CHAMELEON：OutSystems 实时双向转换

Hugo Lourenc。
o*OutSystems*hugo.lourenco@outsystems.com

Jo˜ao Costa Seco
里斯本新大学
joao.seco@fct.unl.pt

Carla Ferreira里斯本
新大学
carla.ferreira@fct.unl.pt

Tiago Sim˜
o
es
*OutSystems*tiago.simoes@outsystems.com

Vasco
S
i
l
v
a
*OutSystems*vasco.silva@outsystems.com

Filipe Assunç˜
ao
*OutSystems*filipe.assuncao@outsystems.com

Andr´e
M
e
n
e
z
es
*OutSystems*andre.menezes@outsystems.com

arXiv:2305.03361v1 [cs.SE] 5 May 2023

摘要——**在模型驱动工程中，模型的双向变换对于促进使用运行在不同抽象层次上的编辑器发挥着关键作用。这一点对于 OutSystems 这类工业级低代码平台尤为重要，因为此类平台拥有完善的工具生态系统，通过领域特定构建器和抽象模型查看器，为标准集成开发环境提供补充。**

我们介绍了 CHAMELEON，这是一种通过利用简单直观的组件模式，以声明式方式动态定义实时双向模型变换的工具。借助这种方法，我们可以逐步定义基于低代码元模型构建的抽象模型的视图路径和综合路径。

我们设计了一种标准的解析器生成技术，用于处理树状模型；该技术基于带有约束和名称绑定器的扩展语法定义。我们允许模型模式之间具有更大的重叠，同时仍可对其进行消歧，从而实现清晰的透镜式变换行为。

CHAMELEON在面向用户界面定义的OutSystems语言片段上进行了评估。为评估性能，我们使用了大量真实的OutSystems应用程序，其中约包含20万个UI组件，以及一个经过整理的控件模式数据库。在基准测试中，我们发现完整模型的最坏处理时间为92毫秒，仍适合交互式模型编辑器的操作。

索引词—**低代码、模型变换、虚拟化**

I. 引言

低代码框架的圣杯，是服务于广泛的专业人员群体。低代码使非专业开发者能够以平缓的学习曲线构建软件，同时为经验丰富的开发者提供更高效的环境。

低代码平台通常通过充分发挥特定平台或领域专用语言（DSL）的全部能力，在特定领域内开展工作。然而，采用 DSL 或低代码方法往往伴随着表达能力受限 [1]，，从而迫使开发者遵循一组编程模式。基于模型的方法以及采用单向模型转换的 DSL 通常具有一次性编写的特性，即只能在较高的抽象层级进行修改。当也可以在更具体的层级进行修改时，如果较高的抽象层级发生演变，这些修改就会丢失。在一种抽象表示的场景中当其展示给更高抽象层级的用户时，他们通常无法了解具体模型中那些未被更抽象模型涵盖的部分。在先前的工作中，我们为“应用程序构建器” [2]，[3]，[4] 定义了一种双向方法，该方法不受这种一次性写入限制 [5]，，但仍然忽略了应用程序构建器既不会生成、也不符合既定抽象定义的具体模型部分。我们在 [5] 中采用的方法是定义抽象模型之间的固定转换。这些转换针对每个“应用程序构建器”以及实际的 OutSystems 应用程序元模型分别进行定制。例如，有一个用于设计业务流程的构建器 [3] ，还有一个利用现有应用程序屏幕来定义应用程序导航的构建器 [2]。这种方法仅适用于异步编辑流程，即将模型加载到构建器中，进行修改，然后再将其转换回原始元模型。

本文提出了一种多层双向、基于模型的设计，通过渐进式方法高效地抽象具体的（原生）模型。尽管我们的方法可应用于模型的任何部分，但本文聚焦于用户界面，其中具体组件称为（原生）控件，抽象组件称为虚拟组件。我们从当前的OutSystems应用程序模型开始，在该平台的后台应用程序中，由专家逐步定义一组控件模式。呈现给用户的抽象模型以虚拟控件树的形式存在，其规模明显小于原生控件树，但仍能捕获所有未匹配任何指定虚拟控件模式的组件。这组未匹配且未被抽象的组件不会被隐藏，而是以边界清晰的原生控件形式继续呈现给用户。因此，我们的方法支持渐进式模型变换，使用户能够定义新的虚拟控件模式，以捕获具体模型中越来越多的部分。通过逐步处理，原生模型中复杂的控件树被抽象为更加直观、简洁的虚拟控件树。

我们特别关注解析过程的效率，以便应用程序模型中的更改能够立即在抽象层次上体现，反之亦然。我们使用解析

生成器技术来高效扫描并抽象树状模型。不同于传统的双向变换引擎逐一查询模型中的单个基于图的模式，[6], [7], 我们采用解析方法，在一次遍历中处理所有模式。通过改进一种现有但速度较慢的检测方法，我们旨在实现模型的实时编辑体验，支持在不同抽象层次进行更改。

CHAMELEON 改进了 OutSystems 当前的虚拟化技术，这些技术加载并处理一个典型应用程序模型需要几十秒。相比之下，新的虚拟化过程处理每个应用程序模型所需的时间不到 100 毫秒，处理每个应用程序屏幕所需的时间不到 10 毫秒。此外，我们的方法可以按更小的增量运行，只解析并显示当前与用户相关的屏幕。总体目标是支持模型编辑器在不同抽象层级上实时运行，同时涵盖整个低代码元模型，并允许不同开发者角色进行模型可视化和编辑。

我们为虚拟组件引入了一种元模型，它同时定义了抽象层中要捕获的模式，以及将虚拟组件综合到原生模型中的综合过程。我们使用组件标识符来绑定不同定义，并表达原生模型与抽象模型的属性之间的关联。根据虚拟组件定义，我们生成带有条件 [8], [9] 的属性语法。虚拟组件可以由多个不同模式构成，例如，“布尔输入框”虚拟组件可以表示为复选框、开关，或由两个按钮组成、分别代表“真”和“假”备选方案的控件组。模式定义可能会重叠——两个成组的按钮也可以视为“枚举输入框”的一种特例，用于从一组固定的备选方案中进行选择。

生成的语法本质上存在歧义，我们通过构建一个自定义的广义 LR 解析器来解决这一问题 [10]。值得注意的是，Tomita 的 GLR 解析器 [10] 会并行探索所有备选方案，而我们则按顺序探索备选方案，并优先处理移进-归约冲突，以获得最长序列（从而确保更高的虚拟化层次）。我们的实现采用回溯来恢复解析错误，并继续尝试下一个备选方案。

为描述原生控件被虚拟组件替代的过程，我们会通过动态添加类来扩展原始（原生）元模型，以表示虚拟组件。因此，抽象元模型是原生元模型的真正扩展，因为它能够完整表示原生控件应用程序。抽象模型的编辑器仍可表示并预览原生控件，分别通过完整的控件树和原生控件预览功能实现。

我们使用包含 239 个应用程序模块和约 4K 个屏幕（含约 200K 个 UI 组件）的基准测试来评估 CHAMELEON。我们使用了一个包含 106 种模式、分组在 31 个虚拟组件中的数据库。对于完整的应用程序模块，我们发现该方法的效率和有效性均呈线性趋势。在实际应用中，解析单个屏幕只需一次可忽略不计的固定开销，即可完成屏幕虚拟化，解析时间小于 10 毫秒，因此可以将其集成到抽象模型的交互式协作编辑器中。

总之，这项工作的主要贡献如下：

- 第 II 节介绍的系统架构支持通过动态扩展原始元模型，逐步定义新的模式，并支持在多个抽象层次上编辑模型。
- 一种用于定义虚拟组件的语言，它同时定义了要捕获的模式以及新组件的综合过程，详见第 III 节。
- 一种多层双向的基于模型的设计，能够以渐进的方式高效地抽象原生模型。这意味着解析过程采用尽力而为的方式，利用所有可用的虚拟组件定义（第 IV 节）。
- 一种合成技术，支持互换使用虚拟组件的控件模式（第 V 节）。

我们最后在第 VI 节使用一项基准测试来评估我们的方法，并列出留待未来工作解决的开放问题（第 VIII 节）。

总之，其影响力体现在该方案已列入公司的路线图，计划集成到其主要产品中。

II. 系统架构

系统组件如总体架构图（图1）所示，涵盖三种不同角色的功能：使用当前开发环境和原生模型创建应用程序的 *OutSystems* 开发者；使用原生模型定义模式，并对其进行分组和参数化以定义虚拟部件的 模式设计者；以及公民开发者 [11], [12]。公民开发者是我们工作的核心，他们使用虚拟部件创建应用程序。他们使用虚拟模型编辑器，该编辑器呈现简化但无损的小部件树，支持编辑和创建新的虚拟部件实例、以只读方式预览原生控件（虚拟元模型的一部分），还提供可直接转换原生模型的选定操作，并将这些操作编码到编辑器中。出于实用和历史原因，此类底层硬编码操作可用于支持虚拟模型编辑器中的基本构建操作，而这些操作不涉及创建控件（例如，以预定义方式更改数据模型）。

虚拟元模型以原生元模型的定义和模式设计者定义的虚拟组件集合为输入生成。图中左侧的粗箭头表示了这一过程。随后，虚拟模型编辑器使用虚拟元模型来支持交互。正如预期，原生元模型是原生模型编辑器操作的基础。虚拟部件的定义也用于生成虚拟化（解析）和合成操作。

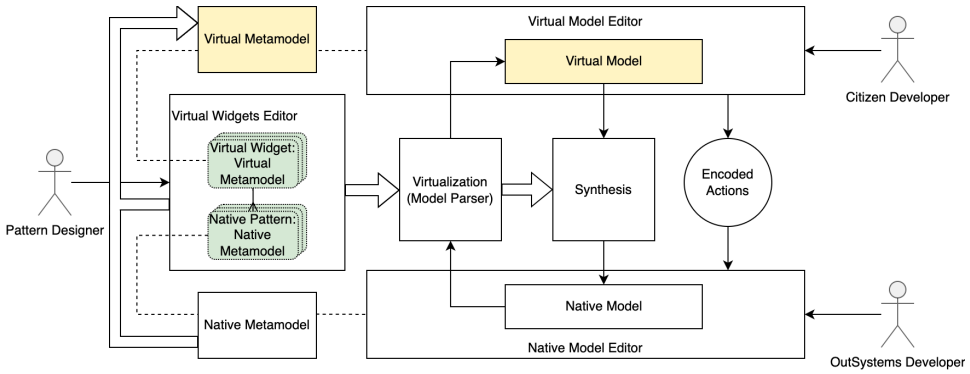


图1：所提出方法的概述

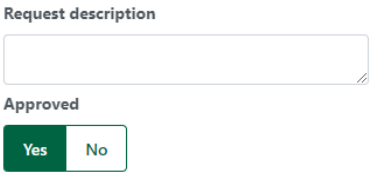


图2：请求编辑表单

在第III节中，我们将介绍一个案例研究，以支持对我们的解析方法（见第IV节）以及合成过程（见第V节）的详细说明。

III. 案例研究：用户界面模式

我们的工作重点是通过引入更高层级的构造来简化用户界面的构建；与底层模型——低代码 OutSystems 应用模型相比，这些构造更易于操作和理解。本节将介绍一个贯穿全文的示例：在一个虚构的企业应用程序中，用于编辑请求的简单表单（图2）。该表单允许填写请求的描述以及审批信息。图 3 展示了用于定义表单的元模型，它是实际 OutSystems 元模型的一个子集。表单 由一棵小部件树组成。输入框 小部件用于获取用户输入的数据，并将其绑定到低代码应用程序中的状态变量。输入框 小部件根据数据类型进行专门化：类型文本区域 用于文本数据，类型复选框 和开关适用于布尔数据，而类型按钮组 用于呈现多个备选方案，每个备选方案都由一个按钮组项 小部件表示。容器 小部件是一种布局组件，用于对其他小部件进行分组。标签 小部件用于实现可访问性，描述与之关联的输入控件。

在清单 1 中，我们展示了图2的模型，并在图4a中描绘了控件树的可视化表示。出于将在下一节中变得明确的原因，我们选择使用 XML表示法来呈现所有模型。

在我们的示例中，表单中的“已批准”输入框用于获取布尔数据。开发者选择了用户友好的表示方式——包含两个备选方案（是或否）的按钮组控件，但也可以使用复选框控件或开关控件。这个简单的示例说明，通常，同一信息有不止一种等价的表示方式（参见图5）。

清单 1：表单模型

```
1  <表单 ID="f">
2    <容器标识符="c1">
3      <标签 Id="l1" 控件="ta">
4        <文本 ID="t1" 样式="bold"值="请求
             描述"/>
5      </标签>
6      <文本区域 ID="ta"
           变量="Request.Description"/>
7    </容器>
8    <容器 ID="c2">
9      <标签 Id="l2" 控件="bg">
10       <文本 Id="t2" 样式="bold"值="Approved
             由经理"/>
11     </标签>
12     <按钮组 Id="bg"
           变量="Request.Approved">
13       <escape></escape>按钮组项 ID<escape></escape>"g1" 值<escape>=</escape>"true"<escape></escape>
14         <文本 Id="t3" 值="是"/>
15       </按钮组项>
16       <按钮组项 ID="g2" 值="false">
17         <文本标识="t4" 值="否"/>
18       </按钮组项>
19     </按钮组>
20   </容器>
21 </表单>
```

快速检查清单 1（其中包含示例表单的原生模型）后可以发现，其中有大量重复和不必要的细节，开发者不应为此操心。两个输入项都遵循相同的基本模式：一个标签包含一个文本 控件，其中显示描述性文本，后面跟着一个输入框 控件，该控件复杂度不一，并绑定到一个变量。通过引入高层元素（我们称之为虚拟组件），并为其提供简化的应用程序接口，我们可以减少开发者的工作量：开发者只需决定要绑定哪个变量。随后，可以手动或自动从一组选项中选择具体的控件表示，并在需要时根据变量推断标签文本的初始值。

在图 6 中，我们展示了虚拟控件的元模型，它是图 3 中原生元模型的扩展。在必要时，为避免混淆，我们将具体元模型及其控件称为原生元模型

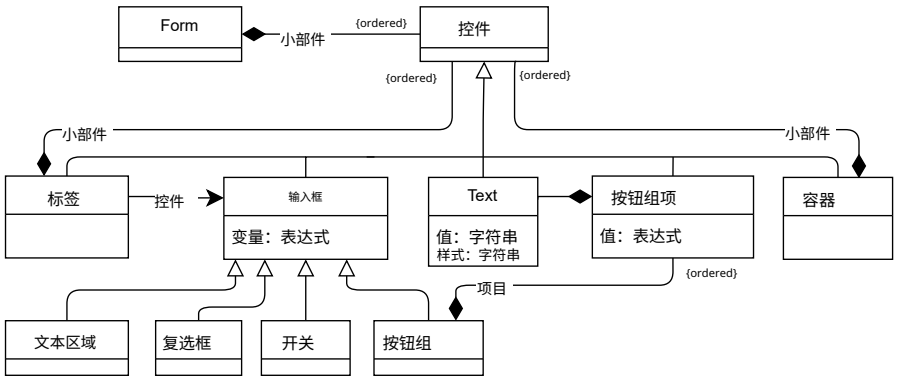
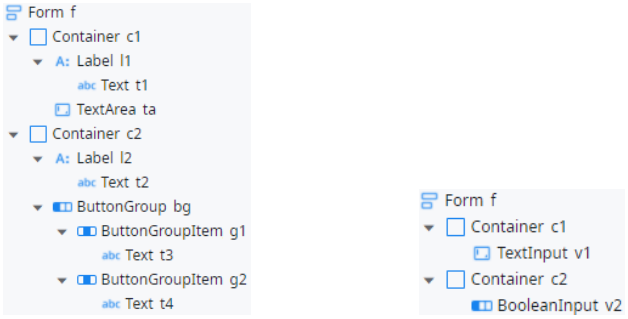


图 3：小部件元模型



(a) 原生树　　(b) 虚拟树。

图 4：部件树



图5：表示布尔输入的三种方式

和原生控件。在我们的示例中，虚拟元模型包含三个新类[1]：文本输入、布尔输入和枚举输入。这些控件中的每一个都表示一个带标签的输入，因此包含变量 和标签 属性。每个元素可能对应于若干不同的具体模式之一，这些模式由模式 属性标识。虚拟元模型包含原生元模型中各类的略微修改版本，其中聚合关系经过放宽，可以引用控件 基类，而不是其某个子类[2]。例如，可以比较按钮组虚拟控件（图 6）及其原生对应控件（图 3）。下一节将详细解释为什么这一变更能够实现原生模型与虚拟模型之间的渐进式转换。

利用新型虚拟部件文本输入 和布尔输入，我们得到了一个简化的虚拟模型（清单 2）。在集成开发环境中并排显示原生控件树与虚拟部件树（图 4），进一步凸显了复杂度的降低。

[1]新类在元模型图中以黄色背景表示。[2]修改后的关系以红色表示。

清单 2：表单虚拟模型

```
1  <表单 ID="f">
2    <容器 Id="c1">
3      <文本输入 Id="v1" 标签="Request
         description" 模式="1"
         变量="Request.Description"/>
4    </容器>
5    <容器 Id="c2">
6      <布尔输入 Id="v2" 标签="由.....批准
         经理" 模式="3"
         变量="Request.Approved"/>
7    </容器>
8  </表单>
```

虚拟部件由属性列表和一个或多个模式定义（见表 I）。这些模式用于在虚拟模型中表示虚拟部件、查看/解析从原生控件到虚拟部件的转换，以及合成原生控件。每种模式都由一棵简化的原生控件树组成，其中仅包含匹配该模式所需具有特定值的属性。名称带有前缀`Default` 的属性仅用于合成阶段，不参与模式匹配阶段。模式还必须指定虚拟部件属性与原生控件属性之间的绑定。表 I 的最后一行以可视化方式展示了该模式的一个实例；此行是为了方便读者而添加的，并不属于模式定义的一部分。

在表 I 中，我们介绍了布尔输入虚拟部件的定义。该虚拟部件声明了两个属性：变量和标签，并由三种不同的模式组成。通过提供默认值表达式，可以将属性设为可选；该表达式仅在合成操作期间使用。在我们的示例中，属性标签 的默认值取自值 属性，如`GetLabelFor(Value)` 表达式[3]所指定。模式 #1 由一个包含文本 控件的标签 控件（其标识符为`t`）以及一个标识符为`i`的复选框 控件组成。该模式规定，虚拟属性变量 绑定到复选框的变量（即变量 `= i.Variable`），并且

[3]函数 `GetLabelFor` 会根据模型表达式计算出合适的易读标签。

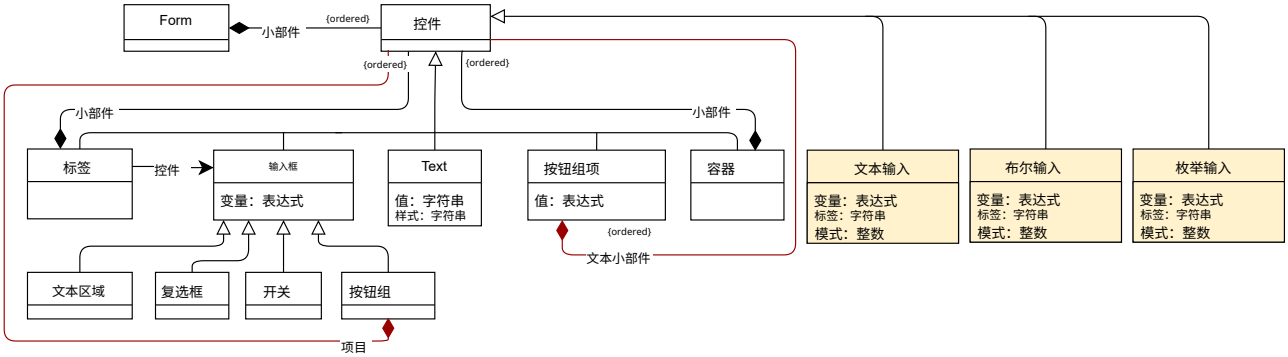


图 6：虚拟部件元模型

标签 属性绑定到文本的值（即标签 `=t.Value`）。请注意，我们没有指定任何文本 控件的值，因为任何值都可以接受[4]。但是，对于模式 #3 的两个按钮组项 控件，我们明确将其值分别指定为`true` 和`false`，因为我们只希望在该特定情况下识别该模式。

我们会检查模式是否符合虚拟控件定义（模式必须为虚拟控件的每个属性提供绑定）和原生元模型，以确保实例化虚拟控件定义后，始终能得到正确的原生模型。

这种虚拟组件表示方式足够灵活，能够支持简便且安全的模型变换。例如，原生模型最初可能由一组符合模式 #3 的小部件定义，然后解析为虚拟模型中的一个 布尔输入 虚拟部件。开发者可以迭代地选择改用模式#1，最终再将其变换回原生模型，得到一组完全不同的原生控件。这样，用户便能以更抽象的方式构建应用程序，并进行配置，而无需复杂地编织属性、值和小部件。

IV. 基本模型解析器

虚拟化过程采用基于解析器的方法实现。在这项工作中，我们构建了一个解析器生成器，它以一种语法作为输入，该语法将虚拟元模型与一组虚拟控件定义结合起来，从而生成一个用于 [10] 具体模型的通用解析器。我们示例中使用的输入语法如图7所示。该语法是根据虚拟元模型的定义（图 6）和虚拟控件的定义机械生成的。请注意，这个上下文无关语法的规则*BooleanInput.3* 和 *EnumInput.2*之间存在歧义。鉴于语法存在歧义，有必要使用一种能够探索单词 [13]所有可能推导树的解析算法。我们改编了 Tomita [10]提出的解析器生成算法，该算法为这类语法提供了一种高效的解析算法。

[4]我们为模式 #3 的两个文本小部件提供了默认值，但如前所述，这不会影响模式匹配阶段。

在图7所示的语法定义中，非终结符以斜体表示。终结符表示原生模型实例，以 XML标签表示，并带有可选的下标标识符和可选的上标条件。除了元模型主要元素的结构外，该语法还包含各虚拟控件中每种模式对应的规则（规则文本输入.*1*、布尔输入.*1*、布尔输入.*2*、*BooleanInput.3*和枚举输入.*1*），以及每种扩展原生组件对应的规则。

虚拟部件的规则可直接根据其定义得出。表 I 中的定义对应规则*BooleanInput.1*、*BooleanInput.2*和 *BooleanInput.3*。为简洁起见，我们省略了`TextInput` 和 `EnumInput` 虚拟部件的定义，但在语法中包含了相应的规则。模式可以通过提供*Id* 属性来捕获其匹配的模型控件的标识符；该属性对应于语法规则中的终结符下标，并在语义动作中将标识符绑定到变量。属性值限制会形成解析过程中必须满足的条件。在语法中，条件以右上标表示。例如，在*BooleanInput.3* 规则中，`<ButtonGroupItem>` 终结符的第一次出现标注了`Value = "true"`。解析时，如果当前词元不满足该条件，则立即拒绝该规则。如果此时没有其他规则处于激活状态，解析器就会回溯到最近的前一个决策点。这种方法与 pred-LR(k)解析器的谓词类似，但限制较少：当当前激活规则中有多个谓词时，该解析器要求其中恰好有一个谓词为真。 [8],

该语法还包括（扩展）原生控件的规则，以支持逐步扩展虚拟组件集。这确保每个原生模型都能被解析为虚拟模型，并对与已定义虚拟组件匹配的模式进行抽象。查看我们的虚拟模型示例（清单 2）可以发现，同一个模型中混合了扩展原生控件（表单 和 容器）与虚拟组件（文本输入.*1*和 *BooleanInput.3*）。

虚拟控件的定义中，模式可以相互重叠。再加上纳入了

表 I：布尔输入模式

布尔输入（表达式变量，字符串标签 = GetLabelFor（值））

模式#1 变量 =i.变量 标签 = t.值	**模式#2** 变量 =i.变量 标签 = t.值	**模式#3** 变量 = i.变量 标签 = t.值
<标签> <文本 ID="t"/> </标签> <复选框 ID="i"/>	<标签> <文本 ID="t"/> </标签> <开关 Id="i"/>	<标签> <文本 ID="t"/> </标签> <按钮组 Id="i"> <按钮组项 值="true"> <文本 默认值="是"/> </按钮组项> <按钮组项 值="false"> <文本 默认值="否"/> </按钮组项> </按钮组>
Approved	Approved	Approved Yes No

Form ::= <表单>控件* </表单>

控件 ::= 虚拟控件
| 原生控件

虚拟控件 ::=
- <标签> <文本/>$_t$ </标签> <文本区域/>$_i$ (文本输入.1)
- <标签> <文本/>$_t$ </标签> <复选框/>$_i$ （布尔输入.1）
- <标签> <文本/>$_t$ </标签> <开关/>$_i$ (布尔输入.2)
- <标签> <文本/>$_t$ </标签> <按钮组>$_i$ 值 = *"true"* <文本/> </按钮组项> <按钮组项>值 = *"false"* <文本/> </按钮组项> </按钮组> (BooleanInput.3)
- <标签> <文本/>$_t$ </标签> <按钮组>$_i$ (<按钮组项文本/> </按钮组项>)* </按钮组> (枚举输入.1)

NativeWidget ::=
- <按钮组>控件* </按钮组> (按钮组)
- <按钮组项>控件* </按钮组项> (按钮组项)
- <复选框/> (复选框)
- <容器>控件* </容器> (容器)
- <标签>控件* </标签> (标签)
- <开关/> (开关)
- <文本/> （文本）
- <文本区域/> （文本区域）

图7：虚拟控件语法

原生控件的规则，这会导致文法具有高度歧义。例如，考虑清单 1 第 9 至 19 行的控件序列。该序列可以用多种不同方式解析，即使用规则 *BooleanInput.3*、*EnumInput.1*，或 <标签> 后接 <按钮组>。如前所述，我们采用了一种受 Tomita 的 GLR 解析器启发的方法 [10]。Tomita 会并行探索所有解析树，而我们每次只保留一棵活动解析树，但会记住移进-归约冲突和归约-归约冲突情况下先前作出的选择，从而允许回溯。请注意，我们的目标是尽可能快速地找到一棵解析树，而非找到所有可能的解析树。根据我们的基准测试（详见第 VI 节），每处理一个词法单元，回溯发生的概率为 2.6%，这在实践中使解析器能够快速收敛。遇到移进-归约冲突时，我们会优先尝试移进路径，因此优先选择较长的

序列。对于归约-归约冲突，我们根据定义虚拟组件时确定的显式顺序，优先探索优先级较高的规则。此外，虚拟组件的规则优先级高于原生控件的规则。

每条虚拟部件规则的语义动作（我们未在图7中表示）相当于创建对应虚拟部件的一个实例。该实例的属性根据模式属性绑定从原生控件中提取。例如，规则 布尔值.*3* 用于归约清单1第9至19行的控件序列。根据模式 #3（表 I）中指定的属性绑定，我们得到终结符与模型控件之间的以下映射：{t ↦ t1,i ↦ bg}。此外，这种关联还产生了虚拟部件属性的以下值（清单 2 第6行）：

- 变量 = Request.Description

- 标签 = "请求描述"

变量 的值计算如下：1) 模式#3 指定绑定 变量 = `i.Variable`，2)模式控件 `i`与原生控件 `bg`匹配，3) 在绑定中将`i` 替换为 `bg` ，我们得到 变量 = `bg.Variable` ，4) `bg.Variable` 的值为 `Request.Description`。

鉴于处理具体低代码模型的特殊情况，以及需要采用特殊的冲突解决策略和回溯机制，我们开发了自己的GLR解析器生成器。由于以下特定应用场景的特点，直接使用任何广泛可用的解析器生成器都存在困难：

- 文法歧义：大多数解析器要求语法不存在歧义。如上所述，我们的文法始终存在歧义，这是有意的设计。
- 歧义解决与回溯：移进-归约和归约-归约冲突并非以单一方式解决。相反，它们代表按顺序探索的备选方案，我们可能需要从中进行回溯。
- 适应新的虚拟控件定义：我们的语法及相应的解析器会即时生成（在内存中）。当模型在集成开发环境中加载时，会获取最新的定义。
- 终结符中的条件：模式中的属性值限制通过相关终结符中的条件来表示。对解析时间的严格限制要求尽可能早地对这些条件进行求值，而不是等到语义动作（可能）拒绝该规则时才进行求值。

在终结符中包含条件，对于精简文法和提高解析器效率至关重要。另一种表示条件的方式是克隆相关的终结符。例如，给定终结符 按钮组项 的现有条件，我们可以将其拆分为三个不同的终结符： <按钮组项_真>、<按钮组项_假>，以及<按钮组项_其他>（加上相应的闭合标签），分别表示 值 属性的三种情况：真、假，以及任何其他表达式。规则 *BooleanInput.3* 可以直接改为使用新的终结符，但规则 按钮组项 则需要拆分为三条不同的规则，每条规则使用一个新终结符。克隆终结符会导致文法组合爆炸，并使解析器中的状态数量呈指数级增长。

请注意，根据构造，终结符上的条件始终采用属性 = 值的形式。我们的解析器生成器利用这一特性，高效地生成解析表。解析表中的一个条目会根据当前词元，为每个终结符／非终结符指定要执行的操作（移进／归约／无操作）。在我们的表示中，每个操作都可能包含一个决策表，例如表 II 所示，其中展示了针对示例语法的部分解析表。当我们在表 II 的状态 32 中处理完以下序列时

表II：解析器表的一个片段

	词法单元	
状态	</按钮组>	<按钮组项>
32	Reduce(枚举输入.1)	Shift(值:{true $\mapsto$ 37, other $\mapsto$ 36 })
48	Reduce(枚举输入.1)	Shift(值:{ false $\mapsto$ 50, other $\mapsto$ 36 })

<标签> <文本/> </标签> <按钮组>

此时，根据解析表，唯一预期的词法单元是 </`ButtonGroup`> 和<`ButtonGroupItem`>[5]。第一个词法单元会触发使用规则 *Enum.Input.1* 执行归约动作，并创建一个*EnumInput* 虚拟部件。第二个词法单元则更有意思。请回想一下，*ButtonInput.3* 规则附带了一个条件：第一个*ButtonGroupItem*的值 = `"true"` 。表 II 中的解析表通过指定将执行移进操作，并根据值 属性的值确定目标状态，记录了这一信息：如果值为*true* ，则移进到状态 37；否则移进到状态 36。

V. 原生模型合成

如表 I 所示，虚拟控件定义包含两方面内容：定义用于生成解析器的模式，该解析器负责虚拟化原生模型并实例化虚拟组件；同时定义如何将每个虚拟组件映射回原生模型，在本例中，即为同一虚拟组件指定多个备选映射。在以下情况下，虚拟模型会被转换回原生模型（即必须对原生模型进行合成）：1）实例化虚拟组件；2）切换虚拟组件的模式；3）更改虚拟组件属性的值。

虚拟组件定义包含一组属性定义，这些属性通过原生控件模式中的注释以及各模式中用于建立关联的方程，与原生控件的属性绑定。以表 I 中定义的虚拟控件布尔输入为例，其参数为类型为表达式的变量 和类型为字符串的标签 。该定义中的方程可以双向解读——既可以从虚拟控件映射到原生控件，也可以反向映射。在虚拟化阶段，模式 #1 中的方程变量 = i.变量表示，虚拟控件的变量 属性值取自由 i 标识的原生控件变量属性值，该原生控件为复选框 。在合成阶段（当创建新的虚拟控件实例，或开发者更改现有虚拟控件的模式时发生），同一方程表示，新创建的、由 i 标识的原生控件复选框 的变量 属性值，取自虚拟控件的变量属性值。

[5]任何其他词法单元都会导致解析错误，并使解析器回溯到最近的决策点。

表 III：为扩展了 31 个虚拟小部件的 OutSystems 元模型生成的解析器统计信息

语法规则数	188
解析器状态数	926
移进-归约冲突数	216
归约-归约冲突数	982

这些方程为合成的原生控件的部分属性提供值。其余属性则根据模式中的指定进行设置，可以采用匹配值或默认值。例如，在实例化 布尔输入 的模式#3 时，第一个按钮组项小部件的值 被设置为`true`，其文本 小部件的值 则被设置为`Yes`。

此外，如果目标模式中存在对应控件，那么更改模式时会保留所有原生控件属性。例如，考虑虚拟模型清单 2，并将 id 为 *v2* 的 布尔输入实例从模式#3更改为模式#1。模式#3 中 id 为 t 的控件对应于 id 为 $t3$ 的原生控件（清单 1）。模式#1 中也包含一个 id 为 t 的控件。因此，合成的 文本控件的 样式属性将具有非空值，尽管该模式未为此属性提供值。清单 3 展示了生成的原生模型。

清单 3：将 布尔输入 从模式#3 更改为模式#1 的结果（片段）

```
1     <标签 ID="l2" 控件="bg">
2       <文本 ID="t2" 样式="bold"
            值="经理已批准"/>
3     </标签>
4     <复选框 ID="bg"
          变量="Request.Approved"/>
```

当然，这种双向读取能力限制了定义中可使用的方程类型。目前，我们将方程限制为直接读取/设置虚拟组件和原生控件的属性。

VI. 评估

我们设计了一项性能基准测试，并选取了一组大型示例来检验我们方法的效率和有效性。在这一批测试中，我们使用了真实的 OutSystems 元模型（包含 30 个原生控件组件类），并以 31 个虚拟控件定义和总计 106 种模式对其进行了扩展；这对应于测试时逐步构建的模式数据库的全部内容。我们使用这组虚拟控件定义，进一步评估解析器生成过程的正确性，并抽样检验在不同模式之间切换的综合过程的正确性。我们先进行预热，然后针对每个模块重复实验 9 次，剔除 2 个最高值和 2 个最低值，以减少噪声并单独测得解析时间。如表 III 所示，生成的解析器具有很高的歧义性。

为了评估我们方法的效率和有效性，我们设计了一项简单实验，在大型代码库中使用虚拟控件定义。样本包含 239 个 OutSystems 模块，其中有 3 718 个屏幕，共计 198 072 个原生控件。图 8 使用双对数刻度展示了每个模块中虚拟化（解析）时间与原生控件数量的关系，以便同时呈现规模较小和较大的应用程序。我们发现，解析时间（毫秒）呈明显的线性趋势（$t = 0.0336 * n - 0.0277$），每解析一个新控件，解析时间就会增加 $33.6\mu s$（t）n。线性回归的决定系数（$R^2 = 0.611$）以及 p–值 < 0.001表明拟合效果良好，但仍有一些情况的耗时与原生控件数量并非线性相关。图 8 显示，约有 14% 的情况比趋势线所示更慢，这可从图中明显可见的离群值看出。在原始数据中，我们观察到某个模块的最长解析时间约为 $92ms$，仍在交互式系统可接受的时间范围内；在这类系统中，$100ms$ 是用户在用户界面中实时操作对象时所能感受到的时限参考值 [14]。在大多数情况下，耗时相当短；出现离群值的原因与极端的歧义情况有关，这些情况会在解析过程中触发过多回溯。对于大多数较直接的歧义情况，广义 LR 解析算法会立即加以处理，并被设置为选择最长的原生控件序列。

就有效性而言，我们从图 9 中观察到，经过抽象处理的小部件平均数量（v）呈现出更加稳定的线性趋势（$v = 0.444 * n + 1.3$），且决定系数较高（$R^2 = 0.977$），并且p–值 < 0.001。该比率基本保持恒定，接近 1:2，这意味着在抽象模型中，小部件的平均数量减少了一半，而实现这一效果仅需定义 31 个虚拟部件。随着逐步添加新的虚拟部件定义，预计这一比率还会进一步提高。

如表 III 所示，我们为基准测试生成的解析器存在大量歧义。大量归约-归约冲突是由于虚拟部件和原生控件的规则相互重叠所致。然而，高度歧义对解析器性能的实际影响很小。表 IV 中的统计数据支持这一结论。在运行完整基准测试时，解析器共处理了 307 884 个词法单元。在回溯从未发生的理想情况下，移位操作的数量应与之相同，即每个词法单元恰好处理一次。可以看出，移位操作总数更高（318 515），但差距不大——平均而言，解析器对每个输入令牌执行 1.035 次移位操作。7.95% 的词法单元会遇到冲突。请注意，冲突是指对于同一个前瞻词法单元存在多个解析器操作；我们的解析器通过优先选择一条规则，并记录其余规则以备需要回溯时使用，来处理这种情况。回溯发生率较低，为 2.57%。这一数值较低是意料之中的，因为并非所有冲突都会导致回溯。回溯的影响很小。共有 10 631 个词法单元需要

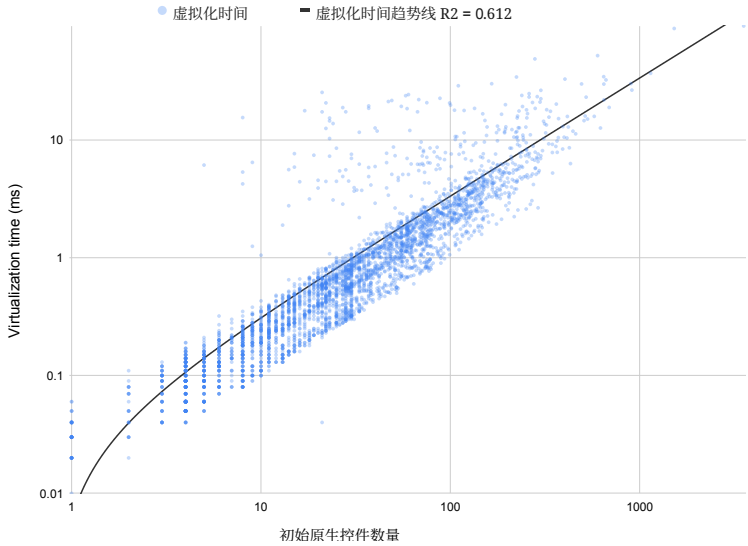


图 8：原生控件数量对应的解析时间（双对数）

表 IV：基准解析统计数据

输入令牌总数	307 884
移位操作次数	318 515
冲突总数	24 477
回溯操作次数	7 927
回溯令牌总数	10 631
每个输入令牌的平均移位操作次数	1.035
每个输入令牌的冲突率	7.95%
每个输入词法单元的回溯率	2.57%
每次回溯操作的平均回溯令牌数	1.341

回溯处理，即重新处理。平均而言，回溯会导致退回 1.341 个词法单元。

总之，这些结果表明，我们的渐进式方法适用于模型的实时双向转换，能够让用户同时编辑处于不同抽象层次的应用模型，而不会覆盖或丢失模型可视化或合成中的任何细节。

VII. 相关工作

双向转换已应用于各种上下文 [15]。例如，在数据库领域，关于视图更新机制已有大量研究 [16]，[17]，[18]，[19]。因此，在许多场景中，关注这些转换的效率至关重要。

在模型驱动工程（MDE）的语境中，双向转换发挥着关键作用 [6], [20]，它是定义灵活的模型到模型转换（M2M）的基石。Guerra 等人 [20] 认为，M2M转换通常通过专门的M2M转换语言来表达，并且主要聚焦于实现阶段。他们的工作 [20] 通过提出一组语言来覆盖完整的转换开发生命周期，从而解决了这一问题。我们的方法可以被视为一种我们工具中所提出方法的一个实例：我们涵盖了转换开发的整个生命周期阶段，从需求到设计和验证。通过提供后台和元模型来定义虚拟组件，我们让模式设计人员能够扩展转换；通过在此编辑器中提供验证，我们确保生成的转换在每个步骤中都是正确的。

将 CHAMELEON 与 EMF [21]或 MPS [22], [23] 等模型到模型转换方法进行比较，缺点在于这些方法将转换定义为语言本身之外的内容（参见 ATL [24], [25]）。CHAMELEON 使用原生元模型，并扩展了名称绑定器和方程，以表达虚拟化与合成所需的模式。

三元图文法（TGGs） [26]，[27] 是一种基于规则的方法，旨在定义基于图的语言之间的转换，同时维持输入元素与输出元素之间的双向关系。TGGs 对于同步不同模型以及确定源模型与目标模型之间的兼容性尤其有用；在 CHAMELEON 中，这种兼容性（大体上）通过构造得到保证，因为虚拟组件的元模型涵盖了原生虚拟组件的元模型。然而，TGGs 更关键的问题在于效率：TGGs 的解析算法十分复杂，简单粗放的解决方案可能导致指数级分析 [27]。即使限制 TGG 的表达能力，专用解析器的复杂度至多也只能达到多项式级。相比之下，CHAMELEON 采用线性解析算法，这得益于它处理的是抽象语法树。在 CHAMELEON 的低代码环境中，存在类似流程图（动作流）的基于图的结构，但它们仍可表示为结构化的（树状代码），因此能够得到高效处理。

Viatra [28]，[29] 采用图转换和抽象状态机的形式化数学技术。该框架可以通过使用

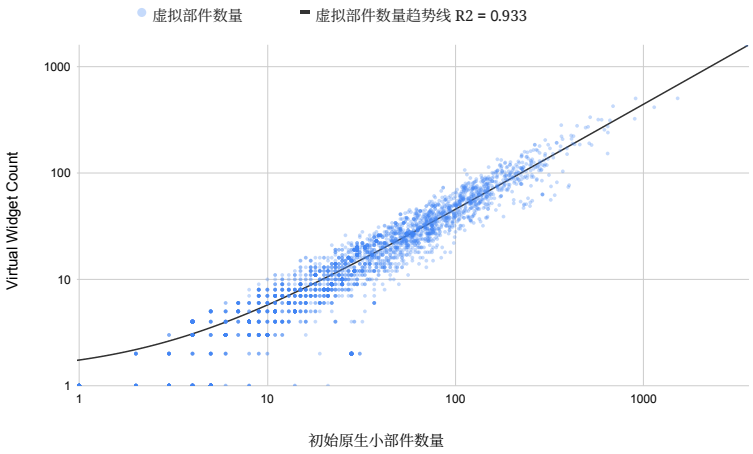


图 9：每个初始原生小部件所解析的虚拟组件数量（双对数坐标）

增量模式匹配方法，在进行增量修改后确定并更新匹配结果。然而，对于大型模型，转换的执行时间可能较长，从而限制了其在交互式系统中的应用。

QVT [30], [31], [32] 是一系列 MDE 语言，主要为定义不同类型模型之间的模型转换提供标准。QVT relations [30]支持复杂的对象模式匹配和对象模板创建，并将不同域之间的对应关系指定为关系。尤其是，QVT relational 采用一种策略，同时将转换规则的所有匹配应用于给定输入模型，然后合并生成的输出模型元素。然而，这种方法容易出错，并且要求深入理解规则如何匹配及彼此交互 [33]。因此，现有的 QVT relational 工具在处理相同输入时，可能会产生明显不同的结果 [7]。

Theworks [34], [35] 解决并发模型同步问题，即转换可能同时应用于源模型和目标模型。目前，CHAMELEON 不支持此功能，但我们正在探索如何将并发模型同步纳入我们的解决方案。

VIII. 未来工作

虽然 CHAMELEON 为模型驱动工程提供了一种颇具前景的方法，但需要注意的是，它并非一种完全通用的解决方案。具体而言，CHAMELEON 并未涵盖本质上属于图结构的模型，因为这类场景通常不会出现在 OutSystems 环境中。

我们在这项工作中施加的约束之一是，模型只能对枚举值使用简单的解析器条件。尽管我们的实现能够处理一般条件，但我们发现，这一限制可能导致解析器中的状态组合爆炸，从而妨碍系统的交互性。然而，在实践中，这一限制表明，它并不会限制虚拟组件的定义。

在本提案中，一个可能影响结果的尚未解决的问题是解析过程中发生的回溯次数。未来，我们计划深入研究必须进行回溯的情况，并寻找避免回溯或对其进行优化的方法。

我们还指出了进一步整合和扩展这项工作的若干可能方向。其中一项最重要的扩展是允许使用此前开发的模板语言 OSTRICH [36], [37], [38] 来定义虚拟控件。通过模板实现模型的可视化，可以提高代码的复用性，并将该系统与平台中已有的精选模式数据库关联起来。此外，绑定机制也可以从使用模板参数中受益，而不是直接将名称绑定到模型上。

IX. 结论

在这项工作中，我们研究了如何应用标准解析技术来实现模型抽象，并定义相互关联的原生模式以定义虚拟部件。虚拟部件在元模型中定义为原生元模型的扩展，可简化部件树中的模型可视化，同时不舍弃模型的任何部分。

通过允许逐步定义虚拟组件与原生控件模式之间的对应关系，以及名称绑定器与方程之间的对应关系，我们支持以渐进式方法对低代码元模型进行虚拟化，从而进一步提升原生语言的抽象层次。

我们引入的虚拟元模型是动态的，由一组虚拟控件定义生成，随后在高效的模型解析器中实例化。我们使用规模显著庞大的真实低代码基准测试，根据效率和有效性参数评估了由此生成的系统。总的来说，结果表明，该方法适用于在不同抽象层级上同时对低代码进行协作式和交互式编辑。

参考文献

[1] E. Visser，“WebDSL：领域专用语言工程案例研究”，载于《生成式与变换式软件工程技术》（GTTSE 2007），《计算机科学讲义》系列，R. Lammel、J. Saraiva 和 J. Visser 编。Springer，2008年。[2] OutSystems。（2019）体验构建器。 [在线]。网址：https: // www.outsystems.com/low-code-platform/experience-builder/ [3] ——。（2019）工作流构建器。 [在线]。网址：https://www. outsystems.com/low-code-platform/workflow-builder/ [4] ——。（2019）集成构建器。 [在线]。网址：https://www. outsystems.com/low-code-platform/integration-builder/ [5] J. Ramalho、H. Lourenc¸o 和 J. C. Seco，“从构建器到编辑器：低代码模型的双向转换”，载于 ACM/IEEE 模型驱动工程语言与系统国际会议伴随会议，MODELS 2021 Companion，日本福冈，2021年10月10日至15日。IEEE，2021年，第108–117页。 [在线]。网址：https://doi.org/ 10 1109 53483 2021 00022 . /MODELS-C . . N. [6] Kahani、M. Bagherzadeh、J. R. Cordy、J. Dingel 和 D. Varr´o，“模型转换工具的调查与分类”，《软件与系统建模》，第18卷，第4期，第2361–2397页，2019年。[7] J. Greenyer 和 E. Kindler，“关系模型转换技术比较：使用三元图文法实现查询/视图/转换”，《软件与系统建模》，第9卷，第1期，第21–46页，2010年。[8] T. J. Parr、R. Quong 和 H. Dietz，“谓词在 LL (k) 和 LR (k) 解析器生成器中的应用（技术摘要）”，普渡大学，技术报告 7-1-1993，1993年。[9] T. J. Parr 和 R. W. Quong，“ANTLR：谓词 LL(k) 解析器生成器”，《软件：实践与经验》，第25卷，第7期，第789–810页，1995年。[10] M. Tomita，“一种高效的增广上下文无关解析算法”，《计算语言学》，第13卷，第1-2期，第31–46页，1987年。[11] OutSystems。（2022）什么是公民开发者？ [在线]。网址：https://www.outsystems.com/glossary/what-is-citizen-developer/ [12] ——。（2022）低代码与传统开发：架构师视角。 [在线]。网址：https://www.outsystems.com/blog/ posts/low-code-vs-traditional-development/ [13] D. E. Knuth，“从左向右翻译语言”，《信息与控制》，第8卷，第6期，第607–639页，1965年。[14] S. K. Card、G. G. Robertson 和 J. D. Mackinlay，“信息可视化工具：信息工作空间”，载于 SIGCHI 人机交互系统因素会议论文集，CHI ’91 系列。美国纽约州纽约：计算机协会，1991年，第 181 186 [ ] 10 1145 108844 108874 – 。在线。[15] 网址：https://doi.org/ . / . K. Czarnecki、J. N. Foster、Z. Hu、R. Lammel、A. Schurr 和 J. F. Terwilliger，“双向转换：跨学科视角”，载于模型转换理论与实践——第2届国际会议 ICMT@TOOLS 2009，瑞士苏黎世，2009年6月29日至30日。会议论文集，《计算机科学讲义》系列，R. F. Paige 编，第5563卷。Springer，2009年，第260–283页。 [ ] 10 1007 978 3 642 02408 5 19 在线。网址：[16] https://doi.org/ . / - - - - F. Bancilhon 和 N. Spyratos，“关系视图的更新语义”，《ACM 数据库系统汇刊》，第6卷，第4期，第557–575页，1981年12月。[17] Y. Masunaga，“关系数据库视图更新转换机制”，载于第十届超大型数据库国际会议，1984年8月27日至31日，新加坡，会议论文集，U. Dayal、G. Schlageter 和 L. H. Seng 编。Morgan Kaufmann，1984年，第309–320页。 [在线]。网址：http://www.vldb.org/conf/1984/P309.PDF [18] G. Gottlob、P. Paolini 和 R. Zicari，“一致视图的属性与更新语义”，《ACM 数据库系统汇刊》，第13卷，第4期，第486–524页，1988年10月。[19] A. Bohannon、B. C. Pierce 和 J. A. Vaughan，“关系镜片：一种用于可更新视图的语言”，载于第二十五届 ACM SIGACT-SIGMOD-SIGART 数据库系统原理研讨会论文集，2006年6月26日至28日，美国伊利诺伊州芝加哥，S. Vansummeren 编。ACM，2006年，第338–347页。 [在线]。网址：https://doi.org/ 10 1145 1142351 1142399 . / . E. Guerra、J. de Lara、D. S. [20] Kolovos、R. F. Paige 和 O. M. dos Santos，“transml：一组用于建模模型转换的语言”，载于模型驱动工程语言与系统——第13届国际会议 MODELS 2010，挪威奥斯陆，2010年10月3日至8日，会议论文集，第一部分，《计算机科学讲义》系列，D. C. Petriu、N. Rouquette 和 Ø. Haugen 编，第6394卷。Springer，2010年，第106–120页。

[21] D. Steinberg、F. Budinsky、M. Paternostro 和 E. Merks，《EMF：Eclipse 建模框架》，第2版，Eclipse 系列。新泽西州上萨德尔河：Addison-Wesley，2009年。[22] JetBrains，《JetBrains 元编程系统》，2020年，http://github. com/JetBrains/MPS。最后访问日期：2022年5月11日。[23]V. Pech、A. Shatalin 和 M. Voelter，《将 JetBrains MPS 用作扩展 Java 的工具》，载于《2013年 Java 平台编程原理与实践国际会议：虚拟机、语言与工具论文集》，德国斯图加特，2013年9月11日至13日。ACM，2013年，第165–168页。 [在线]。可访问：https://doi.org/ 10 1145 2500828 2500846 。/。F. Jouault、F. Allilaire、J. [24] Bézivin 和 I. Kurtev，《ATL：一种模型转换工具》，《科学计算机程序》，第 72, 1 2, 31 39, 卷，第-期，2008 [ ] 10 1016 2007 08 第–页。在线。可访问：https://doi.org/。/j.scico.。. 002 [25] – 2015, 《Atlas 转换语言》，https://wiki.eclipse.org/ATL/ User Guide。最后访问日期：2022年5月11日。[26] A. Sch¨urr，《使用三元图文法规范图转换器》，载于《计算机科学中的图论概念》第20届国际研讨会 WG ’94 论文集，德国赫尔兴，1994年6月16日至18日，论文集，计算机科学讲义系列，E. W. Mayr、G. Schmidt 和 G. Tinhofer 编，第903卷。Springer，1994年，第151–163页。[27] A. Anjorin、E. Leblebici 和 A. Sch¨urr，《三元图文法的20年：未来研究路线图》，《欧洲软件科学与技术协会电子通信》，第73卷，2015年。[28] I. R´ath、G. Bergmann、A. ¨Okr¨os 和 D. Varr´o，《由增量模式匹配驱动的实时模型转换》，载于《模型转换理论与实践——第1届国际会议 ICMT@TOOLS 2008 论文集》，瑞士苏黎世，2008年7月1日至2日，计算机科学讲义系列，A. Vallecillo、J. Gray 和 A. Pierantonio 编，第5063卷。Springer，2008年，第107–121页。[29] D. Varr´o、G. Bergmann、 ´A. Heged¨us、 ´A. Horv´ath、I. R´ath 和 Z. Ujhelyi，《迈向响应式增量模型转换平台：VIATRA框架的三代演进》，《软件与系统建模》，第15卷，第3期，第609–629页，2016年。[30] 《Atlas 转换语言》，2016年，https://www.omg.org/spec/QVT。最后访问日期：2023年4月11日。[31] P. Stevens，《QVT中的双向模型转换：语义问题与未决问题》，载于《模型驱动工程语言与系统》第10届国际会议 MoDELS 2007 论文集，美国纳什维尔，2007年9月30日至10月5日，计算机科学讲义系列，G. Engels、B. Opdyke、D. C. Schmidt 和 F. Weil 编，第4735卷。Springer，2007年，第1–15页。[32] ——，《QVT中的双向模型转换：语义问题与未决问题》，《软件与系统建模》，第9卷，第1期，第7–20页，2010年。[33] F. Klar，《基于扩展三重图文法的高效且兼容的双向形式语言转换器》，博士学位论文，达姆施塔特工业大学，2012年2月。 [在线]。可访问：http://tuprints.ulb.tu-darmstadt.de/2894/ [34] F. Hermann、H. Ehrig、C. Ermel 和 F. Orejas，《基于三元图文法并具有冲突解决机制的并发模型同步》，载于《软件工程基础方法——第15届国际会议 FASE 2012 论文集》，爱沙尼亚塔林，2012年3月24日至4月1日，计算机科学讲义系列，J. de Lara 和 A. Zisman 编，第7212卷。Springer，2012年，第178–193页。[35] Y. Xiong、H. Song、Z. Hu 和 M. Takeichi，《基于双向变换同步并发模型更新》，《软件与系统建模》，第12卷，第1期，第89–104页，2013年。[36] H. Lourenc¸o、C. Ferreira 和 J. C. Seco，《OSTRICH——一种用于低代码开发的类型安全模板语言》，载于《第24届模型驱动工程语言与系统国际会议 MODELS 2021 论文集》，日本福冈，2021年10月10日至15日。IEEE，2021年，第216–226页。 [在线]。可访问：https://doi.org/10.1109/MODELS50736. 2021.00030[37] J. C. Seco、H. Lourenc¸o、J. Parreira 和 C. Ferreira，《Nested OSTRICH：低代码模板组合的孵化》，载于《第25届模型驱动工程语言与系统国际会议 MODELS 2022 论文集》，加拿大魁北克省蒙特利尔，2022年10月23日至28日，E. Syriani、H. A. Sahraoui、N. Bencomo 和 M. Wimmer 编。ACM，2022年，第210–220页。 [在线]。可访问：https://doi.org/ 10 1145 3550355 3552442 。/。H. Lourenc¸o、[38] C. Ferreira、J. Costa Seco 和 J. Parreira，《OSTRICH：一种用于低代码开发的丰富模板语言（扩展版）》，《软件与系统建模》，第72卷，第1-2期，第1–19页，2022年。