第32卷 第9期
2009年9月

计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS

Vol. 32 No. 9
Sept. 2009

基于层次状态机的方面化特征模块的增量式验证

叶 俊 谭庆平 李 暾 徐建军

(国防科学技术大学计算机学院 长沙 410073)

摘 要 方面化特征模块(AFM)是最新提出的软件产品线(SPL)编程范式,能解决现有SPL编程范式存在的问题,但由于AFM范式同时存在模块的并发组合和顺序组合,现有的组合验证技术和模块化模型检验技术并不适用于AFM程序的验证,且目前还未见到针对AFM的验证方法,这制约了AFM范式的应用.文中首次为AFM范式建立了形式化模型,并基于此模型提出一种AFM程序的增量式的验证方法.该方法可以从一个小规模的AFM程序的验证开始,以后每次只添加对新组合的AFM模块的验证,因此可避免直接验证大型AFM程序时可能由于模型的规模太大而无法验证的问题.

关键词 方面化特征模块(AFM);层次状态机(HSM);增量式验证;面向特征编程范式(FOP);面向方面编程范式(AOP)

中图法分类号 TP311 **DOI号**:10.3724/SP.J.1016.2009.01773

Incremental Verification of Aspectual Feature Module Based on Hierarchical State Machine

YE Jun TAN Qing-Ping LI Tun XU Jian-Jun

(*School of Computer Science, National University of Defense Technology, Changsha* 410073)

Abstract Aspectual Feature Module (AFM) is a newly-proposed Software Product Line (SPL) programming paradigm, which solves the problems of existing SPL paradigms. But its inherent hybrid of parallel composition and sequential composition of modules prohibits the application of conventional verification technology, e.g. compositional verification and modular model checking. There is no verification method for AFM paradigm now, which restricts its application. This paper firstly establishes a formal model for AFM paradigm, and proposes an incremental verification method based on this model. This method starts from the verification of a small-scale AFM program and increases with the verification of just the recently-composed AFM module. It can avoid the state space exploration problem caused by a direct verification of a large-scale AFM program.

Keywords Aspectual Feature Module(AFM); Hierarchical State Machine(HSM); incremental verification; Feature-Oriented Programming(FOP); Aspect-Oriented Programming(AOP)

1 引 言

如何更加高效地开发甚至自动生成软件始终是软件工程追求的目标之一,软件产品线(SPL[1])被认为是一种可行的解决方案.方面化特征模块(Aspectual Feature Module, AFM[2-3])编程范式是一种最新提出的SPL编程范式,它解决了现有SPL

收稿日期:2009-04-19;最终修改稿收到日期:2009-08-13.本课题得到国家自然科学基金项目(60773025)以及长江学者和创新团队发展计划资助.**叶 俊**,男,1982年生,博士研究生,研究方向为AOP/FOP的形式化验证.E-mail: yejun.golden@163.com.**谭庆平**,男,1965年生,教授,博士生导师,研究领域为分布式软件工程、形式化方法和自适应软件工程等.**李 暾**,男,1974年生,博士,副教授,研究方向为微处理器设计验证技术、电子CAD技术等.**徐建军**,男,1980年生,博士研究生,研究方向为程序分析.

编程范式——面向特征编程范式(Feature-Oriented Programming,FOP[4])存在的问题. 传统编程范式(例如面向对象编程范式)中,程序的组成模块(例如class)对应系统的参与者(也称为角色);FOP 编程范式下,程序的组成模块对应系统的特征(即可以为用户感知的功能). 两者的对比如图 1. 最近的研究表明[5-6],在许多领域,将系统基于特征而非角色划分能够获得重用性更强的设计. 但 FOP 在表达同构横切时非常繁琐,并且无力表达复杂的动态横切,这限制了它全面表达各类特征的能力. AFM 范式引入了 AOP 编程范式(Aspect-Oriented Programming,AOP[7])的相关设施解决了上述问题.

	actor 1	actor 2	actor 3
feature 1	〈code 11〉	〈code 12〉	〈code 13〉
feature 2	〈code 21〉	〈code 22〉	〈code 23〉
feature 3	〈code 31〉	〈code 32〉	〈code 33〉

图 1　基于特征划分系统与传统的基于角色划分系统的对比(图中竖的实线矩形表示传统模块,横的虚线矩形表示特征模块,特征模块是对各个传统模块中服务于同一功能的代码片段的封装. 特征模块横切了传统模块,因此特征也可被称为横切性关注(简称横切))

AFM 范式要求系统的开发者首先开发一个只实现系统基本功能的最简单的程序,这个程序通常被称为基程序;然后开发一组 AFM 模块实现系统的其他功能. 这样,针对不同用户的需求,开发人员挑选实现了相应功能的 AFM 模块,将它们与基程序组合①,从而得到满足用户需求的程序. 因此,以 AFM 范式开发得到的程序定义如下.

定义 1(AFM 程序). 　以 AFM 范式开发得到的程序称为 AFM 程序,它符合以下递归定义:

(1) 基程序是一个 AFM 程序;

(2) 一个 AFM 程序与 AFM 模块组合得到的仍然是一个 AFM 程序.

作为一种新近提出的编程范式,AFM 在软件工程领域正日益受到重视,并实际应用于一些大型项目的开发(例如 FAME-DBMS②),但目前还没有见到与 AFM 范式的形式化验证相关的工作. 由于 AFM 范式同时存在模块的并发组合和顺序组合,因此现有的组合验证技术和模块化模型检验技术[8-9]并不适用于 AFM 程序的验证. 另一方面,AFM 范式融合了 FOP 和 AOP 范式的思想,虽然 Krishnamurthi 对 FOP 和 AOP 的形式化验证都有研究[10-11],但他的工作早于 AFM 范式的提出,因此没有对两者综合考虑,故不能直接应用于 AFM;且他以状态机作为程序的形式化模型,这使得模型在验证数据相关的属性时会非常复杂和庞大,很可能由于状态空间爆炸而导致验证失败.

针对上述问题,本文首次提出以层次状态机(Hierarchical State Machines,HSM[12])作为 AFM 程序和 AFM 模块的形式化模型,并以此模型为基础,提出一种 AFM 程序的增量式验证方法. 所谓层次状态机,直观地说是一种状态本身(或者说节点)也可以是状态机的状态机. HSM 符合人们由简入繁、逐步精化的思维习惯,因此被广泛应用于系统建模语言中,例如 StateChart、ObjectTime 以及 UML,但 HSM 在软件分析中的应用还未见到. 本文提出以 HSM 作为程序的形式化模型,以其顶层状态机表示程序的 main 函数,底层状态机表示程序的子函数. 由于程序中存在大量的函数复用,因此普通状态机模型中存在许多重复的状态机片段(对应被重复调用的函数),HSM 不存在这个问题,因此可以有效降低模型的规模. 更进一步,本文提出的 AFM 程序的增量式验证方法可以避免对大规模 AFM 程序的直接验证,而将验证仅仅集中在 AFM 模块上. 具体来说,先验证基程序(基程序只实现系统最基本的功能,因此规模较小),然后每次当需要组合新的 AFM 模块(以为系统添加新的功能)时仅仅验证这个新模块. 考虑到模型检验的时间复杂性与模型的规模成正比,而 AFM 模块的规模远小于整个 AFM 程序的规模,因此比之直接验证组合后的程序,本文提出的增量式验证方法可大大降低验证的时间复杂性.

本文第 2 节建立 AFM 范式的形式化模型;第 3 节在此模型的基础上讨论 AFM 程序的增量式验证方法;第 4 节总结全文并展望了今后的工作.

2　AFM 编程范式的形式化模型

AFM 范式的形式化模型应该具有如下特点:

(1) 能够为 AFM 程序建模,并且有相应的高效的验证方法可以直接对建立的模型进行验证;

(2) 能够为 AFM 模块建模;

(3) AFM 程序的模型与 AFM 模块的模型组合后得到的仍然是 AFM 程序.

① 目前基程序与 AFM 模块的组合是以预编译的方式实现的,例如扩展了 C++语言以支持 AFM 的 FeatureC++.

② 这是为嵌入式系统开发的一个高度可配置的嵌入式数据库管理系统产品线,详见 http://fame-dbms.org/

本节提出以 HSM 作为 AFM 程序的模型,同时对 HSM 稍为扩展作为 AFM 模块的模型. HSM 目前已有成熟高效的 CTL 模型检验算法[12],另一方面本文提出的 AFM 程序与 AFM 模块的组合算法可以保证组合得到的仍然是 AFM 程序. 因此本文建立的 AFM 范式的形式化模型符合上述要求.

2.1　AFM 程序的形式化模型

定义 2(层次状态机,Hierarchical State Machines).　基于原子命题集合 P 和偏序集 I 的层次状态机 H 是一个状态机多元组$\{K_i \mid i\in I\}$,其中 K_i 由如下元素构成:

(1) 节点(node)构成的有限集合 N_i.

(2) 超节点(supernode,也称 box)构成的有限集合 B_i. B_i和 N_i互不相交.

(3) 一个初始节点 $in_i \in N_i$.

(4) 一个出口节点 $out_i \in N_i$.

(5) 一个标记(labeling)函数 $X_i: N_i \rightarrow 2^P$. 即 X_i 为 K_i的每个节点标记以 P 的一个子集.

(6) 一个索引(indexing)函数 $Y_i: B_i \rightarrow \{j \mid j\in I$ 且 $i<j\}$. 即 K_i的每个超节点通过 Y_i对应一个在偏序关系 I 上比 i“大”的状态机 K_j.

(7) (超)节点与(超)节点之间的关系 E_i. E_i的元素形如(u,v),u 和 v 都可以是 K_i的节点或超节点.

以上定义中,如果用状态机表示构成程序的一个个函数,状态机的节点代表函数中的语句,超节点代表当前函数对其它函数(即超节点对应的状态机代表的那个函数)的调用,迁移关系代表语句间的控制流,那么可以用 HSM 作为 AFM 程序的形式化模型,定义如下.

定义 3(完全偏序集[13],Complete Partial Order).　一个偏序集$(D,\leq)$是一个完全偏序集(CPO)当且仅当以下两个条件成立:

(1) D 有极小元. 我们用$\perp_D$表示这个极小元,也可简记为$\perp$.

(2) D 中的每条链(chain)都有最小上界.

定义 4(AFM 程序).　设 Σ 是 AFM 程序中的函数取名时使用的字母表,$FN\subset\Sigma^*$ 是函数名依调用关系构成的偏序集,则 Σ 之上的层次状态机$\{K_i \mid i\in FN\}$是一个程序当且仅当 FN 是一个以 $main$ 为极小元的 CPO.

为简化问题,目前不允许函数间存在递归调用,因此以上定义用一个以 $main$ 为极小元的 CPO 描述程序中的各个函数间的调用关系是合理的,因为程序中的所有函数都应该被 $main$ 直接或间接地调用. 下文中对$\perp$和 $main$ 可能不做区别.

2.2　AFM 模块的形式化模型

AFM 模块由 refinement、aspect 和 class 三种元素构成[2],本文提出一个统一的模型表示它们. 首先用一个正规集 J 统一表示三类元素的连接点集合:aspect 通常用 pointcut 表示连接点集合,考虑到 pointcut 是一个正则表达式,可直接转换为等价的正规集;refinement 的连接点即被其 refine 的那些函数,因此也可以用包含有穷元素的正规集表示;至于 class,由于其并不存在连接点,因此相应的 J 为空集. 其次,引入一种特殊的超节点——proceed 超节点——代表三类元素对 proceed 语句的调用:aspect有before、after 和 around 三种类型的 advice. around 类型的 advice 会显式调用 proceed 语句,before和 after 类型的 advice 可认为隐式调用;refinement 可以直接引用被其 refine 的方法,这在语义上等价于调用了 proceed 语句;class 不存在 proceed 调用,因此其对应的状态机中无 proceed 超节点.

定义 5(AFM 模块).　可与 AFM 程序$\{K_i \mid i\in FN\}$(其中 $FN\subset\Sigma^*$)组合的 AFM 模块是一个扩展的 HSM,其状态机额外包含两项元素:(1) Σ 上的正规集 J;(2) 有穷的 proceed 超节点. 我们将这样的扩展状态机记为 K^E,则 AFM 模块是满足如下 3 个条件的 HSM $\{K_i^E \mid i\in AN\}$(其中 $AN\subset\Sigma^*$ 是 AFM 模块中各 class,refinement 和 aspect 的名字构成的集合):

(1) $\forall_{i,j\in AN,\, i\neq j}(J_i \cap J_j \cap FN)=\varnothing$;

(2) 如果 $J_i=\varnothing$,则 K^E无 proceed 超节点;

(3) $\forall_{i\in AN}(main \notin J_i)$.

条件(1)中 $J_i \cap FN$ 表示 K_i^E在 AFM 程序上的连接点集合,因此条件(1)保证了同一 AFM 模块的各个 K^E不会对 AFM 程序的同一函数多次 advice(或 refine),否则可能造成优先权的问题. 条件(2)则是针对 AFM 模块中的 class 以及那些被 aspect 和 refinement 调用的普通函数,由于它们不 advice(或 refine)AFM 程序,因此 $J_i=\varnothing$,且无 proceed 超节点. 条件(3)则保证 $main$ 不是连接点,这是 AFM 范式本身的要求.

2.3　AFM 程序与 AFM 模块的组合

算法 1.　AFM 程序与 AFM 模块的组合.

输入:AFM 程序 p 的 HSM $H_p=\{K_i \mid i\in FN\}$(其中 $FN\subset\Sigma^*$)和 AFM 模块的 HSM $H_a=\{K_i^E \mid i\in AN\}$(其中 $AN\subset\Sigma^*$)

输出:组合得到的程序 p'的 HSM $H_{p'}$

1. foreach $K_i^E \in H_a$ do
2. if ($J_i = \varnothing$) then
3. 以 K_i^E 的 $\langle N_i, B_i, in_i, out_i, X_i, Y_i, E_i\rangle$ 创建一个状态机 K_i，将其加入 $H_{p'}$；
4. else
5. foreach $K_l \in H_p$ do
6. if $l \in J_i$ then
7. 从 H_p 中删除 K_l；
8. 以 K_l 的 $\langle N_l, B_l, in_l, out_l, X_l, Y_l, E_l\rangle$ 创建状态机 $K_{l'}$；
9. 以 K_i^E 的 $\langle N_i, B_i, in_i, out_i, X_i, Y_i[\text{proceed}/l'], E_i\rangle$ 创建状态机 K_l；
10. 将 $K_{l'}$ 和 K_l 加入 $H_{p'}$；
11. fi
12. od
13. fi
14. od
15. 将 H_p 中剩余的 K 拷贝到 $H_{p'}$ 中；
16. 清除 $H_{p'}$ 中没有被 $K_\perp$ 直接或间接调用的状态机.

算法第 9 行出现的形如 $Y[s/t]$ 的函数表示除 s 处取值为 t 外处处取值与 Y 相等的函数. 以图 2 为例对以上算法做简单说明，图 2(a)为一个 AFM 程序$\{K_\perp, K_f\}$，$K_\perp$ 包含超节点 call_f，$Y(\text{call}_f)=f$，图 2(b)为一个 AFM 模块$\{K_a^E\}$，其 K_a^E 的 $J_a=\{f\}^*$，故 $f\in J_a$. 按照算法 1，我们首先为 $H_{p'}$ 创建 $K_{f'}$，$K_{f'}$ 的所有元素都拷贝自 H_p 的 K_f；然后为 $H_{p'}$ 创建 K_f，K_f 除 Y_f 外所有元素都拷贝自 H_a 的 K_a^E，只是 $Y_f=Y_a[\text{proceed}/f']$；最后，直接将 $K_\perp$ 复制到 $H_{p'}$ 中；结果如图 2(c).

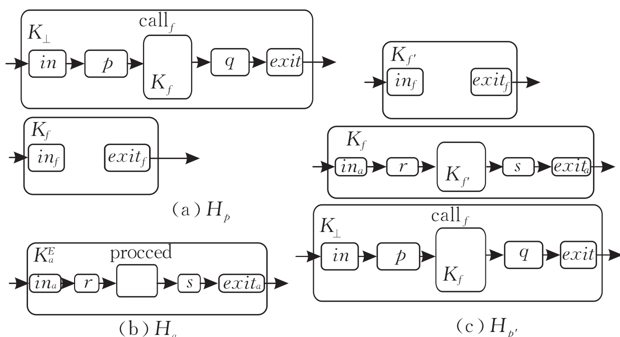

图 2 AFM 程序$\{K_\perp, K_f\}$与 AFM 模块$\{K_a^E\}$的组合（其中$\{K_a^E\}$的 $J_a=\{f\}^*$）

定理 1. 算法 1 生成的 HSM 仍然是 AFM 程序.

证明. 首先，定义 5 的第(3)条保证 $K_\perp$ 不会被算法的第 1～14 行语句处理，因此根据算法的第 15 行，$K_\perp \in H_{p'}$；另一方面，算法的第 1～14 行保证了 $H_{p'}$ 中的状态机都是普通状态机，而非扩展状态机；最后，算法的 16 行保证 $H_{p'}$ 中所有状态机的指标仍然构成以 $\perp$ 为极小元的 CPO；故 $H_{p'}$ 仍然是 AFM 程序. 证毕.

3 AFM 程序的增量式验证

我们称直接针对 AFM 程序的验证为直接验证. 直接验证可能由于 AFM 程序的规模过大而失败. 考虑到 AFM 范式以组合 AFM 程序和各个 AFM 模块的方式生成程序，我们期望能有一种与之匹配的验证方法，即每次只验证新组合的那个 AFM 模块，而不是去直接验证组合后生成的新的 AFM 程序. 本节基于第 2 节建立的 AFM 形式化模型，提出了一种这样的验证方法.

3.1 AFM 程序的 CTL 模型检验

Alur 在文献[12]中给出了的 HSM 的 CTL 模型检验算法，其总体思路是，将待验证公式的子公式依从小到大的顺序依次标记到每个状态机的每个节点上；但在标记每个子公式的前，先将 HSM 的每个状态机 K_i 替换为两个结构完全相同的状态机 K_i^0 和 K_i^1，并假设它们处于不同的上下文中. 具体来说，将 K_i^0 置于封闭环境下，即 K_i^0 的节点上的标记只能依赖 K_i^0 内的路径；而将 K_i^1 置于开放环境下. 举例来说，对于像 $\text{E}(p\cup q)$ 这样的公式，如果 K_i^1 中有一条从 in_i 到 out_i 的路径，其上所有节点都标有 p，就可以为 in_i 标上 $\text{E}(p\cup q)$，因为可以假设 q 的满足发生在 K_i^1 之外；K_i^0 则不然.

基于如下两个原因，本文对 Alur 的算法做了一些修改：(1) 定义 2 给出的 HSM 与 Alur 的定义稍有不同，Alur 的定义中 HSM 的各个状态机指标是全序关系，定义 2 则是偏序关系，因此需要修改算法反映这个差别；(2) 第 3.2 节给出的增量式验证算法依赖于 AFM 程序验证过程中的一些信息，因此需要修改 Alur 算法以收集这些信息.

修改后的 Alur 算法如算法 2，它可在验证 AFM 程序的 HSM 模型的同时收集增量式验证所需信息，其中 $FN\subset\Sigma^*$，且$\{0,1\}\not\subset\Sigma$，P 是原子命题集合，粗体文字为算法 2 与 Alur 算法的不同之处.

算法 2. AFM 程序的 CTL 模型检验算法.

输入：AFM 程序的 HSM $H=\{K_i \mid i\in FN\}$（H 是一个 P 上的 HSM）和待验证 CTL 公式 φ

输出：(1) H 是否满足 φ；

(2) 用于增量式验证的二元关系 IV；

(3) $sub(\varphi)$

算法过程：

$sub(\varphi):=\varphi$ 的子公式列表，按照公式长度依升序排列

IV $:=\varnothing$

foreach $\psi\in sub(\varphi)$ do

case ψ：

$\psi\in P$：skip；

$\psi=\neg\chi$：

foreach $u\in N$ do if $\chi\notin X(u)$ then

$X(u):=X(u)\cup\{\psi\}$ fi od

$\psi=\psi_1\wedge\psi_2$：

foreach $u\in N$ do

if $\psi_1\in X(u)$ and $\psi_2\in X(u)$ then

$X(u):=X(u)\cup\{\psi\}$ fi

od

$\psi=\mathrm{EX}\,\chi$：$H:=CheckNext(H,\chi)$；

$\psi=\mathrm{E}(\psi_1\cup\psi_2)$：$H:=CheckUntil(H,\psi_1,\psi_2)$；

$\psi=\mathrm{EG}\,\chi$：$H:=CheckAlways(H,\chi)$；

od

if $\psi\in X(in_{\perp})$ then return 'yes'; else return 'no'; fi.

其中 $CheckNext$ 以 P 上的层次状态机 $H=\{K_i\,|\,i\in FN\}$ 和子公式 χ 作为输入，输出一个新的层次状态机 H'. 令 $\psi=\mathrm{EX}\,\chi$，$CheckNext$ 假设 ψ 的所有子公式都已经被正确地标记在 H 的节点上了，$CheckNext$ 的目标就是为所有满足 ψ 的节点标上 ψ，这分两步完成，第 1 步是将 FN 划分为两个子集：YES，NO，并记录相应信息于 IV 中以备增量式验证. FN 的元素 $i\in YES$ 表示 $\chi\in in_i$；$i\in NO$ 则相反. 划分的步骤如图 3 上半部分，其中 $last(list)$ 表示 $list$ 的最后一个元素，$first(list)$ 则表示 $list$ 的第一个元素(即 $\perp$). $\models_i$ 是将 K_i 的超节点都视作标记以 YES 或 NO(即超节点对应的状态机的指标所属的集合，下同)的普通节点后，节点在 K_i 内与公式间的满足关系. 例如对 $\forall u\in N_i\cup B_i$，$u\models_i\mathrm{EX}(\chi\vee YES)$ 就表示 u 在 K_i 内存在后继满足：如果是节点，就标记有 χ；如是超节点，就标记有 YES. $CheckNext$ 第 2 步为所有满足 ψ 的节点标记上 ψ，如图 3 下半部分.

$CheckUntil$ 以 P 上的层次状态机 $H=\{K_i\,|\,i\in FN\}$ 和子公式 ψ_1，ψ_2 作为输入，输出一个新的层次状态机 H'. 令 $\psi=\mathrm{E}(\psi_1\cup\psi_2)$，$CheckUntil$ 同样分两步完成，但第 1 步是将 FN 划分为 3 个子集：YES，NO，$MAYBE$. FN 的元素 $i\in YES$ 表示 in_i 可以在 K_i 内满足 ψ；$i\in MAYBE$ 表示 in_i 可能满足 ψ，在 K_i 内无法确定；$i\in NO$ 则表示 in_i 在 K_i 内不满足 ψ. 划分的步骤如图 4 上，$\models_i$ 的意义与 $CheckNext$ 相同. 例如 $u\models_i\mathrm{E}[(\psi_1\vee MAYBE)\cup(\psi_2\vee YES)]$，就表示 K_i 内存在一条以 u 为起点的路径 $uv_1v_2\cdots v_k$ 满足：v_k 如果是节点，则标有 ψ_2，如果是超节点，则标记有 YES；对 $v_i(1\leq i<k)$，如果 v_i 是节点，则标有 ψ_1，如果是超节点，则标记有 $MAYBE$. 需特别指出的是，算法中的 $(\psi_1\wedge out_i)$ 表示 out_i 上标有 ψ_1. $CheckUntil$ 的第 2 步为 H 中所有满足 ψ 的节点标记上 ψ，如图 4 下. $CheckAlways$ 与 $CheckUntil$ 非常类似，在此不再赘述，详见图 5.

将 FN 依其偏序关系表示为图 G；

以宽度优先为序搜索 G，得到函数名列表 $list$；

for $i=last(list)$ downto $first(list)$ do

if $\chi\in in_i$ then $YES:=YES\cup\{i\}$；$IV:=IV\cup\{\langle i,YES\rangle\}$；

else $NO:=NO\cup\{i\}$；$IV:=IV\cup\{\langle i,NO\rangle\}$；

fi

od

---

foreach $i\in FN$ do

将 K_i 替换为两个状态机 K_{i0} 和 K_{i1}：

$K_{i0}=\langle N_i,B_i,in_i,out_i,X_{i0},Y_i',E_i\rangle$，

$K_{i1}=\langle N_i,B_i,in_i,out_i,X_{i1},Y_i',E_i\rangle$，

其中 Y_i'，X_{i0}，X_{i1} 取值如下：

(1) 对 K_i 的超节点 b，如果 $Y_i(b)=j$ 且 $b\models_i\mathrm{EX}(\chi\vee YES)$，

那么 $Y_i'(b)=j1$，否则 $Y_i'(b)=j0$；

(2) 对 K_{i0} 的节点 u，如果 $u\models_i\mathrm{EX}(\chi\vee YES)$，

那么 $\psi\in X_{i0}(u)$，否则 $\psi\notin X_{i0}(u)$；

(3) 对 K_{i1} 的节点 u，如果 $u\models_i\mathrm{EX}(\chi\vee YES)$ 或者 $u=out_i$，

那么 $\psi\in X_{i1}(u)$，否则 $\psi\notin X_{i1}(u)$；

od

return $\{K_i\,|\,i\in FN\cdot\{0,1\}\}$.

图 3　$H:=CheckNext(H,\chi)$，图中 $\psi=\mathrm{EX}\,\chi$

$YES:=\varnothing$；$MAYBE:=\varnothing$；$NO:=\varnothing$；

将 FN 依其序关系表示为图 G；

以宽度优先为序搜索 G，得到函数名列表 $list$；

for $i=last(list)$ downto $first(list)$ do

if $in_i\models_i\mathrm{E}[(\psi_1\vee MAYBE)\cup(\psi_2\vee YES)]$ then

$YES:=YES\cup\{i\}$；$\boldsymbol{IV}:=\boldsymbol{IV}\cup\{\langle \boldsymbol{i},\boldsymbol{YES}\rangle\}$；

else

if $in_i\models_i\mathrm{E}[(\psi_1\vee MAYBE)\cup(\psi_1\wedge out_i)]$ then

$MAYBE:=MAYBE\cup\{i\}$；$\boldsymbol{IV}:=\boldsymbol{IV}\cup\{\langle \boldsymbol{i},\boldsymbol{MAYBE}\rangle\}$；

else $NO:=NO\cup\{i\}$；$\boldsymbol{IV}:=\boldsymbol{IV}\cup\{\langle \boldsymbol{i},\boldsymbol{NO}\rangle\}$；fi

fi

od

---

foreach $i\in FN$ do

将 K_i 替换为两个状态机 K_{i0} 和 K_{i1}：

$K_{i0}=\langle N_i,B_i,in_i,out_i,X_{i0},Y_{i0},E_i\rangle$，

$K_{i1}=\langle N_i,B_i,in_i,out_i,X_{i1},Y_{i1},E_i\rangle$，

其中 Y_{i0}，Y_{i1}，X_{i0}，X_{i1} 取值如下：

(1) 对 K_{i0} 的超节点 b，如果 $Y_i(b)=j$ 且

$b\models_i\mathrm{EX}\{\mathrm{E}[(\psi_1\vee MAYBE)\cup(\psi_2\vee YES)]\}$，

那么 $Y_{i0}(b)=j1$，否则 $Y_{i0}(b)=j0$；

(2) 对 K_{i1} 的超节点 b，如果 $Y_i(b)=j$ 且

$b\models_i\mathrm{EX}\{\mathrm{E}[(\psi_1\vee MAYBE)\cup(\psi_2\vee YES\vee(\psi_1\wedge out_i))]\}$，

那么 $Y_{i1}(b)=j1$，否则 $Y_{i1}(b)=j0$；

(3) 对 K_{i0} 的节点 u，如果 $u\models_i\mathrm{E}[(\psi_1\vee MAYBE)\cup(\psi_2\vee YES)]$，

那么 $\psi\in X_{i0}(u)$，否则 $X_{i0}(u)$ 不变；

(4) 对 K_{i1} 的节点 u，如果

$u\models_i\mathrm{E}[(\psi_1\vee MAYBE)\cup(\psi_2\vee YES\vee(\psi_1\wedge out_i))]$，

那么 $\psi\in X_{i1}(u)$，否则 $X_{i1}(u)$ 不变；

od

return $\{K_i\,|\,i\in FN\cdot\{0,1\}\}$.

图 4　$H:=CheckUntil(H,\psi_1,\psi_2)$，图中 $\psi=\mathrm{E}(\psi_1\cup\psi_2)$

```
YES := ∅; MAYBE := ∅; NO := ∅;
将 FN 依其序关系表示为图 G;
宽度优先为序搜索 G,得到函数名列表 list;
for i=last(list) downto first(list) do
  if in_i ⊨_i E[(χ∨MAYBE)∪(YES)] or
     in_i ⊨_i EG(χ∨MAYBE)
  then
    YES := YES∪{i}; IV := IV∪{⟨i,YES⟩};
  else
    if in_i ⊨_i E[(χ∨MAYBE)∪(χ∧out_i)] then
      MAYBE := MAYBE∪{i}; IV := IV∪{⟨i,MAYBE⟩};
    else NO := NO∪{i}; IV := IV∪{⟨i,NO⟩}; fi
  fi
od
```

```
foreach i∈FN do
  将 K_i 替换为两个状态机 K_i0 和 K_i1:
  K_i0 = ⟨N_i, B_i, in_i, out_i, X_i0, Y_i0, E_i⟩,
  K_i1 = ⟨N_i, B_i, in_i, out_i, X_i1, Y_i1, E_i⟩,
  其中 Y_i0, Y_i1, X_i0, X_i1 取值如下:
  (1) 对 K_i0 的超节点 b,如果 Y_i(b)=j 且
  b ⊨_i EX{E[(χ∨MAYBE)∪(YES)]∨EG(χ∨MAYBE)},
  那么 Y_i0(b)=j1,否则 Y_i0(b)=j0;
  (2) 对 K_i1 的超节点 b,如果 Y_i(b)=j 且
  b ⊨_i EX{E[(χ∨MAYBE)∪(YES∨(χ∧out_i))]∨
          EG(χ∨MAYBE)},
  那么 Y_i1(b)=j1,否则 Y_i1(b)=j0;
  (3) 对 K_i0 的节点 u,如果
  u ⊨_i EX{E[(χ∨MAYBE)∪(YES)]∨EG(χ∨MAYBE)},
  那么 ψ∈X_i0(u),否则 X_i0(u)不变;
  (4) 对 K_i1 的节点 u,如果
  u ⊨_i EX{E[(χ∨MAYBE)∪(YES∨(χ∧out_i))]∨
          EG(χ∨MAYBE)},
  那么 ψ∈X_i1(u),否则 X_i1(u)不变;
od
return {K_i | i∈FN·{0,1}}.
```

图 5 $H:=CheckAlways(H,\chi)$,图中 $\psi=\mathrm{EG}\chi$

显然,算法 2 返回的二元关系 IV 是一个函数:$FN\cdot\{\varepsilon,0,1\}^{|\varphi|}\rightarrow\{YES,MAYBE,NO\}$.设 $name\in FN\cdot\{\varepsilon,0,1\}^{|\varphi|}$,以下我们用 $IV(name)$ 表示 $name$ 在 IV 下的值.

定理 2. 对层次状态机 H 和 CTL 公式 φ,算法 2 是正确的,并且其时间复杂度是 $O(|H|\cdot 2^{|\varphi|})$,式中 $|H|$ 是层次状态机的大小,$|\varphi|$ 是 CTL 公式的长度.

证明. 容易看出图 3 的 *CheckNext* 算法与 Alur 的相应算法是等价的(本文做此修改的目的在于让算法 2 返回一个一致的 IV).因此算法 2 是正确的.另一方面,对图进行的宽度优先搜索并不会影响算法的时间复杂度,因此算法 2 的时间复杂度与 Alur 算法相同,仍然是 $O(|H|\cdot 2^{|\varphi|})$. 证毕.

3.2 AFM 程序的增量式验证算法

基于算法 2 收集的信息,现修改算法 1,在组合 AFM 程序和 AFM 模块的同时进行验证,如算法 3.

算法 3. AFM 程序与 AFM 模块的组合算法,组合的同时对属性进行增量式的保持性验证.

输入:(1) AFM 模块的 HSM $H_a=\{K_i^E \mid i\in AN\}$(其中 $AN\subset\Sigma^*$);

(2) AFM 程序 p 的 HSM $H_p=\{K_i \mid i\in FN\}$(其中 $FN\subset\Sigma^*$);

(3) 应用算法 2 验证 H_p 时返回的 IV 和 $sub(\varphi)$

输出:(1) 组合后程序 p' 的 HSM $H_{p'}$;

(2) 性质 φ 是否保持

```
1.  preserved := true;
2.  foreach K_i^E ∈ H_a do
3.    if (J_i = ∅) then
4.      以 K_i^E 的 ⟨N_i, B_i, in_i, out_i, X_i, Y_i, E_i⟩ 创建一个
        状态机 K_i,将其加入 H_p';
5.    else
6.      foreach K_l ∈ H_p do
7.        if l ∈ J_i then
8.          从 H_p 中删除 K_l;
9.          以 K_l 的 ⟨N_l, B_l, in_l, out_l, X_l, Y_l, E_l⟩ 创建一
            个状态机 K_l';
10.         以 K_i^E 的 ⟨N_i, B_i, in_i, out_i, X_i, Y_i[proceed/
            l'], E_i⟩ 创建一个状态机 K_l;
11.         将 K_l' 和 K_l 加入 H_p';
12.         if (preserved) preserved :=
                    Preservation(IV, sub(φ), K_l);
13.     od
14.   fi
15. od
16. 将 H_p 中剩余的 K 拷贝到 H_p' 中;
17. 清除 H_p' 中没有被 K_⊥ 直接或间接调用的状态机.
```

比较算法 1 可知,算法 3 增加了第 12 行以对每个 aspect 或 refinement 的每次应用进行即时的增量式保持验证,其中 $\mathrm{Preservation}(IV,sub(\varphi),K_l)$ 如算法 4.保持验证的结果为真表示 AFM 程序与 AFM 模块的组合未改变程序原有的性质 φ,为假则不能断定.另外,如算法第 10 行,我们称"H_p 的 K_l 在组合过程中被 $H_{p'}$ 的 K_l 替换了".

Preservation 算法的总体思路是:设 H_p 中某状态机 K_l 在 AFM 程序与 AFM 模块的合并过程中被替换为 K_a.如果 K_l 在 H_p(的验证过程)中所属的集合(即 YES,$MAYBE$ 或者 NO)始终是 K_a 在 $H_{p'}$(的验证过程)中所属的集合,则 K_l 的调用者 $Callers=\{K_i \mid i\in FN, i<l\}$ 将"感觉"不到 K_l 已被替换为 K_a.因此它们在 H_p 中的标记将同样是其在 $H_{p'}$ 中的标记,考虑到 $main\in Callers$,故若 H_p 满足性质 φ,则 $H_{p'}$ 也满足 φ.

算法 4. Preservation.

输入:IV,$sub(\varphi)$ 和 K_a

输出:φ 的保持验证是否成功

算法过程：

```
Let Q := {K_a}
foreach ψ∈sub(φ) do
  let N_a := ∪_{K_i∈Q} N_i
  case ψ:
    ψ∈P: skip;
    ψ=¬χ:
      foreach u∈N_a do if χ∉X(u) then
            X(u) := X(u)∪{ψ} fi od
    ψ=ψ1∧ψ2:
      foreach u∈N_a do
        if ψ1∈X(u) and ψ2∈X(u) then
          X(u) := X(u)∪{ψ} fi
        od
    ψ=EXχ: Q := CheckNextP(Q,χ);
    ψ=E(ψ1∪ψ2): Q := CheckUntilP(Q,ψ1,ψ2);
    ψ=EGχ: Q := CheckAlwaysP(Q,χ);
od
return true.
```

Preservation 算法所以要以 $sub(\varphi)$ 为输入是为了保证算法 4 中状态机的替换过程与验证 H_p 时一致. 算法 4 中 *CheckNextP*，*CheckUntilP*，*CheckAlwaysP* 分别见图 6～图 8，其中 $\{YES, MAYBE, NO\}$ 仍表示超节点上的标记，只是标记值来自 IV：设 $b\in B_i$，则 b 上的标记是 $IV(Y_i(b))$.

```
foreach K_i∈Q do
  if χ∈in_i and IV(i)=NO then
    return false
  fi
  if χ∉in_i and IV(i)=YES then
    return false
  fi
  将 K_i 替换为两个状态机 K_i0 和 K_i1：
    K_i0=⟨N_i,B_i,in_i,out_i,X_i0,Y'_i,E_i⟩,
    K_i1=⟨N_i,B_i,in_i,out_i,X_i1,Y'_i,E_i⟩,
    其中 Y'_i,X_i0,X_i1 同图 3 下的(1)～(3)；
od
return true.
```

图 6　*CheckNextP*(Q,χ)

```
foreach K_i∈Q do
  if in_i ⊨_i E[(ψ1∨MAYBE)∪(ψ2∨YES)] then
    if IV(i)≠YES then return false fi
  else
    if in_i ⊨_i E[(ψ1∨MAYBE)∪(ψ1∧out_i)] then
      if IV(i)≠MAYBE then return false fi
    else
      if IV(i)≠NO then return false fi
    fi
  fi
  将 K_i 替换为两个状态机 K_i0 和 K_i1：
  K_i0=⟨N_i,B_i,in_i,out_i,X_i0,Y_i0,E_i⟩,
  K_i1=⟨N_i,B_i,in_i,out_i,X_i1,Y_i1,E_i⟩,
  其中 Y_i0,Y_i1,X_i0,X_i1 同图 4 下的(1),(2),(3),(4)；
od
return true.
```

图 7　*CheckUntilP*(Q,ψ_1,ψ_2)

```
foreach K_i∈Q do
  if in_i ⊨_i E[(χ∨MAYBE)∪(YES)] or
     in_i ⊨_i EG(χ∨MAYBE)
  then
    if IV(i)≠YES then return false fi
  else
    if in_i ⊨_i E[(χ∨MAYBE)∪(χ∧out_i)] then
      if IV(i)≠MAYBE then return false fi
    else if IV(i)≠NO then return false fi
    fi
  fi
  将 K_i 替换为两个状态机 K_i0 和 K_i1：
  K_i0=⟨N_i,B_i,in_i,out_i,X_i0,Y_i0,E_i⟩,
  K_i1=⟨N_i,B_i,in_i,out_i,X_i1,Y_i1,E_i⟩,
  其中 Y_i0,Y_i1,X_i0,X_i1 同图 5 下的(1),(2),(3),(4)；
od
return true.
```

图 8　*CheckAlwaysP*(Q,χ)

定理 3.　设 AFM 程序 p 的 HSM $H_p=\{K_i \mid i\in FN\}(FN\subset\Sigma^*)$，AFM 模块的 HSM $H_a=\{K_i^E \mid i\in AN\}(AN\subset\Sigma^*)$，设 $JP=\{i \mid \exists k\in AN \text{ s. t. } i\in FN\cap J_k\}$，$Caller=\{K_i \mid i\in FN, \exists k\in JP, i<k\}$，$N=\bigcup_{Ki\in Caller} N_i$，如果按照算法 3 将 H_a 与 H_p 组合，得到 $H_{p'}$ 且验证结果为真，则对 $\forall s\in N$，如果 s 在 H_p 中标有公式 φ，则 s 在 $H_{p'}$ 中也标有公式 φ.

证明.　设 $s\in N$，且 s 在 H_p 中标有公式 φ，对公式 φ 的长度进行结构归纳：(1)若 φ 为原子命题，定理显然成立；(2)若 φ 形如 $\varphi_1\wedge\varphi_2$，则 s 在 H_p 中同时标有 φ_1 和 φ_2，由归纳假设，s 在 $H_{p'}$ 中也同时标有 φ_1 和 φ_2，故 s 在 $H_{p'}$ 中也标有 φ；(3)若 φ 形如 $\varphi_1\vee\varphi_2$，则 s 在 H_p 中标有 φ_1 或者 φ_2，由归纳假设，s 在 $H_{p'}$ 中也标有 φ_1 或者 φ_2，故 s 在 $H_{p'}$ 中也标有 φ；(4)若φ 形如 $\mathrm{EX}\varphi_1$，则 s 存在某后继满足 φ_1，若该后继是一个节点，则由归纳假设，该节点在 $H_{p'}$ 中也标有 φ_1，若该后继是一个超节点，则算法 3 保证其标记(即 YES，$MAYBE$，NO)在 $H_{p'}$ 中保持不变，故 s 在 $H_{p'}$ 中同样标有 φ；(5)若 φ 形如 $\mathrm{E}(\varphi_1\cup\varphi_2)$，则 H_p 中存在一条始于 s 的路径满足 $\mathrm{E}(\varphi_1\cup\varphi_2)$，对该路径上的节点，归纳假设保证其标记不变，对超节点，算法 3 保证其标记不变，故 s 在 $H_{p'}$ 中也标有 φ；(6)若φ 形如 $EG\varphi_1$，同理可证 s 在 $H_{p'}$ 中同样标有 φ.　　证毕.

推论 1.　设 AFM 程序 p 的 HSM $H_p=\{K_i \mid i\in FN\}(FN\subset\Sigma^*)$，AFM 模块的 HSM $H_a=\{K_i^E \mid i\in AN\}(AN\subset\Sigma^*)$，按照算法 3 将 H_a 与 H_p 组合，得到 $H_{p'}$；如果 H_p 满足性质 φ，则 $H_{p'}$ 也满足性质 φ.

证明.　H_p 满足性质 φ，即 $in_\perp$ 在 H_p 中满足 φ，显然 $in_\perp\in N$，故由定理 3，$in_\perp$ 在 $H_{p'}$ 中也满足 φ，即 $H_{p'}$ 满足 φ.　　证毕.

定理 4. 对 φ 和 K_a，算法 4 的时间复杂度是 $O(|K_a| \cdot 2^{|\varphi|})$.

证明. φ 最多有 $|\varphi|$ 个子式，CTL 公式的处理时间与模型的大小成正比，而每个子式都可能使模型的大小翻倍，因此算法 4 的时间复杂度是 $O(|K_a| \cdot 2^{|\varphi|})$. 证毕.

比较定理 2 和定理 4，由于 $|K_a| \ll |H|$，因此增量式验证的时间复杂度远远小于直接验证.

另外，定理 3 说明算法 3 的结果是验证 AFM 程序 p' 具有性质 φ 的充分条件. 但在某些特殊情况下，算法 3 的结果也可以是充要条件. 下面讨论这种特殊情况.

定义 6（可保持属性）. 如果属性 φ 仅由关于 AFM 程序中变量取值的命题构成，并且没有 EX 子公式，则称其为可保持属性.

定义 7（Observer 型 AFM 模块）. 设有 AFM 模块 $\{K_i^E \mid i \in AN\}$（其中 $AN \subset \Sigma^*$），如果对 $\forall i \in AN$，都有如下性质成立，则称该 AFM 模块为 Observer 型 AFM 模块：

(1) in_i 和 out_i 间的任意通路都经过且只经过一个 proceed 超节点；

(2) N_i 及 $\bigcup_{b \in Bi} N_{Yi(b)}$ 中的节点对应的 AFM 程序语句不修改程序中变量的取值.

定理 5. 设 AFM 程序 p 满足可保持属性 φ，则 p 与 Observer 型 AFM 模块的组合得到的程序 p' 一定保持性质 φ.

证明. 为方便说明，设 p 中的 K_f 被替换为 p' 中的 K_a. 只需证明对 EU，EG 子公式，K_f 和 K_a 一定具有相同的标记. 对 $\mathrm{E}(p \cup q)$，设 K_f 属于 YES 集合，说明 K_f 内存在一条以 in_f 为起点的路径 $v_1 v_2 \cdots v_k$ 满足：(1) $v_1 = in_f$；(2) v_k 如果是节点，则满足 q，如果是超节点，则标记有 YES；(3) 对 $v_i (1 \le i < k)$，如果 v_i 是节点，则满足 p，如果是超节点，则标记有 $MAYBE$. K_a 属于 Observer 型 AFM，因此 in_a 和 out_a 间的任意通路都包含 $v_1 v_2 \cdots v_k$ 这段路径. 另一方面，$\mathrm{E}(p \cup q)$ 是可保持属性，因此 p 和 q 都是关于 AFM 程序中变量取值的命题，K_a 属于 Observer 型 AFM，因此 in_a 和 in_f 之间的节点对应的程序语句不修改程序中变量的取值，所有 in_a 和 in_f（注意 $in_f = v_1$）之间的节点都标记有 p（否则 in_f 不可能标记有 p），超节点都标记有 $MAYBE$，故 K_a 仍应标记为 YES. 同理可证其他情况. 证毕.

关于 aspect 对属性的保持，文献[14]也有详细的论述，并且给出了与定理 5 相似的结论，但本文从层次状态机的角度给出了一种全新的解读，并且是针对 AFM 范式的.

4 总结及今后工作

总结全文，本文首先简要介绍了 AFM 编程范式，然后利用层次状态机为 AFM 范式建立了首个形式化模型，并提出了相应的 AFM 程序模型和 AFM 模块模型的组合算法（算法 1）和增量式验证算法（算法 3、4），该算法可以在组合 AFM 程序与 AFM 模块的同时验证 AFM 程序的原有性质是否保持. 相对于直接验证（即验证组合后得到的 AFM 程序）的时间复杂度 $O(|H| \cdot 2^{|\varphi|})$，该方法的时间复杂度为 $O(|K_a| \cdot 2^{|\varphi|})$，其中 $|H|$ 和 $|K_a|$ 分别为整个 AFM 程序的规模和 AFM 模块的规模，显然 $|K_a|$ 远小于 $|H|$，因此该方法可以有效避免大型程序由于规模太大而无法验证的问题.

今后的工作包括：(1) 本文只讨论了层次状态机模型下"程序在组合 AFM 模块后的性质保持"问题，而未涉及"AFM 模块在与程序组合后的性质保持"问题，后一问题同样亟待深入研究；(2) 本文目前只是从理论上指出了基于 HSM 的 AFM 程序的增量式验证的可行性，今后还须开展相应的实验并开发相关的工具.

参 考 文 献

[1] Clements P, Northrop L. Software Product Lines: Practices and Patterns. Addison-Wesley, 2001

[2] Apel S, Leich T, Saake G. Aspectual feature module. IEEE Transactions on Software Engineering, 2008, 34(2): 162-180

[3] Apel S, Leich T, Saake G. Aspectual mixin layers: Aspects and features in concert//Proceedings of the ICSE. Shanghai, China, 2006: 122-131

[4] Batory D. A tutorial on feature oriented programming and product-lines//Proceedings of the ICSE. Oregon, Portland, 2003: 753-754

[5] Batory D. A tutorial on feature oriented programming and the AHEAD tool suite//Proceedings of the Generative and Transformational Techniques in Software Engineering. Springer, 2006: 34

[6] Batory D. Scaling step-wise refinement. IEEE Transactions on Software Engineering, 2004, 30(6): 355-371

[7] Kiczales G, Lamping J, Mendhekar A, Maeda C. Aspect-oriented programming//Proceedings of the ECOOP. Jyvaskyla, Finland, 1997: 220-242

[8] Kupferman O, Vardi M. An automata-theoretic approach to modular model checking. ACM Transactions on Programming Languages and Systems(TOPLAS), 2000, 22(1): 87-128

[9] Jhala R, McMillan K L. Microarchitecture verification by compositional model checking//Proceedings of the CAV. Paris, France, 2001: 396-410

[10] Krishnamurthi S, Fisler K. Foundations of incremental aspect model-checking. ACM Transactions on Software Engineering and Methodology(TOSEM), 2007, 16(2): 36

[11] Fisler K, Krishnamurthi S. Modular verification of collaboration-based software designs//Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. Vienna, Austria, 2001: 152-163

[12] Alur R, Yannakakis M. Model checking of hierarchical state machines. ACM Transactions on Programming Languages and Systems(TOPLAS), 2001, 23(3): 273-303

[13] Loeckx J, Sieber K. The Foundations of Program Verification. 2nd Edition. Wiley, 1987

[14] Djoko S, Douence R, Fradet P. Aspects preserving properties//Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. San Francisco, California, USA, 2008: 135-145

YE Jun, born in 1982, Ph. D. candidate. His research interests include formal verification of AOP/FOP, hardware and software co-verification.

TAN Qing-Ping, born in 1965, Ph. D., professor. His research interests include distributed software engineering, formal methods, adaptive software engineering and so on.

LI Tun, born in 1974, Ph. D., associate professor. His research interests include design verification of micro-processor, electronic CAD.

XU Jian-Jun, born in 1980, Ph. D. candidate. His research interests focus on program analysis.

Background

Aspectual Feature Module (AFM) is a newly-proposed Software Product Line (SPL) programming paradigm. It stems from Feature-Oriented Programming (FOP for short) but introduces Aspect-Oriented Programming (AOP for short) constructs to solve its problems like tedious expression of homogeneous crosscut and no ability to express complex dynamic crosscut. But AFM's inherent hybrid of parallel composition and sequential composition of its modules prohibits the application of conventional verification technology, e.g. compositional verification and modular model checking. Krishnamurthi studied the formal verification of both AOP and FOP, but his works are not quit fit for AFM since it's prior to its proposal. Especially, he uses State Machine as the formal model of program, which will bring a large-scale model when verifying a data flow related property and may cause the failure of the verification.

In this paper, the authors firstly establish a formal model for AFM paradigm. They use Hierarchical State Machines (HSM for short) to be the model of both base program and AFM module. It will bring a model with smaller scale than the State Machine model. Based on this model, the authors propose an incremental verification method. This method starts from the verifications of a small-scale base program and continues with the verifications of merely the AFM module at exactly the same time when it is composed by a base program. This method can avoid the state space exploration problem caused by the direct verification of a large-scale AFM program.

This research is supported in part by National Natural Science Foundation of China under grant No. 60773025, which focuses on SOC system-level functional verification. Their previous work on this subject includes coverage-driven functional verification and compositional verification. But considering the novelty of AFM paradigm and the correspondence of AFM module with system function, we believe that AFM can start a new direction in functional verification. The work in this paper is just a commence, the authors study the preservation of properties on base program when composing with other AFM module, further will study the preservation of properties on AFM modules when composed by a base program. With solution to both problems, the cost of functional verification will be reduced significantly.

This research is also supported in part by Program for Changjiang scholars and innovative research team in university.