ABC方法中体系结构风格建模的研究

谭 佳, 孙艳春, 梅 宏
(北京大学信息科学技术学院软件研究所, 北京 100871)

摘 要: 体系结构风格是体系结构设计的重要指导, 它为设计人员的交流建立了公共的术语空间, 促进了设计复用与代码复用. 本文试图为体系结构风格定义提供一个通用的形式化框架, 支持建模风格的结构约束、拓扑约束和交互行为约束, 并且将风格所蕴含的体系结构变化性特征显式化; 通过将该框架引入 ABC方法, 给出了一种基于风格的体系结构建模方法, 并提供了图形化的建模工具.

关键词: 软件体系结构; ABC方法; 体系结构风格; 形式化规约; 动态体系结构;
中图分类号: TP311 文献标识码: A 文章编号: 0372-2112 (2006) 05-0969-08

Modeling Architecture Style in ABC Methodology

TAN Jia, SUN Yan-chun, MEI Hong
(*Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871, *China*)

Abstract Architecture style is an important guideline for the architecture design. It constructs a common term space for the communication of the stakeholders and benefits the design reuse as well as code reuse. This paper attempts to provide a general formal framework for the style specification and make the variability of software architecture explicit. This framework can support modeling structure constraint, topology constraint and behavior constraint of the style. Through the introduction of the framework into ABC method, the paper offers a style-based architecture modeling method and supplies a graphical architecture modeling tool.

Key words software architecture; ABC (Architecture-based Component Composition); architecture style; formal specification; dynamic software architecture

1 引言

自上个世纪 90年代初期开始, 软件体系结构受到了学术界广泛的关注与重视, 并被认为将会在未来的软件开发中起到重要的作用. 随着软件复杂性的增加, 算法和数据结构的设计已经不再是困扰软件开发人员的主要问题, 如何合理设计系统的全局结构, 特别是如何在问题空间与解空间之间建立关联并保证系统功能性需求和非功能性需求的完整实现, 成为软件开发亟待解决的关键问题之一. 软件体系结构的研究[1, 2]反映了这一趋势. 它作为大型软件系统“全局组织结构”的刻画, 其作用主要在于: 首先, 它为系统提供了一个高层抽象的全局结构视图, 这一视图描述了系统的构成元素以及它们之间的交互, 从而为系统实现与维护提供了蓝图; 其次, 软件体系结构必须解决系统功能指派问题, 即如何在需求与实现之间建立可追踪性的关系, 所以体系结构缩小了问题空间与解空间的差距; 第三, 软件体系结构可以帮助开发人员分析和推理系统的全局性质, 如性能、安全性和伸缩性等.

体系结构风格作为“可复用的组织模式和习语”[3], 其内容也在不断丰富. 本质上, 风格为体系结构设计提供了一个模版, 它大致包含四个方面的内容[4]: 首先, 风格定义了一组构件类型和连接子类型, 即提供了一个系统建模的词汇表, 如管道过滤器风格中的管道和过滤器; 其次是定义了这些类型之间的配置规则, 或是拓扑约束, 这些约束往往表现为模型的全局结构性质; 第三, 为各种建模元素提供了精确的语义解释, 在风格的语法定义和语义模型之间建立映射; 第四是定义了风格特定的质量属性分析方法, 如性能预测和可靠性验证等. 风格的引入为设计人员的交流建立了公共的术语空间, 促进了设计复用与代码复用. ABC方法[5, 6]是一种以软件体系结构作为指导, 通过组装已有构件得到目标系统的软件开发方法, 它以体系结构模型为系统蓝图, 将分布式构件技术作为构件组装的实现框架和运行时支撑, 使用工具支持的映射规则缩小设计和实现的距离, 自动地组装和验证目标系统. 目前, ABC方

收稿日期: 2004-11-15; 修回日期: 2006-01-19
基金项目: 国家 973重点基础研究发展规划 (No. 2002CB312003); 国家自然科学基金 (No. 60125206, 60233010, 60303004, 60503028)

法对于体系结构风格的支持尚存在不足,特别是没有为体系结构描述提供形式化模型,难以验证风格对于系统设计所施加的约束,也无法实现针对特定风格的质量属性分析.以往体系结构形式化建模[7]的对象往往是体系结构自身,如 Wright[8]使用 CSP建模构件的交互行为,Rapide[9]使用部分事件序列描述控制与数据的转移,Darwin[10]使用 p演算建模体系结构的变化性等,但是实践证明,其成效难如人意.工业界以代码为中心的软件开发方法决定了其无法大规模采用难于理解和使用的形式化方法.所以本文尝试在体系结构模型的元层上对风格的约束语义进行描述,使用图形化的建模工具来辅助验证模型的正确性.

本文试图提出一个体系结构风格的形式化框架,支持建模风格的结构约束、拓扑约束和交互行为约束,并且将风格所蕴含的变化性特征显式化;通过将该框架引入 ABC方法,提供了一种基于风格的体系结构建模方法(ABC/SAM, Style-based Architecture Modeling),保证了风格在建模过程中的指导作用;最后给出一个图形化的体系结构建模工具,支持模型的自动生成与验证.

2 相关研究工作

软件体系结构的早期研究者 M. Shaw在文献[11]中将风格定义为"体系结构风格为构件之间的交互提供了一种抽象",即风格的界定仅仅依赖于体系结构中的连接子,构件交互的方式决定了体系结构的风格类型.但是,随着研究者对于风格实例分析[2,12]和形式化建模[13~15]研究的深入,风格定义逐渐演变为"一组已确认的体系结构组织的集合"[4],它刻画了某些体系结构所共享的结构性质和约束规则.从某种意义上来看,体系结构建模的过程即是将风格实例化并规约应用相关的功能语义的过程.

Aesop[4]试图将风格引入体系结构设计过程,它为体系结构风格描述提供一个通用的对象模型,该模型主要包含七类成分:构件、连接子、配置、接入点、角色、表示和绑定,然后针对每一种特定的风格类型将以上实体子类型化,从而达到自定义体系结构风格的目的.另外,Aesop还为开发人员提供了一套工具集,用于从自定义的风格描述自动生成体系结构开发环境,以支持特定风格的体系结构建模.但是该方法并没有为风格描述提供必要的形式化模型,风格约束完全依赖于开发人员在程序中实现,所以其在体系结构的精确描述和性质分析方面存在欠缺,也很难验证风格对于体系结构设计所施加的约束.

风格形式化的研究主要集中在风格的精确描述和高层性质验证上.在文献[14]中,风格被定义为"从语法到语义的解释",这一点是通过为构件、连接子和它们之间的配置规则赋予特定于风格的语义完成的.作者认为,所有的体系结构描述共享一个抽象的语法定义,然后针对每一种特定的风格类型,使用 Z语言定义其语义模型,最后在抽象的语法定义和特定的语义模型之间建立映射从而完成风格的定义.在此基础上,开发人员就可以分析体系结构的性质,比如验证体系结构描述所必须满足的全局结构性质和比较风格之间的异同等,这一建模方法实际上是借鉴了传统编程语言的指称语义定义.在文献[13]中,作者将风格定义为"具有相似形状的体系结构的集合".作者使用图文法来定义体系结构风格,当风格实例化为体系结构时,开发人员使用图重写规则来生成体系结构模型和检查体系结构是否满足预期的约束.这一方法的贡献之处在于将体系结构的拓扑约束作为独立对象显式提出,而不是隐藏在构件交互的行为中.但是,形式化模型难于理解和使用的缺点约束了上述方法的应用,而且它们大多集中于风格的精确描述和高层性质验证上,缺乏一个行之有效的开发方法来指导基于风格的体系结构建模.

Alfa[16]是一种基于体系结构的系统组装语言,它试图提出一套完整的建模概念来支持体系结构风格的构造,其研究重点在于如何更好地构造风格而不是描述风格.Alfa区分了风格的五类构成成分:数据、结构、交互、行为和拓扑,并试图使用这些成分来建模风格模型与体系结构模型.需要指出的是,Alfa并不认为风格是体系结构的"模版",它认为风格就是体系结构,通过对风格求精即可完成体系结构的设计.但是,这一看法弱化了风格对于体系结构建模的指导作用,而且也很难实现针对特定风格的质量属性分析.

3 ABC/SAM 方法概述

ABC方法是一种以软件体系结构作为指导,通过组装已有构件得到目标系统的软件开发方法,它将构件视为黑盒实体,通过描述构件接口的语法模型和语义约束,开发人员即可在构件描述这一抽象层次上进行构件组装.所以,本文将忽略构件的功能计算逻辑的建模,将建模重点放在体系结构风格所包含的约束语义上.

ABC/SAM 将风格视为体系结构设计的模版,它主要包括三个方面的内容:首先,风格定义了一组构件类型和连接子类型,即提供了一个系统建模的词汇表,它们为设计人员的交流提供了公共的术语空间;其次是定义了这些类型之间的语义约束:结构约束用于描述设计元素之间的关联关系和依赖关系,拓扑约束用于规约系统模型的全局拓扑性质,行为约束则用于刻画建模元素之间的交互语义,结构约束和拓扑约束的区别在于前者着眼于局部而后者着眼于全局;第三是定义了风格特定的质量属性分析方法,比如性能预测和可靠性验证等.由于体系结构通常描述的是系统高层的组织结构,所以 ABC/SAM 所适用的范围限于以构件为单元的软件开发方法.开发人员通过规约体系结构风格的性质,ABC/SAM 支持自动验证基于该风格的体系结构模型是否满足预期的风格约束.

风格的定义并不是基于相同的维度,比如面向对象的风格是从构件实现的角度来定义风格,而分层的风格是从

系统拓扑的角度来定义风格. 但是所有的风格都拥有某些公共的特征, 如任何类型的风格一般都会包含构件与连接子两种建模元素, 构件通过连接子发生交互, 构件与构件之间不能直接相连, 必须使用连接子作为中介, 构件之间能够发生交互的前提是二者的接口基调必须保持一致等. 风格的差异性则主要体现在元素类型、控制传递、数据流动和控制/数据交互等四个方面[17], 其中控制视图描述了构件功能执行的时序关系, 数据视图描述了构件交互的数据传递模式, 控制和数据的交互则描述了二者之间关系.

基于以上认识, ABC/SAM 的基本想法是采用继承的思想, 首先定义通用的体系结构风格模型, 然后针对每一种特定风格在其父风格模型的基础上添加其约束语义, 从而完成模型定义. 考虑到形式化模型通常难以为普通的体系结构设计人员所理解和使用, ABC/SAM 区分了风格设计人员与体系结构设计人员两种角色, 其开发流程如图 1 所示: 风格设计人员使用 Alloy 语言为风格建模, 然后将该形式化模型转化为使用 XML 语言描述的风格定义; 体系结构设计人员基于该文本描述使用建模工具设计系统的体系结构模型, 其工作主要包括划分系统功能构件、设计构件接口契约、指派构件类型和连接子类型、在构件和连接子之间建立关联等; 最后开发人员使用映射工具将该模型转化为使用形式化语言描述的实例模型, 并验证系统是否满足预期的风格约束. 这一想法实际上是试图吸收形式化模型的严谨性和文本图形描述的简洁性来辅助体系结构设计, 使我们的方法具有可用性与易用性.

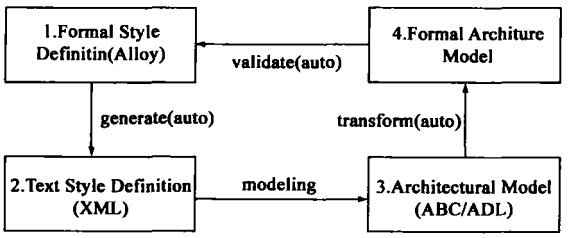


图 1　基于风格的体系结构建模流程

ABC/SAM 使用 Alloy[18,19]作为其形式化工具. Alloy是一种基于集合论和一阶逻辑的形式化语言, 它从 Z中取出了若干适合于描述对象模型的特征, 并扩充了若干新的特征, 比如在其描述中将标量视为单体集合以简化系统描述, 将集合作为退化的关系以提供更加清晰的语义等. Alloy的设计目标是希望能够为系统的结构属性描述提供一个小而完备的建模概念集合, 同时保持语言定义的简洁性. 另外, 通过设计良好的数据结构, Alloy不仅能够被用于描述系统的结构性质, 而且也能够被用于描述系统的操作模型和行为模型.

4　风格约束建模

大致而言, 体系结构风格约束包括三方面的内容: 结构约束, 拓扑约束和交互行为约束. 结构约束用于描述设计元素之间的关联关系和依赖关系, 比如构件之间是否允许交互, 构件包含的接口类型等; 拓扑约束用于刻画系统的高层抽象组织结构, 比如从全局来看体系结构模型中是否允许包含环, 系统全局拓扑是否为星形等; 行为约束则用于规约建模元素之间的交互语义, 比如数据传递是否为先进先出等.

下面的论述将以管道过滤器风格[2]为例, 分别从结构、拓扑和交互行为的角度规约了其约束语义, 并结合经典的编译器实现说明了风格在具体设计过程中的使用. 在管道过滤器风格中, 构件被称为过滤器, 它通常包含一组输入接口和输出接口, 过滤器对输入数据进行处理转换, 产生输出数据; 连接子被称为管道, 它位于过滤器之间, 起到信息流导管的作用. 另外, 过滤器必须是独立的实体, 它并不与其他的过滤器共享状态信息, 也不需要获悉其他过滤器的身份.

4.1　结构约束建模

风格对于体系结构建模所施加的结构约束主要体现在建模元素之间的关联关系和依赖关系上. 体系结构模型描述了系统的构成元素以及它们之间的交互, 从集合论的角度来看, 建模元素代表了基本的原子实体, 具有不可分割性、不可改变性和不可解释性的特点, 它们之间的关系则可以使用数学元组来表示, 每一个元组都包含一个原子实体的序列. 举例而言, 构件包含若干接口这一断言包含两重含义: 首先, 系统包含构件和接口这两种原子实体; 其次, 系统包含从构件到接口的映射关系. 所以, 从数学的角度来看, 结构约束限制了系统中可能出现的关系类型.

ABC方法中的体系结构模型包含两类基本成分: 构件和连接子. 构件是系统中执行计算功能和数据存储的实体, 通常包含若干接口, 这里称之为扮演者 (Player), 它们定义了构件与外界交互的契约, 每一个 Player都会包含若干的基调(Signature), Player之间的继承关系可以使用 Signature的包含关系来描述. 连接子是系统中负责通讯的实体, 具有若干交互角色 (Role), 需要注意的是角色仅仅是一些文字描述, 并不包含类型的概念. 构件和连接子在配置图中完成系统建模, 这里使用 System 来描述. System包含若干的构件实例和连接子实例, 它们分别隶属于上面定义的构件类型和连接子类型, System 还包含三种关系定义: reqBind描述了构件的服务请求接口与连接子角色的绑定关系, provBind描述了服务提供接口与连接子角色的绑定关系, connected则定义了系统中通过连接子关联的构件关系集合. 针对管道过滤器风格而言, 过滤器作为构件类型负责对数据进行转换处理, 通常包含若干数据输入接口和数据输出接口, 管道作为连接子类型负责处理数据流传递, 该风格的结构约束大多继承自通用的体系结构风格. 其 Alloy模型如表 1所示:

事实(fact)原语用于规约总保持为真的系统属性. SoleConfiguration限定体系结构模型中只能包含一个配置

表 1 管道过滤器风格的结构约束模型

```
sig Signature {}
sig Player { sigs: set Signature } { some sigs }
sig Role {}
sig Component { disj reqs, provs: set Player } { some reqs + provs }
sig Connector { disj spons, resps: set Role } { some spons && some resps }
sig InputPort extends Player {}
sig OutputPort extends Player {}
sig Filter extends Component { inputs: set InputPort, outputs: set OutputPort } {…}
sig InputRole extends Role {}
sig OutputRole extends Role {}
sig Pipe extends Connector { input: InputRole, output: OutputRole } {…}
sig System { compInsts: set Component, connInsts: set Connectors,
    reqBind: Player? -> Role,
    provBind: Role->? Player, connected: compInsts -> compInts }
sig PFSystem extends System { filters: set Filter, pipes: set Pipe }
    {…}
fact SoleConfiguration { sole System }
fact Connected {
    all s: System, c: s.compInsts | s.connected[c] = s.reqBind
    [c.reqs].~spons.resps + (s.provBind).~provs + …}
fact BindAcyclic { no System$ connected & iden[Component] }
fact BindSymmetric { all s: System, c: s.connInsts | some s.reqBind
    (c.spons) && some c.resps + (s.provBind) }
fact BindConsistent { all s: System, c: s.connInsts | s.reqBind
    (c.spons).sigs in c.resps + (s.provBind).sigs }
```

元素,即只存在一幅配置图. Connected规约了构件相连必须使用连接子作为其中介. BindAcyclic约束了构件不能请求自己提供的接口. BindSymmetric保证连接子与构件建立绑定关系必须是对称的,即不存在某一特定的构件与连接子的 sponsor建立了绑定关系,却没有另外一个构件与连接子的 responsor绑定. BindConsistent表示构件与构件能够交互的前提是二者接口的基调必须一致,否则无法完成连接. 在此基础上,开发人员就可以针对其构造的系统模型使用以上约束加以验证,保证系统描述的正确性.

表 2 管道过滤器风格的拓扑约束模型

```
fun TopologyChain(s: PFSystem) { some f: s.filters | f*
    (s.connected) = s.filters }
fun TopologyAcyclic(s: PFSystem) { all f: s.filters | f! in f^
    (s.connected) }
```

4.2 拓扑约束建模

风格对于体系结构模型所施加的拓扑约束主要表现在全局结构视图上,比如系统模型中是否允许存在环,是否表现为线性的形状等. 虽然系统建模的目的并不是为了创造某种形状的体系结构模型,但是拓扑约束对于某些系统的功能实现是至关重要的,比如 OSI七层网络协议模型,跨层访问就被认为是非法的. 管道过滤器风格的拓扑约束主要包括:所有过滤器与管道相连表现为线性的形状,但是不能连接成为环状.

4.3 行为约束建模

行为约束模型用于描述构件交互过程中数据和控制的传递约束,刻画了建模元素之间的交互语义. 传统软件体系结构描述的行为模型通常隐藏在配置部分,风格约束也往往散布于构件与连接子的绑定中,如 Wright使用 CSP建模构件的交互行为. 本文将风格所包含的行为约束作为独立实体提出,使用建模元素之间关系的变化描述系统控制和数据的转移,限定了系统中可能出现的交互模式,其约束主要通过断言的形式给出.

以下模型赋予管道过滤器风格中的管道先进先出的数据传递语义,其中,每一个过滤器都包含一个输入数据队列和一个输出数据队列,每一个管道都包含一个数据缓存队列. 数据队列使用 DataList来定义,每一个 DataList都包含一个 first元素和一个 last元素,分别用于指向数据队列中的第一个和最后一个元素. DataList还包含一个数据的集合,用来保存所有的数据,需要注意的是 next关系的定义,修饰词"!"的使用保证 next关系所定义的数据元素是一一对应的,这样就完成数据队列的描述. Insert操作用于将从 过滤器得到的数据插入管道的数据队列中,WriteAssertion断言则用于保证管道接受数据必须是顺序的,而且数据在传递的过程中不会丢失.

表 3 管道过滤器风格的行为约束模型

```
sig Data {}
sig DataList { dataset: set Data, first, last: dataset, next: (dataset - last)
    ! ->! (dataset - first) } { first.* next = dataset }
sig Filter extends Component { dataIn, dataOut: DataList }
sig Pipe extends Connector { dataBuf: DataList }
sig PFSystem extends System { filters: set Filter, pipes: set Pipe }
fun Insert(p, p': Pipe, d: Data) {
        p'.dataBuf.dataset = p.dataBuf.dataset + d
        p'.dataBuf.next = p.dataBuf.next + (p.dataBuf.last -> d)
        p'.dataBuf.last = d }
assertWriteAssertion { all p, p': Pipe, d: Data | Insert(p, p', d) =>
    d in p'.dataBuf.dataset && d = p'.dataBuf.last }
```

5 动态体系结构建模

动态体系结构的出现从本质上来看是提高软件可维护性和适应性的需要. 在某些任务关键的系统中,系统维护无法通过停机来完成,所以系统需要良好的体系结构设计以支持运行时构件的增加、删除和重新配置等. 当前软件系统中包含很多动态体系结构的例子,如微软 IE的插件机制,它允许用户插入新的构件以实现新的功能; JavaGroup的自定制协议栈技术,它依据用户选择的协议能够

自动生成一整套全新的网络多播协议栈; 还比如 OGSA 规范所定义的 Factory接口, 它允许系统根据需要自动生成服务实例等.

软件的变化性主要来自于两个方面原因[20]: 第一是应用需求的变化, 需要开发人员重新设计新的构件或是修改原有构件以体现功能规约的变化; 第二是实现的变化, 出于提高软件质量的原因需要替换构件实现, 如提高系统性能或是增强系统的安全性等, 但是功能接口保持不变. 当前动态体系结构建模的研究主要针对体系结构自身, 研究者在体系结构描述的层次上引入了若干机制用于建模体系结构的变化性. 比如 Darwin[10 21]提供了一整套声明式语言用于描述动态体系结构, 并在 Legis系统中提供了延迟实例化和直接动态实例化两种机制作为其运行支撑, 前者类似于编程语言的模版, 通过为复合构件提供不同的参数, 如构件实例数量等, 从而使其表现为不同的形式; 后者则类似于设计模式中的工厂模式, 能够根据用户的需要生成特定的服务, 这些都需要设计人员精确预测系统所有可能发生的变化. C2[20 22] 是一种基于消息的体系结构风格, 它也提供了一套命令式的脚本语言用于规约体系结构的变化特征, 并提供了图形化的动态体系结构建模工具.

从以上的研究可以看出, 现有的动态体系结构建模往往集中于某些风格特定的系统, 其建模对象大多是体系结构自身, 而且通常需要运行平台的支持. 另外, 并不是所有风格都支持体系结构运行时刻的变化, 如面向对象的风格, 对象交互紧耦合的特点限制了模型变化的能力. 本项研究的特点是在风格的层面上提供了一套通用的、独立于运行平台的机制来描述动态体系结构, 将风格所蕴含的体系结构变化性特征显式化. 从 Alloy的角度来看, 体系结构的变化映射到底层模型实际上就是建模元素之间关系的添加和删除. 下面以支持构件动态添加、删除和重配置的管道过滤器风格为例加以说明:

表 4　管道过滤器风格的变化性模型

```
fun AddComponent(s, s′: DynamicPFSystem, DynFilter f) { s′. filters
    = s. filters + f}
fun DynBindPreCond(s: DynamicPFSystem, f1, f2: DynFilter, p: Pipe)
{
    f1 in s. filters && f2 in s. filters && p in s. pipes; }
fun DynBinding(s, s′: DynamicPFSystem, f1, f2: DynFilter, p: Pipe) {
    DynBindPreCond(s, f1, f2, p)
    s′. reqBind = s. reqBind + f1. reqs -> p. spons;
    s′. provBind = s. reqBind + p. resps -> f2. provs
    s′. connected = s. connected + f1 -> f2 }
```

AddComponent用于向系统添加新的构件, DynBindPreCond原语用于判定是否可以执行动态配置操作, DynBinding原语则用于描述系统的动态重配置. 在此基础上, 开发人员就可以验证系统变化之后是否满足预期的约束. 当然, 该形式化模型仅仅从高层描述了体系结构在什么地方能够发生变化, 并规约了变化之前与之后的区别, 具体变化策略的描述还需要脚本语言的支持, 而且由于 Alloy的限制, 某些量化的性质很难被规约, 如构件的实例数量等, 所以未来我们会考虑在动态体系结构描述中引入某些量化性质的描述.

6　模型生成与验证

模型的生成与验证主要使用建模工具自动完成. ABC/SAM 方法一个重要的优点在于支持风格规约的重用, 在开发人员完成风格性质的规约后, 任何基于该风格的体系结构模型都可以使用工具自动验证该模型是否满足预期的风格约束. 在风格设计人员完成模型定义后, 建模工具支持将风格的形式化描述自动转换为 XML格式的文本描述, 后者主要包含风格定义的构件类型信息和连接子类型信息. 体系结构开发人员基于该文本模型设计系统的体系结构, 其工作主要包括根据应用需求划分系统功能构件、设计构件接口契约、指派构件类型和连接子类型、在构件和连接子之间建立关联等. 在完成系统建模后, 开发人员再次使用映射工具自动地将基于 ABC/ADL[5] 的体系结构描述转化为 Alloy分析器[18] 接受的实例模型信息. 最后是模型验证. 由于一阶逻辑不可判定的特点, 一阶逻辑公式的判定必须要基于特定论域. 体系结构模型包含的建模元素以及它们之间的关系构成了一阶逻辑公式的判定论域, Alloy分析器基于该论域将风格的形式化模型转化为布尔表达式, 然后验证以上表达式是否为真, 即系统模型是否满足预期的风格约束, 详细的验证过程请参见[18]. 需要说明的是当前工具仅仅能够验证模型的静态性质, 动态性质的验证在设计阶段尚缺乏足够的信息.

仍然以管道过滤器风格及基于该风格的编译器设计为例, 在风格设计人员完成模型定义后, 使用工具映射得到的文本模型如下所示, 其中 ComponentType描述了系统允许包含的构件类型, ConnectorType描述了系统允许包含的连接子类型. 在此基础上, 体系结构开发人员根据应用需求, 使用 ABC方法设计基于管道过滤器风格的系统模型, 最后根据模型中的构件类型信息和连接子类型信息与风格的形式化模型建立关联, 并验证风格约束.

表 5　管道过滤器风格的文本描述

```
< Style name= "Pipe-and-Filter" extends= "GeneralStyle" >
  < ComponentType name= "Filter" >
    < PlayerType name= "InputPort" multiplicity= "* " >
    < PlayerType name= "OutputPort" multiplicity = "* " >
  < /ComponentType>
  < ConnectorType name= "Pipe" >
    < RoleType name= "InputRole" multiplicity = "1" >
    < RoleType name= "OutputRole" multiplicity = "1" >
  < /ConnectorType>
< /Style>
```

当体系结构设计人员基于以上风格模型设计程序语言编译器的体系结构时,首先会根据系统需求划分系统的功能部件,传统编译器设计通常包括词法分析器、语法分析器、语义分析器、代码优化器和代码生成器等构件,词法分析器接受用户输入,产生词法单元传递给语法分析器,语法分析器构造语法树并发送给语义分析器,后者生成中间代码,最后经过优化器优化后由代码生成器生成实际可运行的二进制代码.显然,以上每一个构件的类型均为过滤器.其次,体系结构设计人员必须使用合适的连接子类型将识别出来的构件组合成为完整的系统模型,其中一个重要的步骤是确定连接子中所传递数据的格式,数据格式的确定间接决定了系统中每一个构件所请求和提供的接口规约.最后,开发人员在配置图中完成编译器的体系结构设计,其模型如表 6所示.当然,以上步骤都是通过建模工具实现.

表 6 程序语言编译器的体系结构模型

```
< Components>
  < Component name= " LexAnalyzer"  type= " Filter" >
    < Player name= " TokenProvider"  type= " OuputPort" />
  < /Component>
  < Component name= " Parser"  type= " Filter" >
    < Player name= " TokenRequestor"  type= " InputPort" />
  < /Component> …
< /Components>
< Connectors>
  < Connector name= " TokenPipe"  type= " Pipe" >
    < Role name= " TokenSponsor"  type= " InputRole" />
    < Role name= " TokenResponsor"  type= " OutpuRole" />
  < /Connector> …
< /Connectors>
< Configuration>
  < Binding>
    < Requestor role= " TokenSponsor"  componentType= " LexAnalyzer"  Player= " TokenProvider" />
    < Provider role= " TokenResponsor"  componentType= " Parser"  Player= " TokenRequestor" />
  < /Binding> …
< /Configuration>
```

7 工具支持

工具支持是 ABC方法付诸实用的基础.工具对用户屏蔽了 ABC 方法的技术细节,如 ABC/ADL 语言的详细语法、风格约束的验证过程等,这一方面减轻了用户的学习成本,另一方面避免了用户疏忽造成的错误,保证了开发质量.当前我们已经实现了 ABC 方法的支持工具原型 ABC-Tool[23],其功能主要包括:(1)图形化的体系结构建模:用户可以用直观的图形建模的方式生成系统的类型图和配置图;(2)模型映射:工具在风格模型与体系结构模型的形式化描述和文本描述之间分别建立关联,支持格式的自动转换;(3)模型验证:工具将用户提供的风格模型与体系结构模型作为 Alloy分析器的输入,验证系统模型是否满足预期的风格约束;(4)构件库管理:提供了用户管理构件库中可复用构件的能力;(5)代码和文档生成:根据体系结构模型生成代码框架与文档框架,以作为进一步开发实现的基础;(6)构件组装:自动打包部署应用,包括生成部署描述符和将应用部署至中间件运行支撑平台.仍然以基于管道过滤器风格的传统编译器设计为例,使用 ABC-Tool 建模如图 2所示.

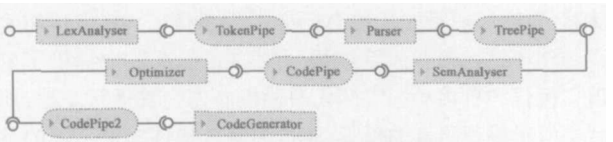


图 2 传统编译器体系结构模型

在完成系统建模后,开发人员可以使用建模工具验证系统的体系结构模型是否满足预期的风格约束,其中关键的一步是将基于 ABC/ADL的体系结构描述转化为 Alloy分析器可接受的实例模型信息,后者主要包含系统模型中定义的原子实体和它们之间的关系.以编译器为例,工具转化得到的形式化模型如图七所示,该模型表明系统包含词法分析器等五类构件,同时包含词法单元管道等四类连接子,connected则定义了通过管道关联的过滤器的关系.

表 7 传统编译器体系结构模型的 Alloy描述

```
Filter  { LexAnalyser, Parser, Optimizer, SemAnalyser, CodeGenerator }
Pipe  { TokenPipe, TreePipe, CodePipe, CodePipe2 }
connected  {
    (LexAnalyser, Parser), (Parser, SemAnalyser), (SemAnalyser, Optimizer), (Optimizer, CodeGenerator) }
…
```

在以上模型的基础上,Alloy分析器即可对于体系结构模型是否满足预期的风格约束做出判断.以上文提到的构件不能请求自己提供的接口为例,其规约形式为 fact BindAcyclic { no System$ connected& iden[Component] }.在基于编译器体系结构模型所提供的论域基础上,以上公式可以转换为表 8所定义的内容.显然,System$ connected所构成集合与 iden[Component]所构成集合的交集为空,从而验证了风格性质的正确性.

表 8 体系结构风格性质验证示例

```
System$ connected = {(LexAnalyser, Parser), (Parser,
    SemAnalyser), (SemAnalyser, Optimizer),
    (Optimizer, CodeGenerator) }
    iden[Component] = { {(LexAnalyser, LexAnalyser), (Parser,
    Parser), (SemAnalyser, SemAnalyser), (Optimizer, Optimizer) }
System$ connected
```

8 结束语

本文为 ABC方法提供了一个体系结构风格的形式化框架,并基于该框架描述了体系结构的变化性. 在体系结构建模的过程中,形式化并不是必须的,但是为 ABC方法提供一套形式化机制,可以提高体系结构模型的描述能力,也可以帮助开发人员验证体系结构模型是否满足预期的风格约束,并在系统设计早期识别系统所存在的问题. 当前 ABC/SAM 方法的重心在于解决以构件为基础的软件开发方法中风格性质验证的问题,支持自动验证风格所包含的各类约束. 未来的工作我们会尝试在风格描述中加入若干质量属性信息,以支持风格特定的体系结构质量分析方法.

参考文献:

[1] Dewayne E Perry, Alexander L Wolf. Foundations for the study of Software Architecture[A]. ACM SIGSOFT Software Engineering Notes[C]. 1992, 17(4): 40-52.

[2] David Garlan, Mary Shaw. An introduction to software architectures[A]. Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering, Vol 2[C]. Singapore: World Scientific Publishing Company, 1993. 1-39.

[3] David Garlan. What is Style? [A]. In Proceedings of Dagshtul Workshop on Software Architecture[C]. Saarbruecken, Germany, 1995. 96-100.

[4] David Garlan, Robert Allen, John Ockerbloom. Exploiting Style in Architectural Design Environments[A]. In Proc Second ACM SIGSOFT Symp on Foundations of Software Engineering[C]. New Orleans, Louisiana, United States: ACM Press, 1994. 175-188.

[5] Hong Mei, Feng Chen, Qianxiang Wang, Yaodong Feng. ABC/ADL: An ADL supporting component composition [A]. In George C M iao Huaikou, eds. Formal Methods and Software Engineering, LNCS 2495, Proceeding of 4th International Conference on Formal Engineering Method, ICFEM 2002[C]. Shanghai, China: Springer-Verlag, 2002. 38-47.

[6] 梅宏, 陈锋, 冯耀东, 杨杰. ABC: 基于体系结构、面向构件的软件开发方法[J]. 软件学报, 2003, 14(4): 721-732.

[7] Mary Shaw, David Garlan. Formulations and formalisms in software architecture[A]. In J van Leeuwen editor, Volume 1000 of Lecture Notes in Computer Science[C]. Springer-Verlag, Berlin, 1995. 307-323.

[8] Robert Allen. A Formal Approach to Software Architecture [D]. Ph D Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, 1997.

[9] David C Luckham, John J Kenney, Larry M Augustin. Specification and analysis of system architecture using rapide[J]. IEEE Transactions on Software Engineering, 1995, 21(4): 336-355.

[10] Jeff Magee, Naranker Dulay, Susan Eisenbach, Jeff Kramer. Specifying distributed software architectures[A]. In Proceedings of the 5th European Software Engineering Conference[C]. London: Lecture Notes In Computer Science, vol 989. Springer-Verlag, 1995. 137-153.

[11] Mary Shaw. Toward higher-level abstractions for software systems[J]. Data & Knowledge Engineering, 1990, 5(2): 119-128.

[12] Mary Shaw. Making Choices: A comparison of styles for software architecture[J]. IEEE Software, special issue on software architecture, 1995, 12(6): 27-41.

[13] Daniel Le Metayer. Software architecture styles as graph grammars[A]. In Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering[C]. San Francisco, USA: ACM Press, 1996. 15-23.

[14] Gregory D Abowd, Robert Allen, David Garlan. Formalizing style to understand descriptions of software architecture [J]. ACM Transactions on Software Engineering and Methodology, 1995, 4(4): 319-364.

[15] Elisabetta D Nitto, David Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures[A]. In Proceedings of the 21st International Conference on Software Engineering[C]. Los Angeles, United States: IEEE Computer Society Press, 1999. 13-22.

[16] Nikunj R Mehta, Nenad Medvidovic. Concise composition of architectural styles from architectural primitives[A]. In Proceedings of the Joint 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering[C]. Helsinki, Finland, 2003.

[17] Mary Shaw, Paul Clements. A Field Guide to Boxology: Preliminary classification of architecture styles for software systems[A]. In Proceedings of COMPSAC97, 21st International Computer Software and Application Conference[C]. 1997. 6-13.

[18] Daniel Jackson. Automating first-order relational logic[A]. In Proceedings of the 8th ACM SIGSOFT Symposium on Foundations of Software Engineering: Twenty-First Century Applications[C]. San Diego, California, United States: ACM Press, 2000. 130-139.

[19] Daniel Jackson, Ilya Shlyakhter, Manu Sridharan. A micro-modularity mechanism[A]. In Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT international Symposium on Foundations of Software Engineering[C]. Vienna, Australia: ACM Press, 2001. 62-73.

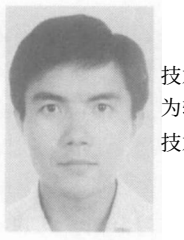
[20] Peyman Oreizy, Nenad Medvidovic, Richard N Taylor. Architecture-based runtime software evolution[A]. In Proceedings of the 20 International Conference on Software Engineering[C]. Kyoto, Japan: IEEE Computer Society, 1998. 177-186.

[21] Jeff Magee, Jeff Kramer. Dynamic structure in software architectures[A]. In Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering[C]. San Francisco, United States: ACM Press, 1996. 3-14.

[22] Richard N Taylor, Nenad Medvidovic, Kenneth M Anderson. A component and message-based architectural style for GUI software[J]. IEEE Transactions on Software Engineering, 1996, 22(6): 390-406.

[23] 向俊莲, 杨杰, 梅宏. 基于软件体系结构的构件组装工具ABC-Tool[J]. 计算机研究与发展, 2004, 41(6): 956-964.

作者简介:



谭 佳 男, 1979年出生. 北京大学信息科学技术学院软件研究所硕士研究生. 主要研究领域为软件工程、软件体系结构、软件复用及软件构件技术等.

孙艳春 女, 1970年出生. 北京大学信息科学技术学院软件研究所副教授. 1999年在东北大学获得博士学位, 同年到北京大学计算机科学技术系工作. 已发表学术论文30余篇, 参加国家"九五"、"十五"重点科技攻关项目、国家"973"重点基础研究发展计划项目、国家"863"高科技发展计划项目、国家自然科学基金项目等国家级项目十余项. 主要研究领域为软件工程、软件开发环境、软件复用及软件构件技术、计算机支持的协同工作等.
E-mail: sunyc@pku.edu.cn

电子学报 2006年第5期 Acta Electronica Sinica No. 5 2006

(总期272期) (Monthly) (Series No. 272)

主管单位	中国科学技术协会	China Association for Science and Technology
主办单位	中国电子学会	Published by the Chinese Institute of Electronics, Beijing
编辑	《电子学报》编辑委员会	Edited by Editorial Board of Acta Electronica Sinica
主编	王守觉	Chief Editor: WANG Shou-jue
总编辑	刘力	Director: LIU Li
通信处	北京165信箱 (邮政编码 100036)	Add: Editorial Office of Acta Electronica Sinica (PO Box 165, Beijing 100036, China)
电话	(010)68279116, 68285082	Tel: 86-10-68279116, 68285082
传真	(010)68173796	Fax: 86-10-68173796

Home page: http://www.elecjournal.org; http://dzxu.chinajournal.net.cn

Email: cje@elecjournal.org; wanghuf@ejournal.org.cn

排版印刷	北京中铁建印刷厂	Printed by Zhongtiejian, Beijing, China
国内总发行	北京市报刊发行局	Distributed by Domestic: Beijing Baokan Faxingju, China
国外总发行	中国国际图书贸易总公司	Foreign: China International Book Trading Corporation
国内订购处	全国各邮电局	Subscription Office——All Local Post Offices in China

国内统一刊号: CN11-2087/TN 邮发代号(国内/国外): 2-891/M436 国内定价 ¥32.00