arXiv:2303.10381v1 [cs.PL] 18 Mar 2023

使用 VDM++对国际象棋进行建模

Morten Haahr Kristensen[1] 和 Peter Gorm Larsen[1]

DIGIT，奥胡斯大学电气与计算机工程系，丹麦，201807664@post.au.dk，pgl@ece.au.dk

摘要。 国际象棋广为人知，风靡世界。然而，国际象棋的规则相当复杂，因为棋子有不同类型，而棋子的移动方式取决于其类型。本文探讨了使用 VDM++ 对国际象棋对局规则进行建模时可采用的不同范式，并展示我们认为最好的模型。文中还说明了如何将此模型与国际象棋对局走法的标准文本记谱法相连接。这样便可以将形式化模型与更便捷的接口结合起来。

1 引言

许多人们可以相互进行的游戏，都包含基于逻辑的规则，用以规定哪些行为是允许的。棋盘游戏通常包含不同种类的棋子，玩家轮流移动棋子。国际象棋是较为复杂的棋局之一。本文使用 VDM++ 对国际象棋进行建模，并讨论不同建模风格 [2]的优缺点。该模型作为教学示例编写，可通过执行来验证国际象棋对局规则是否被违反。

形式化模型的目的，是对所关注系统的理想属性进行形式化分析。然而，每种形式化模型都存在不同的抽象方案，值得针对每种方案讨论描述相关系统的最佳范式。规格说明侧重于阐明与模型目的相关的关键方面；在此背景下，对人类的可解释性远比执行速度重要。本文将以国际象棋规则为例，讨论不同范式的可解释性。

本文结构如下：在本引言之后，第 2 节简要介绍国际象棋对局的规则。随后，第 3 节探讨使用面向对象范式或声明式范式对国际象棋进行建模的思考。接着，第 4 节介绍采用声明式范式构建的 VDM++模型的核心内容。之后，第 5 节简要说明如何纳入便携式棋局记谱法（Portable Game Notation），以便将标准文本格式作为 VDM++ 模型的输入，从而轻松将已进行过的棋局导入模型。第 6 节简要阐述本文的贡献与相关工作的关系。最后，第 7 节总结全文，并展望未来的研究方向。

2 国际象棋的规则

国际象棋是一种由两名玩家参与、按回合进行的棋类游戏，一名玩家控制白方棋子，另一名玩家控制黑方棋子。棋局在国际象棋棋盘上进行，棋盘由 64 个棋格（格子）组成，呈 8×8 网格排列。网格的列称为“纵线”，行称为“横线”。棋格浅、深交替排列；棋盘上由同一颜色的格子组成、从一侧延伸至相邻一侧的线称为“斜线”。棋盘上的横向棋格以字母“a”至“h”标记，纵向棋格则以数字 1 至 8 标记。

起初，每位玩家在棋盘上控制 16 个棋子，如图 1 所示。每个棋子可通过三个不同的属性加以区分。第一项是棋子类型，它决定棋子的走法。第二项是位置，它决定棋子所在的棋格。最后，每个棋子都有一种颜色，用来表明由哪位玩家控制。



图 1. 国际象棋棋子的初始局面：第一排从棋格“a”到“e”依次显示车、马、象、后和王的位置。第二排显示八个白方兵。图片来源： [5]。

棋局中有六种棋子类型，每位玩家开局时各控制八个兵、两个车、两个马、两个象、一个后，最重要的是，还有一个王。每种棋子类型在棋盘上都有独特的走法。例如，车可以沿着所在的横线或纵线移动，而象只能沿斜线移动。如果起始棋格与目标棋格之间有其他棋子，就会阻挡棋子移动到目标棋格。不过，如果目标棋格上有敌方棋子，玩家可以吃掉该棋子，将其从棋盘上移除。当玩家可以吃掉某个棋子时，该棋子就受到攻击。马的走法有一个例外：其行进路径不会被其他棋子阻挡。

棋局的目标是将对方的王置于无法阻止其在下一回合被吃掉的位置[1]

国际象棋中有一些走法可以称为“特殊”走法，因为只有满足特定条件时才能执行。例如，兵可以从初始位置向前移动两格。另一个例子是“王车易位”，这是唯一涉及两个棋子的走法。王车易位允许棋手将王朝其第一横线上的车移动两格，然后将车移到王刚刚经过的棋格。不过，只有在棋局中王和车都没有移动过、王与车之间的格子均未被占据且未受到攻击，并且王没有受到攻击（即未被将军）时，才允许王车易位。

如果兵到达距离其初始位置最远的横线，就必须将其类型改为以下棋子之一：后、马、车或象。这种走法称为“升变”。考虑到这一点实际上颇具挑战，因为这意味着该棋子的类型会动态变化。

3 国际象棋建模的其他范式

不同的 VDM 方言通常鼓励采用函数式范式（FP），但 VDM++ 方言引入了使用面向对象范式（OOP）来组织模型的可能性。在面向对象编程环境中，通常需要在类中设置实例变量来表示状态；而要访问和调整这些变量，通常就需要使用命令式范式，通过赋值来修改实例变量的操作。是否使用这些特性，是一个需要考虑的问题。

在确定采用哪种范式时，必须考虑是更容易封装那些会变化的部分，还是尽量减少它们。对于国际象棋棋局，面向对象编程和函数式编程范式似乎都颇具吸引力，因为国际象棋的许多规则通常是无状态的，因此没有变化的部分。然而，特别规则会使棋局具有状态性，其中包括吃过路兵和王车易位等特殊走法；这些走法是否合法取决于先前走法。

3.1 考量函数式编程与面向对象编程范式

我们考虑了两种分别遵循面向对象编程和函数式编程的架构。第一种采用典型的面向对象编程结构，以`棋子`作为基类，其中定义了用于确定可能走法的基本方法。随后，每种棋子类型都有一个子类实现，用于定义其独特的移动模式和可能的属性。由于`棋盘`类了解棋局状态，因此特殊走法将通过`棋盘`.`可能的_走法`建模。此类架构的简化类图见图 2。

第二种架构遵循函数式编程的理念编写，只使用不可变变量。在 VDM++ 中，这意味着将所有数据结构定义为复合类型。

[1]规则完整列表请参阅 https://www.fide.com/FIDE/handbook/ LawsOfChess.pdf。

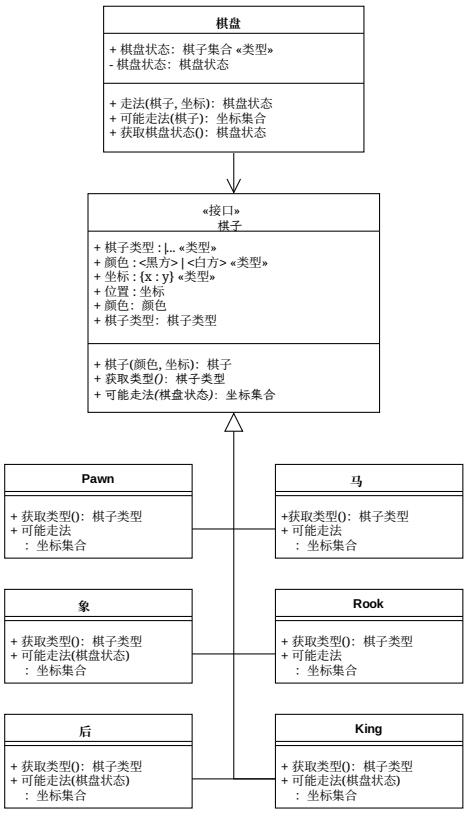

图 2. 国际象棋面向对象架构基础的统一建模语言类图。

这种做法的好处是，规格说明中只有值，这意味着模型是无状态的。以函数式风格编写 VDM++ 时，应将类视为模块，把相关功能封装在同一个命名空间中（因此它看起来会更像 VDM-SL 模型，但为了便于比较，我们仍将其保留为 VDM++ 模型）。图 3 展示了这种架构的一个示例。在这种架构中，所有棋子都共享一个数据结构，其中包含它们的颜色、位置和类型。

采用函数式编程架构的一个好处在于便于对模型进行推理。一般而言，命令式模型的推理可能更加困难，因为其功能可能取决于全局模型状态。要对命令式模型中的某个函数的行为进行形式化验证，必须考虑所有也可能操纵全局状态的方法。相比之下，纯函数式模型由引用透明函数组成。

另一个优点与模型的测试和验证有关。在模型中引入状态后，复杂性会显著增加，因为需要处理的因素更多，因此，若要对模型进行穷尽式测试，就需要编写更多测试组合。在实践中，这意味着采用函数式编程的模型所需编写的测试更少，因为无需考虑模型可能出现的所有状态。

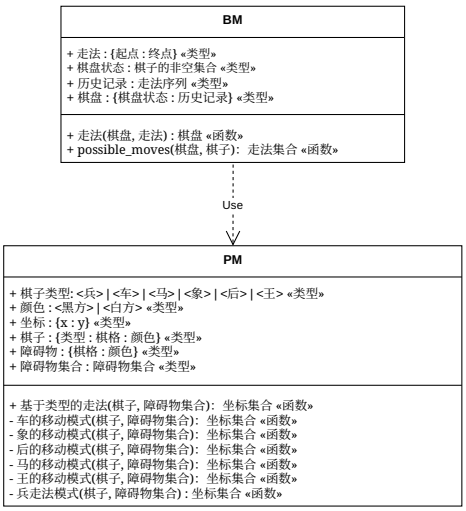

图3. 国际象棋函数式架构基础的统一建模语言类图。

然而，采用函数式编程架构对国际象棋进行建模也有一些局限，尤其是与棋局的有状态特性有关。例如，对王车易位进行建模时，需要知道相关的王和车是否移动过。采用函数式编程，要确定这一点，就必须了解棋局中先前走法，并判断这些棋子是否参与其中。采用面向对象编程时，只需为车和王添加一个布尔属性，用来指示它们是否移动过。虽然采用函数式编程也能做到这一点，但由于不支持继承，所有棋子都必须受到影响。

此时应该已经很清楚，两种架构类型各有优点。编写架构时，应考虑最重要的是尽量减少还是封装会变化的部分，因为这一决定可能会大幅影响实现的难度。最后，还必须考虑模型需要进行多么穷尽式的测试：与有状态模型相比，无状态模型的测试组合数应该更少。

函数式编程（FP）比面向对象编程（OOP）更简洁优雅的一个例子，可以从棋子的移动中看出。假设将*走法* 建模为由两个*棋子*组成的复合类型：一个表示要移动的*棋子*，另一个表示它要移动到的位置[2]，；并且*棋盘状态* 是一个*棋子*集合。在面向对象编程（OOP）的情况下，需要在当前*棋盘状态* 中找到属性与`Move.from_`相似的*棋子*，检查是否有*棋子* 占据与`Move.to_`相同的棋格，必要时移除该*棋子*，并更新找到的*棋子*，使其匹配新的*坐标*。如果发生升变，则需要添加一个升变后类型的新*棋子* 并移除原*棋子*。采用函数式编程（FP）时，只需通过筛选出*走法* 中坐标对应的*棋子*来更新*棋盘状态*，再与`Move.to_`取并集即可。

[2] 后者必须是一个位于某个坐标上的棋子，以便表示升变。

3.2 VDM++ 中复合类型的不变式

在开发早期采用面向对象架构时，VDM工具“VDMJ” [1] 中出现的一个问题引发了一场有趣的讨论，其核心涉及范式的选择。该问题与下方的模型代码片段有关，其中一个棋子 被移动到指定的坐标。该操作通过先移除被吃掉的棋子，再更新被移动的棋子的位置来更新棋盘 的状态。由于在 VDM 中，类实例属于引用类型++，因此可以直接对棋子 使用赋值运算符来完成后一步，因为棋盘_状态 持有对该实例的引用。棋盘状态 是一个棋子集合，并带有一个不变式_，规定集合中棋子的位置必须唯一。

```
/public 走法:棋子*棋子`坐标 ==>()走法(棋子,坐标) == (letcurrent_coords =state_to
_coords_set(board_state)inif坐标insetcurrent_coordsthenletdead_棋子 ={p |p in
setboard_state&p.position =坐标}inboard_state:=board_state\dead_棋子;pi
ece.position:=坐标)prepiece insetboard_stateanddead_棋子 insetboard_state;
```

模型的逻辑看起来是合理的，但在实际运行中，捕获棋子 时，解释器会报告不变式违反。调试该操作时发现，更新棋子的位置时，不变式确实得到了正确检查，但检查所依据的却是旧的棋盘_状态，其中仍包含已死亡_棋子，因而导致不变式被违反。简而言之，问题是由棋子 对象上的“不变式监听器”引起的；修改棋盘_状态 时，该监听器未得到正确更新[3]。虽然本文不关注这一具体问题，但在包含引用的复合类型上设置不变式所带来的复杂性，是一个有趣的话题，也展现了可变性引发的一些问题。

由于 VDM++ 对象是引用，因此它们会在语言中引入别名；也就是说，像棋子 这样的对象可以在规格说明中的多个位置被访问和修改。如果此类对象引用同时也是带有不变式的复合类型实例的成员，例如棋盘_状态，那么每当对象被修改时，都必须检查该复合类型的不变式。复合类型实例本身被修改时，也必须检查不变式。此外，像棋子 这样的对象可能会被放入多个具有不同不变式的不同复合类型实例中，这意味着修改棋子 会导致检查多个不变式。对于国际象棋来说，这可能不是问题，因为其数据结构相对扁平；但在复杂的工业案例中，情况可能并非如此。之所以会出现这个问题，是因为为了提高 VDM 解释器的效率，同时确保不变式不被违反，它只会在发生不变式适用的更改时检查不变式。要让这种机制在存在别名的情况下正常工作，本质上就需要检查相互关联实例的传递闭包。这显然效率不高，因此 VDM 解释器会忽略跨实例的不变式，这也给此处提出的面向对象模型带来了挑战。

[3] 此处可查看该问题的链接：
https://github.com/overturetool/vdm-vscode/issues/197

如果规格说明采用不可变数据类型（例如复合类型）编写，那么不变式问题就不必令人担忧，因为棋盘状态将是一个包含不可变值而非引用的集合。这意味着只有在棋盘_状态发生变化时，才需要检查不变式（或者完全采用函数式编程时，在创建新的棋盘状态之后检查）。

规格说明的撰写者通常不应过多关注工具的细节，但我们认为，这个例子有力地说明了遵循面向对象编程可能会给规格说明引入不必要的复杂性。如果要对遵循面向对象编程、且使用复合类型实例的规格说明进行推理，就需要考虑所有修改该实例的位置，以及所有可能修改其实例成员的位置。这还没有考虑继承等更复杂的面向对象编程概念，而这些概念只会进一步强化这一观点。

4 国际象棋的函数式编程 VDM 模型概述

本节所述模型遵循第 3 节中展示的函数式编程架构，并将自下而上地介绍模型中最有趣的部分。

4.1 棋子模块

棋子模块（PM）类负责定义描述国际象棋棋子所需的类型，并提供描述其基本移动模式所需的函数。

棋子类型可以建模为由以下引号类型构成的并集：<兵> | <车> | <马> | <象> | <后> | <王>。使用复合类型定义坐标，由两个自然数（含 0）组成，用于描述棋子所在位置的 x 坐标和 y 坐标。将以字母标注 x 轴这一惯例抽象掉，以保持两个坐标轴的一致性。对坐标设置不变式，以确保位置合法，即数值小于九。

现在可以将棋子的数据结构定义为一个复合类型，其中包含以下属性：棋子、类型 : 棋子类型、棋格 : 坐标和颜色 : 颜色。

给定一个棋子，可以使用类型_驱动的_走法函数来查找其普通走法（即不包括特殊走法）。该函数以棋子 * 障碍物集合为参数，并返回一个坐标集合，其中包含该棋子可以移动到的有效坐标。需要提供障碍物集合，以过滤掉因棋子的路径被障碍物阻挡而无效的走法。一个障碍物包含一个坐标和一个颜色，其中颜色用于指示该障碍物是否可以被吃掉。另一种做法是，不使用障碍物集合来建模该函数，而是假定棋盘为空，返回合法坐标的集合，因为这样可以进一步降低棋子模块与棋盘状态的耦合。不过，这会使棋盘模块与棋子类型的耦合更紧密，因为它需要针对兵的移动方式添加特殊分支：兵向前移动，但沿斜线攻击。

类型驱动的_走法_函数使用分支表达式编写，通过模式匹配棋子类型来调用定义该棋子特定移动模式的函数。马和王的移动模式可以使用一个通用的辅助函数可能_走法_方向来建模，如下方代码片段所示。调用时，除棋子和障碍物集合外，还会提供一对整数作为参数，指示需要考虑可能走法的方向。随后，通过坐标_工厂，根据方向生成一个新的 [坐标]；如果所提供的输入会导致坐标无效，该工厂函数就返回空值。如果生成的坐标无效或被己方棋子占据，可能_走法_方向的计算结果就是空值。否则，它会返回新的坐标。

```
/possible_move_direction:棋子*ObstacleSet*(int*int) -> [坐标]
possible_move_direction(p,os,dir) ==letnew_c =coordinate_facto
ry(p.square.x +dir.#1,p.square.y +dir.#2)inif(new_c =nil)orexistspiece
insetos&(piece.square =new_c)and(piece.colour =p.colour)thennilelse
new_c;\
```

现在已经定义了可能的_走法_方向，马的移动模式可定义如下。

```
/马_走法_模式: Piece*ObstacleSet ->setof坐标 knight_走法_模式
(p,os) ={possible_move_direction(p,os,mk_(1,2)), -- 2
Up1Right possible_move_direction(p,os,mk_(-1,2)), -- 2
Up1Left possible_move_direction(p,os,mk_(1, -2)), -- 2
Down1Right possible_move_direction(p,os,mk_(-1, -2)), -- 2
Down1Left possible_move_direction(p,os,mk_(2,1)), -- 1
Up2Right possible_move_direction(p,os,mk_(-2,1)), -- 1
Up2Left possible_move_direction(p,os,mk_(2, -1)), -- 1
Down2Right possible_move_direction(p,os,mk_(-2, -1)) -- 1
Down2Left}\\{nil}prep.type = <马>;\
```

类似地，其他棋子类型——车、象和后——也采用如下代码片段所示的 可能_走法_方向进行建模。这一次需要使用递归，因为这些棋子可以沿一个方向持续前进，直到遇到阻挡或吃子。再次生成一个新的 [坐标]。第一个条件判断与可能 _ 走法 _ 方向 中的相同，但此处返回空集合。接着进行检查，以确定对手是否位于正在评估的棋格上；如果是，则终止递归，并返回一个包含该坐标 的集合。最后定义递归情形：返回一个集合，该集合是包含坐标的集合与沿同一方向递归调用函数所得结果的并集。

```
/可能_移动_方向: 棋子*障碍集合*(整数*整数) ->坐标集合 可能_移动_方向(p,os,dir) ==令new_c =
坐标_工厂(p.棋格.x +dir.#1,p.棋格.y +dir.#2)在如果(new_c =为空)或存在棋子属于集合os&(
棋子.棋格 =new_c)且(棋子.颜色 =p.颜色)则{}否则如果存在棋子属于集合os&(棋子.棋格 =new_c)
且(棋子.颜色 =对立_颜色(p.颜色))
```

```
    则 {new_c}否则 {new_c} 并集 possible_moves_direction(mk_Piece(p.type,ne
    w_c,p.colour),os,dir);
```

原则上，这个递归函数应该有一个合适的度量来确保终止，但这并不简单。

后棋子的移动模式可定义如下。

```
后_走法_模式:              棋子 *    ObstacleSet  -> 坐标集合 坐标
后_走法_模式(p,            os) ==
  d并集{possible_moves_direction(p,    os, mk_(0,  1)),
          可能_移动_方向(p,            os, mk_(0,  -1)),
          可行_移动_方向 (p,           os, mk_(1,  0)),
          可行_移动_方向 (p,           os, mk_(-1, 0)),
          可 行_移 动_方 向(p,         os, mk_(1,  1)),
          可 行_移 动_方 向(p,         os, mk_(-1, -1)),
          可行_移动_方向(p,            os, mk_(-1, 1)),
          可行_移动_方向(p,            os, mk_(1,  -1))}
  pre p.type  = <后 >;
```

4.2 棋盘模块

棋盘模块（BM）类负责定义和更新国际象棋棋盘的状态。此外，它还会判断特殊规则走法是否可行。

BM 类中定义的第一个类型是复合类型 走法。最初，走法被建模为由一个棋子 * 坐标[4]组成的乘积类型。直到实现升变之前，这种结构都很合理，因为升变允许更改棋子类型，而旧定义无法表示这一点。因此，走法 被建模为具有起点_ 和终点_ 属性的类型，二者均为棋子类型。为走法 设置了一个不变式，内容如下：m.from_.颜色 = m.to_.颜色且 m.from_.棋格 <> m.to_.棋格，因为走法 不能改变颜色，且必须更新棋子的位置。

随后便可以将历史记录 定义为走法的序列。选择序列是因为走法的顺序很重要，而且如果玩家让棋子来回移动，序列中可能会出现重复项。

接着可以将 棋盘状态 定义为一个Piece 的集合1，因为集合的元素无序且不允许重复，所以选择集合；否则就意味着同一颜色和类型的两个棋子被放在同一个棋格上。为限制两个棋子占据相同位置，编写了以下不变式：forall p1,p2 in set b & p1 <> p2 => p1.square <> p2.square。最后，引入了一个棋盘类型，作为包含棋盘状态和历史记录的复合类型。

函数 possible_moves 负责查找某个 棋子 的合法走法集合，同时考虑简单移动模式和特殊移动模式。该函数会查找基于类型的简单走法集合、有状态的特殊走法集合，以及非法的有状态走法集合。随后，它计算前两者的并集，再减去后一集合。其逻辑本质上可以归结为“找出所有可能的走法，然后移除不可能的走法”。

[4] 由于为字段命名比通过“.#1”和“.#2”引用字段更清晰，因此将走法从乘积类型改为复合类型。

此处使用集合推导式，通过辅助函数 piece_坐标_to_move，将 simple_走法中的坐标转换为走法。

```
public 可能_走法 : 棋盘 * PM‘ 棋子  -> set of 走法
可能_走法 (棋盘,              棋子 )  ==(
   让   状态_p_走法  = stateful_可能_走法 (棋盘,              棋子 ),
        状态_imp_走法  = stateful_impossible_moves ( 棋盘 , 棋子 ),
        simple_走法  = PM‘ 类型_based_走法 ( 棋子 ,
            PM‘pieces_to_obstacles( 棋盘 . board_state )) 在
            ({ 棋子_坐标_到_走法 ( 棋子 ,          c )  |c 集合simple_走法 } 并集
                状态_p_走法 ) \ 状态_imp_走法
)
pre  棋子   in  集合 棋盘 . board_状态 ;
```

函数状态性_不可能_走法 会产生导致玩家失败的走法集合，因为国际象棋规则不允许执行此类走法。因此，该集合包含会使玩家被将军的走法；如果玩家已经被将军，则会筛除无法解除将军的走法。此外，如果棋子类型 为 <兵> 且可以升变，那么集合还包含走法：兵走到位于底线的棋格上，但不改变棋子类型，因为不升变棋子是非法的。

状态性_可能的_走法 是一个分派器，会考虑棋子的特殊规则，即王车易位、升变、吃过路兵，以及兵在首次走棋时可以前进两格的规则。

```
  (状 态 性 _ 可 能 的 _ 走 法 : 棋盘 * PM‘ 棋子  -> 集 合 走 法 状 态 性 _
可 能 的 _ 走 法 ( 棋盘 , 棋子 )  ==( 分情况 棋子 . 类型 :<兵>  -> 并集 { 兵_走
法_两步 ( 棋盘 . board_状态 , 棋子 ) , 吃_过路兵 ( 棋盘 , 棋子 ) , 兵_升变 (
棋盘 . board_状态 , 棋子 ) } ,<王 >  -> 王车易位 _可能 ( 棋盘 , 棋子 ) , 其他
 ->{} 结束 ) ;
```

下面的代码片段展示了如何实现一种特殊走法，其中对升变进行了建模。首先，定义局部底线_y 和可升变_类型。然后构造一个包含升变_格子的集合，其中仅包含兵通常可以走到且位于底线上的坐标。最后，使用多态辅助函数集合_组合_元组生成由可升变类型组合构成的元组集合可升变_类型

s

以及升变_格子，可用于返回升变走法的集合。

```
兵_升变 : 棋盘状态 * PM‘ 棋子  -> 走法集合
兵_升变 ( 棋盘_状态 , 兵 )  ==(
    令底线_y      = 如果  兵 . 颜色  = <白方 > 则          8  否则  1 ,
        可升变类型_类型  ={<马 >, <象 >, <车 >, <后 >} 在
        设   升变_格子  = { 坐标  |  坐标 在                集
          PM‘ 基于类型_的_走法 ( 兵 ,       PM‘ 棋子 _到 _障碍物 （棋盘 _状态）
            & 坐标 .y  = 最后_y } 在
            {mk_走法 （兵,     mk_PM‘ 棋子 （t _c _元组 .#1,     t _c _元组 .# 2, 兵 . 颜色 ））  |
              t _c _元组集合 集 _组合_元组 [PM‘ 棋子类型,                PM‘ 坐标 ]
              ( 可升变_类型,          升变_格子 )}
)
前 兵。类型  = <兵> 和 兵 嵌 入 棋 盘 _状 态 ;
```

到目前为止，这些函数主要关注如何确定合法走法。执行走法的方式也类似，通过函数走法 实现；其中，王车易位和吃过路兵需要采用不同的处理方式，因为前者会改变棋子类型，而后者会吃掉一个棋子，但不会移动到该棋格。其他类型的走法可通过走法_其他建模。前置条件规定走法 必须合法，后置条件则规定返回的棋盘 具有不同的棋盘状态，且历史记录更长。

```
㇀public 走法 : 棋盘 * 走法 -> 棋盘 走法 ( 棋盘 , 走法) ==if 走法 . from_ . 类型
= <王 >and is_castling ( 棋盘 , 走法)then 走法_王车易位 ( 棋盘 , 走法) else if (
走法 . from_ . 类型 = <兵>and is_en_passant ( 棋盘 , 走法 ) )then 走法_en_
passant ( 棋盘 , 走法)else 走法_other ( 棋盘 , 走法) pre 走法 in set 可能_走法 ( 棋盘 , 走法
. from_ ) and 走法 . from in set 棋盘 . boardstate_ _post l
en 棋盘 . history <len RESULT. history and 棋盘 . board_state <>
RESULT. board_state;⩡
```

定义 走法_其他如下所示。首先，找出可能被吃掉的棋子，并将其定义为局部定义中的 被吃_棋子。根据 棋盘状态上的不变式，可以保证它包含一个 棋子，或者是空集。然后，可以将新的 棋盘状态定义为 新_状态：它由先前的状态去掉 被吃_棋子，再加上更新后的已移动棋子组成。最后，返回新的棋盘。

```
㇀走法 其他 : 棋盘 * 走法 -> 棋盘_走法_其他 ( 棋盘 , 走法) ==( 让被吃_棋子 ={p |p 属于集合棋盘 . 棋盘
状态_ 状态 & p . 棋格 =走法 . 到达_ . 棋格 } 于 让新_状态 =( 棋盘 . 棋盘状态_ 状态 \ ( 被吃_棋子 并集{走法 . 起点
_} ) ) 并集{走法 . 到达_} 在构造_棋盘 ( 新_状态 , [走法] ^棋盘 . 历史记录 ) )前置条件前置条件_走法 ( 棋盘 , 走法)
后置条件后置条件_走法 ( 棋盘 , 走法 , 结果) ;
⩡
```

此外，可以定义一个默认_棋盘 辅助函数，用于创建如图 1 所示的初始局面棋盘，并将历史记录设为空。这将作为新棋局的起始局面。

```
㇀公开 default_board:() ->棋盘 default_board() ==(设 board_state : 棋盘状态 =
并集 { {mk_PM‘ 棋子 (<兵 >, mk_PM‘ 坐标 ( x , 2) , <白方 >) |x 属于集合 { 1 , . . . , 8 } } , {mk_PM‘ 棋子 (<兵
>, mk_PM‘ 坐标 ( x , 7) , <黑方 >) |x 属于集合 { 1 , . . . , 8 } }-- 对其他棋子类型重复此操作} 在mk_棋盘 ( b o
a r d _状态 , [] ) ) ;
⩡
```

4.3 游戏模块

游戏模块（游戏模块）类负责控制当前回合由谁进行，并宣布游戏获胜者。该模块定义了引号

类型 胜者，其取值为 [PM‘颜色 | <和棋>]，其中 空值 表示棋局仍在进行。此外，该模块定义了复合类型 棋局，其中包含一个棋盘 和一个表示回合的 PM‘颜色 。

函数走法 用于在棋盘 上执行走法，并可能据此确定胜者。首先，通过BM‘move 执行走法，并将结果保存到局部定义new_board中。然后检查对手在下一回合是否有任何有效走法。如果没有，则根据对手是否被将军，判定该棋局由当前玩家获胜或以和棋结束。

```
public 走法 : 棋局 * BM‘Move  -> (棋局 * 胜者 )
move( game , mov)  == (
   令new_棋盘  = BM‘走法( 棋局 . 棋盘 ,          mov ) ,
     opposite_c  = PM‘opposite_color( 棋局 . turn ) 在
     对所有分分彩  将新棋盘设为_棋盘的状态 &
       p.颜色  = 对方_c  => BM‘ 可能的走法（新_棋盘，p）  = {} 则
        如果BM‘ 在_将军（新_棋盘.棋盘状态,          对方_c）则
          mk_（棋局，棋局.回合）
        否则
          mk_( 棋局 ,  <和棋 >)
     否则
       mk_(mk_棋局( new_棋盘 ,   对手_c ) ,       无)
)
前置走法。来自。颜色  = 棋局。轮到且
  走法包含BM‘ possible_走法（棋局。棋盘,          走法。来自)
后置len 棋局。棋盘。历史          <len 结果。# 1。棋盘。历史
     且棋局。棋盘。棋盘_状态  <> 结果。# 1。棋盘。棋盘 _状态;
```

4.4 Runner

模型的最后一部分由一个名为 Runner的类组成，它读取 PGN 文件（第 5 节）的内容，通过 PGN 模块将其转换为走法，并逐步执行这些走法。

从 Runner 的角度来看，以图形方式呈现棋盘本身也可能很有趣。这可以通过 VDMPad [8] 或 ViennaTalk [9]实现。在 Overture 环境中，基于 Eclipse 时也支持此类可视化，但尚未在 VSC 版本中完全实现 [7]。

5 便携式棋谱记法

便携式棋谱记法 [11](PGN) 是许多在线国际象棋网站用于记录国际象棋棋谱的事实标准。[5] PGN 包含与国际象棋对局相关的信息（例如棋手信息）和走法文本，其中走法文本描述了棋局中棋子的实际移动。PGN 使用字母表示 x 轴，使用数字表示 y 轴，如第 2 节所述。一般而言，PGN 中的一步走法由作为第一个字符的棋子类型和作为第二、第三个字符的坐标 组成。此外，王车易位、将军和将死均有特殊记法，也可以添加额外信息以消除歧义。

[5]PGN 受到 chess.com https://lichess.org/ 和 chess24.com 等网站的支持。

引入 PGN 是为了通过使用一些真实数据进行测试，验证 VDM 模型的整体完整性。由于 VDM++ 不包含字符串处理库，因此有必要自行定义其中一部分。此外，PGN 类负责解析以 PGN 记谱法描述国际象棋棋局的有效 字符串，将其转换为 VDM++走法 表示，反之亦然。

```
值
 数字字符 =_ " 0123456789 ";
 数字_字符_到_自然数: 字符到自然数的映射 =
                { 数字_字符 ( i ) |-> i -1 | i 在 seq 数值型 _字符 };

 有效 x 字符 =_ _ " abcdefgh ";
 x_字符 _到_ 自然数 1: 字符到自然数的映射 1 =
                  { 有效 _x _字符 ( i ) |-> i | i在 seq 有效 _x _字符 };

 棋子 _类型_ 到 _字符串: 映射到 PM‘ 棋子类型 到 字 符 串 =
                              {<兵> |-> "", <车 > |-> "R", <马 > |-> "N",
                               <象 > |-> "B", <后 > |-> "Q", <王 > |-> "K" }

函 数_久久爱

public 走法_到_PGN_字符串: BM‘走法 ->字 符 串
走法_到_PGN_字符串（走法） ==
    令 棋子_类型 = 棋子 _类型 _转换为_字符串（走法.来自.类型），
       x = ( 逆函数 x _字符 _转换为_nat1)(走法.至_.棋格.x),
       y = （逆函数数值型 _字符 _转换为 _nat） （走法.至_.棋格.y）
    in
       棋子_类型 ^ [x] ^ [y ];
```

在解析国际象棋对局并将结果记录到文本文件时，走法_到_pgn_字符串 辅助函数会在Runner 中使用。将类型应用于棋子_类型_到_字符串，即可将棋子类型 映射为字符串。类似地，可以找到 x 坐标和 y 坐标，但此处使用的是逆映射，因为这些映射是单射。

这个类目前是用 VDM 编写的，但由于 VDM 并非专为解析字符串而设计，因此将这部分改用 Java 编写为一个新库会更合理，以便直接将这种外部格式读入 VDM 结构中。如果需要，甚至可以在其中使用扫描器和解析器生成器；不过，主要问题在于当提供的字符串不符合 PGN 语法时，如何报告潜在错误。

6 相关工作

国际象棋对局的规则也已被纳入其他形式化体系，但从不同的出版物中难以看出它们在多大程度上涵盖了王车易位等较复杂的规则 [3,4,10]。大多数其他针对国际象棋对局的形式化方法分析都是从模型检测的角度进行的。因此，重点主要在于分析国际象棋棋局可能出现的未来结果，或仅仅检查某个走法序列是否合法。相比之下，本文介绍的工作更侧重于以最自然的方式表示国际象棋对局规则。不过， [4] 尝试先对棋局规则进行概括，再将其具体化以适用于国际象棋等不同棋类。也可以在本文介绍的工作中更系统地复用这种方法，从而获益。

7 总结与未来工作

本文介绍了一个功能完备的 VDM++ 国际象棋模型，可用作教学示例，也可作为对其他国际象棋相关主题进行形式化分析的基础。该模型已被添加为可导入 Overture [6] Visual Studio Code 版本的模型之一，以便其他人也能使用完整的 VDM++模型进行试验。

总体而言，在考虑使用哪种范式对 VDM++ 中的系统进行建模时，我们发现各种范式各有优缺点。对于国际象棋棋局，我们发现函数式范式比面向对象范式（OOP）更合适，但也存在不足。如果为某个系统选择 OOP，就必须留意它可能带来的复杂性，尤其是与跨对象引用的不变式有关的问题。具有后置条件的操作也会面临同样的挑战，因为这些后置条件也可能涉及操作执行前其他对象中的实例变量。

致谢 我们要感谢匿名审稿人对本文初稿提出的宝贵意见。

引用

1. Battle, N.: VDMJ 用户指南。技术报告，Fujitsu Services Ltd.，英国（2009） 2. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: 面向对象系统的经过验证的设计。施普林格，纽约（2005）。https://doi.org/10.1007/b138800, http://overturetool.org/publications/books/vdoos/ 3. Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., Leuschel, M.: 将形式化规约集成到应用程序中：ProB Java API。《系统设计中的形式化方法》**58**(1), 160–187（2021） 4. Krings, S., Körner, P.: 使用形式化方法进行游戏原型开发。载于：Cerone, A., Roggenbach, M.（编）《形式化方法——人人皆可乐在其中》。第124–142页。施普林格国际出版，Cham（2021） 5. Lichess：国际象棋棋盘编辑器，https://lichess.org/editor 6. Lund, J., Jensen, L.B., Macedo, H.D., Larsen, P.G.: 在 VS Code 环境中实现 UML 和 VDM 支持。载于：Macedo, H.D., Pierce, K.（编）第20届 Overture 国际研讨会论文集。第51–66页。Overture（2022年7月） 7. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: 将 VDM 与可执行代码相结合。载于：Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E.（编）《抽象状态机、Alloy、B、VDM 与 Z》。计算机科学讲义，第7316卷，第266–279页。施普林格出版社，柏林，海德堡（2012），http://dx. doi.org/10.1007/978-3-642-30885-7_19，ISBN 978-3-642-30884-0 8. Oda, T., Araki, K., Larsen, P.G.: VDMPad：用于探索性 VDM-SL 规约的轻量级集成开发环境。载于：Plat, N., Gnesi, S.（编）FormaliSE 2015。第33–39页。与 ICSE 2015 联合举办，佛罗伦萨（2015年5月） 9. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk 和 Assertch：在 Pharo 4 上构建轻量级形式化方法环境。载于：国际 Smalltalk 技术研讨会论文集。第4:1–4:7页。布拉格，捷克共和国（2016年8月）

10. Saralaya, V., Kishore, J.K., Reddy, S., Pai, R.M., Singh, S.：使用 NuSMV 对国际象棋游戏进行建模与验证。载于：Abraham, A., Lloret Mauri, J., Buford, J.F., Suzuki, J., Thampi, S.M.（编）《计算与通信进展》。第 460–470 页。施普林格柏林海德堡，柏林，海德堡（2011年）11. 维基百科：便携式棋谱记法（2007年），https://en.wikipedia.org/w/index.php?title=Portable_Game_Notation&oldid=1101954154