基于抽象状态机的网格系统设计和分析

刘 晖, 李明禄
(上海交通大学计算机科学与工程系, 上海 200030)

摘 要: 基于可执行规范的实现-测试同步开发模式可以将错误尽早消灭在各个开发阶段的初期. 其理论基础是抽象状态机 ASM, 实现工具是支持. NET 的 AsmL. 本文首先介绍了基于可执行规范的实现-测试同步开发模式、ASM 起源和定义, 然后采用 ASM 描述了网格高层次系统语义, 并举例说明了采用 AsmL 生成有限状态机分析模型语义的方法步骤. 本文认为基于 ASM 的网格系统设计和分析值得学术界和工业界的共同关注.

关键词: 抽象状态机; 抽象状态机语言; 网格; 软件测试; 可执行规范; 有限状态机
中图分类号: TP311, TP393 文献标识码: A 文章编号: 0372-2112 (2003) 12A-2096-05

Abstract State Machine Based System Design and Analysis for Grids

LIU Hui, LI Ming-lu
(*Department of Computer Science and Engineering Shanghai Jiao Tong University, Shanghai* 200030, *China*)

Abstract: Synchronizing software implementation and test with executable specifications could find flaws as early as they occurred in each development stages. The mathematical foundation of this paradigm is that abstract state machine (ASM) and its testing tools are AsmL for .NET. After introducing of the paradigm based on executable specifications and the definitions of ASM, a high level semantic model for grids written in ASM is presented. In order to show the complete processes of analyzing and testing software design with ASM, using AsmL to produce finite state machines for software model of resources mapping and resources request is also illustrated. This paper argues that AMS based system design and analysis for grids deserves attentions from both computer academy and computer industries.

Key words: ASM; AsmL; grids; software test; executable specification; finite state machine

1 引言

与传统的瀑布模型、螺旋模型、渐增模型不同, Microsoft 提出了一种新型软件开发模式: 基于可执行规范(Executable specifications)的实现-测试同步模型[1]. 这是减少大规模开发项目测试代价的一种新途径. 其理论基础是抽象状态机 ASM (Abstract state machine)[2], 实现工具是支持. NET 的 AsmL(Abstract state machine language)[3]. 显然, 这种模式必将有助于已经采用 ASM 来定义的新一代网格系统(Grids)[4]各种软件的开发. 本文在简要介绍相关知识背景的基础上, 举例说明了这种开发模式在网格高层语义模型设计和分析上的应用.

2 基于可执行规范的实现-测试同步模型

基于可执行规范的实现-测试同步开发模式的典型过程如图 1 所示(具体阶段因项目而异). 其核心思想是: 在每个阶段, 开发过程都有与之相应的测试活动; 下一阶段只能在对当前阶段已具有足够满意程度时才能开始. 首先, 开发团队通过用例场景研究并测试软件特性需求(并不需要编码实现). 其次, 开发团队将系统定义为一些主要组件之间的交互, 它们应该支持先前确定的系统功能(还是不需要编码实现). 对主要组件之间的交互关系具有足够的了解后, 开发团队可以根据系统演化时内部状态的需要深入设计组件, 当然, 新组件应该

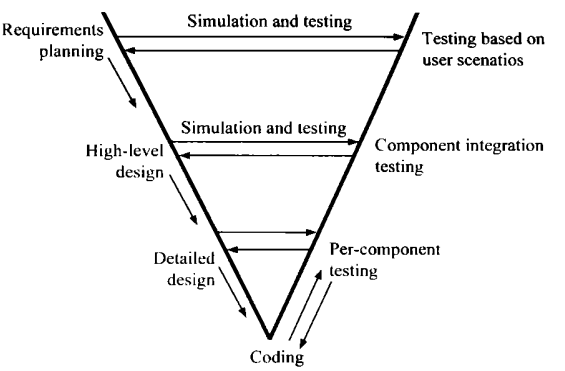

图 1 基于可执行规范的实现-测试同步模式

收稿日期: 2003-08-15; 修回日期: 2003-12-18
基金项目: 国家重点基础研究发展规划 973(No. 2002CB312002); 上海市科委科技攻关重大项目(No. 03dz15027); 上海市科委科技攻关重点项目(No. 025115033)

与先前定义的组件集成方式保持一致. 当详细设计的组件在通过可执行规范实现-测试同步检验后, 可以进行实际编码与编码测试. 在整个开发过程中, 除最终的组件编码外, 其余实现和测试都是通过可执行规范来进行的. 这种开发模式的优点是可以把各个层次的 Bug 消灭在软件生命周期的初期, 减少测试成本中的滚雪球效应[1].

显然, 可执行规范应该具有这样一些特点: (1) 适合严格描述各个层次的语义(Semantic); (2) 能够精确定义各个层次的功能和状态演化; (3) 可以执行. 采用抽象状态机 ASM 书写的规范就符合这些要求.

3　抽象状态机 ASM

3.1　ASM 的发展及应用

ASM 是 Dijkstra 抽象机(Abstract machine) 与采用 Tarski 结构(Structure) 描述抽象状态(Abstract state) 两个概念的结合[5]. Gurevich 对其进行研究的初衷是: (1) 为 Church-Turing 论题引入资源边界的限制; (2) 采用动态结构(Dynamic structures) 定义更为抽象的计算设备. 随后, Börger 认识到动态结构在实践中的潜力, 用其解决了 Prolog 设计中几年来悬而未决的困难并定义了 Prolog 的动态语义(Dynamic semantic) 标准. 这时, 人们了解到 ASM 适合于为变化的情况建模: 从一个可靠的基准模型(Ground model) 开始任意进行水平和垂直调整. ASM 的严格数学定义首次形成于 1993 年. 那时, 它被称为演化代数(Evolving algebra)[6]. 1997 年后, ASM 的最终定义逐渐形成[7, 8].

ASM 已经成功应用于设计包括虚拟机、微处理器体系结构、协议、嵌入式系统和大型软件需求等在内的众多领域[5]. 在微处理器体系结构中, 采用 ASM 的例子包括控制处理单元 zCPU、RSIC 机 DLX 的参考结构、DEC Alpha 处理器系列、TMS3200 C6200 系列. 虚拟机则包括 PVM、JVM. 协议设计中采用 ASM 的有 Lamport 的互斥协议、Kermit 的 FTP 协议、处理器间的 Group membership 协议、UPnP 协议、Kerberos 认证协议. 采用 ASM 设计的语言或标准有 Prolog、C、C++、Java、VHDL、SDL 等[1, 5].

3.2　ASM 的重要定义

能够在任意结构上进行抽象描述的 ASM 包括以下重要定义[2, 6, 7].

定义 **1**　抽象状态机 A 包括有限数目的转移规则(Transition rule) 和初始状态 S_0.

定义 **2**　函数签名(Signature 或 Vocabulary) Y 由函数名称、关系(谓词) 名称及其固定的参数组成. 规定 true, false, undef, = , 以及常规逻辑运算均为函数签名.

定义 **3**　状态 S 表示系统的一个瞬间配置, 是 A 中函数签名(Signature) 上的一个一阶结构. 由一个非空基集 X 和与 X 对应的函数签名 Y 构成.

定义 **4**　域(Universe) 是 1 元关系的特殊名称. 规定 undef 不是域.

定义 **5**　项(Term) 采用迭代方式定义: (1) 一个变量是一个项; (2) 如果 f 是一个 n 元函数签名, 且 $t_1, \cdots t_n$ 是项, 则 $f(t_1, \cdots t_n)$ 也是项.

定义 **6**　更新规则(Update rule) R 通常表示为: $f(t_1, \cdots t_n) := t$. 设当前状态为 S, 执行 R 后, $t_1, \cdots, t_n$ 上的函数解释 f 被换为 t, 进入新状态 S'.

定义 **7**　条件规则(Conditional rule) R 通常表示为: if g then R_1 else R_2 endif.

定义 **8**　并行规则(Do-in-parallel rule) R 通常表示为:

do in-parallel
R_1
R_2
enddo

其中, R_1, R_2, 必须同时执行, 新状态与其出现顺序无关.

定义 **9**　全称规则(Do-for-all rule) R 通常表示为:

do forall v: $g(v)$
$R_0(v)$
enddo

其中, 所有满足 $g(v)$ = true 的项都需要执行规则 R_0.

定义 **10**　非确定规则(Nondeterministic rule) R 通常表示为:

choose v: $g(v)$
$R_0(v)$
endchoose

其中, 如果满足 $g(v)$ = true 的项有多个, 无法确保会选择哪一个执行 R_0.

抽象状态机还包括其他一些定义, 这里不再赘述.

3.3　抽象状态机语言 AsmL

AsmL 是 Microsoft 开发的支持. NET 的 ASM 工具[3]. ASM 的创始人 Gurevich 也是开发人员之一. AsmL 在 ASM 语法基础上扩充了运行程序时应该具备的子机(submachine)、Object、异常处理等机制, 属于非商业性研究、教学软件. 在 Microsoft 内部被用来建模、开发快速原型、分析、生成半自动测试用例、检验 API、设备和协议[9]. AsmL 的语法定义可以参考相关手册[3].

4　基于 ASM 的网格高层语义

采用 ASM 可以为网格各层语义进行规范的说明. 网格的形象比喻是人们在使用各种服务和资源时, 如同将插头插入插座使用电能或者拧开水龙头使用自来水一样方便. 其核心是虚拟组织 VO 通过标准协议和开放式网格服务体系结构 OGSA 进行资源动态共享. 目前, 网格得到了学术界、工业界共同关注[11, 12]. 鉴于对网格的理解、解释、定义还缺乏严格形式化的描述, 学术界已经开始采用 ASM 定义网格的高层语义[4].

根据文献[4] 提供的思路, 网格高层语义包括下列域: APPLICATION(应用), PROCESS(进程), USER(用户), ARESOURCE(抽象资源), PRESOURCE(物理资源), NODE(节点), TASK(运行的进程), ATTR(属性), MESSAGE(消息). 同时, 网格高层语义包括下列函数签名:

app: PROCESS $\rightarrow$ APPLICATION(获得进程所属应用)

attr: {ARESOURCE, PRESOURCE, NODE, TASK} $\rightarrow$ ATTR(获得属性)

belongsTo: PRESOURCE × NODE $\rightarrow$ {true, false}(获得物理资源布局)

canLogin: USER × NODE $\rightarrow$ {true, false}(获得用户在节点上的认证)

canUse: USER × PRESOURCE $\rightarrow$ {true, false}(获得用户对物理资源的授权)

compatible: ATTR × ATTR $\rightarrow$ {true, false}(获得属性的兼容性)

event: TASK $\rightarrow$ {req_res, spawn, send, receive, terminate}(环境事件)

expecting: PROCESS $\rightarrow$ PROCESS × {any}(封锁通信)

from: MESSAGE $\rightarrow$ PROCESS(接收消息)

globalUser: PROCESS $\rightarrow$ USER(获得进程所属全局真实用户)

handler: PRESOURCE $\rightarrow$ PROCESS(物理资源的句柄)

installed: TASK × NODE $\rightarrow$ {true, false}(获得运行进程布局)

localUser: PROCESS $\rightarrow$ USER(获得进程所属局部用户)

location: PRESOURCE $\rightarrow$ NODE(获得物理资源的定位属性)

mapped: PROCESS $\rightarrow$ NODE(获得进程宿主)

mappedResource: PROCESS × ARESOURCE $\rightarrow$ PRESOURCE(将进程和抽象资源映射为物理资源)

request: PROCESS × ARESOURCE $\rightarrow$ {true, false}(进程请求抽象资源)

state: PROCESS $\rightarrow$ {running, waiting, receive_waiting}(获得进程的状态)

task: PROCESS $\rightarrow$ TASK(获得进程有关任务)

to: MESSAGE $\rightarrow$ PROCESS(发送消息)

type: PRESOURCE $\rightarrow$ {reousrce$_0$, …}(获得资源类型)

userMapping: USER × PRESOURCE $\rightarrow$ USER(获得全局用户及物理资源所映射的局部用户)

uses: PROCESS × PRESOURCE $\rightarrow$ {true, false}(进程使用物理资源)

采用 ASM 描述的网格语义的初始状态为:

(1) $\exists p_1, \cdots, p_k \in$ PROCESS, $\forall p_i$, $1 \leqslant i \leqslant k$: $app(p_i) \neq$ undef $\wedge$ globalUser$(p_i) = u \in$ USER.

(2) $\forall p_i$, $1 \leqslant i \leqslant k$: $r \in$ ARESOURCE: request(p_i, r) = true $\forall r \in$ PRESOURCE uses(p_i, r) = false.

(3) $\forall p_i$, $1 \leqslant i \leqslant k$: mapped$(p_i)$ = undef.

(4) $\forall u \in$ USER, $r_1, \cdots, r_k \in$ PRESOURCE: CanUse(u, r_i) = true, $1 \leqslant i \leqslant k$.

采用 ASM 描述的网格语义的更新规则包括:

规则 **1** 资源匹配

if $\exists ar \in$ ARESOURCE $\wedge p \in$ PROCESS: mappedResource(p, ar) = undef $\wedge$ request(p, ar) = true then

choose $r \in$ PRESOURCE where compatible$(attr(ar), attr(r))$

mappedResource$(p, ar) := r$

endchoose

规则 **2** 用户匹配

if $\exists ar \in$ ARESOURCE $\wedge p \in$ PROCESS: request(p, ar) = true $\wedge$

$r \in$ mappedResource$(p, ar) \neq$ undef $\wedge$ canUse(globalUser(p), r) then

if type(r) = resource$_0 \vee \neg(\exists p' \in$ PROCESS): handler$(r) = p'$ then

choose $u \in$ USER where canLogin$(u$, location$(r))$

userMapping(globalUser(p), $r) :=$ localUser(handler(r))

endchoose

elseif $(\exists p' \in$ PROCESS): handler$(r) = p'$ then

userMapping(globalUser(p), $r) :=$ localUser(handler(r))

endif

规则 **3** 资源请求

if state(p) = running $\wedge$ event(task(p)) = req_res(reslist) then

state$(p) :=$ waiting

do forall $r \in$ ARESOURCE: $r \in$ reslist

request$(p, r) :=$ true

enddo

规则 **4** 资源选择

if $\exists ar \in$ ARESOURCE: request(p, ar) = true $\wedge$ $r \in$ mappedResource$(p, ar) \neq$ undef $\wedge$

canUse$(u \in$ userMapping(globalUser(p), $r)$, $r)$ then

if type(r) = resource$_0$ then

mapped$(p) :=$ location(r)

installed(task(p), location$(r)) :=$ true

elseif $\neg(\exists p' \in$ PROCESS): hander$(r) = p'$ then

extend PROCESS by p' with

mapped$(p') :=$ location(r)

installed(task(p'), location$(r)) :=$ true

hander$(r) := p'$

do forall $ar \in$ ARESOURCE

request$(p', ar) :=$ false

enddo

endextend

endif

request$(p, ar) :=$ false

uses$(p, r) :=$ true

规则 **5** 状态迁移

if $\forall r \in$ ARESOURCE: request(p, r) = false $\wedge$ expecting(p) = undef then state$(p) :=$ running

规则 **6** 进程交换

if state(p) = running $\wedge$ event(task(p)) = spawn(reslist) then

extend PRCOCESS by p' with

globalUser$(p') :=$ globalUser(p)

app$(p') :=$ app(p)

state$(p') :=$ waiting

mapped$(p') :=$ undef

do forall $r \in$ ARESOURCE: $r \in$ reslist

request$(p', r) :=$ true

enddo

endextend

规则 **7** 发送

if state(p) = running $\wedge$ event(task(p)) = send(p') then

extend MESSAGE by msg with

to$(msg) := p'$

from$(msg) := p$

endextend

[if expecting$(p') \neq (p$, any) then

expecting$(p) :=$ $(p'$, any)

state$(p) :=$ waiting

endif]

规则 **8**　接收

if state(*p*) = running ∧ event(task(*p*))
= receive(*p*', any) then
if ∃ *msg* ∈ MESSAGE: to(*msg*)
= *p* ∧ from(*msg*) = *p*' then
MESSAGE(*msg*) := false
[expecting(from(*msg*)) := undef
else
expecting(*p*) := (*p*', any)
state(*p*) := receive_waiting]
endif
[if state(*p*) = receive_waiting ∧ (∃ *msg*
∈ MESSAGE): to(*msg*)
= *p* ∧ expecting(*p*)
= (from(*msg*), *any*) then
MESSAGE(*msg*) := false
state(*p*) := running
expecting(from(*msg*)) := undef
expecting(*p*) := undef]

规则 **9**　终止

if state(*p*) = running ∧ event(task(*p*))
= terminate then PROCESS(*p*):
= false

以上就是采用 ASM 定义的网格高层语义的基准模型.

5　基于 ASM 的网格高层语义测试

上述基准模型是对非数学世界的一个数学描述,如何测试是一个相当繁琐的问题. 通常,软件模型的测试方式分为概念合理化测试(Conceptual justification)、试验合理化测试(Experimental justification)以及数学合理化测试[13]. 概念合理化要求软件模型准确捕获系统需求和功能需求. 试验合理化要求为软件模型运行引入测试和验证的可变准则(Falsifiability criteria). 数学合理化强调模型内部一致性,其本质问题是高层次推理和证明.

基于 ASM 的软件模型具有精确性和自然性,对于前两种测试提供了一个介于伪码和文档说明之间的规范. 同时,ASM 的逻辑性和可执行性为数学合理化测试提供了便捷. 借助 ASM,我们既可以从逻辑角度验证软件的内部一致性[14],也可以通过执行工具验证软件的内部一致性. 本文采用后一种方式.

显然,在基准模型中,资源匹配、资源请求与其他规则隶属于不相交的划分. 因此,将其采用 AsmL 语言书写装入 AsmL 测

图 2　在 *AsmL* 测试生成器中编辑规范的配置

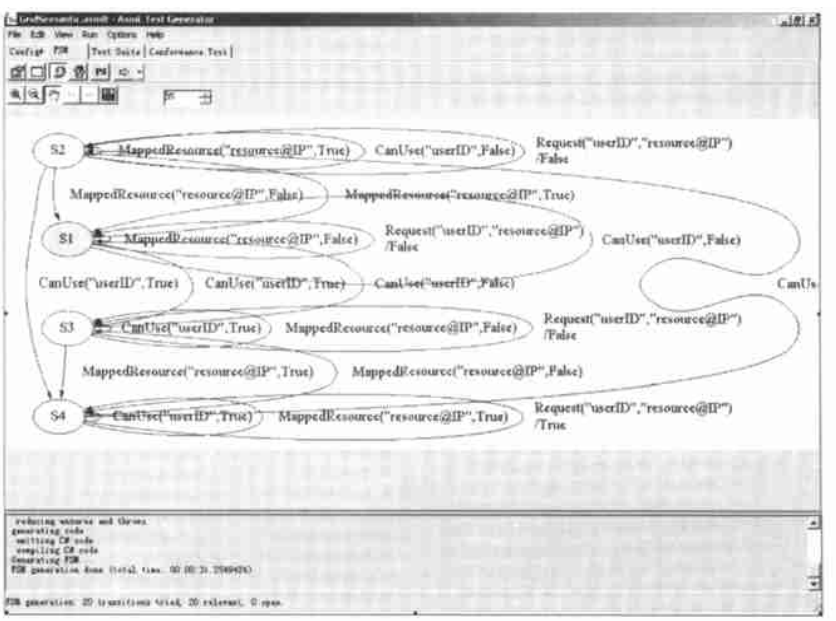

图 3　可执行规范的有限状态机

试生成器中,并按照图 2所示的列表设置模型的初始状态和每个域的取值范围,点击运行,可以生成如图 3所示的有限状态机.

有限状态机有助于用户从逻辑角度和数学角度验证软件设计的合理性.通过有限状态机 FSM 分析测试软件内部一致性更为详细的方法可以参考文献[15].

6 结论

基于可执行规范的实现-测试同步开发模式可以将错误尽量消除在每个开发阶段的初期,其理论基础是抽象状态机 ASM.本文通过实例说明了 ASM 的定义和应用.不难从中理解 ASM 的自然性、逻辑性、精确性、可变的抽象性和描述性.AsmL 则可以从 ASM 书写的可执行规范中生成有限状态机辅助分析、测试软件内部一致性.

采用 ASM 可以在指定抽象层次上精确定义软件模型语义.经过分析测试后,可以进一步在水平方向或垂直方向精炼模型,也就是对指定的语义进行分化及细化.这可以看作是本文说明的过程的新一轮迭代.

最后,我们认为,采用 ASM 定义、描述、设计、分析、测试网格软件,是一种值得研究探索的方式,希望能够引起学术界和工业界的共同关注.

参考文献:

[1] Modeled Computation LLC. Executable specifications: creating testable, enforceable designs[Z]. USA: Microsoft Press, 2001.

[2] Abstract State Machines, the website[DB/OL]. http://www.eecs.umich.edu/gasm.

[3] AsmL, the website[DB/OL]. http://research.microsoft.com/fse/asml.

[4] Zsolt Nemeth, Vaidy Sunderam. Characterizing grids: attributes, definitions and formalisms[J]. Journal of Grid Computing, 2003, 1(1): 9–23.

[5] Egon Börger. The origins and development of the ASM method for high level system design and analysis[J]. Journal of Universal Computer Science, 2002, 8(1): 2–74.

[6] Yuri Gurevich. Evolving algebras 1993: Lipari guide[A]. Egon Börger, ed. Specification and validation methods[C]. BR: Oxford University Press, 1995. 9–36.

[7] Yuri Gurevich. May 1997 draft on the ASM guide[EB/OL]. http://www.eecs.umich.edu/gasm/papers/guide97.html. 1997.

[8] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms[J]. ACM Transactions on Computational Logic, 2000, 1(1): 77–111.

[9] Mike Barnett, Wolfram Schulte. Runtime verification of. NET contracts[J]. Journal of Systems and Software, 2003, 65(3): 199–208.

[10] Mike Barnett, Wolfram Schulte. The ABCs of specification: AsmL, behavior and components[J]. Informatica, 2001, 25(4): 517–526.

[11] I Foster, C Kesselman, J M Nick, S Tuecke. Grid services for distributed system integration[J]. IEEE Computer, 2002, 35(6): 37–46.

[12] Global Grid Forum, the website[DB/OL]. http://www.gridforum.org/.

[13] Egon Börger. High level system design and analysis using abstract state machine[A]. D Hutter, et al, eds. Current trends in applied formal methods[C]. LNCS 1641, Germany: Springer, 1999. 1–43.

[14] Giampaolo Bella, Elvinia Riccobene. Formal analysis of the Kerberos authentication system[J]. Journal of Universal Computer Science, 1997, 3(12): 1337–1381.

[15] I B Burdonov, A S Kossatchev, V V Kulyamin. Application of finite automatons for program testing[J]. Programming and Computer Software, 2000, 26(2): 61–73.

作者简介:

刘 晖 男,1972 年生于湖南长沙人,博士后,主要研究领域为 Web Services 与网格计算.E-mail: liuhui@cs.sjtu.edu.cn;

李明禄 男,1965 年生于重庆,博士,教授,博士生导师,主要研究领域为多媒体计算与生物医学信息学、Web Services 与网格计算、物流与海量空间信息处理.Email: li-ml@cs.sjtu.edu.cn.