软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(7):1436−1454 [doi: 10.3724/SP.J.1001.2013.04333]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

语义可配置的模型转换*

何 啸[1,2,4], 麻志毅[3,4], 王瑞超[3,4], 邵维忠[3,4]

[1](北京科技大学 计算机与通信工程学院,北京 100083)

[2](材料领域知识工程北京市重点实验室,北京 100083)

[3](北京大学 信息科学技术学院 软件研究所,北京 100871)

[4](高可信软件技术教育部重点实验室(北京大学),北京 100871)

通讯作者: 麻志毅, E-mail: mzy@sei.pku.edu.cn

摘 要: 模型转换是模型驱动体系结构的核心技术之一.在一个复杂的模型驱动的开发过程中,可能同时使用多种转换语言及相应的工具实现转换程序.这一方面增加了开发人员的学习负担,也会导致各种兼容性问题的出现.提出一种语义可配置的模型转换技术,通过重新定义转换语言的语义,允许开发人员使用一种转换语言解决不同的转换问题.首先,总结出一组常见的转换原语;然后,利用一种基于 OCL 的脚本语言 TSS 来描述转换语言的语义;最后,对该方法的完全性、表达能力和复杂度进行了讨论,并通过一组案例对该方法进行了验证.

关键词: 模型驱动体系结构;模型转换;转换语言;执行语义

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 何啸,麻志毅,王瑞超,邵维忠.语义可配置的模型转换.软件学报,2013,24(7):1436−1454. http://www.jos.org.cn/1000-9825/4333.htm

英文引用格式: He X, Ma ZY, Wang RC, SHAO WZ. Semantics-Configurable model transformation. Ruan Jian Xue Bao/Journal of Software, 2013,24(7):1436−1454 (in Chinese). http://www.jos.org.cn/1000-9825/4333.htm

Semantics-Configurable Model Transformation

HE Xiao[1,2,4], MA Zhi-Yi[3,4], WANG Rui-Chao[3,4], SHAO Wei-Zhong[3,4]

[1](School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China)

[2](Beijing Key Laboratory of Knowledge Engineering for Materials Science, Beijing 100083, China)

[3](Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[4](Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871, China)

Corresponding author: MA Zhi-Yi, E-mail: mzy@sei.pku.edu.cn

Abstract: Model transformation is a vital technique of MDA. In a complex model-driven development process, it is most likely capapble of employing multiple transformation languages, along with their corresponding tools, to develop a set of model transformations. This increases the learning costs, and also leads to some compatibility problems. The paper proposes a technique of semantics-configurable model transformation, which enables developers to solve different problems using one transformation language, by redefining the semantics of the language. First, a set of common primitive actions are proposed. Then, TSS, an OCL-based scripting language, is employed to specify the logic of a primitive action. Finally, the paper discusses the completeness, expressiveness, and complexity of this approach, and evaluates the approach with some case studies.

Key words: model-driven architecture; model transformation; transformation language; executable semantics

* 基金项目: 国家自然科学基金(61272159); 国家重点基础研究发展计划(973)(2011CB302604); 国家科技支撑计划(2011BAK08B04); 北京市自然科学基金(4122036); 材料领域知识工程北京市重点实验室 2012 年度阶梯计划(Z121101002812005)

收稿时间: 2011-11-15; 修改时间: 2012-06-29; 定稿时间: 2012-10-10

模型驱动体系结构(model driven architecture,简称 MDA)是一种以模型为核心的软件开发和维护手段[1].模型转换技术是 MDA 的核心技术,随着模型转换技术的发展,它已经被运用于多种开发场景,例如:将抽象层次较高的平台无关模型(platform independent model,简称 PIM)转换成抽象层次较低的平台相关模型(platform specific model,简称 PSM);将质量相对较低的设计模型重构为质量相对较高的模型[2];实现不同模型之间的双向转换和同步操作[3].

为了实现一个模型转换,主流的方法是利用专门的转换描述语言编写一个转换程序,再将其部署在相应的执行引擎上运行.每一种转换语言都有各自的执行语义,按照这个语义对转换程序进行解释,就可以完成各种转换操作.不同的转换语言有不同的语义,且不互相等价.因此,转换语言的描述能力并不相同.一般而言,每种转换语言只能解决特定类型的问题,且转换语言 *A* 所能实现的转换程序可能无法用转换语言 *B* 实现.

目前存在多种转换描述语言,用于解决不同开发场景中的转换问题.而在一个相对复杂的模型驱动的开发过程中,可能存在多个需要运用模型转换的场景.为此,开发人员常常需要使用不同的转换描述语言编写转换程序.但这样做会导致以下两个问题:

- 学习负担的增加.开发人员需要掌握不同的转换描述语言以及相应的执行引擎,但是不同的转换描述语言在语法、语义等方面存在较大的差异性[4],不同的工具在使用和执行方式上也各不相同.这就增加了学习不同语言及其相应执行引擎的难度和复杂度,不利于模型驱动方法的实施;
- 执行引擎的兼容性问题.因为不同的转换语言需要配合不同的执行引擎,如果使用不同的转换语言,就必须同时使用不同的执行引擎.然而,目前的转换引擎在存储格式等方面不完全相同[4],工具之间存在一定的兼容性问题,这可能导致额外的使用开销.例如:ATL Engine 是 ATL(ATLAS transformation language)[5]的执行引擎,它能够读取 ECore[6]格式的模型;AGG(attributed graph grammar)[7]的执行引擎[8]则只支持自定义的文件格式.这就是说,不同工具所产生的模型有可能不被其他工具读取.因此,开发人员还需要编写相应的格式转换程序才能实现不同工具的互通.

一种简单的解决方案是直接运用通用的编程语言(如 Java,而非转换描述语言)实现转换程序——开发人员不需要学习新的语言和工具,也能够控制和避免各种兼容性问题.但这种做法降低了转换程序的抽象层次,增加了编写转换程序的复杂度.此外,编程语言是一种实现技术,而 MDA 中的几种关键技术(如 MOF[9]和 OCL[10])则主要面向模型(也包括元模型和元元模型).因此,编程语言和这些技术不能做到无缝结合.

对此,本文提出一种语义可配置的模型转换技术,通过对某种转换语言的语义进行重新定义,开发人员便可以利用同一种语言及其对应的执行引擎实现不同类型的转换程序,从而降低学习负担,并可避免由于使用不同执行引擎所造成的兼容性问题,同时也能最大程度地保留这种转换语言所具有的优点.本文首先对转换语言的语义进行分层,并提炼出一种 3 层结构,其中的每个层次都从某个角度刻画了一部分语义,而 3 个层次的相互组合,便构成了完整的转换语义;接着,本文针对每个层次进行分解,从中提取出一组可扩展的转换原语;利用这组转换原语,开发人员可以重新配置和定义这个层次所表示的转换语义.

1 问题分析

假设 *L* 表示一种模型转换描述语言,*L* 可以用来编写模型转换程序的实现代码.本文假设符号 G_L 表示所有的用语言 *L* 实现的模型转换程序的集合,即 G_L 中的任意一个元素 *p* 表示一个模型转换程序,且 *p* 是用语言 *L* 所描述的.

对于 G_L 中的一个模型转换程序 *p*,它实现了从 *A* 类型模型到 *B* 类型模型的转换操作.抽象地看,*p* 可以看做是一个从 *A* 类型到 *B* 类型的对应关系 $=_p$,即 *p* 可以被表示为

$$A=_pB.$$

由于现有的模型转换语言大部分是声明式和混合式语言(如 QVT-R,AGG,TGG,ATL 等),这些语言描述了转换成功的条件,但可能没有描述具体的转换算法.因此,模型转换程序 *p* 往往不能像一般的程序那样直接执行.为了执行一个用语言 *L* 编写的模型转换程序 *p*,需要将其部署在语言 *L* 的执行引擎上,按照 *L* 的执行语义解释 *p*

中所包含的从类型 A 到类型 B 的转换规则和逻辑,这样便可以将一个符合类型 A 的输入模型 a 转换成一个符合类型 B 的输出模型 b.

语言 L 的执行引擎实现了 L 执行语义(或 L 的某种语义),它赋予 p 中的转换规则和表达式可执行的语义,从而实现需要的转换操作.本文将 L 的执行引擎抽象为一个解释函数 I_L,而 L 的执行语义可以被抽象为 I_L 的实现算法,其中,I_L 可以定义为

$$I_L:G_L\times\Pi\rightarrow Boolean\times\Pi,$$

其中,Π都表示模型空间(model space)[9].在当前语境下,Π可以看做是所有可能的输入模型和输出模型所构成的集合.I_L 描述了如何根据某个转换程序中定义的规则实现某种转换操作,其参数为某个转换程序 p 和转换之前的模型,其返回值表示转换操作是否成功(布尔值)及转换之后的模型.

目前,每种转换语言都有各自的执行语义及相应的执行引擎,即每种语言 L 都有自己的解释函数 I_L,且这些函数各不相同.不难看出,用某种语言 L 编写的转换程序所能够实现的功能直接依赖于 L 的解释函数 I_L——如果 I_L 的算法不符合某种转换操作的需要,L 就不能很好地实现这种操作.此时就需要选择其他语言,从而导致之前所述的各种问题的发生.

本文提出一种语义可配置的模型转换运行机制,本质上可以看做是为模型转换定义了一种新的解释函数 U_L,这种解释函数的定义如下所示:

$$U_L:S_L\times G_L\times\Pi\rightarrow Boolean\times\Pi,$$

其中,G_L,Π的含义和 I_L 定义中的含义相同,S_L 则表示语言 L 所能够支持的所有转换语义的集合.需要说明的是,语言 L 的规范中所定义的执行语义可能只是 S_L 的真子集.例如,在 QVT-R 的语言规范中定义了两种执行语义“只检查(check-only)语义”和“强制(enforcement)语义”,它们都属于 $S_{\text{QVT-R}}$,但不是 $S_{\text{QVT-R}}$ 的全部元素.

不难看出,在解释函数 U_L 中,执行语义变成了输入参数.如前所述,现有模型转换的解释函数 I_L 实现了语言 L 的某个特定语义,如果假设这个执行语义是 s,那么一定有 $s\in S_L$.从而可以得出,U_L 和 I_L 的关系是

$$U_L(s,p,\pi)=I_L(p,\pi),$$

其中,$s\in S_L$,$p\in G_L$,$\pi\in\Pi$.之所以称 U_L 是语义可配置的,是因为在 U_L 中,执行语义是一种可变化的成分,即输入参数:通过修改这个参数的取值,使得解释函数 U_L 能够实现不同类型的转换操作,进而增强语言 L 的能力.

假设 $s\in S_L$,由于 s 是 U_L 的输入参数,因此,为了实现 U_L,必须确定 L 的执行语义的基本结构,即 s 的结构是什么?它包含哪些基本成分,即 s 由什么组成?以及在不同的执行语义中存在哪些共性和变化性——即 s 的成分中哪些部分是通用的,哪些部分是可变化、可修改的.此外,由于本文的方法需要对语言 L 的执行语义进行规约,并作为输入参数传给解释函数 U_L,因此,本文还需要定义一种结构化的转换语义的描述手段.

本文将分别讨论上述几个问题,并给出解决方案.首先,第 2 节定义一种模型转换的抽象结构,分别用来刻画模型转换程序的静态和运行时结构,这样可使本文提出的方法不依赖于任何具体的模型转换语言,从而提升了本文方法的通用性.第 3 节分析模型转换程序执行时的基本过程,并提出一种模型转换语义的基本框架.在这个执行语义的基本框架下,第 4 节进一步讨论其中包含的基本成分以及其中存在的共性和变化性,并利用转换原语的概念封装这些基本语义成分.第 5 节在 OCL 的基础上进行扩展,结合转换原语的概念,提出一种模型转换执行语义的描述语言.在给出解决方案后,第 6 节介绍本文工作的工具原形实现.第 7 节通过一组案例对本文方法进行验证.第 8 节讨论本文方法的完备性、描述能力、使用复杂度等方面的内容.第 9 节将本文与相关工作进行比较.第 10 节总结本文工作.

2 模型转换的抽象结构

为了方便讨论,也为了提升本文方法的通用性,本节定义一种模型转换的抽象结构,其中包括一个抽象静态结构和一个抽象运行时结构.本文余下内容都会基于这个抽象结构进行讨论,从而使得本文方法能够独立于具体的模型转换语言.

2.1 抽象静态结构

不同转换语言的语法存在较大差异,但转换程序的基本结构却大致相同.本文使用类图的形式来表示转换程序的抽象静态结构,如图 1 所示.类 Transformation 表示转换程序.每个 Transformation 拥有至少一个形式参数;转换程序的形式参数用 TypedModel 表示.每个 TypedModel 有一个类型,用来说明这个转换程序可接受哪些模型作为输入,或可以产生哪些输出.参数的类型,即元模型,用 MetaModel 表示.一个 MetaModel 由很多 DataType(即数据类型)组成[11].

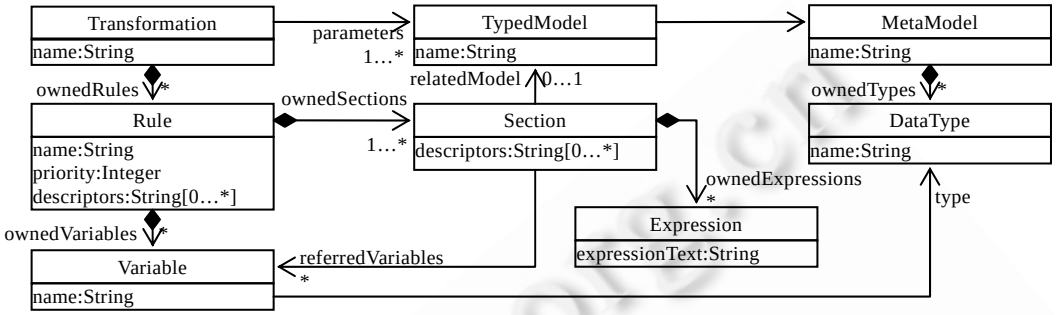
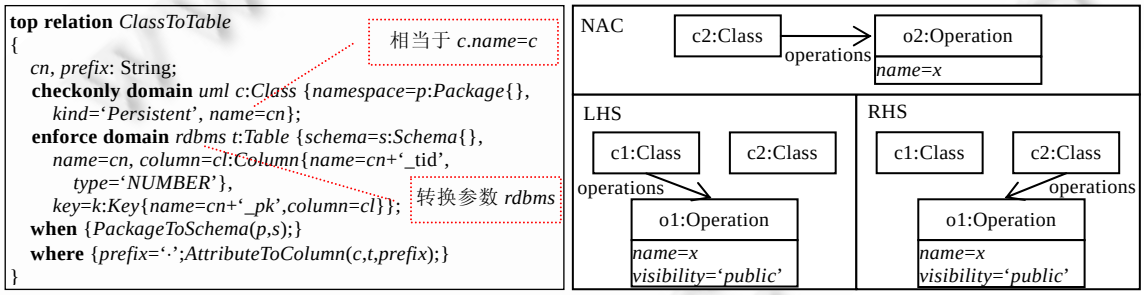


Fig.1 Basic structure of transformation specification

图 1 转换程序的抽象结构

每个 Transformation 都包含一组 Rule,即转换规则.转换规则是转换程序的最重要的组成部分,图 2 展示了两条转换规则的示例:图 2(a)是一条用 QVT-R(query/view/transformation relations)[12]编写的转换规则,它描述了如何将 UML(unified modeling language)[13]模型中的 Class 元素转换成 RDBMS(relational database management system)模型中的 Table 元素;图 2(b)是一条用 AGG 编写的转换规则,它描述了如何在 UML 模型中将某个 Class 所拥有的一个 Operation 元素移动到其父类中.



(a) (b)

Fig.2 Examples of transformation rules

图 2 转换规则示例

如图 1 所示,对于每个 Rule,都可以拥有一个 priority,用来表示它的执行顺序.每个 Rule 还拥有一些 descriptors,可以用来描述 Rule 的性质和种类等.如:QVT-R 规则分为 top-level 和非 top-level 两种[12];而在 ATL 中则分为 matched 和 lazy 等几种[5].此外,每个 Rule 中还拥有很多变量,由 Variable 表示.

如果把一个 Transformation 看做是一个程序,那么每个 Rule 都可以看做是程序中的函数.但与普通程序不同,转换程序中的 Rule 通常还可以分成若干小节,如规则的前置和后置条件、源模式和目标模式等等.如图 2(a)所示,这条 QVT-R 的规则包括 1 组 domain(源模型和目标模式)、1 个 when 子句(前置条件)和 1 个 where 子句(后置条件)[12];而如图 2(b)所示,1 条 AGG 规则由 NAC(negative application condition,否定前置条件)、PAC(positive

application condition,肯定前置条件)、LHS(源模式)和 RHS(目标模式)组成[7].为了统一概念,本文使用类 Section 来表示这些转换规则中的结构.

在图 1 中,Section 关联 *TypedModel*.这是因为,在 QVT-R 和 ATL 中,每个 domain 都可以关联一个转换参数,用来表示该 domain 描述的模式所针对的模型.如图 2(a)中,第 2 个 domain 关联了转换参数“*rdbms*”,这说明转换程序可以根据这个 domain 描述的模式在“*rdbms*”模型中进行模式匹配或模型创建操作.每个 Section 也拥有一些 descriptors,用来说明这个 Section 的类型.例如,在 QVT-R 的规则中有 3 种类型的 Section,即 domain,when 和 where,我们可以利用 descriptors 属性表示它的类型;对于一个 domain,它又可以是 check 或 enforce,这个信息也可以利用 descriptors 来表示.

一个 Section 可以看做是由一组 Expression 组成.类 Expression 用来表示转换规则中的各种表达式.而在一个 Section 中出现的变量则用关联 referredVariables 表示.如在图 2(a)中,where 子句包括两个表达式:

prefix=‘·’; *AttributeToColumn*(*c*,*t*,*prefix*);

其中出现的变量包括 *prefix*,*c* 和 *t*.AGG 规则中的每个 Section 虽然是利用图的形式定义的,但它也可以等价地表示为一组表达式[14],例如,图 2(b)中的 NAC 可以表示为如下的 OCL 表达式:

Class::AllInstances()→includes(*c*2) and
Operation::AllInstances()→includes(*o*2) and
*c*2.*operations*→includes(*o*2) and
*o*2.*name*=*x*

2.2 抽象运行时结构

普通程序在运行的过程中需要使用堆栈来记录程序运行的状态和临时数据,例如,在当前执行的过程(或函数)中,每个变量的取值、计算的中间结果等都会保存在该过程对应的堆栈结构中.转换程序在运行时也需要类似的结构记录一些信息.每种执行引擎在实现时可能使用不同的数据结构存储运行时的信息,为了统一这些结构,本节提出一种模型转换的抽象运行时结构,如图 3 所示.

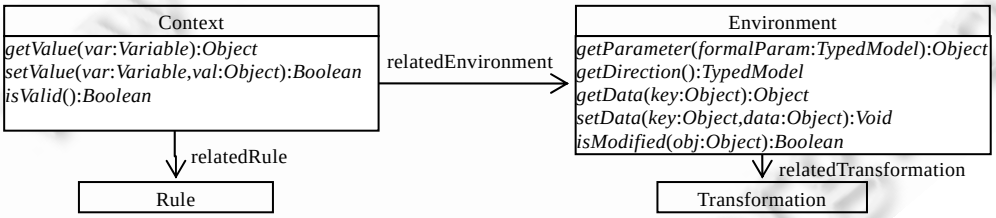


Fig.3 Runtime structure of model transformation

图 3 转换程序的运行时结构

首先,根据图 1 定义的抽象结构可以看出,每条转换规则都包含很多的变量.当执行这条转换规则时,需要使用一种结构来保存所有变量的取值,其作用类似于程序的堆栈.本文将这一运行时结构称作转换规则的 Context,即上下文,其定义如图 3(a)所示.当执行一条转换规则时,就需要创建一个 Context 对象来保存运行时的信息.因此,每个 Context 关联一个 Rule,表示这个 Context 所对应的转换规则.Context 中主要包括以下几种方法:

- *getValue*(*var*:*Variable*):*Object*.该方法用于返回变量 *var* 在当前上下文中的取值;
- *setValue*(*var*:*Variable*,*val*:*Object*):*Boolean*.该方法用于将变量 *var* 在当前上下文中的取值设置为 *val*.如果设置成功,则方法会返回 true;否则,会返回 false;
- *isValid*():*Boolean*.该方法检查当前的 Context 对象是否有效,即在这个 Context 对象中保存的所有变量的赋值都是有效的.对于基本类型的变量,任何赋值都是有效的.而对于复杂类型的变量,它的取值是一个模型元素,如果这个模型元素属于当前的模型,则这个取值是有效的;否则,变量的取值无效.

此外,整个转换程序也需要一个运行时结构来记录全局信息和初始化信息,比如转换的实际参数、转换执行的方向等等.本文将这一结构称为转换程序的 Environment,即环境,其定义如图 3(b)所示.当执行一个转换程序时,就会创建一个 Environment 对象.Environment 中保存的信息在程序初始化时存入,且在转换的执行过程中不会被修改.因此,Environment 提供的方法主要是获取信息的方法,其中主要包括以下几种方法:

- *getParameter*(*formalParam*:*TypedModel*):*Object*.该方法将返回形式参数 *formalParam* 所对应的实际参数;
- *getDirection*():*TypedModel*.该方法将返回转换程序的执行方向,即表示输出(形式)参数;
- *getData*(*key*:*Object*):*Object*.该方法用于获得其他任意的初始化信息,参数 *key* 是一个关键字,而返回值是这个关键字所对应的信息;
- *setData*(*key*:*Object*,*data*:*Object*):*Boolean*.该方法将 *key* 所对应的取值设为 *data*.如果赋值成功,则返回 true;否则,返回 false.在模型转换中之所以可能存在赋值不成功的情况,是因为在赋值操作时需要检查一些前置条件,只有前置条件满足了才可以进行赋值;否则,赋值操作将失败;
- *isModified*(*obj*:*Object*):*Boolean*.该方法用于判断转换过程中,对象 *obj* 是否发生了变化.这个 *obj* 可以表示一个模型、模型中的一个元素或者模型元素的一个属性.

需要说明的是,图 3 所示的运行时结构是一种抽象的定义,它并不依赖于任何一种转换语言和具体实现方法.在具体的转换引擎中,都会存在与 Context 和 Environment 类似的数据结构,虽然名称不同,但它们都具有与图 3 定义类图相似的作用和能力.

3 转换语义的 3 层结构

本节将讨论模型转换语言执行语义的基本结构.虽然每种转换语言的语义并不相同,但转换程序的执行过程却大致类似,可以简要表述如下:

a) 转换程序判断是否满足终止条件,如果条件满足,则停止终止程序;
b) 转换程序按照特定的原则选择一条转换规则(rule);
c) 按照一定的语义解释执行选定的转换规则;规则执行的一般逻辑是这样的:
   c1) 首先检查代表前置条件的 Section,通过计算其中的表达式,判断前置条件是否满足,如果不满足,则终止该规则的执行;
   c2) 利用代表源模式的 Section 在输入模型中进行模式匹配,找到需要转换的输入模型片段;
   c3) 利用代表目标模式的 Section 在输出模型中进行转换操作,即创建、删除和修改模型.

根据上面的描述,为了执行一个模型转换程序,需要解决 3 个主要问题:

1) 如何控制一个转换程序的整体流程;
2) 如何执行转换程序中的每一条转换规则;
3) 如何计算规则中的每一条表达式.

三者缺一,转换程序都会因为语义不清而无法顺利执行.换言之,模型转换的执行语义由 3 个主要层次构成:模型转换程序的整体控制逻辑、转换规则的执行逻辑和表达式的计算逻辑.因此,本文使用一个 3 层框架来刻画模型转换的执行语义,如图 4 所示.对于任意的 $s \in S_L$,s 都包含这 3 个层次.

在这个 3 层结构中:

- 转换控制层决定了转换程序执行的整体流程,例如,何时终止转换程序以及下一步运行哪些转换规则等等;
- 规则执行层定义了一条转换规则如何执行,例如,执行规则时首先应检查前置条件,然后在输入模型中寻找需要转换的模型元素,最后在输出模型中创建相应的模型元素等;
- 表达式计算层解释了转换规则中的一条表达式如何执行,例如,对于一个等号表达式 $a=b$,如果它出现在前置条件中,则会比较 a 和 b 的取值是否相等;如果它出现在代表目标端 Section 中,则其可能会被当作

一个赋值表达式.

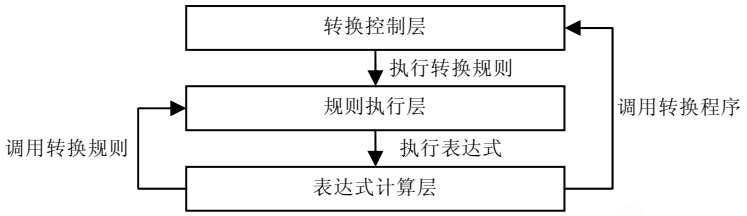


Fig.4 Three layers of transformation semantics

图 4 转换语义的 3 个层次

不同层次之间存在相应的调用关系:

- 首先,当转换控制层选择了一条转换规则后,会将控制权交给规则执行层来运行这条规则;
- 其次,规则执行层在执行转换规则时会调用表达式计算层来计算转换规则中出现的表达式.

除了普通的算术表达式,转换规则中还可能出现“规则调用表达式”和“转换调用表达式”.规则调用表达式的主要作用是调用一条转换规则,而转换调用表达式则调用另一个转换程序.例如,在图 2(a)中,when 子句里包括一个规则调用表达式 *PackageToSchema*(*p*,*s*).因此,表达式计算层在计算这两种表达式时又会调用规则执行层和转换控制层来完成具体的功能.

对于目前主流的转换语言,它们的语义都符合本文提出的 3 层语义结构.

以 QVT-R 为例,在执行一个 QVT-R 的转换程序时,首先需要指定转换的输入和输出模型,这样,转换规则中的 domain 可以分成源端和目标端两组.例如,在图 2(a)中,如果指定转换的参数 *rdbms* 为输出模型,那么与 *rdbms* 关联的那个 domain 就是目标端,而与参数 *uml* 关联的 domain 就是源端.接着,转换程序中所有 top-level 的规则会被反复执行(顺序随机),直到目标模型不再发生任何变化.QVT-R 规则的执行逻辑简单来说是这样的:先利用源端 domain 在输入模型中进行模式匹配,对于每个匹配,再根据目标端 domain 的定义在输出模型中创建相应的模型片段.

再以 ATL 为例,ATL 是一种混合式的语言,在执行 ATL 转换之前,ATL 工具会将 ATL 转换编译成 ATL 字节码,并由 ATL 虚拟机负责执行.虽然 ATL 的执行方式与 QVT-R 不完全一样(目前,大部分的 QVT-R 工具都是解释执行),但其执行过程也基本符合图 4 所示的 3 层结构.当执行一个 ATL 转换时,转换引擎首先会依次执行所有的 matched rule.在执行每条 ATL 规则时,ATL 引擎首先会执行 init 子句,然后会在模型中寻找 from 子句描述的模式的匹配,之后会按照 to 子句描述的模式在输出中创建模型,最后会执行 do 子句中的表达式.在执行过程中,如果执行了规则调用表达式,执行引擎就会根据调用规则的类型(matched,lazy,unique lazy,called 等)选择适当的方式执行规则.例如:如果是 matched rule,那么引擎首先会检查该规则是否已经执行过了,如果没有,则会执行规则,否则,会将上次执行的结果直接返回;如果是 lazy rule,则会直接执行.

4 转换原语

上一节分析了模型转换程序执行的基本过程,并提出一种 3 层框架来刻画这个过程.目前,主流的模型转换语言的执行语义都符合这个 3 层框架,这意味着用不同技术实现的模型转换的执行过程具有一定的相似性.但实际上,每种语言的语义却各不相同,且有较大差别.这就意味着不同的执行语义之间存在很多的共性和变化性成分,其中,共性成分决定了它们之间的相似性,如它们都符合图 4 所示的 3 层框架;而变化性成分则造成了它们之间的差别.

本文使用转换原语的概念来刻画不同执行语义之间的共性和变化性.转换原语可以看做是对模型转换执行过程中存在一些基本操作的抽象和封装,按照一定顺序组合使用这些转换原语,就能够构成不同的执行语义.第 3 节提出的 3 层结构是执行语义的基本框架,而转换原语则是组成并丰富该框架的具体成分.

转换原语又可以分成两类:语义固定的转换原语和语义可配置的转换原语.所谓语义固定的转换原语,其执行逻辑是固定的且不可变;而语义可配置的转换原语,它的执行逻辑随不同语言而发生变化,需要在定义转换语义时明确说明.语义固定的转换原语反映了不同转换语言中语义的共性,它们只要被定义一次,就可以适用于不同的转换语言;语义可配置的转换原语则反映了语义的差异性和变化性,需要针对每种语言进行定义.本文方法——即语义可配置的模型转换,其基本原理就是,通过重新定义某种转换语言中的语义可配置的转换原语,从而改变该语言的语义,使其能够解决不同类型的转换问题.

我们总结了几种主流转换语言的语义(例如 QVT-R,AGG,ATL 等),通过对不同语义进行分析和对比,提取出一组基本的转换原语.举例来说,在执行规则时,需要利用表示源端的 Section 在输入模型中进行模式匹配.其中,“模式匹配”就可以看做是一个转换原语.在不同语言中,模式匹配的含义基本相同,即根据一个 Section 中的表达式在模型中找到符合能够满足这组表达式的模型片段.因此,它是一个语义固定的原语.此外,“执行转换规则”也可以看做是一个转换原语,转换控制层和表达式计算层都需要利用这个原语来执行一条新的转换规则.但在不同的语言中,如何执行一条转换规则却各不相同.因此,这是一个语义可配置的原语,每种转换语言在定义语义时必须对其进行详细说明.

本节下面将讨论在转换语义的 3 层结构中都包含哪些基本的转换原语.对于语义固定的转换原语,本文将给出定义;而对于语义可配置的转换原语,本文将讨论如何定义它们.本节将使用一种类似于 OCL 的语言来定义和描述转换原语的逻辑,称其为转换语义脚本,第 5 节将介绍这种语言.需要说明的是,本节不打算定义转换原语的完全集,而事实上,这样的全集也是不存在的.本节的目标是定义一些常见的转换原语.

4.1 转换控制层

转换控制层控制转换程序的整体流程,它包括一个语义可配置的转换原语——“执行转换”.

定义 1(原语 *ExecuteTransformation*). 转换原语 *ExecuteTransformation* 可以被声明为

ExecuteTransformation(*trans*:*Transformation*,*env*:*Environment*):*Boolean*,

其中,参数 *trans* 表示要执行的转换程序;而 *env* 则表示该转换程序的初始化信息,包括实际参数、执行方向功能等等,如第 2.2 节所述.该原语返回一个布尔值,如果转换程序执行成功,则返回 true,反之,返回 false.

这个转换原语必须实现转换控制层的主要功能,它负责控制转换程序执行的整体流程,决定需要运行的转换规则以及何时终止转换程序.由于这是一个语义可配置的转换原语,因此对于不同的语言需要定义不同的实现逻辑.例如对于 QVT-R,它所对应的 *ExecuteTransformation* 原语定义如例 1 所示.

例 1:QVT-R 中 *ExecuteTransformation* 的定义:

```
1. QVT-R::ExecuteTransformation(trans:Transformation,env,Environment):Boolean
2. =
3. repeat
4.     trans.ownedRules→select(r:Rule|r.descriptors→includes('top'))     //选择需要执行的转换规则
5.         →forAll(r:Rule|ExecuteRule[enforce](r,null,env))              //依次执行选中的规则
6. until not env.isModified(env.getParameter(env.getDirection()))       //判断终止条件
```

在例 1 中,第 1 行表示要定义的转换原语,前缀“QVT-R::”代表所针对的转换语义,在本例中表示 QVT-R 的语义.根据 QVT-R 的语义,一个 QVT-R 的转换程序会反复执行所有 top-level 的规则,直到输出模型不再发生任何变化为止.如果一条规则是 top-level 的,那么它的 descriptors 属性中一定包含值“top”.因此,在判断一条规则是否是 top-level 时,例 1 使用了条件 *r.descriptors*→includes('top').在执行规则时,我们使用了原语 *ExecuteRule*,在第 4.2 节中将详细讨论这个原语.最后需要说明的是,例 1 中的 repeat $expr_1$ until $expr_2$ 表达式将返回一个布尔值:如果在执行 $expr_1$ 时其结果是 false,那么整个表达式将返回 false;否则,当终止条件满足,即 repeat 表达式停止时,将返回 true.

4.2 规则执行层

规则执行层定义了转换规则的执行语义,用于控制一条转换规则的执行过程,其中存在一组转换原语,包括执行规则、匹配模式、创建对象、删除对象.

原语“执行规则”用于执行一条转换规则.由于每种转换语言执行转换规则的方式有所不同,因此这个原语是一个语义可配置的转换原语.其定义如下:

定义 2(原语 *ExecuteRule*). 转换原语 *ExecuteRule*,即“执行规则”,可被声明为

ExecuteRule[*mode*](*rule*:*Rule*,*initialContext*:*Context*,*env*:*Environment*):*Boolean*,

其中,参数 *rule* 表示要执行的转换规则;*initialContext* 表示初始的上下文,其中包含了对规则中变量的初始赋值;*env* 表示转换执行的环境.如果规则执行成功,则该原语返回 true;否则,返回 false.

转换规则可以按照不同的方式去执行.例如在 QVT-R 中,一条转换规则可以按照“只检查”的方式执行,此时,该规则可以看做是输入和输出模型之间的一条一致性约束,按照“只检查”的方式执行这条规则可以检验输入和输出是否满足这条约束;一条规则也可以按照“强制”方式执行,此时,转换程序会根据规则创建输出模型.也就是说,在同一种转换语言中可能存在不同版本的 *ExecuteRule* 原语.为了加以区分,定义 2 中使用符号[*mode*]来表示 *ExecuteRule* 的不同版本.例如,我们可以利用 *ExecuteRule*[*check*]来表示 QVT-R 的只检查语义,而用 *ExecuteRule*[*enforce*]来表示强制语义.利用与例 1 类似的方式,我们可以定义每个版本的 *ExecuteRule*.

“匹配模式”是规则执行层中的一个语义固定的转换原语,它用来在给定的模型中寻找所有符合模式的模型片段.在很多转换语言中,通常把需要进行转换的模型表示成一个模式.利用这个模式,在输入中寻找满足条件的匹配,然后再根据转换规则将其转换成输出模型.“匹配模式”的定义如下所示:

定义 3(原语 *MatchPattern*). 原语 *MatchPattern* 表示“匹配完整模式”,是一个语义固定的转换原语.它可以被定义为

MatchPattern(*patterns*:*Section*[1…*],*initialContext*:*Context*,*env*:*Environment*):*Context*[*]

post: *patterns*=*null* or *patterns*→size()=0 implies *result*=*initialContext*

post: *result*→forAll(*m*:*Context*|*patterns*→forAll(*p*:*Section*|*EvaluateExpression*[*check*](*p.ownedExpressions*,*m*))

其中,*patterns* 表示需要匹配的模式;*initialContext* 表示初始的上下文,其中包括一些变量的初始赋值;*env* 表示转换的环境.“*post*:”后面的表达式是这个原语的后置条件,其中,*result* 表示原语的返回结果.这个后置条件的含义是:如果 *patterns* 为 *null* 或空集,则直接返回 *initialContext*;否则,该原语返回一组 *Context*,其中,每个 *Context* 都可以满足模式中的约束表达式.

本文没有规定如何实现 *MatchPattern*,而是定义了它的后置条件.因此,我们可以使用不同的机制和算法实现这个原语,只需要保证其后置条件的满足.在定义 3 给出的后置条件中,*EvaluateExpression*[*check*]表示“检查约束表达式”,它是表达式计算层中的一个转换原语.它会按照给定的上下文(即变量赋值)计算指定的表达式,如果所有的计算结果为真,那么该原语返回 true.

转换规则在执行过程中还会创建和删除模型元素,为此,本文还分别定义了如下两个原语:

定义 4(原语 *CreateElement*). 转换原语 *CreateElement* 表示“创建模型元素”,它被定义为

CreateElement(*var*:*Variable*,*rule*:*Rule*,*currentContext*:*Context*,*env*:*Environment*):*Boolean*

pre: *var*.*getType*().*oclIsKindOf*(*PrimitiveType*) or *currentContext*.*getValue*(*var*).*oclIsUndefined*()

post: *result*=true implies *var*.*getType*().*oclIsKindOf*(*PrimitiveType*) or

not *currentContext*.*getValue*(*var*).*oclIsUndefined*()

其中,*var* 表示一个变量,*CreateElement* 会根据这个变量的类型在模型中创建新的模型元素,并将这个元素赋值给 *var*;*rule* 表示目前正在执行的转换规则,*var* 应该是 *rule* 包含的一个变量;*currentContext* 表示当前的上下文;而 *env* 则表示当前转换运行的环境.如果元素创建成功,则该原语返回 true;否则,返回 false.“*pre*:”和“*post*:”后面的表达式分别表示原语的前置和后置条件.这个原语会为 *var* 创建一个模型元素,并将这个新创建的元素赋值给 *var*.因此,这个原语的前置条件是在执行 *CreateElement* 之前,*var* 在当前的上下文中没有赋值.而原语的后置

条件则是,当这个原语执行成功后,*var* 在当前的上下文下会拥有一个取值.

与之相对应,规则执行层中还包括一个“删除模型元素”的原语,其定义为:

定义 5(原语 *DeleteElement*). 转换原语 *DeleteElement* 表示“删除模型元素”,它被定义为

DeleteElement(*var*:*Variable*,*rule*:*Rule*,*currentContext*:*Context*,*env*:*Environment*):*Boolean*

pre: *var*.*getType*().*oclIsKindOf*(*PrimitiveType*) or *currentContext*.*getValue*(*var*).*oclIsUndefined*()

post: *result*=true *implies var*.*getType*().*oclIsKindOf*(*PrimitiveType*) or

currentContext.*getValue*(*var*).*oclIsUndefined*()

在 *DeleteElement* 中,每个参数的含义、前置和后置条件的含义都与 *CreateElement* 类似,这里不再赘述.

4.3 表达式计算层

表达式计算层主要负责转换规则中各种表达式的计算,其中包括一个转换原语,即“计算表达式”.理论上,不同的转换描述语言可能使用不同的语法来编写规则中的表达式,因此,“计算表达式”是一个语义可扩展的转换原语,其定义如下所示:

定义 6(原语 *EvaluateExpression*). 原语 *EvaluateExpression* 表示“计算表达式”,它是一个语义可配置的转换原语,并可被声明为

EvaluateExpression[*mode*](*exprs*:*Expressions*,*currentContext*:*Context*):*Boolean*,

其中,*exprs* 表示要执行的表达式;*currentContext* 则表示当前的上下文,其中包括对表达式中出现的变量的赋值.该原语返回一个布尔值,如果所有的表达式执行成功,则返回 true;否则,返回 false.

EvaluateExpression 是一个语义可扩展的转换原语,可以有不同的实现.但是由于目前的转换语言主要使用 OCL(或者经过扩展的 OCL)来编写表达式,所以可以总结出几种常见的实现方式:

- *EvaluateExpression*[*check*]:以检查方式计算 OCL 表达式,根据给定的变量赋值,计算 OCL 表达式的结果.这种方式相当于标准的 OCL 语义;
- *EvaluateExpression*[*enforce*]:以强制的方式计算 OCL 表达式,通过修改模型使得 OCL 表达式得以满足.

EvaluateExpression[*check*]不会修改模型,与之相反,*EvaluateExpression*[*enforce*]则会通过修改模型使表达式得到满足.*EvaluateExpression*[*enforce*]虽然可以修改模型,但也有一个限制:同一个地方只能修改 1 次.这样,可以避免因为反复修改模型从而导致转换程序无法达到不动点的问题.

熊英飞等人在文献[15]中介绍了一种利用 OCL 语言检查和修复模型一致性约束的方法.在他们的方法中,每个 OCL 表达式都被赋予了两种不同的语义,即检查语义和修复语义.这两种语义基本上可以看做是本节提出的 *EvaluateExpression*[*check*]和 *EvaluateExpression*[*enforce*].因此,这里不再赘述它们的详细定义.

5 转换语义脚本

在第 4 节中,本文总结了 6 个基本的转换原语.为了定义转换语义,就需要描述所有语义可配置的转换原语的实现逻辑:*ExecuteTransformation* 对应转换控制层的语义,*ExecuteRule* 对应规则执行层的语义,*EvaluateExpression* 对应表达式计算层的语义.对于这 3 种语义可配置的转换原语,每种转换语言都有各自的实现逻辑;通过重新定义这 3 个原语,就可以做到配置和改变某个转换语言的语义.此外,在必要时,转换程序的开发人员还可以定义新的转换原语,从而扩充某个转换语言的语义.

本文提出一种转换语义脚本(transformation semantics script,简称 TSS)来描述转换原语的实现逻辑,第 4.1 节中的例 1 就是用 TSS 描述转换原语的例子.TSS 扩展自 OCL 语言,其核心语法如图 5 所示.

在图 5 中:语法 *decl* 表示转换原语的声明,在定义转换原语时已经多次用到;语法 *def* 用于描述转换原语的实现逻辑;一个转换原语的实现逻辑是一个表达式(用 *expr* 表示),*expr* 可以是任何一个 OCL 表达式.与此同时,TSS 还增加了几种新的表达式类型:

- *expr*→‘{‘*exprs*’}’:*exprs* 表示一组用“;”分隔的表达式,这个表达式返回值是 *exprs* 中最后一个表达式的结果.例如,{true=false;true}是符合这一语法的表达式,它的计算结果是 true,因为在大括号中,最后一个

表达式是 true;

- *expr*→repeat *expr*[1] until *expr*[2]:这个表达式表示循环,它会反复执行 *expr*[1],直到 *expr*[2] 表示的终止条件成真.如果在执行 *expr*[1] 时,*expr*[1] 的结果为 false,那么整个表达式终止,返回 false;当 *expr*[2] 为真时,循环终止,返回 true;
- *expr*→while *expr*[1] do *expr*[2]:这个表达式也表示一个循环,*expr*[1] 表示循环的条件,*expr*[2] 表示要执行的表达式.与 repeat…until 表达式类似,它也会返回执行 *expr*[2],直到 *expr*[1] 表示的循环条件为假.在这个过程中,如果 *expr*[2] 执行结果为 false,那么循环终止,整个表达式返回 false;当 *expr*[1] 为 false 时,表达式也将终止,并返回 true;
- *expr*→*opName* '('*actparams*')':该表达式表示调用一个转换原语,*actparams* 表示实际参数;
- *expr*→*simpleName* ':='*expr*[1]:该表达式表示一个赋值,其中,*simpleName* 表示一个变量名,*expr*[1] 的计算结果是右值.如果赋值成功,则该表达式返回 true.

decl	→*opName* '('*params*')' ':' *typeName*	*expr*	→'{'*exprs*'}'
opName	→*simpleName*		|repeat *expr until expr*
	|*simpleName* '['*modeName*']'		|while *expr* do *expr*
modeName	→*simpleName*		|*opName* '('*actparams*')'
typeName	→*simpleName*		|*simpleName* ':=' *expr*
params	→*simpleName* ':' *typeName* ',' *params*		|any OCL *expression*
	|*simpleName* ':' *typeName*	*exprs*	→*expr* ';' *exprs*
	|*ε*		|*expr*
def	→*decl* '=' *expr*	*actparams*	→*expr* ',' *actparams*
simpleName	→*IDENTIFIRE*		|*expr*

Fig.5 Core syntax of transformation semantics script

图 5 转换语义脚本的核心语法

TSS 采用一阶谓词逻辑的形式定义转换原语的执行逻辑,或者用来描述某个模型转换语言的执行语义(实际上就是定义 *ExecuteTransformation* 的逻辑).用 TSS 描述的执行语义是一种机器可读的脚本,解释函数 U_L 的实现核心是 TSS 的解释器.需要说明的是,虽然可以用它来描述表达式计算层的原语(即 *ExecuteExpression*),但一方面,表达式的执行逻辑不仅规模庞大而且算法复杂,不适宜用 TSS 直接进行描述;另一方面,常见的表达式执行逻辑可以被枚举出来.所以,本文的原形工具把 *ExecuteExpression* 的几种版本看做是语义固定的转换原语,并通过编码的方式加以实现.因此,TSS 主要用来定义转换控制层与规则执行层中存在的转换原语与执行逻辑.

6 工具实现

为了验证方法的可行性,本文实现了一个语义可扩展的模型转换引擎(http://code.google.com/p/pku-motif/),该工具的体系结构如图 6 所示.首先,转换程序的开发人员使用转换编辑器编写转换程序;之后,该工具利用一个格式转换器将开发人员写好的转换程序变成一个工具可读的内部表示.这个内部表示符合本文所定义的转换程序的抽象结构(如图 1 所示).这个步骤使得本文的方法与工具可以独立于某个具体的转换语言——无论开发人员用什么语言编写转换程序,经过格式转换后,都会变成一个统一的内部表示.

当执行转换程序时,控制器首先会读入转换语义脚本,并控制模型转换器进行初始化,包括读取转换程序、读取模型信息等等.之后,控制器会按照转换语义脚本中描述的执行语义控制模型转换器的行为,使其可以按照转换程序中描述的转换规则和逻辑实现输入模型到输出模型的转换操作.该工具中还包括一个转换原语库,其中包括了一些预先实现好的转换原语,例如语义固定的转换原语以及 *EvaluateExpression* 的几个不同版本.当转换语义脚本中不包含某个转换原语的实现逻辑时,控制器就会调用转换原语库来执行该原语.最后,转换程序终止后,模型转换器会将输出模型保存.通过对比第 1 节中定义的解释函数 U_L 不难看出,图 6 中的由模型转换器和控制器组成的执行引擎完全符合 U_L 的形式化定义:它读取执行语义脚本 *s*、转换程序 *p* 和输入模型,按照 *s* 解释 *p*,并产生输出模型.

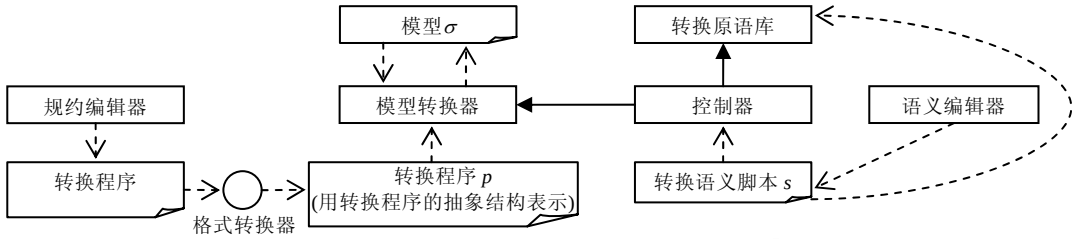


Fig.6 Architecture of semantics configurable transformation engine

图 6 语义可配置的转换引擎的体系结构

7 案例研究

QVT-R 是对象管理组织(Object Management Group)提出来的一种标准化的模型转换语言.QVT-R 是一种声明式的转换语言,它允许开发人员通过描述两个模型之间存在的一致性关系(每个关系相当于一个转换规则)实现模型之间的转换.作为一种标准化的转换语言,QVT-R 常用于实现 PIM 和 PSM 之间的转换,但由于语义的限制,它不能很好地支持模型重构和模型同步等其他类型的转换.

本节将以 QVT-R 为例,利用本文提出的语义可配置的模型转换技术,通过重新定义 QVT-R 的转换语义,使其能够支持模型重构和模型同步.如果在一个开发活动中同时需要实现 PIM 到 PSM 的转换、模型重构和模型同步,利用本文的方法,开发人员无需学习新的转换语言就可以实现这两种类型的转换.

7.1 标准的QVT-R语义

本节介绍如何利用本文方法描述 QVT-R 的标准语义.首先,QVT-R 本身包括两种语义,即“只检查”语义和“强制”语义.“只检查”语义会判断输入模型和输出模型是否满足转换规则所描述的一致性关系;而“强制”语义在输入和输出不满足一致性关系时,通过修改输入模型使得关系得到满足.这两种语义可以在文献[12]中找到准确定义,下面分别讨论如何利用本文方法描述这两种语义:

- QVT-R 的“只检查”语义

在“只检查”语义中,转换控制层的逻辑是这样的:以“只检查”的方式执行所有的 top-level 的转换规则,如果有一条规则的执行结果为 false,则转换返回 false;否则,返回 true.该逻辑可以表示成如下的 TSS 脚本:

1. QVT-R::*ExecuteTransformation*[*check*](*trans*:*Transformation*,*env*,*Environment*):*Boolean*=
2. *trans*.*ownedRules*→select(*descriptors*→includes(‘top’))
3. →forAll(*r*:*Rule*|*ExecuteRule*[*check*](*r*,*null*,*env*))

在上面的脚本中,我们在定义 *ExecuteTransformation* 时借用了符号“[*check*]”来区分“只检查”和“强制”的转换语义.这个脚本需要调用 *ExecuteRule*[*check*],即以“只检查”的方式执行转换规则.*ExecuteRule*[*check*]的基本逻辑可以表述为:在 when 子句中的条件得到满足的情况下,对于每个源端 domain 的匹配,如果存在一个目标 domain 的匹配与之对应,并使 where 子句中的条件得到满足,则规则成立.这个逻辑可以表示为如下的 TSS 脚本:

1. QVT-R::*ExecuteRule*[*check*](*rule*:*Rule*,*initialContext*:*Context*,*env*:*Environment*):*Boolean*=
2. *MatchPattern*(*rule*.*ownedSections*→select(*descriptors*→includes(‘when’)),
   initialContext,*env*)→forAll(*whenContext*:*Context*| //检查 when 子句,当 when 子句满足时检查源端
3. *MatchPattern*(*rule*.*ownedSections*→select(*relatedModel*<>*env*.*getDirection*(),
   whenContext,*env*)→forAll(*sourceContext*:*Context*| //检查源端 domain,当源端满足时检查目标端
4. *MatchPattern*(*rule*.*ownedSections*→select(*relatedModel*=*env*.*getDiretion*()),
   sourceContext,*env*)→exists(*targetContext*:*Context*| //检查目标端 domain
5. *EvaluateExpression*[*check*](*rule*.*ownedSections*→select(//最后检查 where 子句

descriptors→includes(‘where’)).*ownedExpressions*,*targetContext*)

6.) //结束检查目标端

7.) //结束检查源端

8.) //结束检查 when 子句

最后,*ExecuteRule*[*check*]在执行时会调用 *EvaluateExpression*[*check*]来检查规则中的表达式.

EvaluateExpression 虽然是语义可配置的转换原语,但如第 4.3 节中的讨论,我们可以总结出几种固定的实现版本,即 *EvaluateExpression*[*check*]和 *EvaluateExpression*[*enforce*].因此,本节中将它们看做是语义固定的转换原语,不再加以定义.

- QVT-R 的“强制”语义

在“强制”语义中,QVT-R 的转换程序会以“强制”方式反复执行 top-level 的转换规则,直到输出模型不再发生变化;如果有一条规则返回 false,则整个转换就返回 false.该逻辑就是例 1 中的 TSS 脚本,这里不再赘述.

例 1 中的脚本使用 *ExecuteRule*[*enforce*]来表示“按照‘强制’方式执行转换规则”.*ExecuteRule*[*enforce*]的基本逻辑可以分成两部分,即创建逻辑和删除逻辑,即可以表示为:

1. QVT-R::*ExecuteRule*[*enforce*](*rule*:*Rule*,*initialContext*:*Context*,*env*:*Environment*):*Boolean*=
2. *CreateLogic*(*rule*,*initialContext*,*env*)
3. and
4. *DeleteLogic*(*rule*,*initialContext*,*env*)

其中,创建逻辑(CreateLogic)可以表述为:在 when 子句中的条件得到满足的情况下,对于源端 domain 的每个匹配,找到或者创建一个目标端 domain 的匹配,使得目标端 domain 中的约束和 where 子句中的约束得到满足.这一逻辑可以表示为下面的 TSS 脚本:

1. QVT-R::*CreateLogic*(*rule*:*Rule*,*initialContext*:*Context*,*env*:*Environment*):*Boolean*=
2. let *whensec*=*rule*.*ownedSections*→select(*descriptors*→includes(‘when’)) in
3. let *srcsec*=*rule*.*ownedSections*→select(*relatedModel*<>*env*.*getDirection*()) in
4. let *tarsec*=*rule*.*ownedSections*→select(*relatedModel*=*env*.*getDirection*()) in
5. let *wheresec*=*rule*.*ownedSections*→select(*descriptors*→includes(‘where’)) in
6. *MatchPattern*(*whensec*,*initialContext*,*env*)→forAll(*whenContext*:*Context*|
7. *MatchPattern*(*srcsec*,*whenContext*,*env*)→forAll(*srcContext*:*Context*|
8. *MatchPattern*(*tarsec*,*srcContext*,env)→exists(*tarContext*:*Context*|
9. *EvaluateExpression*[check](*wheresec*.*ownedExpressions*,*tarContext*)
10. or (*tarsec*.*referredVariables*→forAll(*v*:*Variable*|*CreateElement*(*v*,*rule*,*srcContext*,*env*))
11. and *EvaluateExpression*[*enforce*](
tarsec.*ownedExpressions*→union(*wheresec*.*ownedExpressions*),*srcContext*)))
12.)
13.)

此外,删除逻辑(DeleteLogic)可以表述为:对于目标端 domain 的任意一个匹配,如果找不到一个源端 domain 的匹配,使得 when 子句中的约束和 where 子句中的约束得到满足,则删除这个目标端 domain 的匹配.这一逻辑(DeleteLogic)也可以用 TSS 描述:

1. QVT-R::*DeleteLogic*(*rule*:*Rule*,*initialContext*:*Context*,*env*:*Environment*):*Boolean*=
2. let *whensec*=*rule*.*ownedSections*→select(*descriptors*→includes(‘when’)) in
3. let *srcsec*=*rule*.*ownedSections*→select(*relatedModel*<>*env*.*getDirection*()) in
4. let *tarsec*=*rule*.*ownedSections*→select(*relatedModel*=*env*.*getDirection*()) in
5. let *wheresec*=*rule*.*ownedSections*→select(*descriptors*→includes(‘where’)) in

```
6.   MatchPattern(tarsec,initialContext,env)→forAll(tarContext:Context|
7.     (MatchPattern(whensec,tarContext,env)→exists(whenContext:Context|
9.       MatchPattern(srcsec,whenContext,env)→exists(srcContext:Context|
10.        EvaluateExpression[check](wheresec.ownedExpressions,srcContext,env))))
12.    or tarsec.referredVariables→select(v:Variable|rule.ownedSections→forAll(s:Section|
           s<>tarsec implies s.referredVariables→excludes(v))
13.      →forAll(v:Variable|DeleteElement(v,rule,tarContext,env)
14.  )
```

根据 QVT 规范[12],在创建元素和删除元素之前还需要进行一些检查.这些检查逻辑都可以使用 TSS 进行描述,但比较繁琐.出于版面和可读性的原因,本文将它们省略.

7.2 利用QVT-R实现模型重构

在第 7.1 节中,我们以 QVT-R 的标准语义为例,利用本文的方法实现了 QVT-R 的基本执行逻辑.不难看出,标准的 QVT-R 语义能够实现两个模型之间的转换,但无法实现模型的重构操作.本节将利用本文的方法重新配置 QVT-R 的执行语义,使其能够实现模型重构.

一般情况下,人们通常使用“图改写”逻辑实现模型重构[2],这种逻辑可以简单地表述为一种替换操作,即在模型中把源端模式的匹配替换成目标模式的实例.为了使用 QVT-R 来描述一个改写逻辑,我们首先需要进行如下规定:1) 每个用 QVT-R 描述重构规则只包含两个 domain,一个用 checkonly 修饰(称为源端),另一个用 enforce 修饰(称为目标端);2) when 子句表示规则的前置条件;3) where 子句表示规则的后置条件.

对于模型重构的语义来说,其转换控制层的执行逻辑与例 1 中的逻辑类似,即反复执行 top-level 的规则,直到模型不发生任何变换.因此,本节不再重复定义这一逻辑.与 QVT-R 的标准语义相比,其主要差别在于规则执行层的逻辑,即原语 *ExecuteRule* 的实现逻辑.本节将其定义为 *ExecuteRule*[*refactor*],表示重构逻辑:

```
1. QVT-R::ExecuteRule[refactor](rule:Rule,initialContext:Context,env:Environment):Boolean=
2. let whensec=rule.ownedSections→select(descriptors→includes('when')) in
3. let srcsec=rule.ownedSections→select(descriptors→includes('checkonly')) in
4. let tarsec=rule.ownedSections→select(descriptors→includes('enforce')) in
5. let wheresec=rule.ownedSections→select(descriptors→includes('where')) in
6.   MatchPattern(srcsec,initialContext,env)→forAll(srcContext:Context|
7.     EvaluateExpressions[check](whensec.ownedExpressions,srcContext) implies
9.       EvaluateExpressions[enforce](wheresec.ownedExpressions,srcContext) and
10.      srcsec.ownedVariables→select(v1:Variable|not tarsec.ownedVariables
           →exists(v2:Variable|srcContext.getValue(v2)=srcContext.getValue(v1))
11.          →forAll(v:Variable|DeleteElement(v,rule,srcContext,env)) and
12.      tarsec.ownedVariables→select(v:Variable|srcContext.getValue(v).oclIsUndefined())
           →forAll(v:Variable|CreateElement(v,rule,srcContext,env)) and
13.      EvaluateExpressions[check](tarsec.ownedExpressions,srcContext)
14.  )
```

7.3 利用QVT-R实现模型同步

模型同步可以使得两个相互关联的模型保持一致,而每个转换规则相当于描述了模型之间的一个一致性关系.QVT-R 也可以用来实现模型同步,但 QVT-R 本身的语义(见第 7.1 节)并不是同步逻辑.限于篇幅,本节仅以一种基本的同步算法为例,讨论如何用本文方法通过重新定义 QVT-R 的语义使其支持模型同步操作.在这样的方法中,转换程序首先需要检查已有的一致性关系是否成立:如果成立,则继续保持这些关系;如果不成立,则需

要删除参与到关系中的模型元素.接着,转换程序会尝试寻找新的一致性关系.在上述过程中,所谓“已有的一致性关系”可以理解为转换规则成功执行的记录,因为每成功执行一次转换规则,就意味着在模型之间找到了一种一致性关系.我们可以用 *Environment* 中的 *getData* 方法来获得以前的执行记录.在模型同步语义中,转换控制层的逻辑可以用下面的 TSS 脚本来表示:

```
1. QVT-R::ExecuteTransformation[sync](trans:Transformation,env,Environment):Boolean=
2. trans.ownedRules→forAll(r:Rule|ExecuteRule[recheck](r,null,env))
3. and repeat
4.    trans.ownedRules→select(descriptors→includes('top'))
          →forAll(r:Rule|ExecuteRule[sync](r,null,env))
5. until not env.isModified(env.getParameter(env.getDirection())
```

在上面的脚本中,*ExecuteRule*[*recheck*]表示重新检查已存在的一致性关系,而 *ExecuteRule*[*sync*]则表示查找新的一致性关系.*ExecuteRule*[*recheck*]的逻辑可以表述为:

```
1.  QVT-R::ExecuteRule[recheck](rule:Rule,initialContext:Context,env:Environment):Boolean=
2.  env.getData('trace')→select(relatedRule=rule)→forAll(cont:Context|
3.     not (
4.     rule.ownedVariables→forAll(v:Variable|
5.        v.getType().oclIsKindOf(PrimitiveType) implies cont.setValue(v,null))
6.     and
7.     rule.ownedSections→forAll(s:Section|EvaluateExpression[check](s.ownedExpressions,cont))
8.     )     //重新检查现有的一致性关系是否成立
9.     implies
10.    let whensec=rule.ownedSections→select(descriptors→includes('when')) in
11.    rule.ownedVariables→select(v:Variable|whensec.ownedVariables→excludes(v))
           →forAll(v:Variable|DeleteElement(v,rule,cont,env))      //如果不成立,则删除相关的模型元素
12. )
```

在检查了现有的一致性关系之后,同步程序还需要寻找新的一致性关系.本文用 *ExecuteRule*[*sync*]来表示这个逻辑.*ExecuteRule*[*sync*]也可以用 TSS 描述,其基本逻辑与第 7.1 节中 *CreateLogic* 的逻辑类似:

```
1.  QVT-R::ExecuteRule[sync](rule:Rule,initialContext:Context,env:Environment):Boolean=
2.  let whensec=rule.ownedSections→select(descriptors→includes('when')) in
3.  let wheresec=rule.ownedSections→select(descriptors→includes('where')) in
4.  MatchPattern(whensec,initialContext,env)→forAll(whenContext:Context|
5.  let domains=rule.ownedSections
          →select(descriptors→excludes('when') and descriptors→excludes('where')) in
6.  domains→forAll(tarsec:Section|let srcsec=domains→excluding(tarsec) in
7.     MatchPattern(srcsec,whenContext,env)→forAll(srcContext:Context|
8.        MatchPattern(tarsec,srcContext,env)→exists(tarContext:Context|
9.           EvaluateExpression[check](wheresec.ownedExpressions,tarContext)
10.          or (tarsec.referredVariables→forAll(v:Variable|CreateElement(v,rule,srcContext,env))
11.             and EvaluateExpression[enforce](
                      tarsec.ownedExpressions→union(wheresec.ownedExpressions),srcContext)))
12.    )
13. )
```

与第 7.2 节中用 QVT-R 实现模型重构的方法类似,按照 *ExecuteTransformation*[*sync*],*ExecuteRule*[*recheck*]和 *ExecuteRule*[*sync*]的逻辑执行一个 QVT-R 的转换程序,就可以实现一个基本的模型同步操作.

通过上面 3 个案例不难看出,本文提出的方法可以有效地描述不同类型的转换语义,因此本文方法具有足够的表达能力可以对转换语言的语义进行(重)定义.如果不使用本文的方法,为了完成 PIM 到 PSM 的转换、模型重构和模型同步,开发人员可能需要使用 QVT-R,AGG 和 TGG 等多种语言及其相应工具.在使用了本文方法之后,开发人员只需要使用 QVT-R 一种语言就可以实现这 3 种类型的转换程序.

8 讨 论

- 完全性

需要讨论的第 1 个问题是本文提出的转换原语是否完全?是否存在其他的转换原语?首先,应当指出,本文在第 4 节提出的转换原语只是一些公有的转换原语,是一些在不同的转换语言和语义中经常被用到的原语.从这个意义上讲,本文提出的转换原语是不完全的,而且可以肯定地说确实存在其他类型的转换原语.转换原语是对转换语义中某些特定逻辑的封装,因此人们可以根据需要定义新的转换原语.同时,本文也提供 TSS 允许人们定义新的转换原语,例如在第 7.1 节中我们就定义了 *CreateLogic* 和 *DeleteLogic* 这两个针对 QVT-R 标准语义的转换原语.

- 描述能力

既然本文提出的转换原语是不完全的,那么是否会影响本文方法的描述能力?即是否可以用本文方法定义出“所有的”转换语义?首先,文本无法从理论证明本文的方法能够描述“所有的”语义,因为究竟有多少种转换语义仍是一个未知的问题,所以无法排除一些极端复杂的转换语义无法使用 TSS 进行描述(或描述起来很复杂).但从实践中看,本文方法的表达能力还是足够的,原因有三:

- ✧ 其一,通过第 7 节中的 3 个案例不难看出,本文方法在处理常见问题上是充分可行的;
- ✧ 其二,本文给出 TSS 是一种基于 OCL 的脚本语言,由于 OCL 已经被实践证明具有充足的表达能力,因此 TSS 具有足够的能力来描述绝大部分转换原语实现逻辑;
- ✧ 其三,对于某些十分特殊且复杂的转换语义,可以通过编码的方式将其封装成相应的转换原语,并将其加入到转换原语库(见第 6 节)中,从而可以整体上提升本文方法的表达能力.

- 使用复杂度

需要讨论的第 3 个问题是本文方法的复杂度.

* 首先,本文方法的核心思想是,通过重新定义转换语言的语义,使得同一种语言可以支持不同类型的转换操作.为定义转换语言的语义,要求开发人员了解语义的每个细节.从这个意义上讲,本文方法在一定程度上增加了开发人员的负担,提高了复杂度.但用本文方法定义的转换语义是可复用的,并不是每编写一个转换程序就需要定义一套转换语义,相同类型的转换程序可以使用同一个转换语义;
* 其次,如果不使用本文的方法,又要重新定义转换语言及相应的引擎执行逻辑,就需要直接修改转换引擎的源代码.与这种方式相比,本文一方面提出一组公共的转换原语,对常见的执行逻辑进行了封装;另一方面又利用 TSS 定义语义可配置的转换原语的实现逻辑.与直接编码的方式相比,本文方法的复杂度更低.

综上,本文方法的复杂度是可接受的.

- 提高转换程序的可复用性

本文工作的核心思想在于通过重新定义转换语言的语义,使其能够解决不同的问题.除此之外,本文工作还能提高转换程序的可复用性.这是因为:在改变转换语言(特别是声明式的转换语言)的语义后,用这种语言编写的转换程序的行为也随之发生变化;同一个转换程序按照不同的语义进行解释,就可以完成不同的操作.例如,通过定义模型同步语义,一个原本用于实现从 PIM 到 PSM 转换的程序,也可以用来维护这两个模型之间的同步关系.这样,同一个转换程序就可以用在不同的场景中——这个程序的可复用性得以提高.

9 相关工作

目前,很多模型转换语言都有固定的转换语义,适合处理不同类型的转换任务.例如:

- AGG[7]是一种图转换语言,经常被用于实现模型重构.AGG 支持图改写逻辑,每条 AGG 规则包括一个 LHS(left hand side,即左手端)和一个 RHS(right hand side,即右手端)分别描述模型修改前的状态和修改后的状态.因此,AGG 非常适合描述如何修改一个模型,但如果用它来描述两个模型之间的转换则会非常复杂;
- TGG(triple graph grammar)[3]是一种三图转换语言,每条 TGG 规则包括一个 LHS、一个 CS (correspondence,即相关图)和一个 RHS,分别描述了源模型(LHS)、目标模型(RHS)和二者之间的关系(CS);
- GG 适用于描述两种模型之间的一致性关系,因此常被用作模型同步操作;
- QVT-R 是对象管理组织提出来的一种模型转换语言,本文在第 2 节中介绍了它的语法,并在第 7.1 节中描述了它的语义.从中可以看出,QVT-R 适合用来实现两个模型之间的转换,特别是 PIM 到 PSM 之间的转换;
- ATL 和 ETL[16]是两种混合式的模型转换语言,它们同时具有声明式语言(如 QVT-R)及编程语言的特点,常被用来实现两个模型之间的转换.作为一种混合式语言,与 AGG,QVT-R 和 TGG 相比,ATL 和 ETL 能够描述更加复杂的转换逻辑,解决更多类型的转换问题,但也无法很好地支持模型同步和模型重构.

由此可见,尽管目前存在各种各样的模型转换语言,但为了解决一个复杂的转换问题,还是需要同时学习和使用不同语言及其相应的执行引擎.而利用本文的方法,通过重新定义转换语言的语义,允许开发人员使用同一种语言解决不同类型的转换问题,从而降低了学习负担.

本文最终实现了一种语义可配置的模型转换执行机制,因此,现有的模型转换执行引擎也是本文的相关工作.现有的执行工具在执行模型转换时可以分成 3 种方式:解释型、编译型和转化型.

- 解释型的执行工具,如 MediniQVT[17],通过编码的方式实现转换语言的执行语义.如果预先实现的执行语义需要修改或扩展,就必须修改执行工具中的代码:开发人员需要首先理解原有的工具结构和相应的算法,再修改或添加所需要的代码.这种方式不仅复杂度高,而且容易在工具中引入新的错误;
- 编译型的执行工具,如 ATL Virtual Machine[18],会将模型转换程序编译成字节码,然后部署在相应的虚拟机上运行.然而,为了修改或扩展这种工具所支持的执行语义,开发人员可能不仅需要修改编译器——通过更改字节码产生的方式,还可能需要修改虚拟机的指令集——更改字节码的执行逻辑.由于字节码的粒度更细,因此会导致这种方式抽象层次更低,而且复杂度也很高;
- 转化型的执行工具会将用语言 L_1 编写的转换程序转化成用语言 L_2 编写的转换程序,并利用 L_2 的执行工具执行这个程序.换言之,这种工具会利用 L_2 的语义解释 L_1 的语义.与编译型的工具类似,当需要扩展 L_1 的执行语义时,可以修改从 L_1 到 L_2 的转化过程,或者将 L_1 映射到其他语言上.但这种方式依赖于目标语言的执行语义,如果目标语言的语义也不能满足实际需要,就必须修改目标语言所对应的执行工具,从而导致在扩展解释型或编译型工具时所出现的问题.

由此可见,这 3 种类型的执行工具的可扩展性都存在不足.而本文提出的语义可配置的模型转换执行引擎把执行语义作为输入,允许开发人员进行修改和扩展,而不需要修改工具源代码.同时,本文提出的转换原语的概念是对执行语义中特定操作的抽象和封装,其抽象层次高于源代码和字节码等概念.因此,本文方法重新定义模型转换的执行语义,比通过直接编码、修改编译器和字节码的方式复杂度更低.

代码生成(code generation)可以看做是一种模型到文本的转换,属于模型转换的一种,而 Prout 等人提出了一种语义可配置的代码生成技术[19].由于建模语言在不断地演化,当其语义发生变化时,相应的建模工具、模型分析器和代码生成工具都需要进行更新——这极大地增加了工具维护的开销.因此,Prout 等人设计了一个语义模板,利用它来描述建模语言中可能发生变化的语义成分.一旦建模语言的语义发生变化,只需要修改语义模板中对应的描述,然后产生相应的模型分析器、代码生成器等,从而保持相应工具的一致性.Prout 等人的工作与本

文工作的主要差别在于所解决的问题不同:Prout 等人的工作通过重新定义建模语言的语义来应对建模语言的演化;而本文的方法则是通过重新定义转换语言的语义,从而改变转换语言的执行逻辑和转换程序的行为,使得开发人员可以使用一种转换语言解决不同类型的问题.

组合转换(composite transformation)技术可以将一组相对简单的子转换组合起来构成复杂转换程序的技术.如果一个转换程序需要同时解决多个问题,每个问题使用不同的转换语言和工具来实现,那么组合转换技术可以帮助开发人员构造这个复杂的转换程序.虽然关于组合转换的研究已有一些成果[4,20−22],但与本文方法相比,这一技术还有以下两点不足:

1) 它无法降低学习负担,因为每个子转换仍然是用不同的语言和工具来实现,所以开发人员还是需要学习这些语言和工具;
2) 目前的组合转换技术还不能很好地解决不同的转换语言和工具之间的兼容性问题,例如,ATLFlow[21]和 Wires*[22]只能把 ATL 转换程序组合起来,UniTI[4]和 MCC[20]则需要手动编码来弥补转换程序之间的兼容性.

而本文方法通过重新定义某种转换语言的语义,使得利用一种语言可以解决不同类型的问题.这样既降低了开发人员的学习负担,也避免了各种兼容性问题.当然,组合转换技术与本文方法在适用范围上稍有不同:组合转换技术主要用于复用已有的转换程序,而本文方法还适用于开发新的转换程序.同时,用本文方法开发的转换程序也可以利用组合转换技术组装起来,两种方法的结合可以更好地发挥各自的优势.

10 结 论

本文的核心思想是通过配置转换语言的语义,从而扩充转换语言的能力,使得开发人员使用一种语言就可以解决多种类型的转换问题,从而降低学习负担和工具兼容性问题.本文的主要贡献在于:

1) 对转换语义进行了分析和抽象,提出了一种三层转换语义结构、一组转换原语以及一种基于 OCL 的 TSS 脚本来刻画转换语言的语义;
2) 通过一组案例对本文方法进行了验证,并讨论了本文方法的完全性、描述能力和复杂度.

在将来的工作中:首先,我们会进一步识别和丰富转换原语,提升本文方法的描述能力;其次,我们会将本文方法更多地应用于工程实践,从而在实践中对本文方法的可用性和效率等方面进行验证.

References:

[1] Miller J, Mukerji J. MDA guide. 2003-06-12. http://www.omg.org/docs/omg/03-06-01.pdf

[2] Liu H, Ma ZY, Shao WZ. Graph transformation based description language for model refactorings. Ruan Jian Xue Bao/Journal of Software, 2009,20(8):2087−2101 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3469.htm [doi: 10.3724/SP.J.1001.2009.03469]

[3] Giese H, Wagner H. Incremental model synchronization with triple graph grammars. Lecture Notes in Computer Science, 2006, 4199:543−557. [doi: 10.1007/11880240_38]

[4] Vanhooff B, Ayed D, Baelen SV, Joosen W, Berbers Y. UniTI: A unified transformation infrastructure. Lecture Notes in Computer Science, 2007,4735:31−45. [doi: 10.1007/978-3-540-75209-7_3]

[5] Jouault F, Kurtev I. Transforming models with ATL. Lecture Notes in Computer Science, 2006,3844:128−138. [doi: 10.1007/11663430_14]

[6] EMF project. 2013. http://www.eclipse.org/modeling/emf/

[7] Taentzer G. AGG: A graph transformation environment for modeling and validation of software. Lecture Notes in Computer Science, 2004,3062:446−453. [doi: 10.1007/978-3-540-25959-6_35]

[8] AGG homebase. 2013. http://user.cs.tu-berlin.de/~gragra/agg/

[9] OMG. Meta object facility (MOF) core specification 2.0. 2006. http://www.omg.org/spec/MOF/2.0/PDF/

[10] OMG. Object constraint language. 2006. http://www.omg.org/spec/OCL/2.0/PDF

[11] Steel J, Jézéquel JM. On model typing. Software and System Modeling, 2007,6(4):401−413. [doi: 10.1007/s10270-006-0036-6]

[12] OMG. Meta object facility 2.0 query/view/transformation specification. 2008. http://www.omg.org/spec/QVT/1.0/PDF/

[13] OMG. OMG unified modeling language superstructure. 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/

[14] Cabot J, Clarisó R, Guerra E, de Lara J. Analyzing graph transformation rules through OCL. Lecture Notes in Computer Science, 2008,5063:229−244. [doi: 10.1007/978-3-540-69927-9_16]

[15] Xiong YF, Hu ZJ, Zhao HY, Song H, Takeichi M, Mei H. Supporting automatic model inconsistency fixing. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering (ESEC/FSE 2009). New York: ACM Press, 2009. 315−324. [doi: 10.1145/1595696.1595757]

[16] Kolovos DS, Paige RF, Polack FAC. The epsilon transformation language. Lecture Notes in Computer Science, 2008,5063:46−60. [doi: 10.1007/978-3-540-69927-9_4]

[17] MediniQVT project. 2012. http://projects.ikv.de/qvt/

[18] ATL Project. Specification of the ATL virtual machine. 2005. http://www.eclipse.org/m2m/atl/doc/ATL_VMSpecification[v00.01].pdf

[19] Prout A, Atlee JM, Day NA, Shaker P. Semantically configurable code generation. Lecture Notes in Computer Science, 2008,5301:705−720. [doi: 10.1007/978-3-540-87875-9_49]

[20] Kleppe A. MCC: A model transformation environment. Lecture Notes in Computer Science, 2006,4066:173−187. [doi: 10.1007/11787044_14]

[21] ATL flow project. 2013. http://opensource.urszeidler.de/ATLflow/

[22] Rivera JE, Ruiz-González D, López-Romero F, Bautista J, Vallecillo A. Orchestrating ATL model transformations. In: Jouault F, ed. Proc. of the 1st Int'l Workshop on Model Transformation with ATL. Nantes, 2009. 34−46.

附中文参考文献:

[2] 刘辉,麻志毅,邵维忠.一种基于图转换的模型重构描述语言.软件学报,2009,20(8):2087−2101. http://www.jos.org.cn/1000-9825/3469.htm [doi: 10.3724/SP.J.1001.2009.03469]

何啸(1983－),男,北京人,博士,主要研究领域为面向对象建模,元建模,模型驱动的体系结构,模型转换技术.
E-mail: hexiao06@sei.pku.edu.cn

王瑞超(1987－),男,硕士生,主要研究领域为模型驱动开发,模型转换.
E-mail: wangrc10@sei.pku.edu.cn

麻志毅(1963－),男,博士,副教授,CCF 高级会员,主要研究领域为软件工程与支撑环境,软件建模技术,面向对象技术.
E-mail: mzy@sei.pku.edu.cn

邵维忠(1946－),男,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程环境,面向对象方法,软件复用,软件构件技术.
E-mail: wzshao@pku.edu.cn