arXiv:2304.15006v1 [cs.PL] 1 Apr 2023

# 用于 Isabelle/HOL 翻译的 VDM-SL 定义拓扑排序

Leo Freitas[1] 和 Nick Battle[2]

[1] 纽卡斯尔大学计算机学院，leo.freitas@newcastle.ac.uk[2] 独立研究者，nick.battle@acm.org

**摘要。** VDM 库及其扩展构成了一个生态系统，其中包括面向 Isabelle 中 VDM 的翻译与证明环境。该环境支持 VDM-SL 的大部分子集，并会根据需要进一步添加其他构造。对新手用户而言，一个主要障碍是 Isabelle/HOL 要求所有定义必须先声明后使用，而（相互）递归定义**必须** 同时定义。本文介绍了这一问题的解决方案，使新手用户和实际模型都能更广泛地使用翻译器插件。

**关键词：** VSCode、VDM、排序、Isabelle/HOL

## 1 引言

维也纳开发方法（VDM）已广泛应用于工业和学术领域，涵盖安全 [4, 5],容错 [8], 医疗器械 [6], 等多个领域。我们通过一套工具和数学库扩展了对 VDM 规约的支持 [1]。

VDM 还有一个具备多种功能的 Visual Studio Code（VS Code）集成开发环境，其中包括将内容翻译为 Isabelle/HOL 的策略 [9]。然而，用户需要遵循严格且特定的规约风格，否则翻译就会失败或无法进行。例如，每个 Isabelle/HOL 理论**都** 必须写在一个同名文件中，而 VDM 文件可能不含模块，也可能包含多个模块。此类 VDM 规约的翻译应强制采用所要求的 Isabelle 风格。目前，用户必须留意这些问题，否则翻译就会失败。

这些问题涉及多个方面，相关内容列于发行包中的各个示例文件中，并在 [2] 中进行了详细说明。这些 VDM“惯用模式”不仅对于实现翻译至关重要，而且有助于确保证明过程易于管理，并使证明策略更容易确定。

然而，Isabelle 翻译的一大障碍在于，VDM 规**必须**按顺序排列（即，声明必须先定义，然后才能使用）。这对大多数 VDM 用户来说相当不自然，因为他们通常会自顶向下编写规格说明，从较大或较复杂的概念开始，逐步写到较简单或较容易的定义。此外，由于遗留模型编写时并未考虑这种顺序要求，因此很难对其进行翻译。手动

[1] https://github.com/leouk/VDM_Toolkit/

重新排列并不困难，但既费时又容易出错，因此会成为使用转换器的门槛。

解决这一特定的顺序要求是本文的主要贡献。我们按照 byNaur 的 N皇后算法 [7] 的风格介绍解决方案：回顾 VDM工具 中的各种发展如何最终促成模块排序的实现。这些内容分布在多个章节中，其中包括对相关发展的历史概述。

## 2 背景

将 VDM 翻译为 Isabelle/HOL，需要遍历其抽象语法树（AST），并为VDM具体语法中每个兼容翻译的部分生成相应的 Isabelle/HOL 定义。这个过程可能很复杂。例如，VDM联合类型的表达能力很强，允许一些特殊的定义，如下面的交叉联合选择示例。

```
类型 TUnion1 = int | nat inv u == (is_nat(u) => u > 0) and (is_
int(u) => u < 0); TUnion2 = seq of nat | set of real; 函数 f: TUnion2
* TUnion1 -> bool f(u1, u2) ==(is_int(u2) =>((is_(u1, seq of nat) =>u2 i
n set elems u1) and (is_(u1, set of real) => u2in set u1)) ) and (is_nat(u2)
 =>((is_(u1, seq of nat) => not u2 in set elems u1) and (is_(u1, set of
real) =>not u2 in set u1)) );
```

VDM联合类型会在合并后的类型上形成一个最大类型集合，同时保留不变式。这可能导致选择和不变量检查过程相当复杂且令人困惑。当然，这类规范很少编写。不过，从翻译的角度来看，这说明了 VDM 的表达能力给如何将其翻译为 Isabelle 带来的一些挑战。

VDM 中另一个相当复杂的定义空间涉及值和模式。例如，可以在值和类型中定义一些复杂的 VDM模式，如下所示：

```
类型T = 个自然数集合 inv {a,b,c} == a < b 且b <
c;值[i,j]: 自然数序列 = [1,2]; {k,m}: 自然数集合 =
{2,1};
```

类型定义说明它是一个由三个显式 nat 值组成的有序集合，而值定义则将每个名称绑定到对应的值（即，i=1、j=2，其中 k 和 m 将从该集合中随机选取！）。鉴于 Isabelle 要求名称、类型和值之间的绑定关系更严格，这给翻译带来了各种挑战。实际上，这类示例很少构成障碍（或者说很少有人编写），但排序算法也必须能够处理它们。尤其是，像 those[i,j]，这样的值定义，实际上会为 i 和 j 分别创建两个独立的值定义。

因此，转换器针对的是 VDM 中较为“温和”（且大多会用到）的子集。转换器配备了 VDM 分析工具（名为 exu[2]），可检查各种此类语法难题和限制，并告知用户应如何处理以及问题所在。修正这些错误和警告十分重要，以确保能够完成翻译，并避免用户不得不自行解决类型错误。

exu 还提供支持，帮助用户为翻译做好准备（例如，exu 能报告 50 多种错误和 20 种警告）。例如，它会检查函数定义中的任何函数调用是否都包含该函数的前置条件（如果存在）。然而，仍有一个关键障碍：模块定义及其依赖关系的排序。本文将讨论这一问题，而且这一功能最近已在最新版本的转换器中实现（见第 5 节）。

## 3 项重要的历史发展 მოვლენ

在介绍我们的解决方案之前，首先需要指出对主要 VDM工具所做的一些关键扩展（尽管它们在技术上并不相关）。这些扩展为 exu 最终能够对定义进行排序以供翻译铺平了道路。下面我们将在主要 VDM工具中着重介绍这些扩展的动机及其实现方式。

### 3.1 加载时问题

处理大型 VDM 模型 [3]（例如， 150+ 模块， 60+ 千行代码）时，类型检查耗时（例如，$2-4$ 分钟）可能成为阻碍，而初始化耗时（例如， 15+分钟）可能影响后续开发。

如此漫长的初始化时间，是因为 VDM 运行时需要多次遍历 AST，以处理前向引用。遍历次数取决于模块间的依赖关系数量（也就是说，模块 A 导入模块 B，反之亦然），以及模块内定义的顺序（也就是说，在定义之前使用定义），等等。

为了优化类型检查和初始化，大型模型的用户必须仔细检查导入链和模块内定义，以跟踪依赖关系。例如，用户必须精心构建依赖关系，并明确记录下来。这容易出错，也难以维护。

[2] 这个名称源自约鲁巴神祇*Èṣù*，祂是凡人与神祇之间的守门者（兼信使）。

详细输出。为了解决这个问题，VDMJ [1] 增加了一项扩展，用于跟踪跨模块的顶层依赖关系。一个简单易行的解决方案是，让类型检查器在遍历 VDM 抽象语法树并查找循环依赖时，输出每个顶层定义的依赖关系详情。通过这些详细输出，用户可以了解已执行的轮数，以及是否存在循环依赖及其位置。随后，用户便可手动修复这些环。这意味着用户可以利用详细输出来减少轮数，方法是减少前向引用，从而缩短加载时长。

排序，使用依赖关系与自由变量访问器。为了确定 VDM-SL 中最优的模块排序，只需考虑每个模块的导入项。这样便创建了依赖图，目标是优先处理“被依赖最多”的模块。在 VDM++ 中，类没有显式导入项，而是可以使用其他类中的任意公共定义。因此，类排序使用了与类型检查器相同的访问器来搜索循环依赖——也就是说，它会搜索当前定义在初始化时依赖于哪些其他公共定义。目标是优先处理“被依赖最多”的类文件。最后，在模块或类内部，最优定义排序会尽量减少前向引用的数量。这需要使用另一种访问器，用于查找每个定义所使用的自由变量。循环依赖访问器与自由变量访问器非常相似，但循环依赖的情况不得产生误报，即不得在不存在环时声称存在环。这意味着必须忽略不可判定的情况，例如包含条件“if”语句的环。另一方面，自由变量访问器只关注分析时的排序，可以容忍环，因此会返回所有自由变量。

模块依赖关系的拓扑排序。利用收集到的依赖关系信息，可以对模块名称进行拓扑排序，尽可能减少遍历次数，最终得到模块名称列表。这在大多数情况下有效解决了详细输出中的依赖警告，因此可以向最终用户提供按顺序排列的模块名称列表，以便加载。随后可以在加载时直接将该列表传给 VDMJ。借此还可以将模块依赖关系以有向无环图的形式输出为 DOT 文件。

### 3.2 递归循环的检测与移除。

通常，依赖图中可能存在环。在 VDM 中，这不一定是错误。例如，存在相互递归函数或相互依赖的模块完全合理；但值初始化之间不能相互依赖（例如，A = B 和 B = A）。因此，VDMJ 类型检查器会查找初始化依赖循环，并将其视为错误；同一依赖搜索也用于支持对模块或类进行最优排序。

但是，要从图中生成节点有序序列，必须先将图简化为有向无环图（DAG）。这意味着必须以适当的方式打破合理的环，确保所得有向无环图的排序仍然最优——在我们的例子中，也就是前向引用最少。这是一个难题（已知为 NP-难）；而且，由于源文件中的定义目前无法由类型检查器重新排序，问题变得更加复杂。

因此，图中包含固定的子图（源文件定义），其依赖关系无法更改；而子图间的依赖关系（文件之间）则可以更改。[3]，

VDMJ 采用的模块/类排序方案很简单：反复移除最后找到的、会导致环的边，直到不存在环为止。这种方法速度很快，而且在大多数情况下都能显著改善排序（也就是说，远胜于猜测）；即使在最坏情况下，次优的模块顺序仍能通过类型检查并正确初始化，因此这并非关键问题。然而，对于 Isabelle 定义排序，前向引用会导致错误，因此在移除环以及选择有向无环图的起始节点时必须格外谨慎。

## 4 Exu 排序

对模块之间的依赖关系进行拓扑排序，解决了类型检查和初始化耗时过长的问题。然而，这并未优化模块内部的定义。此外，模块依赖搜索只考虑了导入项，没有考虑本地定义（例如，let-in 定义）和函数命名空间，也没有检查类型命名空间。该搜索还针对顶层导入定义（即类型、值、函数等。），而非其结构。因此，需要采用一种不同的拓扑排序算法。新算法分为五个阶段：

1. 收集所有具名定义；2. 处理非函数空间依赖关系：– 遍历所有定义依赖；– 为所有声明类型 T 创建缺失的 inv_T；– 关联类型与定义空间；

3. 处理函数定义命名空间依赖：– 遍历名称空间依赖；– 忽略递归调用（这些由 VDMJ 的循环依赖检查处理）；– 链接命名依赖； 4. 拓扑排序：– 检查是否需要拓扑排序；– 使用 Kahn 算法对顶层名称进行有向无环图排序； 5. 模块重构：– 组织顶层名称，将类型名称空间与函数命名空间分开；– 重新排序模块定义；– 可选地再次进行模块类型检查；

第一阶段实际上会将模块顶层定义展平为一个列表，其中包含模块内定义的所有内容。例如，显式函数可能会定义前置条件、后置条件以及度量，而类型可能会定义不变式、相等性或顺序谓词。这些定义本身都是显式函数，必须对其进行显式处理，以确定（自由变量）前向依赖关系。

第二阶段会搜索顶层定义依赖，并为所有未显式声明不变式函数定义的类型创建相应定义。

[3] 这或许可以作为 VDMJ 的一项未来改进。

确保每个没有显式状态不变式的类型 T 仍依赖于函数 inv_T，这一点非常重要。这实际上在用户声明的类型、已声明的值和函数之间创建了函数空间依赖。此类合成构造在结构上等价于类型检查器在用户显式编写类似 inv_T = true 的内容时所创建的构造。这适用于记录和命名类型不变式函数的构造方式。例如，记录会创建一个从记录名称到布尔值的函数，而命名类型 S = RHS 会创建一个从 RHS 类型到布尔值的函数。接下来，对于已声明的值，不需要创建隐式函数，但会记录命名的（类型）依赖链接。这会深入处理值，而值可以包含多个局部定义，正如上面第 2 节中的示例所述。

第三阶段会搜索所有顶层定义之间的自由变量依赖，同时考虑局部上下文，例如函数参数或局部绑定变量（例如，量化变量、let-def 构造等。）。此搜索还会查找递归和相互递归的函数调用。前者会被忽略或移除，因为转换器会以不同方式处理它们。后者必须得到解决，否则拓扑搜索阶段将会失败，因为该阶段要求图为有向无环图（DAG）。

第四阶段是 VDMJ 实际使用的拓扑排序算法，用于对上述模块名称进行排序。它是 Kahn 有向无环图排序算法的一种变体[4]。该排序算法还会检查给定图中是否存在循环依赖（并且无法容忍此类依赖）。必须提前解决此类循环依赖。对于定义而言，循环依赖只可能出现在相互递归的类型定义或函数定义之间。幸运的是，VDMJ 已经具备移除循环依赖的机制，我们在这里也采用了这一机制。实践中效果良好。不过在一般情况下，根据所涉及的相互递归的复杂程度，此方法可能会失败（见上文）。

前三个阶段被视为模块定义处理阶段。它们会创建排序所需的所有结构和隐式定义。不过，在正式排序之前，需要提供两项信息。首先，排序必须以所谓的图起始点为依据：即没有入边的节点。其次，只有当模块中存在先使用后声明的定义时，才会尝试排序。这是因为对已经排序的模块再次排序可能会导致结果不一致（也就是说，排序顺序仍无法避免先使用后声明）。因此，我们确保只有在必要时才运行 Kahn 算法。是否有必要运行该算法，是通过对 VDMJ 用于检查循环依赖的详细输出警告进行变通处理来确定的。

最后，在第五阶段，如果不需要模块排序，则跳过拓扑排序，且不修改模块。否则，如果需要排序，就必须重新组织拓扑排序后的名称列表，以考虑模块顶层定义中类型和函数各自独立的两个命名空间（在第一阶段进行定义扁平化之前）。

例如，考虑下面的模块 M。它采用自顶向下的方式定义类型和函数，即先使用后声明，其中某些定义没有显式定义的不变式。

[4] https://en.wikipedia.org/wiki/Topological_sorting#Kahn's_算法

*1* 模块 *M 2* 导出全部 *3* 定义 *4* 类型 *5 --@doc* 在声明类型 *S* 和 *T* 之前使用它们；需要隐式不变式_*Rec 6 Rec:: s: S t:T; 7 8 --@doc* 在声明类型 *T* 和函数 *tail* 之前使用它们。 *9 S = T* 不变量 *== head(s) > 0* 且 *len tail(s) > 0; 10 11 --@doc* 需要隐式不变式_*T 12 T = seq1 of nat; 13 14* 函数 *15 tail: seq1 of nat -> seq of nat 16 tail(s) == tl s; 17 18 head: seq1 of nat -> nat 19 head(s) == hd s; 20* 结束*M*

当用户在此模块上运行 exu（并设置调试标志）时，这就是该工具的输出。

> 正在调用 Exu VDM 分析器.....正在计算声明依赖关系，**模块**‘M‘.....已打印**模块**M.dot 的依赖关系，位于 ./M.dot M‘T 声明在 Rec 之后 M‘S 声明在 Rec 之后 M‘T 声明在 S 之后 tail 声明在 S 之后 tail 声明在 inv S 之后_ 发现 5 处定义先用后声明。需要进行拓扑排序。原始名称：Rec、S、T、tail、head 起始点：Rec、S 排序后的名称：tail、head、inv_T、inv_S、T、inv_Rec、S、Rec 整理后的名称：tail、head、T、S、Rec Exu 已成功排序**模块**M **定义**

原始名称是在声明顺序中由用户定义的顶层名称，可在第一阶段之后获取。起始点是图中位于顶端的名称（即 没有入边），可在第三阶段之后以按声明位置排序的名称列表形式获取。排序后的名称是第四阶段之后得到的名称。请注意，其中包含了对 R 和 T 的（隐式声明的）类型不变式的依赖关系。最后，整理后的名称是名称空间已合并且仅包含原始声明集合中的名称。

此外，请注意，函数 head 和 tail 与类型 T 之间的顺序是“不稳定的”，即在这种情况下它们不可比较。也就是说，无论它们以何种顺序声明都有效，只要它们在 S 之前声明，而 S 又必须在 Rec 之前声明。正是这些名称之间的不可比较性，使得在所有名称都必须先声明再使用的情况下，模块排序可能产生无关的排序结果。我们尝试按照声明顺序对起始点进行排序，以尽量减少这种情况，但这无法完全避免无关的排序结果。

该算法满足以下不变式：原始名称集合与整理后的名称集合相等，且排序后的名称列表包含所有整理后的名称。这样，所有名称都已纳入考量，且隐式创建的名称（例如，inv_T）不会出现在结果中。

该翻译提供可选标志，用于生成 VDM 源代码位置信息，以及将原始 VDM 源代码作为 Isabelle 注释。这样，模型的任何重新排列都不会影响翻译。VDM 用户不应修改 Isabelle 翻译，而应处理证明义务证明脚本。

exu 还会生成依赖关系的 DOT 文件视图，其形式与 VDMJ 的排序插件针对模块名称依赖关系生成的输出类似（见图 1）。它展示了排序前用户声明的依赖关系，并标注了相应的源代码行位置：起始节点以红色倒三角形表示，隐式创建的不变式以双圆圈表示，而没有出向链接的（终端）节点则以三角形表示。对于复杂（且规模较大）的模块，这类可视化“线索”可能极具价值

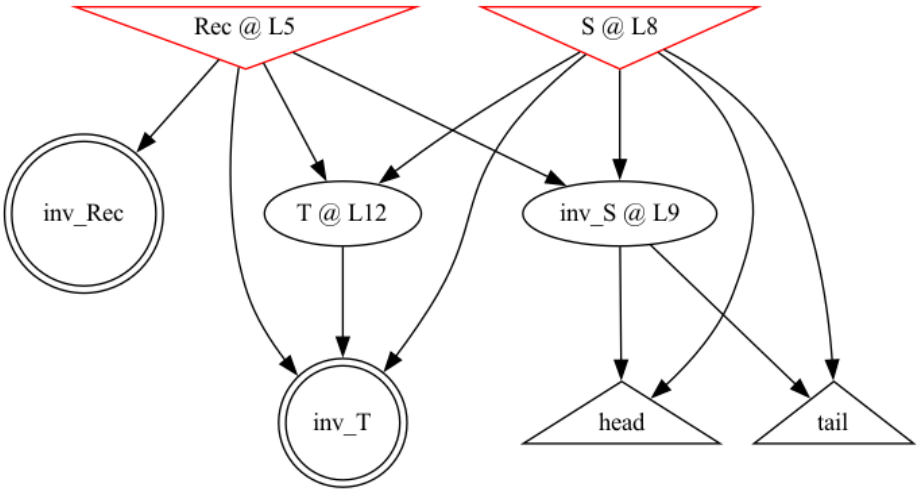


图 1：M.vdmsl 定义依赖关系。

以便找出需要进行哪些调整，从而避免不稳定排序产生的结果不完全符合先声明后使用的要求，并可能导致 Isabelle 错误。

## 5 VDMJ 和 VDM-VSCode 集成

我们的插件基础设施遵循 UNIX 原则，严格区分命令行操作及其相应的图形用户界面输出。插件架构始于一个 VDMJ CommandPlugin 扩展。这样便可通过命令行在 VDMJ 控制台中执行 exu 插件。该插件经过配置，可供所有界面使用。

对于 VDMJ 控制台接口，它遵循 VDMJ 的惯例，根据插件调用来命名插件类。这使 VDMJ 调试控制台能够像调用 C/Java 程序的主函数一样，使用用户提供的参数来调用相应功能。

对于 VDM-VSCode 接口，它遵循 VDM-VSCode 语言服务器协议（LSP）惯例 [9]：插件类封装了图形用户界面编辑器的响应式功能以及调试协议（DAP）命令行调试。对于 DAP VDM 调试会话，它提供相同的 VDMJ 控制台插件命令；其中的任何问题或错误都会直接报告到 DAP 输出控制台，而不是通过图形用户界面可视化来显示这些错误。这主要仅用于调试和底层应用程序。建议用户改为通过图形用户界面可视化流程进行操作。

对于 GUI 可视化流程，ISA LSP 插件会响应两个 LSP 事件：CompleteCheckEvent 和 UnknownTranslateEvent。前者发生在 VDMJ 对 VDM-VSCode 项目中所有已加载的 VDM-SL 模块完成类型检查之后；后者则发生在用户明确选择与 VDM 关联的项目弹出菜单项 VDM->Translate to Isabelle 时。

对于 CompleteCheckEvent，插件会分阶段执行各个插件的相关功能，包括执行 exu 进行检查和排序，执行*vdm2isa* 进行翻译，以及执行 *isapog* 进行证明义务翻译。最终输出包括模块定义依赖图、VDM 源代码和证明义务到 Isabelle 的翻译结果，以及此过程中可能出现的错误和警告。

对于 UnknownTranslateEvent，会考虑 VDM-VSCode 范围内的翻译选项，并以相同流程处理待翻译模块。这在一定程度上造成了浪费，因为翻译既会在类型检查后立即进行，也会在用户明确提出请求时进行。这是在速度与可用性之间作出的折衷：成功翻译所需的分析与预期的翻译输出紧密相关。值得庆幸的是，平均而言，“延迟”时间尚可接受，用户也可以避免这种延迟（即，只有用户通过点击 UnknownTranslateEvent 弹出菜单发出请求时才会进行翻译）。

这种用例的权衡在于，错误会一并处理，而且只有在发出翻译请求后才会处理。如果已加载项目的 CheckCompleteEvent 处理速度很快，那么这种方式更适合用于翻译，因为它能在模型开发过程中持续向用户提供反馈。否则，可以由用户请求触发翻译，并据此报告和处理任何剩余问题。

对于 VDM-VSCode 集成，我们尽量使这 3 种用例（即 VDMJ 控制台、VDM-VSCode DAP 控制台和 VDM-VSCode LSP 图形界面响应）尽可能相互独立，同时也尽可能复用。这样做很重要，这样就不必以略有不同的方式重复编码各项功能。我们通过采用 UNIX 原则实现了这一点：将基于参数的插件命令行执行作为所有类型接口的基础。具体而言，这意味着 exu 工具既集成在命令行中，也作为独立组件集成在 VSCode 中，且不依赖于 Isabelle 翻译器。

实际上，这意味着每个插件都会响应特定命令。这与其他 UNIX 命令类似（例如，git 有 push、pull 等命令。）。系统设置了各种默认选项，可通过属性文件、命令行 set 命令或 VSCode 属性进行更改。属性的读取和设置过程与 VDMJ 自身的属性文件相近。对于 VDM-VSCode，图形界面也会通过

其设置界面公开这些属性。最终，VDM-VSCode 插件会根据具体情况，向特定插件发出特定命令调用：如果属于 LSP 的响应式行为（例如，在代码位置显示黄色/红色波浪线，以提示警告/错误），则执行相应调用；如果属于 DAP 的交互式行为（例如，在调试控制台中调用特定的 DAP 插件），则执行相应调用。相应地，也可以通过命令行使用 VDMJ 控制台实现相同功能。翻译输出默认保存到 ./.generated/isabelle，但用户可以将其设置为自己选择的任意目录。

## 6 结果与局限性

通过对模块定义进行排序，我们克服了阻碍 VDM 用户使用 VDM 到 Isabelle/HOL 转换器的主要障碍。不过，VDM 语言本身仍存在一些使用限制，必须加以遵守，例如值声明中某些联合类型的构造方式。此外，一些 VDM 构造尚未得到支持，例如各种声明的参数中包含多个记录模式（例如，显式函数签名的不变式声明）。这是因为记录翻译需要局部上下文。我们预计，其中一些限制将在未来得到解决。

此外，尽管针对各种 VDM 语句已有相应的翻译策略，但对它们的支持 [2]，仍相当有限。这主要是因为它们的隐式规范（例如，语句执行时对状态的隐式更新）相当复杂 [10]。因此，对 VDM 状态和操作的支持相当有限。状态仅被翻译为一种特殊的记录，操作签名可用，同时也支持一些语句（例如，if 语句、赋值、状态指示符、等。）。许多其他语句尚未得到支持。

我们已经处理了大量 VDM-SL 示例，以及 100 多个涉及各种（有时较为特殊的）VDM 构造的测试模块。完整的示例集可在 VDM_Toolkit 的 GitHub 页面[5]上找到。

通过 VDM-SL AST 的这些探索，有助于发现 VDM工具遗漏的边界情况。例如，映射推导式表达式中包含多个类型绑定的情况，例如：

```
值
  v = { 1 |-> 5 | x 属于集合 {<A>,<B>,<C>}, x 属于集合 {1,2,3} & x = <A>}
```

$x$ 这里的类型最终变成了各个集合类型的并集。这现在是一个类型错误：VDM 推导式中不应允许这种隐式类型并集。

## 7 结果与讨论

本文提出了一种解决方案，以应对将 VDM-SL 模型转换为 Isabelle/HOL 时的主要问题：必须对声明进行排序，才能使用它们。这涉及对 VDM(J) 工具进行各种基础扩展，以及针对转换器进行相应调整。

[5] https://github.com/leouk/VDM_Toolkit/tree/main/plugins/vdm2isa/src/test/resources

该成果也已集成到 VDM-VSCode 扩展中 [9]。这样，大多数用户无需费心重新组织 VDM 文件即可使用 VDM 到 Isabelle/HOL 的转换器。这对于旧有（以及规模较大的）模型尤其重要，因为在开发期间并未考虑此类因素。

我们力求保持生态系统的稳定性，并使其不依赖特定环境。因此，同一种解决方案适用于用户偏好的三种特定 VDM 环境：VDMJ（UNIX）命令行、VDM-VSCode 编辑器以及 VDM-VSCode 调试会话。这三种接口使用相同的代码，表现出相同的行为，从而避免了任何形式的代码重复，这得益于 UNIX 原则所倡导的命令行一致性。

未来工作。 我们计划通过使用代码透镜，加强 VDMJ 控制台插件与 VDM-VSCode 图形用户界面之间的集成。代码透镜是由用户在图形界面中执行的操作，可触发特定的插件事件。例如，JUnit 测试会在每个定义为测试用例的 Java 方法左侧显示一个绿色或红色圆圈（即，带有 @Test 标记）。用户可以点击该圆圈来运行、调试特定测试，或跳转到相应测试。我们设想为 VDM 模块的翻译和不同的证明脚本策略提供类似的代码透镜。这些策略可以自动依次尝试，以便为各种 VDM 证明义务“发现”证明，类似于 Isabelle 自带的 sledgehammer 工具。

致谢。我们感谢 Peter Gorm Larsen 及其多位学生与我们讨论转换器的各个方面（以及其局限性的影响），也感谢他鼓励我们将工具的各个部分呈现给最终用户。

## 参考文献

1. Battle, N.：《VDMJ 用户指南》。技术报告，Fujitsu Services Ltd.，英国（2009） 2. Freitas, L.：面向 Isabelle 的 VDM 翻译策略（Nill Full game）（2016） 3. Freitas, L.：VDM 大规模建模：EMV 第二代内核建模。载于：第 21 届巴西形式化方法研讨会，SBMF。第 26–30 页（2018） 4. Kulik, T., Macedo, H.D., Talasila, P., Larsen, P.G.：使用 VDM 对 HUBCAP 沙盒架构建模——一项安全研究。载于：Fitzgerald, J.S., Oda, T., Macedo, H.D.（编），第 18 届国际 Overture 研讨会论文集。第 20–34 页。Overture（2020 年 12 月） 5. Kulik, T., Talasila, P., Greco, P., Veneziano, G., Marguglio, A., Sutton, L.F., Larsen, P.G., Macedo, H.D.：扩展对 HUBCAP 沙盒的形式化安全分析。载于：Macedo, H.D., Thule, C., Pierce, K.（编），第 19 届国际 Overture 研讨会论文集。Overture（2021 年 10 月） 6. Macedo, H.D., Larsen, P.G., Fitzgerald, J.：使用 VDM 对心脏起搏系统的分布式实时模型进行增量式开发。载于：Cuellar, J., Maibaum, T., Sere, K.（编），FM 2008：形式化方法，第 15 届形式化方法国际研讨会。《计算机科学讲义》第 5014 卷，第 181–197 页。Springer-Verlag（2008） 7. Naur, P.：《程序开发实验》。BIT 12，347–365（1972） 8. Nilsson, R., Lausdahl, K., Macedo, H.D., Larsen, P.G.：将一个工业案例研究从 VDM++ 转换为 VDM-SL。载于：Pierce, K., Verhoef, M.（编），第 16 届 Overture 研讨会。第 107–122 页。纽卡斯尔大学计算机学院，牛津（2018 年 7 月），TR-1524

9. Rask, J.K. 等：Visual Studio Code 中的高级 VDM 支持。载于：第20届国际 Overture 研讨会。第 35–50 页（2022年） 10. Tran-JØrgensen, P.W.、Larsen, P.G.、Leavens, G.T.：将 VDM 自动翻译为 JML 带注释的 Java。《国际软件工具与技术转移期刊》20(2)，211–235（2018年4月）