特征模型融合研究

引言

特征模型是组织和复用特定领域的软件制品的有效手段。特征是可被用户感知的、有价值的软件特点。特征模型对领域内的多个相似产品（又称产品家族）进行抽象，提取产品家族中的共性和变化性特征，并且建立特征之间的依赖关系。在构建了特征模型之后，可以通过选择一个不违反依赖关系的特征子集来实现软件制品的复用。总之，特征模型为复用相似产品来构建新的产品提供了一个解决方案。

随着特征模型应用的不断普及，在同一个领域内常存在多个相似的特征模型。这种相似特征模型的现象之所以产生，主要有如下几个原因：

1、不同建模者（团体或个人）在不同时期对同一个领域进行特征建模

2、（补充文献中的场景）

一方面，特征模型的建立需要大量领域知识；另一方面，对很多领域而言，又已经存在了多个特征模型。因此，这就促使我们去思考这样一个问题：能否复用相似特征模型来构建新的特征模型？ 对于这个问题，一个解决方案是特征模型融合，即：将一组相似的特征模型进行合并，生成一个新的特征模型。

本文对特征模型融合的研究现状进行了调研。现有的特征模型融合算法主要分为两大类。

本文的章节组织如下：第二章详细介绍特征模型融合算法的分类标准，第三章分类介绍典型的特征模型融合算法，第四章对典型算法进行了比较和分析，第五章提出了特征模型融合算法的研究展望，第六章对全文进行了总结。

概念框架（新）：

输入模型

根特征一致

父特征一致

关系：精化、约束

输出模型

产品保持

规模保持

精化保持

约束保持

融合

算法：组合、规则、逻辑公式

实现难度

时间复杂度

特征模型融合算子

特征模型

特征模型是对特定领域内一个产品家族的共性和变化性的抽象。图1是一个音频播放软件领域的特征模型示例。特征模型包含一组特征和它们之间的关系。特征是用户可感知的、有价值的软件特点，例如*解码器*、*运行平台*等。特征之间的关系主要由两类：精化关系和约束关系。精化关系将特征组织为一个层次结构，主要包含以下几类：

* 必选特征：其语义是当一个必选特征的父特征出现在一个产品中时，该必选特征也必须出现在同一个产品中。例如，*音频播放软件*必须包含*解码器*、*运行平台*两个特征。
* 可选特征：一个可选特征可以不与其父特征出现在同一个产品中。
* 异或（xor）组：当一个异或组的父特征出现在一个产品中时（例如图1中的*运行平台*），该组特征中有且仅有一个特征出现在同一个产品中（例如图1中的*手机*和*桌面*）。
* 或（or）组：当一个或组的父特征出现在一个产品中时，该组特征中可以有一到多个特征出现在同一产品中。

约束关系表达了特征之间额外的依赖。基本的约束关系主要有两种：

* requires关系：如果特征X requires 特征Y，其语义是当X出现在一个产品中时，Y必须出现在同一个产品中。
* excludes关系：互为excludes关系的两个特征不能出现在同一个产品中。

F:\Doc\paper\jos\Figures\1 - fm.tif

图1

给定一个领域特征模型，我们可以定制出一系列特定的产品，其具体做法是选择一个不违反精化和约束关系的特征子集。在本文余下的文字中，我们用特征的集合表示一个产品。例如，{音频播放软件，解码器，WAV，音频CD播放，运行平台，桌面}是图1的一个合法产品，而形如{…, 运行平台，手机，音频CD播放, …}的产品则不是图1的合法产品，因为它违反了模型中的excludes约束。

特征模型融合算子的定义

根据前面的讨论，我们看到特征模型在其构建阶段是抽象一个产品集合，而在其复用阶段又是定制得到该集合中的特定产品。因此，我们基于特征模型代表的产品集合来给出特征模型融合算子的定义。

给定特征模型FM，用符号[[FM]]表示其对应的产品集合。我们可以定义两种特征模型融合算子如下：

交算子

并算子

，当等号满足时，我们又称其为严格并算子。

严格并算子

特征模型融合算子的使用场景（可与上一节合并）

交算子（列相关工作，下同）

并算子

特征模型融合算子的实现及比较

在这一章节中，我们将首先提出融合算子实现的一系列性质，然后我们根据这些性质来对典型的实现进行比较。

比较框架

我们主要基于以下三个方面的性质对融合算子的实现进行比较。

融合结果的质量（树结构保持，规模保持，约束保持）

首先，一个融合算子的实现必须能够保持该算子所要求的语义（如2.2节所定义）。因此，*语义保持*这一性质是融合结果最基本的质量属性。但是满足语义保持性质的实现可能有多个，且它们所给出的结果特征模型是不同的。例如，图2给出了交算子两种实现的输出结果，其中之一来自【文献-generic semantics of feature diagram】，该输出结果中引入了三个“虚特征”（R、A’、B’）和一个额外约束（B mutual-requires B’）。当从输出模型中定制产品时，这些虚特征将被移除，由此我们可以验证：

,



图2

因此c和d均满足语义保持性质。但是从特征模型本身的质量上来看，c和d是有明显差别的。由于“特征模型质量”一词尚无精确定义，在本文中我们把它具体化为以下两个方面，即*模型规模*和*精化关系*。

*模型规模*指的是特征模型所含特征的数量。一般来说，特征模型规模越大，就越不易被人所理解。对特征模型融合而言，并算子的理想输出模型应恰好包含输入模型中所有不同特征，而交算子的理想输出模型应恰好包含输入模型中的所有共同特征。图2中的c情形属于理想情况，而d情形则引入了三个多余特征，显然增加了模型的复杂度。

*精化关系*体现的是特征模型的层次结构。对特征模型融合而言，输出模型的层次结构应该尽量与输入模型保持一致，否则会增加理解的困难。图2的d情形说明了这一点，额外引入的根特征和虚特征子树都增加了复杂性，难以一目了然的看出原特征（A和B）之间的关系。

对输入模型的限制

当前研究中，特征模型融合算子的实现对输入模型都具有一定的限制，总的来说主要有*父特征一致性*的限制和*约束关系*的限制。

给定两个特征模型，如果对于任一个它们共有的特征X，X在两个模型中的父特征也相同，那么我们称这两个特征模型具有父特征一致性【文献-SPLC Workshop】。在比较融合算子实现时，我们关注当输入模型不满足父特征一致性时，该实现能否进行处理。由于在实际应用中，输入模型很难完全满足父特征一致性，因此若一个实现无法处理这种不一致性，其应用场景必大受限制。

*约束关系*是特征模型的重要组成部分，因为只有约束关系才能保证定制出的产品都是合理的。在比较融合算子实现时，我们关注该实现能否正确的处理输入模型中的约束关系。

实现难度和复杂度

最后，我们比较不同实现所需的技术难度，以及该实现的时间复杂度。在图2的示例中，d实现的技术难度几乎可忽略不计，且融合可以在线性时间内完成。

基于简单组合的实现

算法介绍

基于简单组合的实现的基本思想是引入一些“虚特征”和“虚关系”将输入模型简单的拼接在一起，从而得到结果模型。【文献-generic semantics】和【文献-Supplier Independent】属于此类，前者仅从理论上探讨了这一实现的可能性，而后者则给出了这一实现的实际应用场景。在【文献-supplier】中，同一领域（在文献中是“汽车集成电路领域”）有多个供应商提供定制产品的服务，每个供应商发布一个特征模型用来描述自己可提供的产品集合。某一下游生产商希望将所有供应商的特征模型进行融合，从而得到一个能够描述所有供应商的所有产品集合的“全局特征模型”。然而在文献所述场景中，从全局特征模型中定制出来的一个产品必须能够追踪到一个特定供应商的特征模型中，因为只有供应商才具备实际生产能力。因此全局特征模型的构造是通过简单拼接输入特征模型来完成的。

下面首先介绍【文献-generic semantics】对交算子和并算子的实现，该实现本质上是对“交算子和并算子可以在线性时间内运算”这一命题的证明。实现的描述详见算法1。

算法1

|  |
| --- |
| Merge\_Intersection (x: FeatureModel, y: FeatureModel): FeatureModel {  // 把x和y的相同叶子特征改名，以便最后添加额外的mutual-requires约束  S ← { };  for each (Feature f ) {  rename y.f as y.f’;  S ← S{f};  }  R ← new Feature(); // 新建一个“虚根特征”R  // 并且把x和y作为R的必选（Mandatory）子特征  R.addMandatoryChild(x.root);  R.addMandatoryChild(y.root);  FM ← new FeatureModel(R); // 以R为根新建特征模型  // 为S中的每个特征及其对应的改名后的特征添加mutual-requires约束  for each (Feature f S) {  FM.addConstraint(f mutual-requires f’);  }  return FM;  }  Merge\_Union (x: FeatureModel，y: FeatureModel): FeatureModel {  R ← new Feature(); // 新建一个“虚根特征”R  R.childrenGroup ← new XORGroup(); // R的子特征之间是异或(XOR)关系  // x, y成为R的子特征  R.childrenGroup.add(x.root);  R.childrenGroup.add(y.root);  return new FeatureModel(R); // 返回以R为根的特征模型  } |

图3给出了算法1运行的一个例子。需要注意的是，算法中引入的虚特征（包括虚根特征R以及交算子实现中的重命名特征）将从融合特征模型的产品中移除，从而得以保持算子的语义。

【文献-supplier】对算法1的并算子实现做了扩充，其详细描述如算法2所示。主要扩充点如下：

* 可以对2个以上的特征模型进行一次性融合；
* 额外建立了一个只含必选、可选关系的特征模型SIFM来方便用户浏览输入特征全集；
* 额外建立了一个简单特征模型来显式表达只能从多个供应商中选其一的约束。

算法2

|  |
| --- |
| Merge\_Union (m1, m2, …, mk: FeatureModel): FeatureModel {  //1、建立Supplier Independent FM (SIFM)  SIFM ← new FeatureModel();  for each (Feature f ) {  if (f在m1, m2, …, mk中的父特征均为p) {  把父子特征p, f添加到SIFM中;  } else {  SIFM.root.addChild(f);  }  if (f在m1, m2, …, mk中均为必选特征) {   f为必选特征;  } else {  f为可选特征;  }  }  //2、建立供应商特征模型（SFM）  R← new Feature();  R.childrenGroup ← new XORGroup(); // R的子特征之间为异或（XOR）关系  for each Supplier S {  R.childrenGroup.add(new Feature(S)); // 创建以供应商命名的“特征”  }  SFM ← new FeatureModel(R);  //3、添加SIFM、SFM和输入模型之间的追踪关系  Constraints ←{ };  for each (Feature f SIFM) {  Constraints.add( f requires XOR{t | });  }  for each (Feature f ) {  Constraints.add( f requires SIFM.f );  }  for () {  Constraints.add( Supplieri mutual-requires mi.root );  }  finalRoot ← new Feature();  finalRoot.addMandatoryChild ( SIFM.root );  finalRoot.addMandatoryChild ( SFM.root);  result ← new FeatureModel(finalRoot);  result.addConstraints (Constraints);  return result;  } |

性质

语义保持：容易证明

模型规模、精化关系：不佳（即可读性差）

对输入模型：无限制

实现难度：低

时间复杂度：与输入特征总数成线性

额外性质：如果一个相同特征又映射了不同的制品，那么该映射可以被保留

基于规则的实现

算法介绍

基于规则的实现【文献-automated graph】、【文献-composing fm】和【文献-splc workshop】的基本思想是定义一系列规则来处理输入模型中相匹配的部分。这些规则可以分为两类，分别处理精化关系和约束关系。因此各文献中的算法也均可分为两大步骤，即首先匹配并输出精化关系，得到特征树，然后匹配并添加约束关系。各文献的区别在于具体规则的不同，因此下面仅以【文献-composing fm】为例介绍算法的基本框架，如算法3所示。

算法3

|  |
| --- |
| merge\_tree\_intersection (root1: Feature, root2: Feature): Feature {  if (root1 does not match root2)  return null;  R ← root1.copy();  // 根据规则计算R及其子特征之间的精化关系，具体规则见表1  relation ← compute\_by\_intersection\_rules (root1, root2);  // 递归的融合root1和root2的每一对相同子特征  for each (Feature c ) {  merged\_child ← merge\_tree\_intersection (root1.c, root2.c);  R.add\_child (merged\_child, relation);  }  return R;  }  merge\_tree\_union (root1: Feature, root2: Feature): Feature {  if (root1 does not match root2)  return null;  root ← root1.copy();  // 规则见表2  relation ← compute\_by\_union\_rules (root1, root2);  common ← ;  for each (Feature c common) {  merged\_child ← merge\_tree\_intersection (root1.c, root2.c);  root.add\_child (merged\_child, relation);  }  // 与Intersection不同的是，还需要添加非公共的特征  for each (Feature u1) {  root.add\_child (u1, OPTIONAL);  }  for each (Feature u2) {  root.add\_child (u2, OPTIONAL);  }  return root;  }  merge\_union (x: FeatureModel, y: FeatureModel): FeatureModel {  // 1、融合精化关系  root ← merge\_tree\_union (x.root, y.root);  // 2、根据规则融合约束关系，具体规则见表3  constraints ← compute\_constraints\_by\_union\_rules (x.constraints, y.constraints);  result ← new FeatureModel (root);  result.add\_constraints (constraints);  return result;  }  // merge\_intersection类似，在此略去 |

算法3中引用的具体规则如表1、表2和表3所示。我们通过示例来说明该算法是如何通过规则来实现融合算子的语义的。图3(a)中，（以下介绍交和并算子对精化关系和约束关系进行处理的规则示例）

性质

语义保持：不易证明（部分不等于整体）

模型规模、精化关系：佳

对输入模型：有诸多限制（根匹配、父特征匹配）

实现难度：中等

时间复杂度：（待证）

基于逻辑公式的实现

算法介绍

基于逻辑公式的实现【文献-managing multi spl】的基本思想包含三个步骤：首先将输入特征模型转换为逻辑公式，然后将融合算子表达为与输入逻辑公式有关的一个融合逻辑公式，最后将融合逻辑公式反向转换为特征模型，即得到融合后的特征模型。在这个过程中，第一步和第三步都有专门的研究者提出了较为成熟的实用算法，因此该实现的核心工作在于定义融合逻辑公式。下面我们分别就这三个步骤进行介绍。

特征模型转换为逻辑公式

【文献-fm，logic】提出了一种将特征模型转换为逻辑公式的方法。在该方法中，特征模型中每个特征对应逻辑公式的一个变量，而每个精化或约束关系对应一个或多个逻辑蕴含式（如表4所示），最终整个特征模型表示为所有逻辑蕴含式的合取。图4给出了一个简单的示例。

表4：关系与逻辑蕴含式对应一览

|  |  |
| --- | --- |
| 关系 | 逻辑蕴含式 |
| 精化关系 | |
| 特征C是特征F的必选子特征 |  |
| 特征C是特征F的可选子特征 |  |
| 特征F有一组互为OR关系的子特征C1, C2, …, Cn |  |
| 特征F有一组互为XOR关系的子特征C1, C2, …, Cn | 以及 |
| 约束关系 | |
| X requires Y |  |
| X excludes Y |  |

将特征模型表示为逻辑公式以后，该特征模型的一个合法产品即满足该逻辑公式的一组变量赋值，其中变量赋值为true意味着对应特征在产品中出现，赋值为false则相反。

定义融合逻辑公式

根据【文献-managing multi spl】，设两个输入模型1和2对应的逻辑公式分别为，且对应的特征集合分别表示为，那么1和2对应的产品集合：

从而直接根据交算子和并算子的定义就可以得到融合以后的特征模型对应的逻辑公式，如下所示：

逻辑公式转换为特征模型

【文献-fm，logic，back】提出了一种根据逻辑公式构造特征模型的算法，其主要框架如算法4所示。根据前述讨论，特征模型可以转换为一系列逻辑蕴含式的合取，因此算法4的基本思想是先构造一个逻辑蕴含图（有向图），然后逐步恢复精化关系，最终把逻辑蕴含图变为一个有向森林，该森林对应了特征模型的层次结构。算法4的主要缺点在于无法区分精化关系和约束关系，也无法区分相互蕴含的一组特征，因此最终得到的特征模型还需要一定的人工重构。图5的示例摘自【文献-fm，logic，back】，5(a)中的特征模型首先被转换为逻辑公式，然后再用算法4重新转换为特征模型得到5(b)，可以看到二者的差别，尤其是5(a)中的约束关系在5(b)中被当作精化关系，严重影响了特征模型的可理解性。

算法4

|  |
| --- |
| formula\_to\_feature\_model (: Formula): FeatureModel {  // 检查公式可满足性  if (not SAT()) return null;  F ← 的变量集合（即特征集合）;  // 找出“死特征”(dead feature)，即不可能出现在任一产品中的特征  D ← {};  // 构造逻辑蕴含图（Implication Graph），该图为一有向图（可能有环）  V ← ;  E ← {(u, v)};  G ← (V, E)  // G中的一个强连通分支包含的特征为一原子集（即这些特征要么全部出现在一个产品中，要么全部不出现在一个产品中）  // 抽取原子集  for each (Strongly Connected Component scc G) {  // 将scc收缩为一个顶点  n ← new Node();  *V* ← ;  *E* ←  ;  }  // 至此G为一有向无环图  // 抽取OR和XOR精化关系  for each () {  // 查找f的最小OR子特征组，参见【文献-prime implication】  k ←;  if (SAT() {  n ← new Node();  V ← V ;  E ← E ;  if ( {  n为XOR组;  }  }  }  // 抽取可选子特征关系  Compute transitive reduction of G; // 参见【文献-transitive reduction】  // 至此G成为森林  for each ((u, v)) {  u是v的可选子特征;  }  return new FeatureModel(G); //返回由森林G代表的特征模型  } |

图5（摘自【文献-fm logics back】

性质

语义保持：可证明

模型规模：佳（恰好一个变量一个特征）

精化关系： 不佳

对输入模型：无限制

实现难度：困难

时间复杂度：（待证）

小结

（父特征不一致的解决必须靠人来进行）各算法是如何处理处于不同层次的相同特征的？

研究展望

相同特征的判定

叉积算子

总结