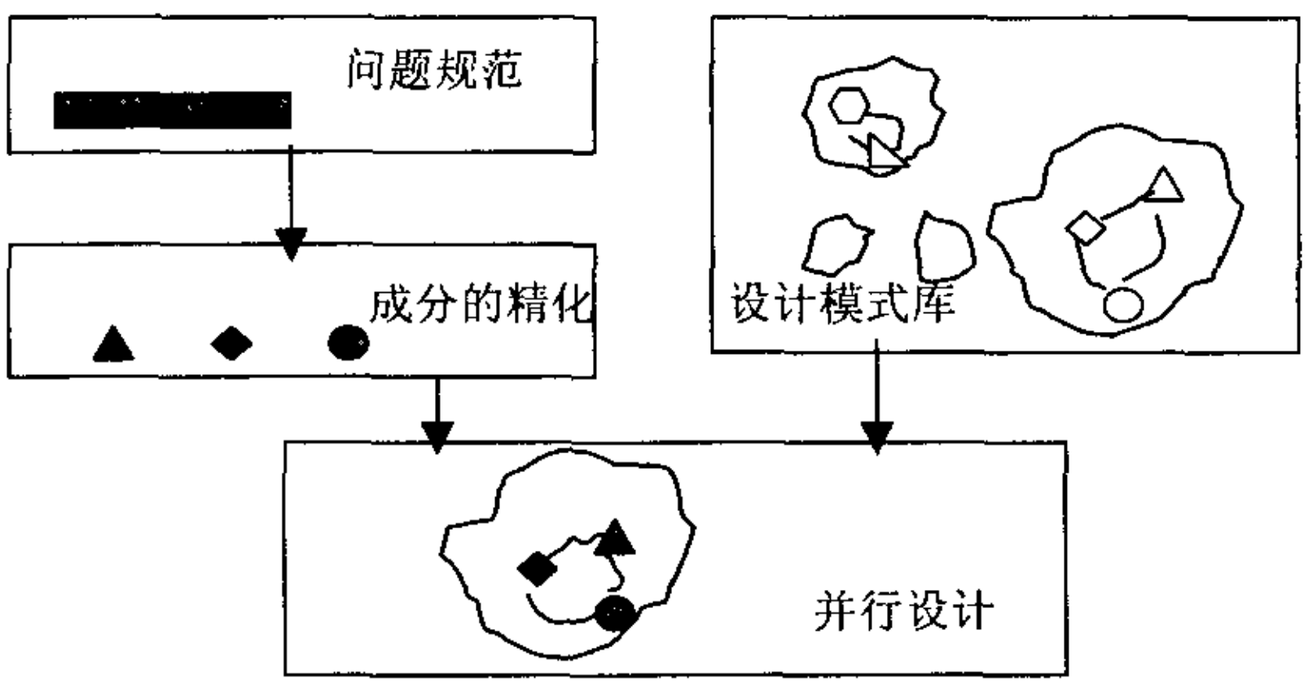


图 4.1 设计模式的使用机制

Fig. 4.1 Using of Design Patterns

DPaPD 系统中，设计模式的使用如图 4.1 所示。设计模式库中的每个设计模式

描述了一个特定的算法设计方法或策略(参见定义 4.1)，而不包含与应用问题相关的因素。图中设计模式中的各种形状的对象表示设计模式的基本组成成分。灰色表示与应用问题有关的性质。根据应用问题的规范描述可以知道问题的基本性质与类别，从而选取一个设计模式对问题进行求解。根据设计模式结构部分的描述，将问题规范分解成符合设计模式成分的一些定义，这些定义是问题相关的，是针对具体的应用问题对设计模式中对应成分的精化，将他们与选定的设计模式组合在一起，则产生求解该问题的一个并行设计。

4.3DPaPD 系统中的设计模式实例

4.3.1 并行分而治之

名字

Parallel Divide&Conquer

意图

分而治之是一种重要的算法设计技术。对于一个输入规模为 n 的函数或问题，先用某种方法把输入划分成 k 个子集，$1 \le k \le n$，从而产生 l 个类型与原问题相同，但规模更小的子问题。在解出 l 个子问题后，再用某种方法把它们的解组合成原来问题的解。如果子问题相当大，则重复以上的步骤，直到子问题分得足够小，不必再进行分割，而可以直接求出它们的解。

当划分所得的子问题满足并行兼容条件时，可以对这些子问题并行求解，这样的并行求解方法称之为并行分而治之。并行分而治之设计模式是对并行分而治之问题求解方法的抽象，常用于数值计算的问题求解。

结构

```
[DATA,RESULT]
    datas: DATA
    result: RESULT
    divide:   DATA→F DATA
    merge:   F REAULT→RESULT
    baseoperate: DATA→ RESULT
    basedatas: P DATA
    D&C: DATA→RESULT
    D&C datas = if basedatas datas then baseoperate datas
                    Else merge D&C|(divide datas)|
    result = D&C datas
```

效果

从 Parallel Divide&Conquer 的结构中可以看出，该模式的成分包括有类型 DATA、RESULT，数据 datas、result 和函数 divide、merge、baseoperate、basedatas。根据 DPaPD 模型下设计模式的使用方法，在使用该模式时，用户需要根据具体应用问题分别对以上成分进行精化，分别精化为具体问题的数据类型、结果类型、问题的输入、输出数据和对问题进行划分、合并、直接求解(即不进行划分)、直接求解条件的定义，然后与该设计模式进行组合，产生求解该问题的一个并行分而治之的设计。

通过该模式的使用，用户不需了解任何并行设计与实现的知识，也不需进行具体的程序编码，只需在 PaZ 的抽象框架下根据具体问题和模式的要求进行一些有关的定义，即可得到该框架下的一个并行设计，并进而通过 DPaPD 系统的精化、转换最终得到一个求解该问题的递归并行程序。

4.3.2 单步分而治之

名字

One-step Divide&Conquer

意图

分而治之是一种常用的问题求解方法(参见 4.3.1 节)，单步分而治之是对问题只进行一次划分，得到若干个类型与原问题相同，但规模更小的子问题，将所得的子问题并行运算，再将结果合并起来得到原问题的解的方法。设计模式 One-step Divide&Conquer 是对这一问题求解方法的抽象，常用于数值计算问题的求解。

结构

[DATA, RESULT]

```
datas: DATA
result: RESULT
split:  DATA → F DATA
baseoperate:  DATA →  RESULT
merge:  F RESULT →  RESULT
one-step-D&C: DATA→ RESULT
---
one-step-D&C datas = merge baseoperate |( split datas )|
result = one-step-D&C datas
```

效果

模式 One-step Divide&Conquer 中的基本成分包括类型 DATA、RESULT，数据 datas、result 和函数 split、merge、baseoperate。在使用该模式时，用户需要根据具体应用问题分别对以上成分进行精化，分别精化为具体问题的数据类型、结果类型、问

题的输入、输出数据和对问题进行划分、合并、直接求解的定义，然后与该设计模式进行组合，产生求解该问题的一个单步并行分而治之的设计。同样地，使用该模式，用户不需了解任何并行设计与实现的知识，也不需进行具体的程序编码，只需在 PaZ 的抽象框架下根据具体问题和模式的要求进行一些有关的定义，即可得到该框架下的一个并行设计，并进而通过 DPaPD 系统的精化、转换最终得到一个求解该问题的并行程序。

从问题求解的基本思想上看，该模式与 Parallel Divide&Conquer 非常相似，但由于该模式只对问题进行一次划分，因此适用于规模不太大的问题的求解。另一方面，从模式的结构可以看出，与 Parallel Divide&Conquer 不同，该模式所产生的设计中不包含递归，因此实现效率更高。因此，用户可以根据应用问题的规模和具体运行环境的特点进行权衡，选择适合的设计模式进行问题的求解。

4.3.3 并行分枝限界设计模式

名字

Parallel Branch&Bound

意图

分枝限界算法设计策略是一种用于求解优化组合问题的有效方法。简而言之，分枝限界方法的搜索过程可以用树表示。树的根就是搜索的开始，考虑所有的可能解。对可能解的组成作出限制将所有的可能解划分成子集，在树中用根的子节点表示。对这些子集再作限制，又可将子集划分成更小的子集，在树中得到更多的子节点。有时，对可能解作了某些限制以后，可以确定得到的子集中所有的可能解都不是问题的解，则可以停止对这个子集的可能解作进一步的限制，即不必产生这个节点的子节点了，可以停止对以这个节点为根的子树(即可能解的子空间)的搜索。

并行分枝限界设计模式是基于 Mitten [9]的抽象并行分枝限界模型构造的。在该模型中，分枝以算子 β 来表示，它以一个子问题的集合作为参数，产生一个新的子问题集合。另一个算子 PRUNE 作用于一个子问题的集合，其结果为从该集合中将那些经探测认为不可能产生可行解的子问题，还有那些已经超越了限制的子问题去除掉所得的集合。 若这些子问题满足并行兼容条件，则可以并行地求解，我们称之为并行分枝限界方法。模式 Parallel Branch&Bound 是对并行分枝限界问题求解方法的抽象，常用于组合优化问题的并行求解。

结构

[subproblem]

| A, A'：F subproblem
| β：F subproblem→F subproblem
| PRUNE: F subproblem → F subproblem
| B&B: F subproblem → F subproblem
| B&B A = $_{\mu X}$• if $_{A'\neq A}$ then A'=PRUNE|(β|(A)|)|;X

模式 Parallel Branch&Bound 的结构中含有一个 μ 表达式。一般来说，μ 表达式可以精化为一个迭代：$\mu X \bullet$ *if* G then S;X else skip $\subseteq$ while G do S od。

效果

该模式中的成分包括有子问题的类型 subproblem，问题的可能解的集合 A、A'，分枝函数 β 和修剪函数 PRUNE。使用该模式时，用户需要根据具体应用问题分别对以上成分进行精化，然后与该设计模式进行组合，产生求解该问题的一个并行分枝限界的设计。通过该模式的使用，用户不需了解任何并行设计与实现的知识，也不需进行具体的程序编码，只需在 PaZ 的抽象框架下根据具体问题和模式的要求进行一些有关的定义，即可得到该框架下的一个并行设计，并进而通过 DPaPD 系统的精化、转换最终得到一个求解该问题的并行程序。

分枝限界方法是一种优化的穷举方法，它是一个对空间状态树的搜索过程，常用于求最优解的问题，其中很多为 NP 完全的问题。通过使用该模式进行问题求解，由于其将独立的可能解的子集进行并行的处理，在实现时可以提高时间效率，这一点对于复杂问题的求解具有重要的意义，然而，使用该设计模式不能提高空间效率，相反，并行的处理还可能导致额外的空间开销，这是使用该模式进行问题求解时需要考虑与权衡的一个因素。

4.3.4 设计模式的使用实例

本节我们通过一个实例来讨论 DPaPD 模型中设计模式的使用情况，以整型数组求和问题为例。该问题定义为：现有 m 个处理器，a: array[1..N] of integer，求 sum=$\sum_{i=1}^{N} a[i]$。

首先进行设计模式的选择。这是一个数值计算的问题，浏览一遍模式，从各设计模式的意图部分来看，并行分而治之模式和单步分而治之模式都可用于该问题的求解。假设 N 不是一个很大的数，那么从并行分而治之模式和单步分而治之模式的效果来看，可以选择单步分而治之模式。

从单步分而治之的结构和效果来看，使用该模式，我们需要根据数组求和问题分

别对模式 One-step Divide&Conquer 中的基本成分 DATA、RESULT、datas、result、split、merge、baseoperate 进行精化，分别精化为数组求和问题的数据类型、结果类型、问题的输入、输出数据和对问题进行划分、合并、直接求解的定义。

先看数据成分的精化。根据问题的定义，其输入类型为整型数组，输出类型为整型，输入数据为 a，输出数据为 sum，显然，数据成分的精化包括：

DATA ⊆ array[1..N] of integer

RESULT ⊆ integer

datas ⊆ a

result ⊆ sum

再看函数成分的精化。假设函数 split、merge、baseoperate 分别精化为函数 split_sum、merge_sum、baseoperate_sum，那么根据问题的定义，split_sum 的功能是将数组 a 进行划分，假设划分成 m 个部分，则得到集合 {a[1..N/m], a[N/m+1..2N/m], … … ,a[(m-1)N/m+1..N] }。使用 PaZ 语言定义 split_sum 为：

a: array [1..N] of integer
split_sum: array[1..N] of integer→ F array[1..N/m] of integer
---
split_sum a = {a[1..N/m], a[N/m+1..2N/m], … … ,a[(m-1)N/m+1..N] }

同理，可以定义 merge_sum 和 baseoperate_sum，其功能分别为对集合中的元素进行求和与对数组中的元素进行求和。

S: F integer
merge_sum: F integer→ integer
---
merge_sum S = $\sum$ (si: si∈S: si)

ai: array [1..N/m] of integer
baseoperate_sum: array[1..N/m] of integer→ integer
---
baseoperate_sum ai = $\sum_{i=1}^{N/m} ai[i]$

通过将以上定义与 One-step Divide&Conquer 相结合，得到求解整型数组求和问题的设计为：

a: array [1..N] of integer

sum: integer
split_sum 的定义
merge_sum 的定义
baseoperate_sum 的定义

one-step-D&C_sum : array[1..N] of integer→integer

one-step-D&C_sum a = merge_sum baseoperate_sum |(split_sum a)|

sum = one-step-D&C_sum a

通过该实例对单步分而治之模式的使用可以看出，问题的求解是从问题的规范定义到设计的一个精化过程，而且，最终是对模式中函数 one-step-D&C 的精化。如果使用 Z 语言中过程精化的思想来解释，这一问题求解的过程就是要找到一个全函数 Abs：

Abs

States of one-step-D&C
States of one-step-D&C_sum

各状态量的对应关系

(其中 states of one-step-D&C 表示 one-step Divide&Conquer 模式的基本成分，states of one-step-D&C_sum 表示基本成分精化后的结果)使得 one-step-D&C 和 one-step-D&C_sum 之间满足过程精化的两个条件[参见 61]，那么称 one-step-D&C_sum 是 one-step-D&C 的一个正确的精化。为了证明 one-step-D&C_sum 是 one-step D&C 的一个正确精化，需要证明 one-step-D&C_sum 中的每一个成分在 Abs 的解释下为 one-step-D&C 相应成分的正确精化。我们非形式地来看 split_sum 函数：对于数组 a 的任意取值，split_sum 将 a 划分成 m 个部分，数组 a 的值不发生改变，$S_c=S_c'$，而 $Abs(S_c)$为 datas 的取值，即为 a 的取值，因此，split_sum 为 split 的正确精化。

在 DPaPD 模型对于精化正确性的解释中，我们不是通过对每一个成分精化的正确性进行证明，而是通过对所得的整个设计的验证，证明其在问题规范前置条件的约束下满足问题规范的后置条件。设问题规范为 S，设计为 D，则定义设计的正确性为：$D \wedge pre\ S \Rightarrow post\ S$。这部分的有关内容我们将在第六章中作具体的讨论。

4.4 小结

并行程序开发的困难存在于两个方面：一是并行设计的困难，二是并行程序编码的困难。前者涉及从问题规范到并行设计的问题并行求解过程，后者涉及并行设计在复杂的并行环境下的实现。重用前人的经验和知识是简化软件设计和程序开发的实用途径。将新的工作建立在已有工作的基础之上，可以避免重复的劳动，有效降低程序开发的难度，这一点对于复杂的并行程序开发尤为重要，这就是我们设计模式的基本思想。在本文的工作中，我们的研究目标是抽象 arb 并行程序的设计开发，因此，我们所定义的设计模式主要用于问题的并行求解。本章描述了我们对设计模式的定义，并讨论了设计模式的描述、分类、选择和使用问题。在 DPaPD 系统中，设计模式采用 PaZ 语言进行描述，这样使得使用设计模式从问题规范到并行设计的问题并行求解过程可以在一个统一的语言框架下平滑的进行。设计模式的使用过程是将具体的、与问题相关的成分来作为设计模式所提供框架中的相应抽象成分的精化，并与设计模式相结合得到并行设计。文中列举了一些设计模式的具体实例，并举例说明了其使用过程。

第五章 PaZ 到 arb 程序的精化

5.1 基本类型定义

PaZ 中所定义的基本类型，在精化为 arb 程序时应将其定义为一个具体的类型。由于在规范定义时，往往对某些类型的具体结构并不关心，而只关心这些基本类型的使用和意义上的区别，例如：

[NAME, DATE]

birthdaybook

known: P NAME

birthday: NAME → DATE

known = dom birthday

在该 PaZ 规范说明中，并不关心 NAME 和 DATE 的具体结构，而只是为了说明 NAME 和 DATE 是两个类型，NAME 表示姓名，DATE 表示日期。而在 arb 程序中，我们则必须关心类型的具体结构，而这具体结构又是根据类型在具体问题中的意义来决定的。对于上例中的类型，我们可以根据其意义定义 NAME 为字符串类型，DATE 为由年月日构成的结构类型（当然，也可以根据实现的需要定义为其他的具体类型）：

```
typedef   char   *NAME;
typedef   struct
   {
      int year;
      int month;
      int day;
    }  DATE;
```

因此，一般而言，基本类型定义的精化规则是：

[A，B，….]

⊆ typedef 具体类型 1 A

typedef 具体类型 2 B

… …

5.2 数据结构

在 PaZ 并行设计中，我们使用的是较为抽象的数学结构来描述数据。而在 arb 程序中，则使用较为具体的数据结构。Arb 模型从语言的角度而言是在 FORTRAN 语言或 C 语言上扩充了表示并行运算的 arb 语言结构的语言。本文中的 arb 语言为在 C 语言基础上的扩充。PaZ 设计比 arb 程序更简短，因为它忽略了数据表示的细节。而作为 arb 程序，其数据表示则更接近于具体的程序。通常，一种抽象的数学结构可以精化为多种不同的数据结构，而数据结构的选择通常依赖于数据的数量、目标系统的资源情况，和运行效率等因素。

5.2.1 集合

一般情况下，集合类型可以精化为带有布尔标志的数组，数组中的元素为集合元素类型，而标志表示该元素是否在集合中。如果集合的数据量很大，此时可以精化为其他的数据结构，例如数据文件等。

例如，对一个预定义类型的集合：

```
┌── S ──
│  …. ….
│  s: F integer
│  …….
└──────────
```

精化为 arb 程序段：

```
#define N #s
int s[N]
```

对于自定义类型的集合：

```
FAULT::=overload | line_voltage | overtemp | ground_short
┌── PS ──
│  … …
│  faults: P FAULT
│   … …
└──────────
```

集合 faults 精化为一个数组：

```
typedef enum {OVERLOAD, LINE_VOLTAGE, OVERTEMP,
GROUND_SHORT} fault;
#define N_FAULTS GROUND_SHORT+1
```

int faults[N_FAULTS]

5.2.2 序列

序列类型通常可以与集合类型一样，精化为数组。序列类型与集合类型的根本区别在于序列类型中的元素排列是有序的，而集合中元素的排列是任意的。数组的下标可以确定元素在一个序列中的位置，因此，数组类型可以很好地作为序列类型的实现。

5.2.3 关系

在 PaZ 中经常使用关系来表示数据结构。例如

Phone:NAME→N

Dom phone=subscribers

这个关系表示 NAME×N 类型的序对的一个集合，因此可以实现为结构类型

```
typedef struct
 { NAME:: name;
    integer:: phone-num;
}phone_rec
phone_rec phone[n]
```

5.2.4 函数

PaZ 中的函数也经常用于表示数据。函数是一个二元关系，其第一元为唯一的。因此作为数据表示的函数可以实现为文件或每个元素只有一个唯一关键字的数据结构。关键字对应函数的定义域，而元素对应值域。例如数组，数组下标对应定义域，数组元素对应值域。例如

| u,v:Z→Z

实现为 integer:: u[N],v[N]

在公理描述中定义的函数，公理描述中的谓词部分刻画了函数的定义域与值域之间存在的约束，此时函数精化为 arb 中的一个子过程，详见 5.6 节。

5.3 状态模式

状态模式通常实现为内容经常变化的数据结构，例如实现为普通的程序变量，而模式的绑定为变量所取的值。例如状态模式 s 含有两个状态变量 x 和 y：

s= [x,y: Z]

实现为 int x,y

另外，状态模式也可以实现为结构类型，结构类型中的成员为模式中的状态变量。例如上式也可实现为：

typedef struct

{int x,y } s

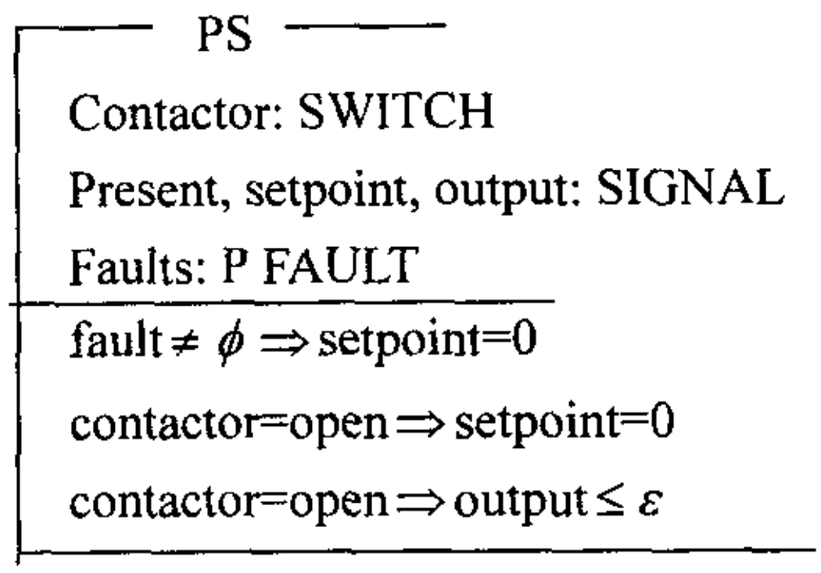
现在我们可以说明一些该类型的变量，因此 PaZ 的声明

| sa,sb,sc: s

精化为 arb 中的声明

s sa,sb,sc

再看一个例子：



可以精化为下面的程序段：

```
typedef int signal;
typedef enum {OPEN, CLOSED} switch;
/*PS state*/
typedef struct power_supply
{
   switch contactor;
   signal preset, setpoint, output;
   int faults[ N_FAULTS]
 } PS;
```

5.4 运算的精化

5.4.1 一般表达式

一般的算术表达式、逻辑表达式、关系表达式的精化只需使用 arb 语言中的算术运算符、逻辑运算符、关系运算符直接代替 PaZ 表达式中相应的运算符即可。例如算术运算符+, *, div, mod 精化为+, *, /, %；关系运算符=和≠分别精化为= =和!=；逻辑运算符∧和∨分别精化为成&&和|| 等等。

看一个简单的表达式的精化：

$$x \bmod 2 \neq 0$$

$$\subseteq \quad x\%2!=0$$

5.4.2 集合运算

集合运算的精化依赖于集合本身的精化，当集合精化成一个数据结构时，集合的运算精化为该数据结构上的一些运算。在 arb 中，通常实现为一个子过程。该子过程依赖于集合类型所精化成的数据结构。在尚不知集合精化成什么数据结构之前，集合运算的精化规则可以非形式化地描述，例如

$x \in s \quad \subseteq$ 在数据结构 s 中搜索 x

若确切地知道表达式中的集合精化为何种数据结构，则可以形式化地描述集合运算的精化规则。例如，若集合类型精化为一个以集合中元素为下标的布尔数组，此时

$x \in S \quad \subseteq \quad S[x]$

有时集合的定义是通过一个谓词进行定义的，此时判断元素是否属于集合的运算可以精化为对该元素的谓词运算，即若有集合{x:X|p(x)},则有精化规则：

$$x \in \{x:X|p(x)\} \subseteq p(x)$$

例如判断一个元素 x 是否属于奇数集合，有以下精化过程：

$$odd(x) \subseteq \quad x \in (odd\_)$$

$$\subseteq \quad x \in \{x:Z \mid x \bmod 2 \neq 0\}$$

$$\subseteq \quad x \bmod 2 \neq 0$$

$$\subseteq \quad x\%2 !=0$$

$$\equiv \quad x\%2$$

而如果集合类型精化为一个数据文件，此时

$x \in S \quad \subseteq$ 在文件 S 中搜索 x 的子过程

5.4.3 函数作用

PaZ 中的函数作用通常精化为 arb 中的数据结构或可执行的 arb 程序段，这要视表达式中函数的精化情况而定。

若函数 u 实现为数据结构，则函数作用 $u(x)\subseteq$ 在数据结构 u 中搜索关键字为 x 的项；若函数 f 实现为可执行的程序段，则 $f(x)\subseteq$ 以 x 为参数计算 f。即：

$u(x) \subseteq u[x]$ （当 u 精化为数组结构时）

$f(x) \subseteq f(x)$ （当 f 精化为一个 arb 子过程时）

5.4.4 赋值

当一个变量的值与原始时的值相同时，该表达式精化为一个空语句：

$x'=x \subseteq$ 空语句

否则，若表达式中有一个变量的值发生了改变，则精化为一条赋值语句，例如 x 的值发生改变成了表达式 e 的值，即：

$x'=e \wedge y'=y \wedge z'=z \wedge \ldots\ldots \subseteq x=e$

该精化规则仅适用于一个变量的值发生变化的情况，若有多个变量的值发生改变，则不能简单地将其精化为多条赋值语句，例如 $x'=y \wedge y'=x$ 不能简单地精化为两条赋值语句 x=y;y=x，此时需要做数据流分析才能决定赋值的先后次序。

我们来看下面的模式：

Op
△ S
$X'=e_1(x,y)$
$Y'=e_2(x,y)$

该操作模式改变两个变量 x 和 y 的值，由于表达式 e_1 和 e_2 中均使用了改变之前的 x 和 y，因此需要引入临时中间变量来保存某些变量的值，在此例中需引入变量 t 保存变量 x 改变之前的值，即其精化规则为：

$x'=e_1(x,y) \wedge y'=e_2(x,y) \wedge z'=z\ldots\ldots$

$\subseteq$ $t=x;\quad x=e_1(x,y);\ y=e_2(t,y)$

当然，如果在变量不进行交互引用的情况下，则不需引入中间变量，如：

$x'=e_1(x) \wedge y'=e_2(y) \wedge z'=z\ldots\ldots$

$\subseteq$ $x=e_1(x);\ y=e_2(y)$

然而，并非所有变量的改变都可以直接精化为赋值语句，因为某些变量并不能作为可赋值的变量，例如具有复杂结构的数据类型的变量。此时，在不知道该变量类型的具体精化类型时，只能使用非形式的精化规则，例如对于集合类型变量的改变：

$S'=S \cup \{x\} \subseteq$ 将 x 加入数据结构 S

而如果变量类型的精化已知的情况下，例如集合类型精化为以元素为下标的布尔数组，则其精化规则可以形式化地描述为：

$S'=S \cup \{x\} \subseteq S[x]=TRUE$

5.4.5 合取运算

一般地说，合取运算的精化没有直接的方法，要视具体情况而定。在合取式之间基本不进行交互的情况下，可以精化为条件语句，称为卫式命令规则：

$p \wedge s \subseteq$ if (p) s

例如 $x=e_1 \wedge x'=e_2$

$\subseteq$ if $(x=e_1)$ $x'=e_2$ （卫式命令规则）

$\subseteq$ if $(x==e_1)$ $x'=e_2$ （关系运算符的精化规则）

$\subseteq$ if $(x==e_1)$ $x=e_2$ （赋值规则）

另外，合取式也经常表示一种并列的关系，特别是当合取分量为表示变量取值的等式时，此时，每个分量分别进行精化。

$p \wedge s \subseteq$ p 的精化; s 的精化;

例如 $x=e_1 \wedge y=e_2$

$\subseteq$ $x=e_1$; $y=e_2$; （e_1 和 e_2 必须满足赋值规则的约束）

而如果各个合取式之间包含有相同的变量，则基本没有合适的精化规则。例如一个整数 n 可以表示为因子 d 的倍数再加上余数的形式，即合取表达式；

$d \neq 0 \wedge n=q'*d+r' \wedge r'<d$

此时我们需要找到满足条件的 q'和 r'才能满足该合取表达式，即可以实现为循环语句：

for(q=0,r=n;r>=d;q++) r=r-d

在这种情况下，其精化所得的结果往往不能从表达式本身的成分上得到，而是需要通过设计一个算法来使得该表达式得到满足，因此无法通过直接的精化规则来得到对应的 arb 程序段。

5.4.6 析取运算

析取运算通常精化为条件分支语句

$(p \wedge s) \vee (q \wedge t) \subseteq if\,(p)s; else \quad if\,(q)t$

这条规则的满足必须在 p 和 q 的表示非重叠的情形下，即其中只有一个谓词可以为真。在该式中 t 可以进一步为一个析取式，这样我们可以重复地使用这条规则得到：

$(p \wedge s) \vee (q \wedge t) \vee (r \wedge u) \ldots\ldots \subseteq if\,(p)s; else \quad if\,(q)t\,; else\ if\,(r)u \ldots\ldots$

另外，析取式中还有一些特殊情形，例如：

$(p \wedge s) \vee (\neg p \wedge t) \subseteq if\,(p)s; else$ t

当 s 和 t 为描述同一变量的等式时，我们还可以使用下面的规则：

$(p \wedge x'=e_1) \vee (\neg p \wedge x'=e_2) \subseteq x'=$ if p then e_1 else e_2

$\subseteq$ $x=p?e_1:e_2$

5.4.7 新变量的引入

在程序的精化、转换过程中，有时需要引入一些在 PaZ 所描述的设计中没有的变量。例如赋值精化规则中引入的变量 t 用以记录变量 x 的原始值，另外，当表达式太长时，需要精化成若干个语句时，也要引入新的变量。

程序中的新变量相当于 PaZ 中的局部变量。下面的规则说明我们可以引入一个新的变量 x 来代替表达式 e 的所有出现：

$s \Leftrightarrow (\text{let } x= =e \bullet s[x/e])$

其中 s[x/e]表示将表达式 s 中所有表达式 e 的出现以变量 x 替代，新变量的精化规则如下：

$(\text{let } x= =e \bullet s(x)) \subseteq x=e;\ s(x)$

下面我们看一个精化的例子：

$p(y') \wedge y'=f\ x$	(PaZ 的表达式)
$\Leftrightarrow p(f\ x) \wedge y'=f\ x$	
$\Leftrightarrow (\text{let } t= =f\ x \bullet p(t) \wedge y'=t)$	（新变量引入规则）
$\subseteq t=f\ x;\ p(t) \wedge y'=t$	（新变量精化规则）
$\subseteq t=f\ x;$ if $(p(t))\ y'=t$	（合取式精化规则）
$\subseteq t=f\ x;$ if $(p(t))\ y=t$	（赋值精化规则）
$\subseteq t=f(x);$ if $(p(t))\ y=t$	（函数作用精化规则）

5.4.8 量词

使用集合的量词精化为对集合中元素的循环迭代的检测过程，量词表达式中的谓词部分为检测过程，全称量词表示集合中的每个元素均使检测结果为真，而存在量词则表示至少有一个元素通过检测使之为真。引入布尔标识变量 b，全称量词表达式和存在量词表达式的精化规则如下：

$\forall x : S \bullet P(x) \subseteq$ b=1; for (x in S) if (!P(x)) then b=0

$\exists x : S \bullet P(x) \subseteq$ b=0; for (x in S) if (P(x)) then b=1

对于集合 S 中的 x，将表达式 P(x)看作作用于 x 的函数，那么当 P(x)满足可并行兼容条件时，即 $\forall x : S \bullet \Updownarrow P(x)$ 时，该精化规则可进一步为：

$\forall x : S \bullet P(x) \subseteq$ b=1; arball (x in S)
if (!P(x)) then b=0
end arb

$\exists x : S \bullet P(x) \subseteq$ b=0; arball (x in S)
if (P(x)) then b=1
end arb

若集合 S 精化为一个数组 S, 则精化规则可完全形式化地表示：

$\forall x : S \bullet P(x)$ $\subseteq$ b=1; for (i=0; i<n; i++) if (!P(S[i])) b=0

$\exists x : S \bullet P(x)$ $\subseteq$ b=0; for (i=0; i<n; i++) if (P(S[i])) b=1

同理，当集合 S 中所有元素 x 的 P(x)满足可并行兼容条件时，该精化规则可进一步为：

$\forall x : S \bullet P(x)$ $\subseteq$ b=1; arball (i=0:n-1)

if (!P(S[i])) then b=0

end arb

$\exists x : S \bullet P(x)$ $\subseteq$ b=0; arball (i=0:n-1)

if (P(S[i])) then b=1

end arb

事实上，对于集合 S 中的所有元素 x，$\forall x : S \bullet \Updownarrow P(x)$ 总是为真的，我们可以通过运算符$\Updownarrow$的定义来证明这一点。

$\forall x : S \bullet \Updownarrow P(x) \quad \Leftrightarrow \quad \forall x_1, x_2 : S \bullet x_1 \neq x_2 \Rightarrow P(x_1) \Updownarrow P(x_2)$

将 P(x)看作作用于 x 上的函数，即：

$P(x_1)$: $\{x_1\} \rightarrow \{TRUE,FALSE\}$

$P(x_2)$: $\{x_2\} \rightarrow \{TRUE,FALSE\}$

根据函数并行兼容运算$\Updownarrow$的定义（参见 3.3.1 节）有：

$op_1 \Updownarrow op_2$ =

$\forall s_1, s_2 : (s_1, s_2) \in op_2 \bullet s_1 \in x_1 = s_2 \in x_1$ .....(1)

$\wedge \forall s_1, s_2 : (s_1, s_2) \in op_1 \bullet s_1 \in x_2 = s_2 \in x_2$ .....(2)

$\wedge \forall s_1 : s_1 \in x_1 \wedge s_1 \in x_2 \bullet$

$(\forall s_2, s_3 : (s_1, s_2) \in op_1 \wedge (s_2, s_3) \in op_2 \bullet (\exists s_2' \bullet (s_1, s_2') \in op_2 \wedge (s_2', s_3) \in op_1)$

$\wedge \forall s_2, s_3 : (s_1, s_2) \in op_2 \wedge (s_2, s_3) \in op_1 \bullet (\exists s_2' \bullet (s_1, s_2') \in op_1 \wedge (s_2', s_3) \in op_2))$..(3)

由于 $P(x_2)$中只有一个转换为$(x_2,TRUE)$或$(x_2,FALSE)$，而

$x_1 \neq x_2 \quad \Rightarrow \quad x_2 \in \{x_1\} \Leftrightarrow FALSE$

同时， $TRUE \in \{x_1\} \Leftrightarrow FALSE$,且 $FALSE \in \{x_1\} \Leftrightarrow FALSE$，因此(1)式为真。同理可以得到（2）式为真。

由于 $x_1 \neq x_2 \Rightarrow \{x_1\} \cap \{x_2\} = \phi$，因此（3）式为真。

综上得 $x_1 \neq x_2 \Rightarrow P(x_1) \Updownarrow P(x_2)$ 对于任意 x_1，x_2 成立，因此 $\forall x_1, x_2 : S \bullet x_1 \neq x_2 \Rightarrow P(x_1) \Updownarrow P(x_2)$。

实际上，由于集合中的元素是不重复的，因此对集合元素的操作 P(x)满足不相交并行的条件，该条件强于可并行兼容条件，因此 P(x)一定满足可并行兼容条件。

5.5 操作模式

操作模式通常用于表示状态的变化。它可以实现为改变程序变量的子过程，该子过程可以改变程序变量的值。在比较简单的情况下，一个模式类型只有一个绑定，即精化为一个程序变量，此时该操作模式精化所得的过程不需传递参数，过程直接修改全局变量，例如 PaZ 的模式：

$s \mathrel{\widehat{=}} [x,y: Z]$

$op \mathrel{\widehat{=}} [\ \Delta s \mid x'=x+y]$

可以精化为：

```
int   x, y;
void op(void)
 {
  x=x+y;
 }
```

而另一种更复杂的情况是 PaZ 中使用绑定运算符的情况。在下面的例子中，操作 SumOp 与上例中的 op 具有相同的功能，然而它是通过函数 sum 作用于模式 S 的绑定来定义的：

$s \mathrel{\widehat{=}} [x,y: Z]$

$$\frac{sum: s \rightarrow Z}{\forall s \bullet sum(\theta s) = x + y}$$

$sumop \mathrel{\widehat{=}} [\ \Delta s \mid x'=sum(\theta s)]$

此时，θs 表示 s 的所有绑定，当模式 s 只有一个实例时，sum 和 SumOp 的实现不需传递任何参数，对绑定的访问就是对全局变量的访问。此时精化结果为：

```
int x,y;
int sum(void)
 {
 return x+y;
 }
void sum_op(void)
 {
  x= sum( );
```

}

而若 S 有多个实例时，绑定则必须显式地通过参数的形式表示出来，此时精化的结果为：

```
typedef struct
 {
  int x, y;
 } s;
 s sa,sb,sc;
 int sum(s *p)
  {
     return(*p).x+(*p).y;
  }
void sum_op(s *p)
 {
   (*p).x=sum(s);
 }
```

5.6 公理描述

PaZ 中的公理描述由一个或多个全程变量与关于它们的谓词组成，谓词表达了这些变量之间所满足的约束。经常地，公理描述用于说明函数变量，其谓词部分则刻划了该函数的性质。例如：

$$\frac{\text{Succeeding: } Z \to Z}{\forall i : Z \bullet succeeding(i) \Leftrightarrow i + 1}$$

此时，公理描述中的函数精化为 arb 中的一个子过程：

```
int succeeding (int i)
   {
     return (i+1);
   }
```

其中，子过程的返回值类型和参数类型从公理描述中的声明部分获得，而其体则由公理描述中谓词部分中与该函数变量相关的谓词表达式精化而来。上例是一个明显的函数变量的声明，在 PaZ 中，还有一种公理描述中变量声明也精化为 arb 中的子过程，例如：

$$\frac{\text{odd: P } Z}{\forall i : Z \bullet odd(i) \Leftrightarrow i \bmod 2 \neq 0}$$

该公理描述中的变量odd也精化为子过程，这是因为PaZ中没有布尔类型的定义，因此往往使用集合类型变量表示一个值域为布尔类型的函数，上例中的 odd 实际上可以看作函数 odd: Z→BOOLEAN，因此，使用公理描述中函数的精化方法，得到：

```
int odd( int i)
 {
    return (i%2);
  }
```

5.7 模式表达式

模式表达式由模式和模式运算符构成，5.3 和 5.5 节分别对状态模式和操作模式的精化进行了说明。尽管如此，在模式表达式的实现时，却可以不必将其中的每个模式进行精化，再将其进行组合，而是可以通过模式演算的规则先得到结果模式，再将其进行精化。例如对于模式的析取运算：

$$op1 \triangleq [\ \triangle\ s \mid p \wedge s1']$$

$$op2 \triangleq [\ \triangle s \mid \neg p \wedge s2']$$

$$op12 \triangleq op1 \vee op2$$

此时可以不必先实现 op1 和 op2，再将其组合，而是先计算出 op12：

$$op12 = [\triangle s \mid (p \wedge s1') \vee (\neg p \wedge s2')]$$

再使用操作模式和表达式的精化规则，op12 精化为：

```
void op_12(void)
 {
     if (p) s1;   else s2;
  }
```

再看一个模式合取运算的例子：

$$Opx \triangleq [\triangle s \mid p(x) \wedge x' = f\ y]$$

$$Opy \triangleq [\triangle s \mid q(y) \wedge y' = g\ x]$$

$$Opxy \triangleq Opx \wedge Opy$$

计算模式表达式的结果 Opxy 为：

Opxy

△s

p(x)∧q(y)

x'=f y ∧ y'=g x

使用操作模式、表达式和函数作用的精化规则，得到 Opxy 的精化结果为：

```
void op_xy(void)
{
        int t;
        if (p(x)&&q(y)) {
            t=x; x=f(y); y=g(t);
        }
}
```

5.8 程序的结构

PaZ 描述是由段落组成，这些段落可以实现为相应的语言结构，例如数据类型、变量、子过程等。其中基本类型定义精化为 arb 程序中的类型定义，公理描述一般精化为子过程，约束也可精化为子过程，模式和模式表达式精化为子过程。

在程序结构的组织上，一个状态模式以及包含它的所有操作模式以及它们所使用的定义可以在 arb 程序中组织成一个模块，当一个状态模式包含另一个时，则形成模块的依赖，或面向对象中的继承。一般情况下，一个 PaZ 的规范的最外层是有一组操作模式构成的，判断最外层操作模式的标准就是看其是否未被任何其他模式所包含。若最外层的模式有 op1,op2,... ...,opn，则主程序可以定义为它们所构成的析取式

$main \triangleq op1 \vee op2 \vee .. \ldots \vee opn \vee Exception$。

5.9 一个例子

本节我们通过一个例子来说明 PaZ 的描述到 arb 程序的过程。给出一个 PaZ 的描述：

[A,X,Y,Z]

x1:X; y2:Y

p:P A

q: $X \leftrightarrow Y$

g,h: $Y \rightarrow Z$

f: $Z \times Z \rightarrow Z$

$S \triangleq [a:A; x:X; y: Y; z:Z]$

$Opx \triangleq [\triangle s \mid a'=a \wedge x'=x1]$

$Op1 \triangleq [opx \mid p(a) \wedge y'=y \wedge z'=z]$

$Opy \triangleq [opx \mid \neg p(a) \wedge y'=y2]$

$Op2 \triangleq [opy \mid q(x',y') \wedge z'=f(z,g(y'))]$

$Op3 \triangleq [opy \mid \neg q(x',y') \wedge z'=f(z,h(y'))]$

$Op \triangleq op1 \vee op2 \vee op3$

根据模式表达式的精化规则，我们先计算出 op：

Op

$\triangle s$

$a'=a$

$x'=x1$

$((p(a) \wedge y'=y \wedge z'=z) \vee$

$(\neg p(a) \wedge y'=y2 \wedge$

$(q(x',y') \wedge z'=f(z,g(y'))) \vee$

$(\neg q(x',y') \wedge z'=f(z,h(y'))))$

然后我们对谓词部分的表达式进行精化，先看外层，定义

$s= (q(x',y') \wedge z'=f(z,g(y'))) \vee (\neg q(x',y') \wedge z'=f(z,h(y')))$

则 Op 的谓词部分精化为

x=x1;

if (!p(a)) { y=y2; s}

下面再对表达式 s 进行精化，通过提出 z'，引入新变量，使用析取式的精化规则得到：

z'=(let t= =if q(x', y') then g(y') else h(y') • f(z,t))

进一步使用新变量的精化规则得到：

```
t= q(x,y)?g(y):h(y);
z=f(z,t)
```

因此，整个实例精化后得到的结果为：

```
typedef int A,X,Y,Z;
A a;
X x,x1;
Y y,y2;
Z z;
Int p(A a) { … ….}
Int q(X x, Y y) { … …}
Z g(Y y) { … … }
Z h(Y y) { … … }
Z f(Z z, Z t) { …. ….}
Void op (void)
{
   Z t;
   x=x1;
   if (!p(a)) {
    y=y2;
    t= q(x,y)?g(y):h(y);
    z=f(z,t);
   }
}
```

当然，由于 PaZ 描述中省略了对函数 p,q,g,h,f 的具体定义，因此这些函数精化所得的子过程的内容不能确定。

第六章 DPaPD 模型下并行程序开发的完整实例

6.1 DPaPD 模型下并行程序开发过程

根据第二章中对 DPaPD 模型的概述，在该模型下进行并行程序的开发过程是从问题规范出发，通过使用系统设计模式库中所提供的问题并行求解策略对问题进行求解，通过精化、验证的手段，得到求解该问题的并行设计，然后，通过使用 PaZ 到 arb 程序的精化、转换规则，将 PaZ 语言描述的并行设计进一步精化为 arb 模型下的抽象并行程序，再通过转换将 arb 并行程序转换为具体并行程序。这一过程可具体划分为以下五个步骤：

第一步：规范描述(**Specification**)

使用 PaZ 语言对应用问题进行形式化的描述，得到问题的功能规范；

第二步：使用设计模式的精化（**Refinement using DPs**）

根据问题规范的描述，判断问题所属的类别，再根据各设计模式的描述，在系统所提供的设计模式中选取适当的设计模式，通过对设计模式的各成分进行具体化、精化，并与设计模式所提供的框架相结合，得到对问题进行并行求解的并行设计；

第三步：验证(**Verification**)

证明所得的并行设计满足问题的规范。

第四步：使用规则的精化(**Refinement using laws**)

使用 PaZ 到 arb 程序的精化、转换规则将并行设计精化、转换为 arb 程序；

第五步：转换(**Transformation**)

使用[51]中的规则对 arb 程序实施转换，得到具体的并行程序。

本章我们将通过使用 DPaPD 模型下开发并行程序的方法实现几个完整实例的并行程序的开发。本章包含三个实例：傅立叶变换问题、快速排序和 0-1 背包问题。我们将按照上述并行程序开发步骤逐一介绍每个实例的并行程序的详细开发过程。

6.2 傅立叶变换

首先，我们以基数为 2 的离散傅立叶变换(DFT)问题的并行程序开发作为实例。DFT 问题的快速傅立叶变换算法(FFT)的形式化推导是许多研究者的研究课题[62,27,81]。在此，我们将说明在 DPaPD 模型下的 FFT 并行程序开发。

6.2.1 DFT 问题的规范描述

首先，使用 PaZ 语言写出离散傅立叶变换的规范，如下所示：

DFT

a: seq complex

b: seq complex

FFT: seq complex→seq complex

n,k: N

---

w: complex

n=#a=#b

$n = 2^k \wedge w^n = 1 \wedge \forall 0 < i < n \bullet w^i \neq 1$

$\forall 1 \le j \le n \bullet b(j) = \sum_{i=1}^{n} a(i) w^{(i-1)(j-1)}$

b=FFT a

PaZ 语言在问题的规范描述中不区别前置条件（pre-condition）和后置条件（post-condition），而将他们统一地在谓词部分进行描述，其前置条件通过对谓词实施 pre 算子来获得。事实上，通过对谓词部分的分析，不难区别前后置条件。由于谓词中有 b=FFT a，因此谓词中与 b 有关的部分应属于问题的后置条件，而与 b 无关的部分为问题的前置条件。即前置条件为 n=#a $\wedge n = 2^k \wedge w^n = 1 \wedge \forall 0 < i < n \bullet w^i \neq 1$，而后置条件为 n= #b∧ b=FFT a∧ $\forall 1 \le j \le n \bullet b(j) = \sum_{i=1}^{n} a(i) w^{(i-1)(j-1)}$ 。

6.2.2 FFT 并行设计

从问题的功能规范可以看出，这是一个数值计算问题。函数 FFT 对一个有限的复数序列施加一定的计算，得到一个新的复数序列。它们之间的关系在谓词部分进行了描述。该问题所涉及的数据结构为一个具有有限规模的一维结构，且结果未对数据结构进行改变。这类问题适合使用分而治之的方法进行求解，因此可以选用设计模式 parallel divide&conquer（参见 4.3.1 节）。

设计模式 parallel divide&conquer 的结构如下：

[DATA,RESULT]

datas: DATA

result: RESULT

divide: DATA→**F** DATA

merge: **F** REAULT→RESULT

baseoperate: DATA→ RESULT

basedatas: P DATA

D&C: DATA→RESULT

D&C datas = if basedatas datas then baseoperate datas

Else merge D&C|(divide datas)|

Result = D&C datas

按照该模式使用的要求，需要根据 DFT 问题的规范对模式中的成分 DATA，RESULT，datas,result, divide, merge, baseoperate, basedatas 进行精化，其中包括对数据成分的精化和对函数成分的精化。即使用 DFT 规范中的数据和类型对 parallel divide&conquer 中的抽象数据和类型进行精化，并使用这些数据和类型将 parallel divide&conquer 中的函数成分进行精化。

根据 DFT 问题的规范描述，相应的数据精化显然包括：

类型的精化：DATA⊆seq complex　　RESULT⊆ seq complex

数据的精化：datas ⊆a　　result ⊆b

假设 parallel divide&conquer 中的运算 basedatas, baseoperate, merge, divide 分别精化为具体的运算 FFTbasedatas, FFTbaseoperate, FFTmerge, FFTdivide，而最终模式中的 D&C 精化成函数 FFTD&C，则这些函数组合成的设计必须满足 DFT 的规范，即：

n=#a ∧ $n = 2^k \wedge w^n = 1 \wedge \forall 0 < i < n \bullet w^i \neq 1$ ∧ b = FFTD&C a

$\Rightarrow$ n=#b ∧ $\forall 1 \le j \le n \bullet b(j) = \sum_{i=1}^{n} a(i) w^{(i-1)(j-1)}$

下面我们来分析这一问题。当 DFT 问题的规模为 1 时，显然不可再分。那么，FFTbasedatas 应定义为：

a: seq complex

FFTbasedatas: P seq complex

$\forall a \bullet FFTbasedatas(a) \Leftrightarrow (\#a = 1)$

相应地，此时对 a（a 中只有一个元素）的傅立叶变换得到的是 a 本身，因此，FFTbaseoperate 定义为：

a: seq complex

b: seq complex

FFTbaseoperate: seq complex → seq complex

---

b=FFTbaseoperate a

b=a

当 $n>1$ 时，按照 parallel divide&conquer 的思想，需将 a 进行分解。若我们将 a 按照其下标的奇偶进行分解，则 $\forall 1 \le j \le n \bullet b(j) = \sum_{i=1}^{n} a(i) w^{(i-1)(j-1)}$ 转换为

$\forall 1 \le j \le n \bullet b(j) = \sum_{i=1}^{n/2} a\ (2i-1) w^{2(i-1)(j-1)} + w^{j-1} \sum_{i=1}^{n/2} a\ (2i) w^{2(i-1)(j-1)}$。进一步地，若将 b 分裂成两个相等长度的序列，根据 w 的性质，有下式：

$$\forall 1 \le j \le n/2 \bullet b(j) = \sum_{i=1}^{n/2} a(2i-1) w^{2(i-1)(j-1)} + w^{j-1} \sum_{i=1}^{n/2} a(2i) w^{2(i-1)(j-1)}$$

$$\forall 1 \le j \le n/2 \bullet b(j+n/2) = \sum_{i=1}^{n/2} a(2i-1) w^{2(i-1)(j-1)} - w^{j-1} \sum_{i=1}^{n/2} a(2i) w^{2(i-1)(j-1)}$$

则 FFTdivide 和 FFTmerge 分别定义如下：

a: seq complex

subdatas: **F** seq complex

a_{even},a_{odd}: seq complex

FFTdivide: seq complex→ **F** seq complex

---

n/2=#a_{odd}=#a_{even}

Subdatas=FFTdivide a

Subdatas={a_{even},a_{odd}}

$\forall 1 \le i \le n/2 \bullet a_{even}(i) = a(2i) \wedge a_{odd}(i) = a(2i-1)$

Subresults: **F** seq complex

b_{even},b_{odd}: seq complex

b: seq complex

FFTmerge: **F** seq complex→ seq complex

---

n/2=#b_{odd}=#b_{even}

b=FFTmerge subresults

Subresults={b_{even},b_{odd}}

$\forall 1 \le i \le n/2 \bullet b(i) = b_{odd}(i) + w^{i-1} b_{even}(i)$

$$\wedge\ b(i+n/2) = b_{odd}(i) - w^{i-1} b_{even}(i)$$

FFTD&C 的定义通过使用上述定义对设计模式 Parallel Divide&Conquer 的结构进行具体化得到：

a: seq complex
b: seq complex
FFTD&C: seq complex→seq complex

FFTD&C a = if FFTbasedatas a then FFTbaseoperate a
else FFTmerge FFTD&C|(FFTdivide a)|
b = FFTD&C a

将以上精化结果组合在一起，构成 DFT 问题的并行设计为：

a: seq complex
b: seq complex
n,k: N
w: complex
n=#a=#b
$n = 2^k \wedge w^n = 1 \wedge \forall 0 < i < n \bullet w^i \neq 1$
FFTbasedatas 的定义
FFTbaseoperate 的定义
FFTdivide 的定义
FFTmerge 的定义
FFTD&C 的定义

6.2.3 设计的验证

下面证明使用 FFTD&C 对 a 进行操作的结果 b 满足问题 DFT 规范中的后置条件，即

$$\forall 0 \le j < n \bullet b[j] = \sum_{i=0}^{n-1} a[i]w^{ij} \quad \text{.....(1)}$$

证明：

1. 当 if 条件成立时，即 n=1 时，由于 n=1，所以 w=1
则 FFTbaseoperate a=a[0]满足 b[0]=a[0]，则(1)式成立；
2. If 条件不成立时，假设当 $n=2^k$ 时满足(1)式，为示区别，此时的 w 使用 w_k 表示，即

$$\forall 0 \le j < n \bullet b[j] = \sum_{i=0}^{n-1} a[i]w_k^{ij} \quad \text{.....(2)}$$

现证明 $n=2^{k+1}$ 时，b=FFTD&C a = FFTmerge FFTD&C$_{\text{FFTdivide a}}$ 满足（1）式。

首先，根据 FFTdivide 的约束可得 FFTdivide a=$\{a_{even},a_{odd}\}$

且 $\forall 0 \le i < n/2 \bullet a_{even}[i] = a[2i] \wedge a_{odd}[i] = a[2i+1]$ .....(3)

又，根据（2）式，有

$$\forall 0 \le j < n/2 \bullet b_{even}[j] = \sum_{i=0}^{n/2-1} a_{even}[i] w_k^{ij} \quad \wedge \forall 0 \le j < n/2 \bullet b_{odd}[j] = \sum_{i=0}^{n/2-1} a_{odd}[i] w_k^{ij} \quad ..(4)$$

那么 FFTmerge FFTD&C$_{\text{FFTdivide a}}$

= FFTmerge$\{b_{even},b_{odd}\}$

(根据 FFTmerge 的约束和约束变量的换名律)

$$= \forall 0 \le j < n/2 \bullet b[j] = b_{even}[j] + w^j b_{odd}[j] \wedge b[j+n/2] = b_{even}[j] - w^j b_{odd}[j]$$

（根据（4）式）

$$= \forall 0 \le j < n/2 \bullet b[j] = \sum_{i=0}^{n/2-1} a_{even}[i] w_k^{ij} + w^j \sum_{i=1}^{n/2-1} a_{odd}[i] w_k^{ij}$$

$$\wedge b[j+n/2] = \sum_{i=1}^{n/2-1} a_{even}[i] w_k^{ij} - w^j \sum_{i=1}^{n/2-1} a_{odd}[i] w_k^{ij} \quad .....(5)$$

(由于 $n=2^k$ 时，$w_k^n=1$，而 $n=2^{k+1}$ 时，$w^n=1$，则 $w_k=w^2$)

$$= \forall 0 \le j < n/2 \bullet b[j] = \sum_{i=0}^{n/2-1} a_{even}[i] w^{2ij} + w^j \sum_{i=1}^{n/2-1} a_{odd}[i] w^{2ij}$$

$$\wedge b[j+n/2] = \sum_{i=1}^{n/2-1} a_{even}[i] w^{2ij} - w^j \sum_{i=1}^{n/2-1} a_{odd}[i] w^{2ij}$$

(根据（3）式)

$$= \forall 0 \le j < n/2 \bullet b[j] = \sum_{i=0}^{n/2-1} a\ [2i] w^{2ij} + w^j \sum_{i=1}^{n/2-1} a\ [2i+1] w^{2ij}$$

$$\wedge b[j+n/2] = \sum_{i=1}^{n/2-1} a\ [2i] w^{2ij} - w^j \sum_{i=1}^{n/2-1} a\ [2i+1] w^{2ij}$$

(由于 w 的性质，$w^j = -w^{j+n/2}$，且 $w^{2ij}=w^{2i(j+n/2)}$)

$$= \forall 0 \le j < n/2 \bullet b[j] = \sum_{i=0}^{n/2-1} a\ [2i] w^{2ij} + w^j \sum_{i=1}^{n/2-1} a\ [2i+1] w^{2ij}$$

$$\wedge b[j+n/2] = \sum_{i=1}^{n/2-1} a\ [2i] w^{2i(j+n/2)} + w^{j+n/2} \sum_{i=1}^{n/2-1} a\ [2i+1] w^{2i(j+n/2)}$$

$$= \forall 0 \le j < n/2 \bullet b[j] = \sum_{i=0}^{n/2-1} a\ [2i] w^{2ij} + w^j \sum_{i=1}^{n/2-1} a\ [2i+1] w^{2ij}$$

$$\wedge b[j+n/2]=\sum_{i=1}^{n/2-1} a\ [2i]w^{2i(j+n/2)}+\sum_{i=1}^{n/2-1} a\ [2i+1]w^{(2i+1)(j+n/2)}$$

$$= \quad \forall 0\le j<n \bullet b[j]=\sum_{i=0}^{n/2-1} a\ [2i]w^{2ij}+\sum_{i=1}^{n/2-1} a\ [2i+1]w^{(2i+1)j}$$

$$= \quad \forall 0\le j<n \bullet b[j]=\sum_{i=0}^{n-1} a\ [i]w^{ij} \quad =(1)\text{式}, \quad \text{得证。}$$

6.2.4 arb 程序的产生

从 6.2.2 节的结果，我们得到了并行求解 DFT 问题的设计（参见 6.2.2 节）。需要指出的是，由于 a 和 b 为全局变量定义，在 FFTbasedatas 等所有函数定义的公理描述中则不需对其进行重复定义。而且，模式中的基本类型定义[DATA，RESULT]由于已在设计中根据具体的问题精化为 seq complex，因此所得的设计中不再有该类型定义。将所得的并行设计精化、转换为 arb 程序，分以下几个方面进行：

6.2.4.1 数据结构的精化

在 DFT 的并行设计中，一个基本的抽象类型为复数序列，按照我们对该类型的精化规则（参见 5.1.2 节），可以以数组类型实现。即

a: seq complex

b: seq complex

转换为

```
#define n1 #a;
#define n2 #b;
complex a[n1],b[n2];
```

由于公理 n=#a=#b，所以优化为：

```
#define n #a
complex a[n], b[n]
```

6.2.4.2 新变量的引入

对于比较长和复杂的表达式，往往需要将其分成若干步。这样就需要引入新的变量。在 DFT 的并行设计中，FFTD&C 中有一个复杂表达式：FFTmerge FFTD&C|(FFTdivide a)|，为了精化该表达式，需要引入一些中间变量。使用新变量引入规则，得到：

FFTmerge FFTD&C|(FFTdivide a)|

$\subseteq$ let t= =FFTdivide a • FFTmerge FFTD&C |(t)|

再使用新变量精化规则，得到：

⊆ t=FFTdivide a; FFTmerge FFTD&C |(t)|

由于 FFTdivide a 产生的结果 t 为一个含有两个元素的集合，我们将 t 的内容使用两个变量显式地描述出来，设 t={aeven,aodd}，则以上部分转变成：

⊆ (aeven,aodd)=FFTdivide a; FFTmerge FFTD&C |(aeven,aodd)|

再对剩余的表达式引入新变量，进一步精化，最后得到：

(aeven,aodd)= FFTdivide a

beven=FFTD&C aeven

bodd= FFTD&C aodd

b=FFTmerge(beven,bodd)

因此引入的变量包括：

complex aeven[n/2], aodd[n/2], beven[n/2], bodd[n/2];

6.2.4.3 各函数定义的精化

将并行设计中的各个公理描述按照 5.6 节所描述的规则精化为 arb 程序。以 FFTdivide 的定义为例：

subdatas: **F** seq complex
a_{even},a_{odd}: seq complex
FFTdivide: seq complex→ **F** seq complex
n/2=#a_{odd}=#a_{even}
Subdatas=FFTdivide a
Subdatas={a_{even},a_{odd}}
$\forall 1 \le i \le n/2 \bullet a_{even}(i) = a(2i) \wedge a_{odd}(i) = a(2i-1)$

为便于精化，将上述定义进行简化，取消变量 subdatas，得

aeven,aodd: seq complex
FFTdivide: seq complex→ **F** seq complex
n/2=#aodd=#aeven
(aeven,aodd)=FFTdivide a
$\forall 1 \le i \le n/2 \bullet aeven(i) = a(2i) \wedge aodd(i) = a(2i-1)$

先看声明部分：

aeven,aodd: seq complex

⊆ #define n1 #aeven;

#define n2 #aodd;

complex aeven[n1], aodd[n2];

FFTdivide: seq complex→ F seq complex

⊆ complex *FFTdivide(complex a[ ])

再看谓词部分：

性质 n/2=#aodd=#aeven 表明 aeven,aodd 的声明可优化为：

```
complex aeven[n/2], aodd[n/2];
```

性质(aeven,aodd)=FFTdivide a 表明(aeven,aodd)为函数的返回值，即精化为:

```
return (aeven,aodd);
```

$\forall 1 \le i \le n/2 \bullet aeven(i) = a(2i) \wedge aodd(i) = a(2i-1)$ 的精化是一个全称量词表达式的精化，首先由于 $aeven(i) = a(2i) \wedge aodd(i) = a(2i-1)$ 中aeven(i)与aodd(i)没有交互引用，因此直接使用赋值规则精化得到赋值语句aeven(i)=a(2i)；aodd(i)=a(2i-1)，再使用全称量词精化规则得到：

```
int i;
arball(i=1:n/2)
    aeven(i)=a(2i)
    aodd(i)=a(2i-1)
end arb;
```

（注：由于 $aeven(i) = a(2i) \wedge aodd(i) = a(2i-1)$ 精化得到赋值语句，因此此式恒为真，因此精化规则中if语句的条件恒为假，则if语句可以省略）

综上，关于函数 FFTdivide 的公理描述精化为：

```
complex *FFTDivide(complex a[ ])
  { complex aeven[n/2],aodd[n/2];
    int i;
    arball(i=1:n/2)
     aeven(i)=a(2i)
     aodd(i)=a(2i-1)
    end arb;
    return (aeven,aodd);
   }
```

其它函数的精化同理。最后得到完整的 arb 程序如下：

```
Int n;
Complex a[n];
Complex b[n];
Complex w;

Int FFTBasedatas(complex a[ ])
{ int bB;
   bB=(#a=1);
   return bB;
}
```

```
complex *FFTBaseoperator(complex a[ ])
{ b[1]=a[1];
   return b;
}

complex *FFTDivide(complex a[ ])
  { complex aeven[n/2],aodd[n/2];
     int i;
     arball(i=1:n/2)
       aeven(i)=a(2i)
       aodd(i)=a(2i-1)
     end arb;
     return (aeven,aodd);
  }

complex *FFTMerge(complex beven[ ],complex bodd[ ])
{    int i;
     arball(i=1:n/2)
       b(i)=bodd(i)+w^(i-1)*beven(i)
       b(i+n/2)=bodd(i)-w^(i-1)*beven(i)
     end arb;
     return b;
}

complex *FFT(complex a[ ])
   { complex aeven[n],aodd[n],beven[n],bodd[n];
     if FFTBasedatas(a) then b=FFTBaseoperator(a)
     else {(aeven,aodd)=FFTDivide(a)
          arb
            beven=FFT(aeven)
            bodd=FFT(aodd)
          end arb;
          b=FFTMerge(beven,bodd)
        }
       return b;
  }
```

6.2.5 并行程序

使用[51]中的规则将 6.2.4 节的 arb 程序转换成具体的并行程序。由于这部分内容不属于本文的工作，因此省略。

6.3 快速排序

快速排序也是一个使用分而治之方法设计的算法，因此也可以使用并行分而治之设计模式进行设计。

6.3.1 排序问题的规范描述

不失一般性，考虑整型数据的排序问题。其规范描述为：

sort

a:seq integer

b:seq integer

sort: seq integer→seq integer

---

a≠<>

b=sort a

b=displace a $\wedge \forall 1 \le i \le \# b - 1 \bullet b(i) \le b(i+1)$

其中，b=displace a 表示 b 中元素为 a 中元素的一次置换。

6.3.2 快速排序的并行设计

根据排序问题的规范描述，显然其中数据成分的精化包括：

DATA ⊆ seq integer RESULT ⊆ seq integer

datas ⊆ a result ⊆ b

当 a 中只有一个元素时，可以直接求解，因此 sortbasedatas 可以定义为：

a: seq integer

sortbasedatas: P seq integer

$\forall a \bullet sortbasedatas(a) \Leftrightarrow (\# a = 1)$

相应地，此时对 a（a 中只有一个元素）进行排序得到的结果是 a 本身，因此，sortbaseoperate 定义为：

a: seq integer

b: seq integer

sortbaseoperate: seq integer→ seq integer

---

b=sortbaseoperate a

b=a

再看划分的情况，当 a 中不止一个元素时，需要对 a 进行划分。可将 a 划分成两部分，其中一部分的所有元素比另一部分的所有元素小，这样，划分 sortdivide 定义为：

a: seq integer
subdatas: **F** seq integer
a_l,a_r : seq integer
sortdivide : seq integer → **F** seq integer
---
subdatas = sortdivide a
subdatas = { a_l,a_r}

$a_l \frown a_r$ = displace a $\wedge \forall i, j \mid 1 \le i \le \# a_l \wedge 1 \le j \le \# a_r \bullet a_l(i) \le a_r(j)$

而合并只需将两部分序列连接起来，因此 sortmerge 定义为：

b: seq integer
subresults: **F** seq integer
b_l,b_r : seq integer
sortmerge : **F** seq integer→ seq integer
---
b=sortmerge subresults
subresults = { b_l,b_r}
b= $b_l \frown b_r$

同样地，sortD&C 的定义通过使用上述定义对设计模式 Parallel Divide&Conquer 的结构进行具体化得到：

a: seq integer
b: seq integer
sortD&C: seq integer→seq integer
---
sortD&C a = if sortbasedatas a then sortbaseoperate a
else sortmerge sortD&C|(sortdivide a)|
b = sortD&C a

将上述精化的结果组合在一起，得到快速排序的并行设计为：

a: seq integer
b: seq integer
sortbasedatas 的定义
sortbaseoperate 的定义
sortdivide 的定义

sortmerge 的定义

sortD&C 的定义

6.3.3 设计的验证

下面证明 b=sortD&C a = if　sortbasedatas a then sortbaseoperate a

Else sortmerge sortD&C|(sortdivide a)|

满足规范 sort 中的约束，即

b=displace a $\wedge \forall 1 \le i \le \#b-1 \bullet b(i) \le b(i+1)$ .....(1)

证明：1. 当 if 条件成立时，n=1，b=a，因此(1)式显然成立；

2. 当 if 条件不成立时，即 n>1 时，要证明 b= sortD&C a = sortmerge sortD&C|(sortdivide)|使得(1)式成立。使用数学归纳法证明如下：

i) 当 n=2 时，sortdivide a={a_l,a_r}，由于 a_l 和 a_r 中分别只有一个元素，因此根据 sortdivide 的 property，有 $a_l \frown a_r$ = displace a $\wedge a_l(1) \le a_r(1)$ 成立；且根据 1. 有 $b_l=a_l \wedge b_r=a_r$，

因此 b=sortmerge{b_l,b_r}= $b_l \frown b_r = a_l \frown a_r$ = displace a $\wedge a_l(1) \le a_r(1)$，因此

b= displace a $\wedge b(1) \le b(2)$，（1）式成立；

ii) 假设第 k 步时有 sortD&C a 满足（1）式，则第 k+1 步时

sortD&C a = sortmerge {b_l,b_r}

b_l=sortD&C $a_l \wedge$ b_r=sortD&C a_r

其中{a_l,a_r}=sortdivide a

据假设，有 b_l=displace a_l $\wedge \forall 1 \le i \le \#b_l - 1 \bullet b_l(i) \le b\ (i+1)$

b_r=displace a_r $\wedge \forall 1 \le i \le \#b_r - 1 \bullet b_r(i) \le b_r(i+1)$

则 b=sortmerge {b_l,b_r} = $b_l \frown b_r$ =displace $a_l \frown a_r$

满足 $\forall 1 \le i \le \#b_l - 1 \bullet b(i) \le b(i+1) \wedge \forall 1 \le i \le \#b_r - 1 \bullet b(i+\#b_l) \le b(i+\#b_l+1)$

又根据 sortdivide a 的定义，有 $\forall i, j \mid 1 \le i \le \#a_l \wedge 1 \le j \le \#a_r \bullet a_l(i) \le a_r(j)$，且据

假设有 b_l=displace a_l，b_r=displace a_r，所以 $\forall i, j \mid 1 \le i \le \#b_l \wedge 1 \le j \le \#b_r \bullet b_l(i) \le b_r(j)$，

所以 b(i)≤b(i+#b_l-1)，所以 $\forall i \mid 1 \le i \le \#b-1 \bullet b(i) \le b(i+1)$，（1）式成立，得证。

6.3.4 arb 程序的产生

从 6.3.2 节的结果，我们得到了快速排序的并行设计。要将它转换、精化为 arb 程序，同样分以下几个方面进行：

6.3.4.1 数据结构的精化

在快速排序的并行设计中，一个基本的抽象类型为整型序列，按照我们对该类型的精化规则，可以以整型数组类型实现。类似于上例，

a: seq integer

b: seq integer

转换为

#define n #a;

int a[n],b[n];

6.3.4.2 新变量的引入

在快速排序的并行设计中，sortD&C 中有一个复杂表达式：sortmerge sortD&C|(sortdivide a)|，为了精化该表达式，将该表达式拆成若干步进行，因此需要引入一些中间变量。其精化过程与上例同，只是为了可读性的需要，给引入的新变量赋予了不同的名字，其结果为：

(al,ar)= sortdivide a

bl= sortD&C al

br= sortD&C ar

b=sortmerge(bl,br)

引入的变量为：

int L,R

int al[L], ar[R], bl[L], br[R];

6.3.4.3 各段落的精化

与上例相类似，将并行设计中的各个段落按照上一章所描述的规则精化为 arb 程序。需要指出的是，在定义 sortdivide 的公理描述中的谓词部分有一个含多个约束变量的全称量词表达式，目前，我们没有确定的转换规则来处理这种情况。因此这里精化所得的结果不能从表达式本身的成分上得到，而是需要设计一个算法使得该表达式得到满足，因此，这种情况的精化，我们不能自动地实现。根据上一步得到的并行设计，精化得到的快速排序 arb 程序如下：

```
Int n;
Int a[n],b[n];
```

```
int sortbasedatas(int a[ ])
  {
   return(#a=1);
  }

int *sortbaseoperate(int a[ ])
    {
     b[1]=a[1];
     return b;
    }

int *sortdivide(int a[ ])
   {
      int al[L],ar[R];
      L=0;
      R=0;
      arball (i=2:n)
         if (a[i]<a[1]) then
            {
               L=L+1;
               al[L]=a[i];
            }
         Else
            {
               R=R+1;
               ar[R]=a[i];
             }
         L=L+1;
         al[L]=a[1];
       end arb
     return (al,ar);
}

int *sortmerge(bl,br)
   {
    arball (i=1:n)
      if (i≤#bl) then
         b[i]=al[i];
      else
         b[i]=ar[i-#bl];
    end arb
    return b;
   }

int *sort(int a[ ])
  {
   int L,R;
   int al[L],ar[R],bl[L],br[R];
   if sortbasedatas(a) then    b=sortbaseoperate(a)
   else {
          (al,ar)=sortdivide(a);
          arb
             bl=sort(al);
             br=sort(ar);
          end arb
          b=sortmerge(bl,br);
        }
 return b;
}
```

6.3.5 并行程序

同 6.2.5 节。

6.4 0-1 背包问题

在以上的两节中，我们讨论了两个数值问题的实例，它们都使用了并行分而治之模式进行求解。本节我们将考虑一个经典的组合优化问题：0-1 背包问题的并行程序开发。该问题中有一个最大承重量为 max_weight 的背包和一些物体 object，每个物体有其重量 w_j 和价值 p_j，现要求在给定的所有物体中取出若干放入背包，使得放入背包的物体的价值达到最大。在文献[82,26]中对这一问题的研究现状进行了全面的阐述，本节我们将使用 DPaPD 模型来对这一问题进行并行求解。

6.4.1 背包问题的规范描述

使用 PaZ 对 0-1 背包问题进行形式化规范描述如下：

Object

W: real

P: real

In-knapsack: -1..1

Objects ::= F object

Max_weight, max_value: real

D,D': F objects

$(\forall 1 \le i \le \#objects \bullet objects(i).in-knapsack = -1) \wedge D = \{objects\}$

$F_{constraint} = \sum_{i=1}^{\#d} d(i).w * d(i).in\_knapsack \le \max\_weight$

$\forall d \in D' \bullet ((\forall 1 \le i \le \#d \bullet d(i).in\_knapsack \ne -1)$

$\wedge \sum_{i=1}^{\#d} d(i).p * d(i).in\_knapsack$ is maximal under $F_{constraint}$)

规范中 object 表示物体的类型，W 为其重量，P 为其价值，in_knapsack 表示物体是否在背包中，1 表示在背包中，0 表示不在，-1 表示尚未考虑。Objects 表示一个部分解的类型，D 为部分解的集合，因此 D'为问题的最后解集。前置条件 $(\forall 1 \le i \le \#objects \bullet objects(i).in-knapsack = -1) \wedge D = \{objects\}$ 表示初始时所有物体均尚未考虑，后置条件

$$\forall d \in D' \bullet ((\forall 1 \le i \le \# d \bullet d(i).in\_knapsack \neq -1) \quad \wedge \sum_{i=1}^{\#d} d(i).p * d(i).in\_knapsack \quad \text{is}$$

maximal under $F_{constraint}$)

表示 D'中的解满足在背包重量限制下价值达到最大。

6.4.2 背包问题的并行求解

由于 0-1 背包问题是一个组合优化问题，因此可以尝试使用并行分枝限界模式进行求解（参见 4.3.3 节）。模式 Parallel Branch&Bound 如下：

```
[subproblem]
  A A': F subproblem
  β : F subproblem→F subproblem
  PRUNE: F subproblem → F subproblem
  B&B: F subproblem → F subproblem
  B&B A = μX• if A'≠A then A'=PRUNE|( β|(A)| )|;X
```

根据 0-1 背包问题的规范，数据成分的精化显然包括：

subproblem ⊆ objects　　A ⊆ D

假设我们使用函数 branching 、eliminating 分别对模式中的 β 、PRUNE 进行精化，而模式中的函数 B&B 精化为 B&B_KP，则下面我们需要通过分析背包问题给出函数 branching 和 eliminating 的定义。

在这一问题中，部分解 d 的分枝可以这样考虑。若 d 还不是一个最后解，则 d 中并非所有的物体均被考虑，假设我们已考虑了 i-1 个物体，则这一步应考虑第 i 个物体。此时有两种可能：物体 i 放入背包或不放入背包，相应地使用 d_1 和 d_2 表示。则 d 分枝成 d_1 和 d_2，d 从部分解集合中去除，而 d_1 和 d_2 为新加入的部分解。进行分枝后，再对当前的最大价值进行刷新。根据这一分析，branching 函数定义如下：

D,D': F objects

branching: F objects→F objects

$$\forall d \in D \bullet (\exists 1 \le i \le \# d \bullet (\forall 1 \le j < i \bullet d(j).in\_knapsack \neq -1 \wedge d(i).in\_knapsack = -1)$$

$$\Rightarrow \quad \forall 1 \le j < i \bullet d_1(j) = d(j) \wedge \forall i < j \le \# d \bullet d_1(j) = d(j)$$

$$\wedge\, d_1(i).in\_knapsack = 1 \wedge d_1(i).w = d(i).w \wedge d_1(i).p = d(i).p$$

$$\wedge\, \forall 1 \le j < i \bullet d_2(j) = d(j) \wedge \forall i < j \le \# d \bullet d_2(j) = d(j)$$

$$\wedge\, d_2(i).in\_knapsack = 0 \wedge d_2(i).w = d(i).w \wedge d_2(i).p = d(i).p$$

$$\wedge\, D' = (D \setminus \{d\}) \cup \{d_1, d_2\})$$

$$\forall d \in D \bullet$$

$$(\forall 1 \le i \le \# d \bullet d(i).in\_knapsack \neq -1) \wedge$$

$$\sum_{i=1}^{\# d} d(i).w * d(i).in\_knapsack \le \max\_weight$$

$$\Rightarrow \max\_value' = \max\Big(\sum_{i=1}^{\# d} d(i).p * d(i).in\_knapsack \;, \max\_value\Big)$$

删除操作则必须将部分解集合中那些不满足问题要求的和那些不可能发展成问题最后解的部分解去除。在这一问题中，若部分解的总重量超出 max_weight，则不满足问题的要求，若部分解的总价值小于当前的 max_value，则不可能发展成问题的最后解。因此 eliminating 定义为：

D,D': F objects

eliminating: F objects→F objects

$$\forall d \in D \bullet (\forall 1 \le i \le \# d \bullet d(i).in\_knapsack \neq -1)$$

$$\wedge \sum_{i=1}^{\# d} d(i).w * d(i).in\_knapsack > \max\_weight \;\Rightarrow D' = D \setminus \{d\}$$

$$\forall d \in D \bullet (\forall 1 \le i \le \# d \bullet d(i).in\_knapsack \neq -1$$

$$\wedge \sum_{i=1}^{\# d} d(i).w * d(i).in\_knapsack \le \max\_weight$$

$$\wedge \sum_{i=1}^{\# d} d(i).p * d(i).in\_knapsack < \max\_value \;) \Rightarrow D' = D \setminus \{d\}$$

B&B_KP 的定义通过使用上述定义对设计模式 Parallel Branch&Bound 的定义进行具体化得到：

B&B_KP: F objects→ F objects

B&B_KP D = μX• if $D' \neq D$ then D'=eliminating|(branching|(A)|)|;X

因此得到求解 0-1 背包问题的并行设计为：

object
W: real
P: real
In-knapsack: -1..1

Objects ::= F object
Max_weight,max_value: real
D,D': F objects
Branching 的定义
Eliminating 的定义
B&B_KP 的定义

6.4.3 设计的验证

证明 6.4.2 节所得的设计满足 0-1 背包问题的规范：

$\mu X \bullet if\ D' \neq D\ then\ D' = eliminating|(branching|(D))|$； X

$$\Rightarrow \mathrm{F}_{constraint} = \sum_{i=1}^{\#d} d(i).w * d(i).in\_knapsack \leq \max\_weight$$

$$\forall d \in D \bullet ((\forall 1 \leq i \leq \#d \bullet d(i).in\_knapsack \neq -1)$$

$$\wedge \sum_{i=1}^{\#d} d(i).p * d(i).in\_knapsack \text{ is maximal under } \mathrm{F}_{constraint})$$

首先将蕴涵式的后件分成两部分：

$\forall d \in D \bullet ((\forall 1 \leq i \leq \#d \bullet d(i).in\_knapsack \neq -1)$ .....(1)

$\forall d \in D \bullet \sum_{i=1}^{\#d} d(i).p * d(i).in\_knapsack$ is maximal under $\mathrm{F}_{constraint}$) .....(2)

则证明分成两部分：（i）证明（1）式；

（ii）证明在（1）式的前提下有（2）式。

(i) 蕴涵式左边是一个迭代式，其终止条件为 D'=D，由于 branching 与 eliminating

不为逆过程，因此 D'=D ⇒ D=branching|(D)| ∧ D=eliminating|(D)|

D=branching|(D)|

$$\Rightarrow \forall d \in D \bullet \neg\exists 1 \le i \le \# d \bullet (\forall 1 \le j < i \bullet d(j).in\_knapsack \neq -1) \wedge d(i).in\_knapsack = -1$$

$$\Rightarrow$$

$$\forall d \in D \bullet \forall 1 \le i \le \# d \bullet \neg((\forall 1 \le j < i \bullet d(j).in\_knapsack \neq -1) \wedge d(i).in\_knapsack = -1)$$

$$\Rightarrow$$

$$\forall d \in D \bullet \forall 1 \le i \le \# d \bullet (\neg\forall 1 \le j < i \bullet d(j).in\_knapsack \neq -1) \vee \neg d(i).in\_knapsack = -1$$

（由于 branching 在迭代过程中的连续性（即物体是按序依次考虑的），有

$$\forall d \in D \bullet \forall 1 \le i \le \# d \bullet (\neg\forall 1 \le j < i \bullet d(j).in\_knapsack \neq -1) = false$$ ）

$$\Rightarrow \forall d \in D \bullet \forall 1 \le i \le \# d \bullet \neg d(i).in\_knapsack = -1$$

$$\Rightarrow \forall d \in D \bullet \forall 1 \le i \le \# d \bullet d(i).in\_knapsack \neq -1 \Rightarrow$$ （1）式

（ii）使用反证法证明

假设（2）式不成立，则至少存在一个 d∈D，使 $\sum_{i=1}^{\#d} d(i).p * d(i).in\_knapsack$ 不为满足条件 $F_{constraint}$ 的最大值，即至少存在一个 d' ∈ D，使 $\sum_{i=1}^{\#d} d'(i).p * d'(i).in\_knapsack > \sum_{i=1}^{\#d} d(i).p * d(i).in\_knapsack$，为方便阅读，我们将 $\sum_{i=1}^{\#d} d'(i).p * d'(i).in\_knapsack$ 记为 V'，$\sum_{i=1}^{\#d} d(i).p * d(i).in\_knapsack$ 记为 V。

解 d 与 d'的产生有三种情况：

① d 与 d'在同一次 branching 中产生；

② d 先于 d'产生；

③ d'先于 d 产生。

我们分别考虑之：

① 根据 branching 的规范 $\forall d \in D \bullet$ if

$$i = \# d \wedge \sum_{k=1}^{i} d(k).w * d(k).in\_knapsack \le \max\_weight$$

then $U' = \max(\sum_{k=1}^{\#d} d(k).p * d(k).in\_knapsack, U)$，在产生 d 和 d'后，根据假设 V'>V，则 U=V'。那么，根据 eliminating 的规范

$$\forall d \in D \bullet (\forall 1 \le i \le \# d \bullet d(i).in\_knapsack \neq -1$$

$$\wedge \sum_{k=1}^{\#d} d(k).w * d(k).in\_knapsack \le \max\_weight$$

$$\wedge \sum_{k=1}^{\#d} d(k).p * d(k).in\_knapsack < U) \Rightarrow D' = D \setminus \{d\}$$，由于 V<U，因此 D'=D\{d}，与假设 d∈D 相矛盾；

②在产生 d 的 branching 中，根据其规范有 U=V，然而等到产生 d'的 branching 时，由于 U'=max($\sum_{k=1}^{\#d'} d'(k).p * d'(k).in\_knapsack$,U)，根据假设 V'>V，则 U=V'，根据 eliminating 的规范，由于 V<U， 因此 D'=D\{d}，与假设 d∈D 相矛盾；

③同理可证。

因此，假设不成立，得证。

6.4.4 arb 程序的产生

同样地，要从 6.4.2 节所产生的并行设计得到对应的 arb 程序，需要对设计的每个段落进行精化，数据结构的精化包括对状态模式的精化（参见 5.3 节）、类型定义的精化以及集合类型变量的精化，因此有：

```
┌── object ──────
│  W:   real
│   P:   real
│  In-knapsack:   -1..1
└──────────────────
Objects ::=   F object
Max_weight,max_value:   real
 D,D':   F objects
⊆
Typedef struct
 {
    real w;
    real p;
    int in_knapsack;
  } object;
typedef object *objects;
real max_weight=0;
real max_value=0;
Objects D[ ], D1[ ];
```

各函数的精化仍然遵循上一章中公理描述和表达式（主要是量词表达式与合取表达式）的精化规则，最后得到的完整 arb 程序如下：

```
Typedef struct
 {
    real w;
```

```
        real p;
        int in_knapsack;
} object;
typedef object *objects;

real max_weight=0;
real max_value=0;
Objects D[ ], D1[ ];

Objects *Branching(objects D[ ])
{
int bImplication;
int indexofd,i,j;
object d[ ],d1[ ],d2[ ];

arball(indexofd=1:#D)
   d=D[indexofd];
   arball(i=1:#d)
        bImplication = TRUE;
        arball(j=1:i-1)
             If d[j].in_knapsack = =-1 Then    bImplication = FALSE;
        end arb;
        if   bImplication && d[i].in_knapsack= =-1
        then   {
                    arball(j=1:i-1)
                          d1[j] = d[j];
                    end arb;
                    arball(j=i+1:#d)
                          d1[j] = d[j];
                    end arb
                    d1[i].in_knapsack = 1;
                    d1[i].w = d[i].w;
                    d1[i].p = d[i].p;
                    arball(j=1:i-1)
                          d2[j] = d[j];
                    end arb;
                    arball(j=i+1:#d)
                          d2[j] = d[j];
                    end arb
                    d2[i].in_knapsack = 0;
                    d2[i].w = d[i].w;
                    d2[i].p = d[i].p;
                    D1=(D\{d})∪{d1,d2};
               }
        end arb;
end arb;

arball(indexofd=1:#D)
      d=D[indexofd];
       bImplication = TRUE;
       arball(i=1:#d)
            If d[i].in_knapsack= = -1 Then    bImplication = FALSE;
       end arb;
       if   bImplication &&     (1<=i<=#d·d(i).w*d(i).in_knapsack)<=max_weight
       then     max_value=max(    (1<=i<=#d·d(i).p*d(i).in_knapsack),max_value);
     end arb
}
```

```
objects *Eliminating(objects D[ ])
 {
    int bImplication;
    int indexofd,i,j;
    object d[ ];

     arball(indexofd=1:#D)
          d=D[indexofd];
           bImplication = TRUE;
           arball(i=1:#d)
                If d[i].in_knapsack= = -1 Then    bImplication = FALSE;
           end arb;
           if   bImplication &&     (1<=i<=#d·d(i).w*d(i).in_knapsack)>max_weight
           then     D1=D\{d};
       end arb

       arball(indexofd=1:#D)
            d=D[indexofd];
             bImplication = TRUE;
             arball(i=1:#d)
                  If d[i].in_knapsack= = -1 Then    bImplication = FALSE;
             end arb;
            if   bImplication &&     (1<=i<=#d·d(i).w*d(i).in_knapsack)   max_weight &&
(1<=i<=#d·d(i).p*d(i).in_knapsack)<max_value
          then     D1=D\{d};
       end arb
   }


  objects *B&B_KP(objects D[ ])
   {
      D1=Eliminating(Branching(D);
      While (D1!=D)
          {
             D1=Eliminating(Branching(D);
          }
      return D1;
   }
```

6.4.5 并行程序

同 6.2.5 节。

6.5 小结

本章我们通过几个完整实例详细描述了在 DPaPD 模型下进行问题求解和并行程序设计开发的过程。这些实例分别是两大类问题集合：数值计算问题集合和组合优化问题集合的问题的典型代表。我们选用这些实例的目的就是为了说明 DPaPD 模型下的并行程序开发方法不仅适用于结构较为简单的数值计算问题的并行程序设计开发，

而且对复杂的非数值问题的并行程序的开发同样适用，这与许多并行程序模型对非数值问题的程序开发比较困难的特点有明显的不同。同时还可以看出，DPaPD 模型下进行并行程序开发具有以下特点：

1. 模型中的设计模式为问题的求解和程序的设计开发提供了基本的思想和实现框架。用户针对具体问题的类型选用设计模式，然后根据设计模式的要求，使用具体问题对设计模式的各个成分进行具体化和精化。由于设计模式提供了问题求解的总体框架，因此用户所考虑的部分比原问题结构更为简单。

2. DPaPD 模型中用户对并行程序的设计局限于一个高抽象层次的 PaZ 框架之下。并行程序设计开发的困难不仅仅在于问题并行求解的困难，还在于并行语言、并行程序环境、并行体系结构的多样性。而在 DPaPD 模型下，用户不需考虑任何并行的实现，甚至不需了解 arb 并行模型和任何并行的知识，只要在 PaZ 的抽象层次上对设计模式中有关的成分根据具体问题进行精化。这是由于系统已将所有有关并行的运算包含在系统所提供的设计模式的定义之中，从而达到并行的目的。不仅如此，系统还将自动地实现从抽象的 PaZ 并行设计到 arb 并行程序的精化、转换过程，并最终产生具体的并行程序。这部分内容将在下一章中进行讨论。

第七章　DPaPD 系统的实现

7.1 DPaPD 系统实现的总体目标

DPaPD 系统的实现是以 DPaPD 并行程序开发模型的思想为依据的，其总体目标是将 PaZ 框架下用户根据所要求解的问题和选定的设计模式进行的有关定义自动进行精化、转换和组合，最终得到求解问题的完整的 arb 程序。由于本系统的输出结果为抽象的 arb 并行程序，因此系统的实现环境采用了本地微机或工作站，本系统在整个并行程序的设计开发过程中的作用如图 7.1 所示。

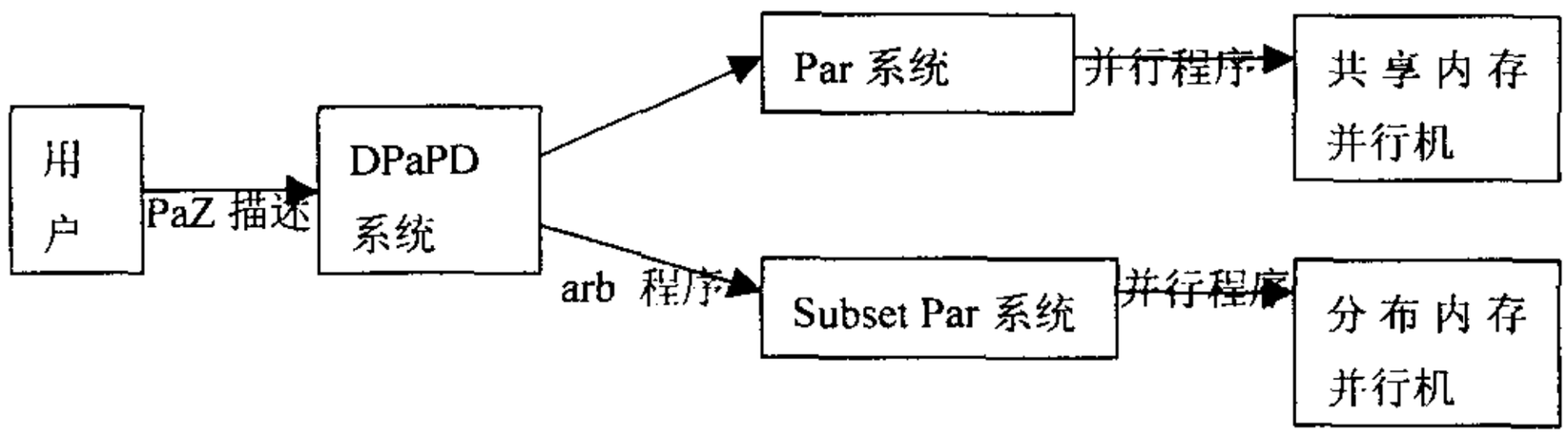


图 7.1　DPaPD 系统实现的总体目标

Fig. 7.1 Total Target of DPaPD System Implementation

7.2 系统的基本结构

系统的基本结构如图 7.2 所示。

系统中的主要功能部件包括有：

1．DPaPD 系统集成环境（Fun.dll）

【功能】：该部件是 DPaPD 系统的前端，能让用户方便地选择系统的设计模式，PaZ 语言的输入编辑或打开已有的 PaZ 程序文件，保存、打印 PaZ 程序及 Arb 程序等功能。其中包括两个主要模块，主要用于装载系统提供的设计模式和提供 PaZ 语言中的特殊字符集。前一个模块是通过访问系统的设计模式库来实现，后一个通过构造特殊字符，并通过系统弹出菜单让用户进行选择。

【引用方法】：MenuFun.funCls

【主要成员函数】：splitMenu

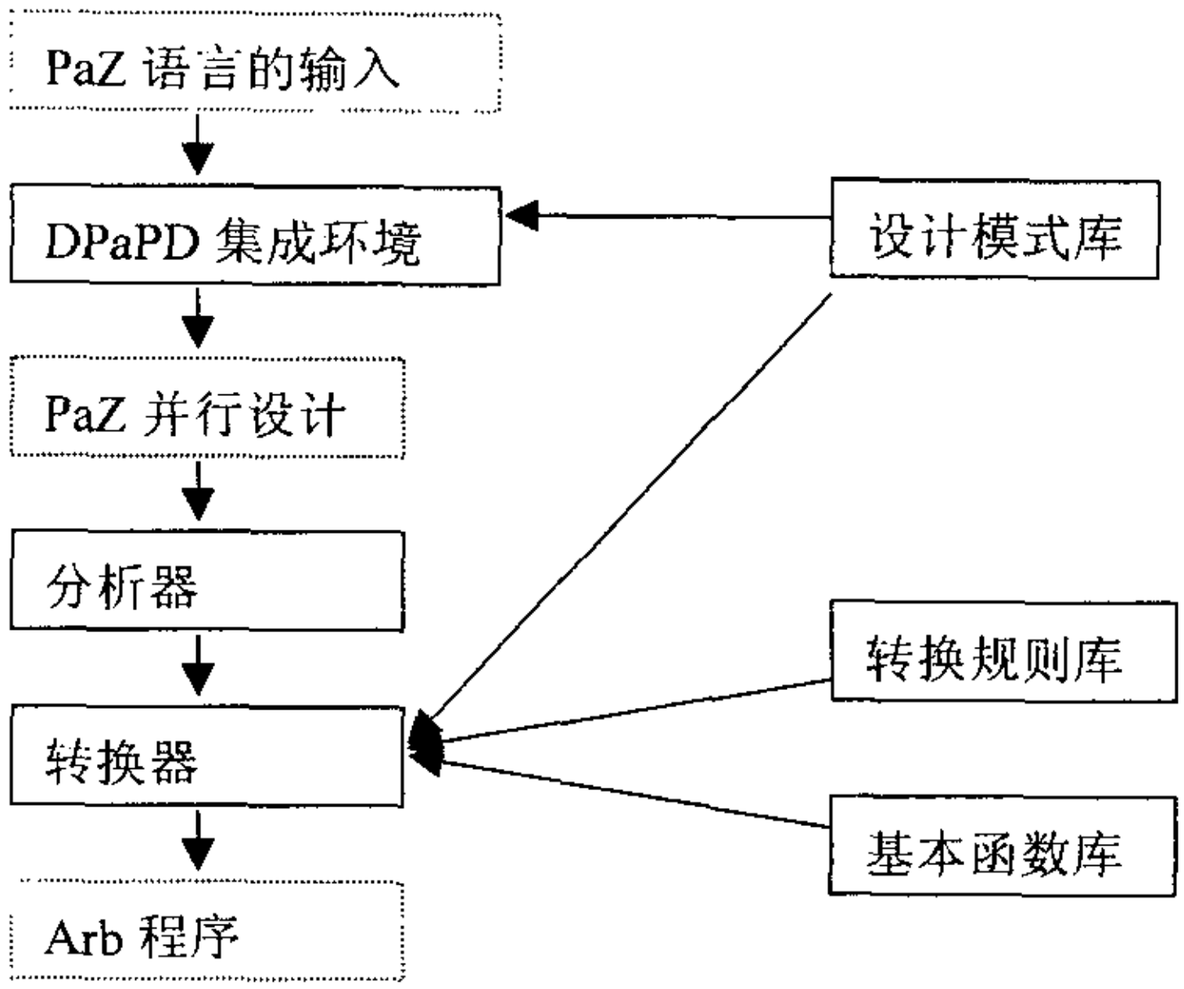


图 7.2 DPaPD 系统的基本结构

Fig. 7.2 Structure of DPaPD System

【输入参数】: MenuFun.menuInfo

其中 menuInfo 的结构为：

mInfo.hwnd 为系统菜单的句柄；

mInfo.Number 将系统菜单分隔的数量；

mInfo.position 要分隔的菜单在系统菜单的位置。

【输出】: 空

2. 分析转换器（Z_pattern.dll）

【功能】: 该部件对用户界面指定窗口中输入的 PaZ 语言文本和系统中内嵌的设计模式进行词法、语法分析，并使用相应设计模式所提供的框架和转换规则库中相应的转换规则将其转换成完整的 Arb 程序。

【引用方法】: Z_patten.Z_patten_compiler

【成员函数 1】: InitEnvirments

【功能】: 在系统启动前，设置系统的有关环境变量。

【输入参数】: 空

【输出】: 空

【成员函数 2】: CompilingSource

【功能】: 对用户所定义并从指定窗口输入的设计模式的各个子项进行分析和转换。

【输入参数】：byval sText as string, byval NoUnit as integer

其中 sText 为设计模式子项的内容。NoUnit 为设计模式子项的编号。

【输出】：空

【成员函数 3】：GetCompilingResult

【功能】：取得转换后的结果。

【输入参数】：byval PattenName string

PattenName 为用户所选定的设计模式名。

【输出】：字符串

3．基本函数库（function.Lib）

对于 PaZ 中的一些运算符，由于其抽象程度较高，在 arb 模型中没有直接与之相对应的运算符，而是需要通过一个 arb 子过程来对其进行精化，例如 $\sum$ 、∪、\等。

对于这些运算符，我们通过在系统中建立一个基本函数库，在精化所得的 arb 程序中的相应部分直接调用对应的函数来实现。

4．设计模式库

设计模式库用于保存和管理系统中所定义的设计模式，这些设计模式提供给用户选用，从而为问题的求解提供一个基本的框架。目前我们系统中只定义了两个设计模式，但系统提供了良好的接口，以便设计模式的增加和修改。

5．转换规则库

转换规则库的建立主要依据第五章所描述的 PaZ 到 arb 的转换规则，它提供给转换器作为转换的依据。

7.3 系统运行平台

DPaPD 系统在微机的 Windows 系列操作系统上均可运行。

7.4 系统的实现技术

7.4.1 系统实现的主要功能

目前，本系统所实现的功能包括：

1.设计模式的选择

目前本系统中只包含两种设计模式（Parallel Divide&Conquer 和 Parallel Branch&Bound）的定义。用户界面的设计模式窗口（pattern）中罗列出系统中定义的所有设计模式的名字，用户可以针对具体的问题在窗口中选择，选定一个设计模式后，其下方则显示出该设计模式所要求定义的各个成分的窗口，用户则将相应的定义输入对应的窗口中。

2.PaZ 语言的输入

本系统中提供了很方便的 PaZ 语句的输入功能。由于 PaZ 语言是一种数学的语言，而不是一种程序语言，因此其中包含了大量的数学符号，而这些符号在键盘上没有对应的字符。为了方便用户的输入和系统的分析，我们在系统中提供了特殊符号的窗口，用户只需在输入窗口中点击鼠标右键，即可得到一个弹出菜单，菜单上列出了特殊字符，再使用鼠标左键选取所需的符号。

3.输入、输出结果的保存功能。

为了避免用户每次重新输入和重新转换，系统实现了对输入输出结果文本的保存功能，只要激活所需保存的窗口，使用主菜单中的文件保存按钮即可进行保存。

4.打开 PaZ 文本

相应地，系统可以打开用户所保存的 PaZ 文本文件，在打开时，系统自动将原先的文本段放入对应的输入窗口。

5.打印输入及输出结果

为了方便用户将文本打印出来进行阅读、分析，系统实现了各窗口文本的打印功能。

6.分析和转换

这是系统的核心功能。在用户完成对应输入后，只需点击转换按钮，系统将完成对输入的分析和转换，得到完整的 arb 程序输出。目前本系统的输出为在 C 语言上进行扩充的 arb 语言，而[51]中实现的 arb 语言为 FORTRAN 语言的扩充。

7.4.2 主要实现技术

下面我们将对系统中的核心功能部件——分析转换器的主要实现技术进行介

绍。

1. PaZ 文本输入分析

对于用户输入的文本，我们将其分为类型定义、变量声明和语句三类，象 PaZ 中的基本类型定义、状态模式定义和自由类型定义等归类为类型定义进行处理，对于全局变量的定义和公理描述中的变量声明也进行统一的处理，公理描述中的谓词和全局的约束作为语句进行处理，目前我们没有对类属式定义的分析处理。因此 PaZ 文本分析如图 7.3 所示：

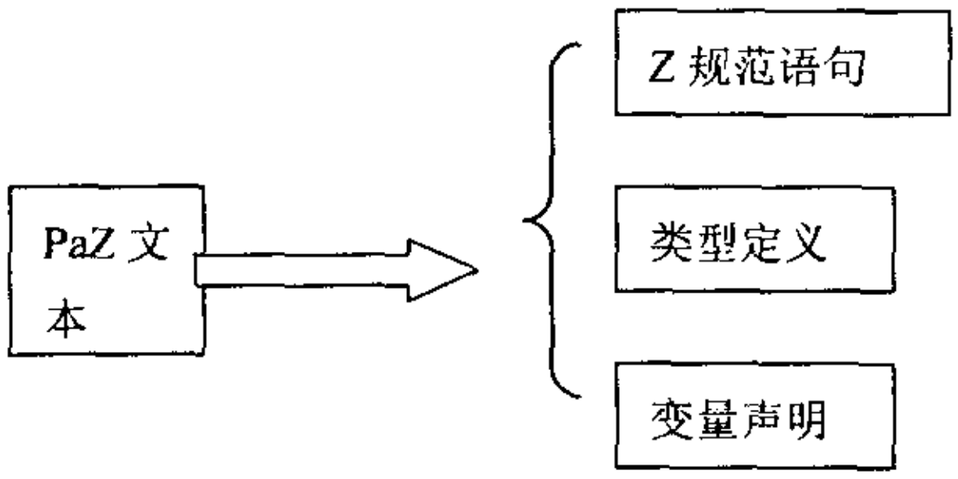


图 7.3 PaZ 文本分析

Fig. 7.3 Text Analysis of PaZ

在具体处理中，我们对以上三类语法成分分别进行：

(1)类型定义的识别

Function TypeAnalysis(ByRef sText As String) As String '≙

为了方便识别和分析，我们要求对状态模式和类型的定义使用符号≙，且模式的定义采用水平形式，用中括号作为分隔符，逗号作为模式中各声明的分隔符，如：object≙ [w:real,p:real,in_knapsack:int] 定义一个状态模式，objects≙F object 定义一个自由类型等等。在分析时，通过检测≙符号来进行确定。

(2)类型定义的处理

Function TypeDefinition(ByVal sText As String) As String

将 PaZ 文本中的类型定义转换为相应的 arb 程序的类型定义。

(3)变量声明

Private Function VariablesAnalysis(By Val Stext As String) As String

将 PaZ 程序中的变量声明转换为相应的 arb 程序变量声明。

(4)语句分析

funtion statementsAnalysis(ByVal sText As String) As String

通过上述分析剔除出类型定义和变量声明，得到语句段。该函数通过对 PaZ 文本中的语句段进行分析，得到单个的语句。

2. PaZ 语句分析

这里我们所指的PaZ语句实际上是谓词。PaZ语句分析是整个分析中的主要部分，语句中包含各种运算符，我们将语句分成三类进行处理，如图 7.4 所示：

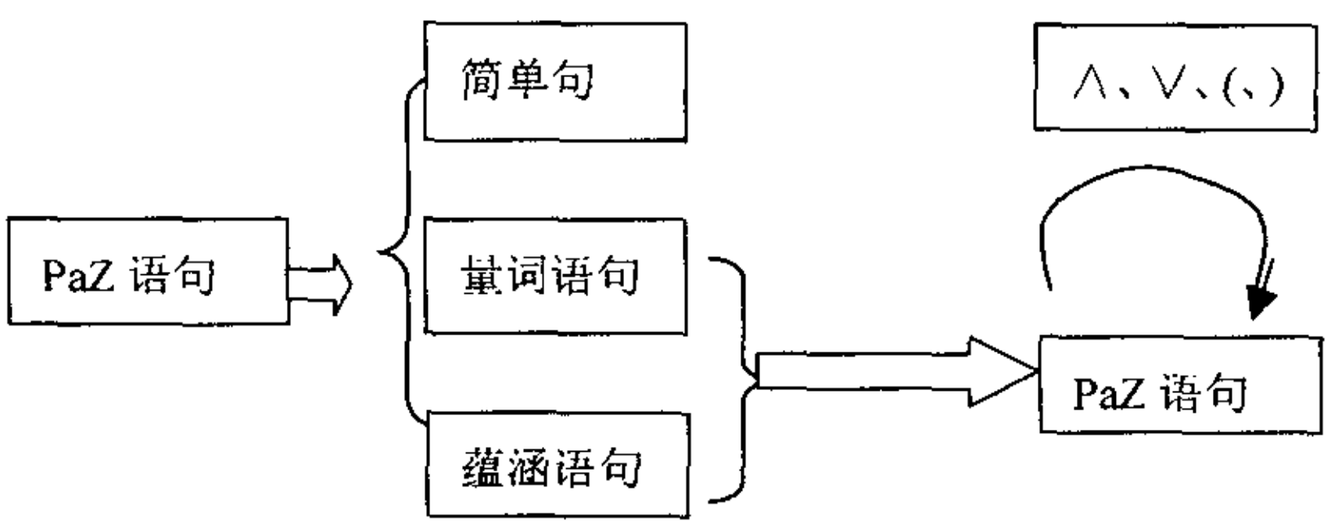


图 7.4　　PaZ 语句分析

Fig. 7.4 Sentence Analysis of PaZ

注：在蕴涵句中，我们规定蕴涵的前件中不可再含有蕴涵句，但后件中可以。

具体处理的实现情况如下：

(1)主分析过程

Function syntaxAnalysis(ByVal sText As String) As String

具体分析时，我们采用图 7.5 所示的流程对语句进行判断：

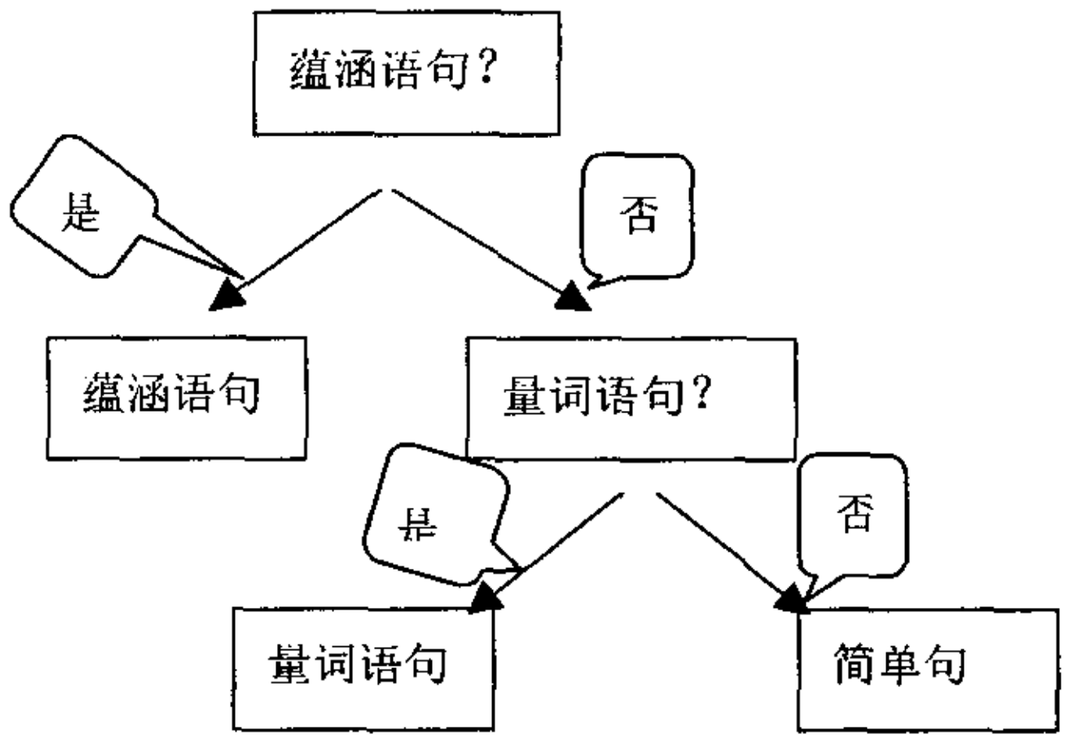


图 7.5　PaZ 语句分析流程

Fig. 7.5 Procedure of PaZ Sentence Analysis

注：若蕴涵前件中含有量词，则切取出并用一个布尔变量代替 bImplication

(2)蕴涵句前件的处理

Function ImplicationCondition(sText As String) As String

将蕴涵前件转换成 if 条件语句中的条件表达式。

Function ConvertToPredication(sText As String) As String

若蕴涵前件中含有量词语句，则转到量词语句的处理，使用一个 arb 程序段实现，并设置布尔变量代替 bImplication。

(3)简单句的处理

Function Convert_Z_To_C_statement(ByVal sText As String) As String

对简单句进行分析并根据转换规则将其转换成 arb 中的对应表达式。

(4)量词语句的识别与处理

Function Qstyle(ByVal sText As String) As styleQuantifier

分析语句是否为量词语句以及是什么类型的量词语句（全称量词或存在量词）。其中 styleQuantifier 的定义为：

```
Enum styleQuantifier
   Existential
   Universal
   Normal      'non-quantifier style
End Enum
```

Function GetQuantifierInfo(ByVal sText As String) As QuantifierInfo

取得量词的信息。其中 QuantifierInfo 的定义为：

```
Type QuantifierInfo
   Quantifier As String
   initialValue As String
   initialSymbol As String
   finalValue As String
   finalSymbol As String
   setOperator As Boolean
   setElement As String
End Type
```

3. 设计模式的实现

设计模式是问题求解的基本框架，因此也是最终程序的主控程序的框架。由于我们在 DPaPD 模型下对设计模式的描述方法和规范定义的描述方法是一致的，即都是统一使用 PaZ 语言的描述方法，因此对于设计模式的实现也使用同样的分析方法。设计模式往往描述为一个公理，它是由类型定义、变量声明和谓词构成，根据上述处理，可以对相应成分进行分析和转换。在 arb 程序的结构上，设计模式转换的结果为

调用关系的最外层，它依赖于其各成分的定义和转换结果。

系统中定义了两个设计模式，其中 Parallel Branch&Bound 的处理由函数 Function addBBFunction() As String 完成，Parallel Divide&Conquer 的处理由 Function addDCFunction() As String 完成。

7.4.3 系统实现的主要特点

本系统在实现上具有以下主要特点：

1．充分体现 DPaPD 并行程序开发模型的主要思想和特点，结构简单，方便易用。通过一目了然的系统界面，简单方便的输入形式，使 DPaPD 模型的主要思想很方便地被用户接受和使用。

2. 良好的开放性和可维护性。在系统的维护过程中，一些对系统功能的进一步需求可以方便地增加进来。例如设计模式的添加、新的 PaZ 语言成分的词法、语法检测部件、错误及异常处理等都可动态添加到系统中。

3. 本系统是一个抽象并行程序的开发系统，但它采用了纯文本的转换系统的实现形式，独立于任何程序语言开发环境，对软硬件运行环境均无特殊要求。

7.5 运行实例及结果

本系统通过一些运行实例的测试，证明其能够获得稳定的结果。上一章中的傅立叶变换和 0-1 背包问题的实例均在系统中测试通过，此外，我们还测试通过了一些其它简单的实例，如数组元素分别加一、数组元素累加等。对于快速排序问题，本系统不能产生最后的结果，这是因为其中 sortdivide 函数的精化中包含了算法设计，不能全自动地完成（参见 6.3.4.3 节）。

对于傅立叶变换问题，用户在规范窗口中输入 DFT 问题的规范，系统所要求的输入仅为问题规范的声明部分和包含主函数的谓词部分，不需输入前后置条件，这是因为系统中不包含程序正确性验证的功能，关于设计是否满足问题的前后置条件需要在 DPaPD 并行程序开发的第三步中手工完成（参见 6.1 节）。因此在规范窗口输入：

```
a,b:seq complex
w:seq complex
FFT:seq complex→seq complex
b=FFT a
```

然后在设计模式窗口选择设计模式 Parallel Divide&Conquer，此时其下方出现四个子窗口，分别为该设计模式的成分 basedatas，baseoperate ,divide 和 merge。将相应的定义输入对应的窗口(定义参见 6.2.2 节) ，确认输入无误后点击转换按钮，系统则

开始对所有输入和所选定的设计模式进行分析、转换，得到快速傅立叶变换的 arb 程序。由于系统未进行优化处理，转换所得的 arb 程序与手工精化的结果(参见 6.2.4 节)略有不同，但不影响程序的实质。

0-1 背包问题的测试过程与上例同，只是在设计模式窗口选择设计模式 Parallel Branch&Bound，再将相应的定义输入对应的窗口，经转换得到求解 0-1 背包问题的 arb 程序。

7.6 存在的问题和系统的局限

由于时间关系，本系统目前只是一个初步的实现，还远远没有达到完整和完善的程度。PaZ 语言中还有一些语法成分的分析转换尚未考虑，例如类属模式等。因此，我们现有的系统所能识别和处理的是 PaZ 语言的一个子集。总结而言，系统的局限主要有以下方面：

1．某些 PaZ 语言成分不能识别

系统目前无法识别的 PaZ 语言成分包括类属模式定义、操作模式定义、数学库中的包、模式的运算、绑定运算、前置条件运算等。

2．量词表达式中约束变量的类型

系统目前处理的量词表达式中约束变量只能为整型。对于其他的类型，例如以集合形式表示约束范围的某种类型约束变量，此时首先将该表达式进行一个预先的转换，使其约束变量转换到整型，例如转变为使用下标对该集合进行遍历：

例如有集合 D : F Objects

现量词表达式为　d∈D •… …

则通过如下转换：

```
{声明部分}
    int index;
    Objects d;
{代码部分}
    arball(index=1:#D)
     d=D[index];
```

此时，集合类型精化为数组，而量词中的约束变量转变为整型进行处理。

3．函数定义域的限制

当函数有多个定义域时，要求其为相同的定义域，例如f: X×X→Y。在分析函数

自变量的问题上，我们采取的是简易的方法，只考虑同类型的自变量而没有考虑多种不同类型的自变量。这一部分的改进不存在技术上的困难，可以进一步完成。

4．错误检测及错误处理

系统目前只能检测PaZ语言中一般性的语法错误，如括号不匹配、量词与蕴涵语句复合错误等，而没有进行比较复杂的检查，如类型检查等。系统目前没有提供错误处理的手段，因此要求用户输入语法上正确的PaZ定义。

5．设计模式的管理

目前尚没有提供设计模式管理的手段，但留下了设计模式添加的接口。

第八章　结束语

8.1 本文工作总结

并行程序设计难是并行计算中的难点之一。而并行程序设计的困难主要存在于两个方面：问题的并行求解和并行程序的编码。并行程序编码的目标是将一个并行算法在具体的并行机上通过程序的形式进行实现，其困难主要在于并行计算机系统结构的多样性所导致的统一并行计算模型的缺乏，多年来并行计算模型的领域集中了大量的研究工作，而统一的并行计算模型是研究的一个理想与目标。而问题并行求解的目标是通过对问题的分析和其中固有并行性的开发得到求解该问题的并行算法，其困难主要在于问题的多样性和求解过程中所需的创造性劳动，使得这一过程难以进行自动化。只有综合考虑这两个方面的因素，才能从根本上解决从问题到求解问题的并行程序的整个并行程序开发的困难。本文的工作正是基于这样的一种考虑进行的。尽管我们只是提出并实现了一个初步的模型，但对于解决并行程序设计难的问题，推动并行计算研究的发展具有一定的现实意义。

本文的具体内容可以从理论和实现两个方面来看。本文在理论方面的工作主要包括：

1．设计模式概念的扩展。

设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述，其目标是将设计经验以人们能够有效利用的形式记录下来。这一概念在面向对象软件工程领域中提出并越来越流行。在并行程序设计领域，尽管也有使用设计模式的研究，但并未对这一概念引入并行程序设计领域进行明确的定义。我们将设计模式引入我们的并行程序设计模型，是根据其思想对这一概念进行了明确的定义，并对其各个主要的方面（如分类、描述、使用等）进行了研究。

2．DPaPD 模型的提出

基于我们所提出的设计模式概念和我们对于并行程序设计的基本思想，本文提出了一个基于设计模式的并行程序设计模型 DPaPD。该模型不仅是一个算法设计和程序编码的并行抽象平台，更是一个支持并行程序设计全过程的方法和辅助工具，包括并行程序的设计、推导、精化、验证、转换等。

3．PaZ 语言

由于本文所提出的模型支持从问题到程序的全过程，因此离不开设计规范描述语言的研究。Z 语言是软件工程中一种常用的规范描述语言，它不仅可以描述问题的规范，而且可以描述设计和进行精化。根据具体系统的研究，Z 语言有许多扩充版本。

在 DPaPD 模型中，我们对 Z 语言进行了一个简单的扩充，得到 PaZ 语言，使其可用于描述模型中的问题规范、设计模式和问题的并行设计。

本文在实现方面的工作主要包括：

1．设计模式的开发和实现

根据我们的设计模式概念和 DPaPD 模型下进行程序设计的需要，我们开发了若干设计模式，包括单步分而治之设计模式、并行分而治之设计模式、并行分枝限界设计模式等。其中单步分而治之设计模式用于对问题进行单步分而治之求解及其相应并行算法的设计，并行分而治之设计模式用于对问题进行分而治之方法的求解及相应并行算法的设计，并行分枝限界设计模式用于对问题进行分枝限界方法的求解及相应并行算法的设计。这些设计模式可分别应用于许多问题的求解。

2．PaZ 语言的实现

在 DPaPD 模型中，PaZ 语言可以用于描述问题的并行设计。我们开发并实现了从 PaZ 语言到一个抽象并行模型 arb 的转换规则，使用这些规则可以将一个描述问题并行设计的 PaZ 文本自动转换、精化得到等价的 arb 并行程序，而 arb 程序可以进一步自动转换成在共享内存或分布式内存并行机上运行的并行程序。

3．DPaPD 并行程序开发系统的设计实现

基于以上的工作，本文最终实现了 DPaPD 并行程序开发系统。该系统通过将问题的规范描述、用户所选择的设计模式和用户针对问题进行的一些定义组合成问题求解的并行设计，并将其自动转换为 arb 并行程序，从而实现了 DPaPD 模型支持并行程序设计全过程的基本思想，为并行程序的设计提供了一种切实可行的方法和支持工具。

8.2 进一步的工作展望

本文提出并实现了一个基于设计模式的并行程序开发系统 DPaPD，但本文目前的工作还仅限于该模型的研究与探索阶段，距离一个成熟完善并可投入实际应用的并行程序开发模型来说还存在不小的差距。因此，我们进一步的工作将从以下几个方面进行开展。

1．更多设计模式的开发使用

目前我们所提出并实现的设计模式相当有限，它们虽然可用于解决许多问题，但对于实际应用的需要还远远不够。因此，我们将进一步对问题领域进行研究，开发更多的设计模式。当设计模式的数量达到一定程度时，将同时带来一个新的课题，就是设计模式的分类、选择问题。在我们目前的系统中，设计模式的数量少、功能的区别明显，用户可以比较容易地选择所需的设计模式。而这一情况发生变化以后，设计模式的选择将变得十分困难。即使在面向对象领域，设计模式的使用已达到十分流行的

程度，但在实际应用中这一问题并未得到很好解决。因此，关于设计模式的研究还有许多工作要做。

2．设计的精化

在并行设计向 arb 抽象程序的精化、转换过程中，有些精化规则是确定的，可以自动地转换，但还有许多不确定的精化情况。例如一些表达式可以对应多种精化的选择，而没有明确的精化选择条件；还有些情况没有直接对应的精化规则，仍然需要创造性的劳动等。这些都给自动的精化、转换带来了难以解决的困难。面对这些情况应该如何有效地处理，是系统中亟待解决的一个问题。

3．程序的正确性

程序的正确性是程序设计中一个普遍关心的问题。DPaPD 模型所依据的程序设计三级结构的基本思想，有利于程序的正确性证明。在 DPaPD 模型下，程序的正确性分成两部分。一是设计的正确性，即设计是否满足问题规范的约束。DPaPD 模型的第三步（验证）所解决的就是这个问题。目前这一步的验证依靠手工完成，如何在系统中实现这一功能是我们进一步的工作之一。在设计的正确性通过验证后，才进入到下一步的精化。另一部分就是从设计到程序的正确性，即精化、转换所得的程序是否与设计等价。这一部分我们将通过精化规则的正确性证明来保证。

4．系统的集成

在 DPaPD 模型所描述的系统中，本文的工作是其中的一部分（参见图 2.3），即从问题规范到 arb 并行程序。而从抽象的 arb 并行程序到具体并行机上运行的并行程序的转换是[51]中所做的工作。当两部分工作都比较完善时，需要将它们集成到一个系统中，这是我们的最终目标。

附录 PaZ 语言的语法

Specification ::= Paragraph NL. . . NL Paragraph
Paragraph ::= [Ident, . . . , Ident]
| Axiomatic-Box
| Schema-Box
| Generic-Box
| Schema-Name[Gen-Formals] $\triangleq$ Schema-Exp
| Def-Lhs==Expression
| Ident::=Branch | . . . | Branch
| Predicate

Axiomatic-Box ::= [| Decl-Part
| Axiom-Part]

Schema-Box ::= ├Schema-Name [Gen-Formals]—
[| Decl-Part
| Axiom-Part]

Generic-Box ::= | [Gen-Formals]
| Decl-Part
[|
| Axiom-Part]

Decl-Part ::= Basic-Decl Sep. . . Sep Basic-Decl
Axiom-Part ::= Predicate Sep. . . Sep Predicate
Sep ::= ; | NL
Def-Lhs ::= Var-Name[Gen-Formals]
| Pre-Gen Decoration Ident
| Ident In-Gen Decoration Ident
Branch ::= Ident
| Var-Name《Expression》
Schema-Exp ::= ∀ Schema-Text • Schema-Exp
| ∃ Schema-Text • Schema-Exp
| $\exists_1$ Schema-Text • Schema-Exp
| Schema-Exp-1
Schema-Exp-1 ::= [Schema-Text]
| Schema-Ref
| ¬ Schema-Exp-1 U
| pre Schema-Exp-1 U
| Schema-Exp-1 ∧ Schema-Exp-1 L
| Schema-Exp-1 ∨ Schema-Exp-1 L
| Schema-Exp-1 ⇒ Schema-Exp-1 R

		Schema-Exp ⇳ Schema-Exp	L
		Schema-Exp ‖ Schema-Exp	L
		Schema-Exp-1 ⌒ Schema-Exp-1	L
		Schema-Exp-1 ↾ Schema-Exp-1	L
		Schema-Exp-1 \ (Decl-Name, , , , , Decl-Name)	L
		Schema-Exp-1 $\fatsemi$ Schema-Exp-1	L
		Schema-Exp-1 >>Schema-Exp-1	L
		(Schema-Exp)	
Schema-Text	::=	Declaration [| Predicate]	
Schema-Ref	::=	Schema-Name Decoration[Gen-Actuals] [Renaming]	
Renaming	::=	[*Decl-Name/ Decl-Name*, . . . , *Decl-Name/ Decl-Name*]	
Declaration	::=	Basic-Decl; . . . ; Basic-Decl	
Basic-Decl	::=	Decl-Name, . . . , Decl-Name : Expression	
	|	Schema-Ref	
Predicate	::=	∀ Schema-Text • Predicate	
	|	∃ Schema-Text • Predicate	
	|	∃$_1$ Schema-Text • Predicate	
	|	*let Let-Def*; . . . ; *Let-Def* • *Predicate*	
	|	Predicate-1	
Predicate-1	::=	Expression Rel Expression Rel . . . Rel Expression	
	|	Pre-Rel Decoration Expression	
	|	Schema-Ref	
	|	true	
	|	false	
	|	¬ Predicate-1	
	|	Predicate-1 ∧ Predicate-1	U
	|	Predicate-1 ∨ Predicate-1	L
	|	Predicate-1 ⇒ Predicate-1	L
	|	Predicate-1 ⌒ Predicate-1	R
	|	(Predicate)	L
Rel	::=	= | ∈ | In-Rel Decoration	
Let-Def	::=	Var-Name = = Expression	
Expression-0	::=	λ Schema-Text • Expression	
	::=	μ Schema-Text [• Expression]	
	::=	Let Let-Def ;…; Let-Def • Predicate	
	::=	Expression	
Expression	::=	if Predicate then Expression else Expression	
	|	Expression-1	
Expression-1	::=	Expression-1 In-Gen Decoration Expression-1	R
	|	Expression-2 × Expression-2 × … × Expression-2	
	|	Expression-2	
Expression-2	::=	Expression-2 In-Fun Decoration Expression-2	L
	|	P Expression-4	
	|	Pre-Gen Decoration Expression-4	
	|	⌒ *Decoration Expression-4*	

	|	Expression-4 (| Expression-0 |) Decoration
	|	Expression-3
Expression-3	::=	Expression-3 Expression-4
	|	Expression-4
Expression-4	::=	Var-Name [Gen-Actuals]
	|	Number
	|	Schema-Ref
	|	Set-Exp
	|	⟨[Expression ,..., Expression]⟩
	|	[Expression ,..., Expression]
	|	(Expression ,Expression,..., Expression)
	|	θ Schema-Name Decoration [Renaming]
	|	Expression-4 . Var-Name
	|	Expression-4 Post-Fun Decoration
	|	$\text{Expression-4}^{\text{Expression}}$
	|	(Expression -0)
Set-Exp	::=	{ [Expression,..., Expression] }
	|	{ Schema-Text [• Expression] }
Ident	::=	Word Decoration
Decl-Name	::=	Ident | Op-Name
Var-Name	::=	Ident | (Op-Name)
Op-Name	::=	_In-Sym Decoration_
	|	Pre-Sym Decoration_
	|	_Post-Sym Decoration
	|	_(| _ |) Decoration
	|	_|(_)| Decoration
	|	_Decoration
In-Sym	::=	In-Fun | In-Gen | In-Rel
Pre-Sym	::=	Pre-Gen | Pre-Rel
Post-Sym	::=	Post-Fun
Decoration	::=	[Stroke ... Stroke]
Gen-Formals	::=	[Ident,..., Ident]
Gen-Actuals	::=	[Expression,..., Expression]
In-Fun	::=	⇳ | ‖ | ∘ | ⊕ | ... ...
... ...		
Word	_	Undecorated name or special symbol
Stroke	_	Single decoration: ', ?, ! or a subscript digit
Schema-name	_	Same as Word, but used to name a schema
In-Fun	_	Infix function symbol
In-Gen	_	Infix generic symbol
Pre-Rel	_	Prefix relation symbol
Pre-Gen	_	Prefix generic symbol
Post-Fun	_	Postfix function symbol
Number	_	Unsigned decimal integer

参考文献

[1] Kuo-Chan Huang, Feng-Jian Wang, Jyun-Hwei Tsai: Two Design Patterns for Data-Parallel Computation Based on Master-Slave Model. Information Processing Letters70 (4): 197-204, 1999

[2] Masashi Toyoda, Buntarou Shizuki, Shin Takahashi, Satoshi Matsuoka, Etsuya Shibayama: Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In Proceedings of 1997 IEEE Symposium on Visual Languages (VL'97): 76-83, 1997

[3] Dhrubajyoti Goswami, Ajit Singh, Bruno R. Preiss: Building Parallel Applications Using Design Patterns. In advances in software Engineering: Topics in Comprehension, Evolution and evaluation, New York, NY, Springer-verlag, 2000

[4] Shin-ichi Minato, Giovanni De Micheli: Finding All Simple Disjunctive Decompositions Using Irredundant Sum-of-Products Forms, In Proc. of ACM/IEEE International Conference on Computer-Aided Design (ICCAD'98): 111-117, 1998

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: Design Patterns: Abstraction and Reuse of Object-oriented Design. In O. Nierstrasz (Ed.), European Conference on Object-Oriented Programming, number 707 in Lecture Notes in Computer Science: 406-431. Springer-Verlag, 1993

[6] Christian Kramer, Lutz Prechelt: Design Recovery by Automated Search for Structural Design Patterns in Object-oriented Software. Proceedings of Working Conference on Reverse Engineering (WCRE'96), 1996

[7] Willam B. Frakes, Thomas P. Pole: An Empirical Study of Representation Methods for Reusable Software Components. IEEE Transactions on Software Engineering, Vol. 20, No. 8, 1994

[8] David E. Hudak, Nathan Baughman, Greg Hodges: Aggregates: Using Design Patterns to Create Implicitly Parallel Data Structures in C++. Aerospace and Electronics Conference (NAECON'97), Vol. 1: 239-246, 1997

[9] Jan Bosch: Language Support for Design Patterns. http://www.pt.hk-r.se/~bosch

[10] Gorel Hedin: Language Support for Design Patterns Using Attribute Extension. In Bosch and Mitchell (Eds.), Object-Oriented Technology, ECOOP'97 Workshop Reader, LNCS 1357: 137-140, Springer-Verlag, 1997

[11] Eugene J. Rollins, Jeannette M. Wing: Specifications as Search Keys for Software Libraries. In Proceedings of the Eighth International Conference on Logic Programming, Paris, June 1991

[12] Kuo- Chan Huang, Feng-Jian Wang, Jyun- Hwei Tsai: Two Design Oatterns for Data-parallel Computation Based on Master-slave Model. Information processing letters 70: 197-204, 1999

[13] David Hemer, Peter A. Lindsay: Reuse of Verified Design Templates through Extended Pattern Matching. Technical Report TR 97-03, http://svrc.it.uq.edu.au

[14] Anthony Lauder, Stuart Kent: Precise Visual Specification of Design Patterns. In Proc. European Conference on Object-Oriented Programming (ECOOP '98): 114-134, Springer-Verlag, 1998

[15] Chan-Tsun Chang, William C. Chu, etc.: A Formal Approach to Software Components Classification and Retrieval. Proceedings of 21st International Computer Software and Applications Conference (COMPSAC'97), 1997

[16] John Penix, Perry Alexander: Using Formal Specifications for Component Retrieval and Reuse: Proc. 31ST Annual Hawali International Conference on System Science, 1998

[17] Tamar Richner: Describing Framework Architectures: more than Design Patterns. Proceedings of European Conference on Object-Oriented Programming, Workshop on Object-Oriented Software Architectures, http://www.iam.unibe.ch/~richner, 1998

[18] Amnon H. Eden, Joseph Gil, etc.: Towards a Mathematical Foundation For Design Patterns. Technical report 1999-004, Department of Information Technology, Uppsala University, 1999

[19] Christaian Peper, Reinhard Gotzhein, Martin Kronenburg: A Generic Approach to the Formal Specification of Requirements. Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM '97), http://church.computer.org/proceedings, 1997

[20] Tommi Mikkonen: Formalizing Design Patterns. Proceedings of the 1998 International Conference on Software Engineering (ICSE'98): 115-124, IEEE Computer Society Press, http://www.acm.org/pubs/articles/proceedings/soft/302163/p115-mikkonen/p115, 1998

[21] Jean-lin Pacherie: Operator Design Pattern for Data Parallel Computation. In Proc. Technology of Object-Oriented Languages and Systems (TOOLS 23), IEEE Computer Society, 1998

[22] Walter F. Tichy: A Catalogue of General-Purpose Software Design Patterns. In Proc. Technology of Object-Oriented Languages and Systems (TOOLS 23), IEEE Computer Society, 1998

[23] Amy Moormann Zaremski, Jeannette M. Wing: Specification Matching Of Software Components. ACM Transactions on Software Engineering and Methodology (TOSEM): 333-369, Volume 6, Number 4, October 1997

[24] Bernd Freisleben, Thilo Kielmann: Coordination Patterns for Parallel Computing. In D. Garlan and D. Le Metayer(Eds.), Coordination Languages and Models, Proceedings of COORDINATION '97, LNCS 1282: 414—417, Springer, 1997.

[25] Brendan Mahony, Jin Song Dong: Overview of the Semantics of TCOZ. In Proceedings of the 20th International Conference on Software Engineering: 95-104, IEEE Computer Society Press, April 1998

[26] S. Martello, D. Pisinger, P. Toth: New Trends in Exact Algorithms for the 0-1 Knapsack Problem. European Journal of Operational Research (123): 325-332, 2000

[27] Sergei Gorlatch, Holger Bischof: Formal Derivation of Divide-and-conquer Programs: a Case Study in the Multidimensional FFT's. In D. Mery (Ed.), Formal Methods for Parallel Programming: Theory and Applications: 80-94, 1997

[28] Berna L. Massingill, K. Mani Chandy: Parallel Program Archetypes. Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, 1999

[29] Maria Smolarova, Pavol Navrat: Reuse with Design Patterns: Towards Pattern-Based Design. Beijing, China, Y. Feng, D. Notkin and M.C. Gaudel (Eds.): 232-235, PHEI - Publishing House of Electronics Industry, 2000

[30] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, K. R. Sujithan: PRAM Programming: Theory

vs. Practice. In Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain, IEEE Computer Society Press, January 1998

[31] David Lecomber: A Semantic for Parallel Programming with BSP. Technical report, Oxford University Computing Laboratory, http://web.comlab.ox.ac.uk/oucl/work/david.lecomber/, Sep. 1999

[32] D. B. Skillicorn: A New Framework for Software Development. Queen's University, Kingston, Canada, External Technical Report ISSN-0836-0227-1999-432, Sep. 1999

[33] S. Radha, C. R. Muthukrishnan: A Portable Implementation of UNITY on Von Meumann Machines. Comput. Lang., Vol. 18, No.1: 17-30, 1993

[34] H. J. M. Goeman, J. N. Kok, K. Sere, R. T. Udink: Coordination in the ImpUNITY Framework. Science of Computer Programming 31: 313-334, 1998

[35] David B. Skillicorn, Domenico Talia: Models and Languages for Parallel Computation. ACM Computing Surveys, Vol. 30, No. 2, June 1996

[36] R. J. R. Back, A. J. Martin, K. Sere: Specification of a Microprocessor. September 1994

[37] Martin Abadi, Leslie Lamport: Composing Specifications. ACM Transactions on Programming Languages and System, Vol. 14, No.4, October 1992

[38] Ketil Stolen: A Method for the Development of Totally Correct Shared-state Parallel Programs. Proc. CONCUR'91, LNCS 527: 510-525, Springer, 1991

[39] Xu Qiwen, He Jifeng: Laws of Parallel Programming with Shared Variables. In D. Till (Ed.), the 6th Refinement Workshop, Workshops in Computing, BCS FACS, Springer-Verlag, 1994

[40] C. A. R. Hoare, He Jifeng: Unifying Theories of Programming. Prentice-Hall International, 1998

[41] D. B. Skillicorn: The Bird-Meertens Formalism as a Parallel Model. In NATO ARW "Software for Parallel Computation", June 1992

[42] Edmund M. Clarke, Jeannette M. Wing etc.: Formal methods: State of the Art and Future Directions. ACM Computing Surveys, Vol.28, No.4, Dec. 1996

[43] Berna L. Massingill: Experiments with Program Parallelization using Archetypes and Stepwise Refinement. Proceedings of the Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98 / IPPS'98), 1998

[44] D. Goswami, Ajit Singh, Bruno R. Preiss: Architectural Skeletons: the Reusable Building Blocks for Parallel Applications. 1999 International Conference on Parallel and Distributed Processing Techniques and Applications, USA, 1999

[45] Kathryn S. Mckinley: Automatic and Interactive Parallelization. A thesis submitted in partial fulfillment of the requirements for the degree PH.D. , Rice University, Houston, Texas, Mar. 1994

[46] Domenico Talia: Parallel Computation still Need a Unifying Model. http://isi-cnr.deis.unical.it:1080/~talia/./cacm97.ps.

[47] P. J. Parsons, F. A. Rabhi: Specifying Problems in a Paradigm Based Parallel Programming System. In Parallel Computing: State-of-the-Art and Perspective, Proceedings of the International Conference ParCo '95: 215-222, Gent, Belgium, Sept. 1995

[48] Rudolf K. Keller, Reinhard Schauer: Design Components: Towards Software Composition at the Design Level. Proceedings International Conference on Software Engineering 1998: 302-311, 1998

[49] 袁崇义，屈婉玲: 从河内塔的并行解到 UNITY 程序设计. 计算机学报, Vol.21 增刊, 1998.8

[50] M. Cole: Algorithmic Skeletons: Structured Management of Parallel Computation. The MIT Press, Cambrige Massachusetts, 1989

[51] B. Massingill: A Structured Approach to Parallel Programming. (Ph.D. thesis), technical report CS-TR-98-04, California Institute of Technology, 1998

[52] D. C. Schmidt: Using Design Patterns to Develop Reusable Object-oriented Communication Software. Communication of ACM, Vol.38, No.10, October 1995

[53] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995

[54] I. T. Foster: Designing and Building Parallel Programs: Concepts & Tools for Parallel Software Engineering. Addison-Wesley Pub. Company, 1995

[55] Jams Noble: Classifying Relationship Between Object-Oriented Design Patterns. IEEE Software Engineering Conference: 98-107, 1998

[56] Uzi Vishkin: Can Parallel Algorithms Enhance Serial Implementation. Communications of the ACM, Vol.39, No.9, Sep.1996

[57] Ben H. H. Juurlink, Harry A. G. Wijshoff: A Quantitative Comparison of Parallel Computation Models. Annual ACM Symposium on Parallel Algorithms and Architectures: 13-24, 1996

[58] Stephen Siu, Ajit Singh: Design Patterns for Parallel Computing Using a Network of Processors. Sixth IEEE International Symposium on High Performance Distributed Computing: 293-304, Oregon, USA, August 1997

[59] J. Darlington, M. Ghanem, H. W. To: Structured Parallel Programming. Proceedings of Programming Models for Massively Parallel Computers: 160-169, 1993

[60] B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi: SkIE: A Heterogeneous Environment for HPC Applications. Parallel Computing 25, 1999

[61] J. M. Spivey: the Z Notation: a Reference Manual. Prentice Hall International Ltd., 1992

[62] Richard B. Pelz: Parallel FFTs. Parallel Numerical Algorithms: 245-266, 1997

[63] Al Geist, Adam Beguelin: PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994

[64] Duncan K. G. Campbell: A Survey of Models of Parallel Computation. 1997

[65] Carroll Morgan: Programming from Specifications. Prentice Hall, 1998

[66] R. M. Burstall, J. A. Goguen: The Semantics of Clear, A Specification Language. Advanced Course on Abstract Software Specifications, Copenhagen, LNCS 86: 292-332, Springer-Verlag, 1980

[67] H. Ehrig, B. Mahr: Fundamentals of Algebraic Specification. Spring-Verlag, 1985

[68] Jens Clausen, Michael Perregaard: On the Best Search Strategy in Parallel Branch-and-Bound: Best-First Search Versus Lazy Depth-First Search. Annals of Operations Research 90: 1-17, 1999

[69] J. A. Goguen, J. Tardo: An Introduction to OBJ: A Language for Writing and Testing Software Specification. Proceedings Conference on Specifications of Reliable Software, Boston, 1979

[70] Bernard Gendron, Teodor Gabbiel Crainic: Parallel Branch-and-Bound Algorithms: Survey and Synthesis. Operations Research, vol. 42, no. 6, November-December, 1994

[71] Bing Wu, L.Lai: Combining Refinement Calculus with Z Schemas: A Case Study. Technical Report, Dept of Computing, University of Bradford, Aug 1999.

[72] J. V. Guttag, J. J. Horning, J. M. Wing: Larch in Five Easy Pieces. Digital System Research Center Report, July 1985

[73] Jim Woodcook, Jim Davies: Using Z: Specification, Refinement, and Proof. Prentice Hall, May 1996

[74] ANSI Technical Committee X3H5 Parallel Extension for Fortran, document number X3H5/93-SD1-Revision M., April, 1994

[75] I. T. Foster, K. M. Chandy: FORTRAN M: A Language for Modular Parallel Programming. Journal of Parallel and Distributed Computing 26(1): 24-35, 1995

[76] 徐家福等：软件自动化. 清华大学出版社，1993

[77] C. Jones: Whither Formal Methods: A Plea to Investigate New Application. Proceedings of The First IEEE International Conference On Formal Engineering Method (ICFEM'97): 5, IEEE CS Press, 1997

[78] J. R. Abrial: The B book - Assigning Programs to Meanings. Cambridge University Press, August 1996

[79] H-E Erikkson, M. Penker: Design Java Apps with UML. Excerpted from UML Toolkit, New York, Wiley&Sons, 1998.

[80] UML Consortium: Object Constraint Language Specification, version 1.1. http://www.rational .com, 1997

[81] Sergei Gorlatch, Holger Bischof: A Generic MPI Implementation for a Data-Parallel Skeleton: Formal Derivation and Application to FFT. Parallel Processing Letters 8(4): 447-458, 1998

[82] Michael G. Lagoudakis: The 0-1 Knapsack Problem: an introductory survey. http://www.cs.duke.edu/~mgl/papers.html, 1996

[83] Sergei Gorlatch: From Transformations to Methodology in Parallel Program Development: A Case Study. Microprocess Micriprogram, Vol. 41, No. 8-9: 571-588, 1996

致 谢

在本文完成之际，我首先衷心感谢敬爱的孙永强教授给予我一次宝贵的学习机会及为我的成长所付出的心血，本文的研究始终是在孙老师的悉心指导下完成的。

三年来，孙老师对我的学习、生活和工作给予了极大的关怀和帮助。孙老师深厚广博的学识修养、严谨求实的治学态度、敏锐活跃的学术思想、崇高的敬业精神和平易近人的长者风范深深地影响和教育了我，并将成为我毕生受益的宝贵财富。

感谢尤晋元教授、张申生教授、陆鑫达教授、沈恩绍教授、李家滨教授等老师精彩的课程，他们渊博的知识、孜孜不倦的育人精神让我获益匪浅。

感谢陆鑫达教授、白英彩教授、徐良贤教授、黄林鹏教授、李明禄教授在对本文预审中提出宝贵的修改意见，使我更好地完成本文。

感谢薛锦云教授给予我学术研究上的指导和生活、工作上的关怀与帮助，感谢王明文教授对本文的完成所给予的关心和提供的帮助，感谢江西师范大学的领导、计算机系主任周定康教授及全体教师的关怀和支持，使我有更多时间投入于学习和研究。

感谢黄林鹏教授、陆朝俊副教授、陈翌佳博士、陈昌生博士、王明文博士、万燕博士、徐淑廷博士、李勇坚博士及博士生韩莹洁、周冲、沈浩和其它师兄弟三年来的关怀和帮助，为我在上海交大的学习时光增添了美丽的色彩。

感谢我的同学们余敏、韩莹洁、敖青云、张勇、汪良主、毛卫良、冯东雷、陈凯、铁玲、李文一等给予我的帮助。

特别感谢我的先生王伟对我外出求学的理解和支持，以及在艰难时所给予我的鼓励。

最后，感谢父母的养育之恩，及所有亲人对我的关怀。

攻读博士期间的研究工作

一、 本人作为第一作者发表的论文有：

[w1] Expanding Design Patterns to Parallel Programming. In Proc. Technology of Object-Oriented Languages and Systems (TOOLS36), IEEE Computer Society, Oct. 2000 (EI 收录，ID: 1823635)

[w2] A Design Pattern for Developing Parallel Divide and Conquer Programs. International Symposium on Future Software Technology (ISFST'2001), Nov. 2001

[w3] Parallel Programming in SRVRT: A Case Study of 0-1 Knapsack Problem. The 6th International Conference for Young Computer Scientists (ICYCS'2001), Oct. 2001

[w4] 一种异构计算系统中考虑通信冲突的有效任务调度算法 《小型微型计算机系统》（已录用，拟刊于 2002.2）

[w5] 一个并行分枝限界算法产生器的设计与实现 《计算机工程与应用》vol. 37, 2001.9.

[w6] 一种设计模式的混合规范描述模型研究 《计算机工程》2001.5

[w7] 使用规范匹配实现设计模式的自动获取 《小型微型计算机系统》（已录用）

[w8] 一种基于设计模式的三阶段并行程序设计方法 《计算机研究与发展》（已录用）

[w9] 一种从 Z 规约到并行程序的精化方法 《软件学报》（已录用）

二、 本人参与的研究课题有：

1. 国家自然科学基金资助课题“实用的软件形式化方法及其开发工具研究”
2. 国家自然科学基金资助课题“分划递推法应用于高可靠 JAVA 程序开发方法研究”