پیوست ۳ روش‌های رسمی
به‌کارگیری نمادگذاری ریاضی برای مشخصات

برای تشریح استفاده از نمادگذاری ریاضی در مشخصات رسمیِ مولفه‌ی نرم‌افزاری، مسأله‌ی اداره‌کننده‌ی بلوک را بازبینی می‌کنیم که در فصل ۱۸ مطرح شد. سیستمی برای مدیریت بلوک‌ها به طور شماتیک در شکل ۲۸–۸ ارائه شد و قبل از ادامه‌ی کار در این‌جا، باید مرور شود.

مجموعه‌ای به نام BLOCKS شامل هر شماره‌ی بلوک است. AllBlocks مجموعه‌ای از بلوک‌ها است که بین 1 و MaxBlocks قرار دارد. این حالت با دو مجموعه و یک دنباله مدل‌سازی می‌شود. دو مجموعه عبارتند از used و free. هر دو شامل بلوک‌ها هستند– مجموعه‌ی used شامل بلوک‌هایی می‌شود که در حال حاضر در فایل‌ها استفاده شده‌اند و مجموعه‌ی free شامل بلوک‌هایی می‌شود که برای فایل‌های جدید موجود هستند.

دنباله شامل مجموعه‌ای از بلوک‌ها می‌شود که آماده‌ی آزادسازی از فایل‌هایی هستند که حذف شده‌اند. حالت می‌تواند به صورت زیر توصیف شود:

```
used free: ℙ BLOCKS
BlockQueue: seq ℙ BLOCKS
```

این حالت خیلی شبیه اعلان متغیرهای برنامه‌نویسی است و بیان می‌کند که used و free مجموعه‌هایی از بلوک‌ها هستند و BlockQueue یک دنباله خواهد بود که هر عنصر آن مجموعه‌ای از بلوک‌ها است. ثابت داده‌ای می‌تواند به صورت زیر نوشته شود:

```
used ∩ free = ∅∧
used ∪ free = AllBlocks ∧
∀ i: dom BlockQueue • BlockQueue i ⊆ used ∧
∀ i, j: dom BlockQueue • i ≠ j => BlockQueue i ∩ BlockQueue j = ∅
```

سطر اول از ثابت داده‌ای بیان می‌کند که بلوک‌های مشترکی در کلکسیون‌های استفاده‌شده و آزاد از بلوک‌ها وجود ندارد. سطر دوم بیان می‌کند که کلکسیون بلوک‌های استفاده‌شده و بلوک‌های آزاد، همیشه برابر با کل کلکسیون بلوک‌ها در سیستم است. سطر سوم نشان می‌دهد که عنصر iاُم در صف بلوک همیشه زیرمجموعه‌ای از بلوک‌های استفاده‌شده خواهد بود. سطر آخر بیان می‌کند که برای هر دو عنصر از صف بلوک که یکسان نیستند، بلوک‌های مشترکی در این دو عنصر وجود ندارد.

اولین عملی که باید تعریف شود، عملی است که عنصری را از جلوی صف بلوک حذف می‌کند. پیش‌شرط این است که حداقل یک آیتم باید در صف باشد:

```
#BlockQueue > 0,
```

پس‌شرط این است که جلوی صف باید حذف شود و در کلکسیونی از بلوک‌ها قرار گیرد و صف طوری تنظیم شود که عمل حذف را نشان دهد:

```
used′ = used\head BlockQueue ∧
free′ = free ∪ head BlockQueue ∧
BlockQueue′ = tail BlockQueue
```

قاعده‌ای که در روش‌های رسمی به کار می‌رود، این است که مقدار یک متغیر پس از یک عمل، اطلاعات لازم را در اختیار دارد. لذا، اولین مولفه از عبارت قبلی بیان می‌کند که بلوک‌های استفاده‌شده‌ی جدید (used′) برابر با بلوک‌های استفاده‌شده‌ی قدیمی منهای بلوک‌هایی است که حذف شده‌اند. مولفه‌ی دوم بیان می‌کند که بلوک‌های آزاد جدید (free′) همان بلوک‌های آزاد قدیمی هستند که جلوی صف بلوک به آن اضافه شده است. مولفه‌ی سوم بیان می‌کند که صف بلوک جدید برابر با دنباله‌ی مقدار قدیمی صف بلوک، یعنی تمام عناصر در صف به جز اولین عنصر است. عمل دوم، کلکسیونی از بلوک‌ها، Ablocks را به صف بلوک اضافه می‌کند. پیش‌شرط این است که Ablocks در حال حاضر مجموعه‌ای از بلوک‌های استفاده‌شده است:

```
Ablocks ⊆ used
```

پس‌شرط این است که مجموعه‌ی بلوک‌ها به انتهای صف بلوک اضافه می‌شود و مجموعه‌ای از بلوک‌های استفاده‌شده و آزاد بدون تغییر می‌ماند:

```
BlockQueue′ = BlockQueue ⌒ <Ablocks> ∧
used′ = used ∧
free′ = free
```

شکی نیست که مشخصات ریاضی صف بلوک به طرز چشمگیری از روایت زبان طبیعی یا مدل گرافیکی دقیق‌تر است. دقت بیشتر، به تلاش نیاز دارد، اما فواید حاصل از سازگاری بهبودیافته و کامل بودن، می‌تواند برای برخی دامنه‌های اپلیکیشن توجیه شود.

زبان‌های مشخصات رسمی

زبان مشخصات رسمی معمولاً ترکیبی از سه مولفه‌ی اصلی است: (۱) نحوی که نمادگذاری خاصی را تعریف می‌کند که مشخصات با آن نمایش داده می‌شود، (۲) دامنه‌ی معنایی برای کمک به تعریف "جهان اشیا" [Win90] که برای توصیف سیستم به کار می‌رود و (۳) مجموعه‌ای از رابطه‌ها که قواعد معنایی را تعریف می‌کند که نشان می‌دهد اشیا چگونه می‌توانند به طور مناسب دستکاری شوند و مشخصات را برآورده کنند.

دامنه‌ی معنایی یک زبان مشخصات رسمی، گاهی مبتنی بر یک نحو است که از نمادگذاری نظریه‌ی مجموعه‌ی استاندارد و حساب مسندها به دست می‌آید. دامنه‌ی معناییِ زبان مشخصات نشان می‌دهد که زبان چگونه نیازمندی‌های سیستم را نمایش می‌دهد.

امروزه انواع مختلفی از زبان‌های مشخصات رسمی مورد استفاده قرار می‌گیرند OCL [OMG03b]، Z [ISO02]، LARCH [Gut93] و VDM [Jon91] نمونه‌هایی از زبان‌های مشخصات رسمی هستند که مشخصات ذکرشده را دارند. در این پیوست مختصری از OCL و Z ارائه می‌شود.

زبان محدودیت شیء[1] (OCL)

زبان محدودیت شیء (OCL) یک نمادگذاری رسمی است که طراحی شد تا کاربران UML بتوانند دقت بیشتری به مشخصات خود اضافه کنند. کل قدرت منطق و ریاضیات گسسته در این زبان فراهم است. به هر حال، طراحان OCL تصمیم گرفتند که فقط کاراکترهای آسکی (به جای نمادگذاری ریاضی معمولی) باید در دستورات OCL استفاده شوند.

برای استفاده از OCL، با یک یا چند نمودار UML شروع می‌کنید- که غالباً نمودارهای کلاس‌ها، حالت یا فعالیت هستند (پیوست ۱). عبارات OCL اضافه می‌شوند و حقایقی درباره‌ی عناصر نمودارها بیان می‌کنند. این عبارات را **محدودیت‌ها** می‌نامند؛ هر پیاده‌سازی حاصل از مدل باید تضمین کند که هر یک از این محدودیت‌ها همیشه درست باقی می‌مانند.

مثل یک زبان برنامه‌نویسی شیءگرا، عبارت OCL شامل عملگرهایی هستند که روی اشیا عمل می‌کنند. به هر حال، نتیجه‌ی یک عبارت پیچیده همیشه باید بولی باشد، یعنی درست یا نادرست باشد. اشیا می‌توانند نمونه‌هایی از کلاس **کلکسیون** OCL باشند که دو زیرکلاس آن **Set** و **Sequence** هستند.

شیء **self** عنصری از نمودار UML در حیطه‌ای است که عبارت OCL از آن ارزیابی می‌شود. سایر اشیا می‌توانند با **گشت و گذار** از طریق نماد نقطه از شیء **self** به دست آیند. به عنوان مثال:

- اگر **self** کلاس **C** با خصیصه‌ی **a** باشد، آن‌گاه **self.a** به شیء ذخیره‌شده در **a** ارزیابی می‌شود.
- اگر **C** یک رابطه‌ی انجمنی یک به یک به نام **assoc** با کلاس دیگر **D** داشته باشد، آن‌گاه **self.assoc** به **Set** ارزیابی می‌شود که عناصر آن از نوع **D** هستند.
- سرانجام، اگر **D** دارای خصیصه‌ی **b** باشد، آن‌گاه عبارت **self.assoc.b** به مجموعه‌ای از تمام **b**های متعلق به تمام **D**ها ارزیابی می‌شود.

OCL اَعمال توکار را فراهم می‌کند که عملگرهای مجموعه و منطق، مشخصات سازنده و ریاضیات مرتبط را پیاده‌سازی می‌کند. نمونه‌ی کوچکی از این‌ها در جدول پ ۳-۱ آمده است.

برای تشریح استفاده از OCL در تعیین مشخصات، مثال اداره‌کننده‌ی بلوک را بررسی می‌کنیم که در فصل ۲۸ معرفی شد. مرحله‌ی اول، توسعه‌ی یک مدل UML (شکل پ۳-۱) است، این نمودار کلاس، رابطه‌های زیادی بین اشیا را مشخص می‌کند. به هر حال، عبارات OCL اضافه می‌شوند تا به پیاده‌سازی‌های سیستم اجازه دهند که دقیقاً بدانند وقتی سیستم اجرا می‌شود، چه چیزهایی باید درست باقی بمانند.

1. Object Constraint Language

جدول پ ۳-۱ خلاصه‌ای از نمادگذاری مهم OCL

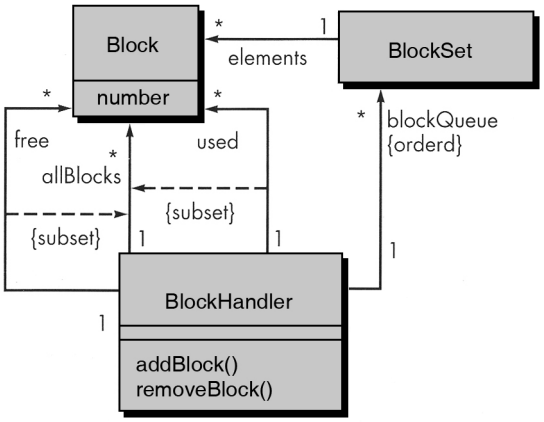
x.y	Obtain the property y of object x. A property can be an attribute, the set of objects at the end of an association, the result of evaluating an operation, or other things depending on the type of UML diagram. If x is a Set, then y is applied to every element of x; the results are collected into a new Set.
c->f()	Apply the built-in OCL operation f to Collection c itself (as opposed to each of the objects in c). Examples of built-in operations are listed below.
and, or, =, <>	Logical and, logical or, equals, not-equals.
p implies q	True if either q is true or p is false.
Sample of Operations on Collections (including Sets and Sequences)	
C->size()	The number of elements in Collection c.
C->isEmpty()	True if c has no elements, false otherwise.
c1->includesAll(c2)	True if every element of c2 is found in c1.
c1->excludesAll(c2)	True if no element of c2 is found in c1.
C->forAll(elem | boolexpr)	True if boolexpr is true when applied to every element of c. As an element is being evaluated, it is bound to the variable elem, which can be used in boolexpr. This implements universal quantification, discussed earlier.
C->forAll(elem1, elem2 | boolexpr)	Same as above, except that boolexpr is evaluated for every possible *pair* of elements taken from c, including cases where the pair consists of the same element.
C->isUnique(elem |expr)	True if expr evaluates to a different value when applied to every element of c.
Sample of Operations Specific to Sets	
s1->intersection(s2)	The set of those elements found s1 and also in s2.
s1->union(s2)	The set of those elements found in either s1 or s2.
s1->excluding(x)	The set s1 with object x omitted.
Sample Operation Specific to Sequences	
Seq->first()	The object that is the first element in the sequence seq.

عبارات OCL که نمودار کلاس را کامل می‌کنند، متناظر با شش قسمت از ثابت‌ها (تغییرناپذیرها) هستند که در بخش ۲۸-۶ معرفی شدند. در مثالی که در ادامه می‌آید، ثابت به زبان فارسی بیان می‌شود و سپس عبارت OCL متناظر آن نوشته می‌شود. فراهم کردن متن زبان طبیعی همراه با منطق رسمی، راهکار خوبی است؛ این کار به شما کمک می‌کند تا منطق را درک کنید و به مرورکنندگان کمک می‌کند اشتباهات را پیدا کنند، مثل وضعیت‌هایی که در آن‌ها منطق و فارسی با هم سازگار نیستند.

۱. هیچ بلوکی هم به عنوان استفاده شده و هم به صورت استفاده نشده علامت‌گذاری نمی‌شود:

```
context BlockHandler inv:
    (self.used->intersection(self.free)) ->isEmpty()
```

توجه کنید که هر عبارت با واژه‌ی کلیدی **context** شروع می‌شود. این واژه عنصر نمودار UML را نشان می‌دهد که عبارت آن را محدود می‌کند. واژه‌ی کلیدی **self** در این‌جا به نمونه‌ای از **BlockHandler** اشاره می‌کند. در ادامه، اگر در OCL مجاز باشد، **self** را حذف می‌کنیم.



شکل پ ۳-۱ نمودار کلاس برای اداره‌کننده‌ی بلوک

۲. تمام مجموعه بلوک‌هایی که در صف نگهداری می‌شوند، زیرمجموعه‌هایی از کلکسیون‌های بلوک‌های تازه استفاده شده‌اند:

```
context BlockHandler inv:
    blockQueue->forAll(aBlockSet | used->includesAll(aBlockSet ))
```

۳. هیچ عنصری از صف شامل شماره‌ی بلوک‌های یکسان نخواهد بود:

```
context BlockHandler inv:
    blockQueue->forAll(blockSet1, blockSet2 |
        blockSet1 <> blockSet2 implies
        blockSet1.elements.number->excludesAll(blockSet2.elements.number))
```

عبارت قبل از **implies** لازم است تا تضمین کند که جفت‌هایی را نادیده گرفتیم که هر دو عنصر در یک بلوک قرار دارند.

۴. کلکسیون بلوک‌های استفاده‌شده و بلوک‌های استفاده‌نشده برابر با کل کلسیون بلوک‌هایی است که فایل‌ها را می‌سازند:

```
context BlockHandler inv:
    allBlocks = used->union(free)
```

۵. کلکسیون بلوک‌های استفاده‌نشده، شماره‌های بلوک تکراری ندارند:

```
context BlockHandler inv:
    free->isUnique(aBlock | aBlock.number)
```

۶. کلکسیون بلوک‌های استفاده‌شده فاقد شماره‌های بلوک تکراری‌اند:

```
context BlockHandler inv:
    used->isUnique(aBlock | aBlock.number)
```

OCL می‌تواند برای مشخص کردن پیش‌شرط‌ها و پس‌شرط‌های اَعمال به کار رود. برای مثال، عبارت زیر اَعمالی را توصیف می‌کند که مجموعه‌ای از بلوک‌ها را به صف اضافه یا حذف می‌کند. توجه کنید که نمادگذاری **x@pre** نشان می‌دهد که شیء **x** قبل از عمل وجود دارد؛ این نمادگذاری، مقابل نمادگذاری ریاضی است که قبلاً بحث شد که در آن **x** پس از عملی وجود دارد که مشخص شده است (مثل **x′**):

```
context BlockHandler::removeBlocks( )
    pre: blockQueue->size( ) >0
    post: used = used@pre-blockQueue@pre->first( ) and
        free = free@pre->union(blockQueue@pre->first( )) and
        blockQueue = blockQueue@pre->excluding(blockQueue@pre->first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)
    pre: used->includesAll(aBlockSet.elements)
    post: (blockQueue.elements = blockQueue.elements@pre
        -> append (aBlockSet.elements) and
        used = used@pre and
        free = free@pre
```

OCL یک زبان مدل‌سازی است ولی تمام خصیصه‌های یک زبان رسمی را دارد. OCL اجازه‌ی عباراتی از محدودیت‌های مختلف، پیش‌شرط و پس‌شرط، نگهبان‌ها و سایر ویژگی‌هایی را مجاز می‌کند که به اشیای نشان داده شده در مدل‌های مختلف UML مرتبط می‌شود.

زبان مشخصات Z

Z یک زبان مشخصات است که به طور گسترده در جامعه‌ی روش‌های رسمی استفاده می‌شود. در زبان Z از مجموعه‌های نوع‌دار، رابطه‌ها و توابع در داخل حیطه‌ی منطق مُسند مرتبه‌ی اول استفاده می‌کند تا شِماها را بسازد که ابزارهایی برای ساخت مشخصات رسمی هستند.

مشخصات Z به صورت مجموعه‌ای از شِماها[1] سازمان‌دهی می‌شوند. شِماها برای ساختاردهی به مشخصات رسمی، درست مثل مولفه‌ها که برای ساختاردهی سیستم به کار می‌روند.

یک شِما، داده‌های ذخیره‌شده را توصیف می‌کند که سیستم به آن دسترسی دارد و آن را تغییر می‌دهد. در حیطه‌ی Z، آن را "حالت" می‌گویند. این نوع استفاده از واژه‌ی "حالت" در Z، کمی متفاوت از استفاده از این واژه در بقیه‌ی این کتاب است[2]. ساختار کلی یک شِما به صورت زیر است:

```
———schemaName———————————————
    declarations
————————————————————————————
    invariant
————————————————————————————
```

که در آن، اعلان‌ها متغیرهایی را مشخص می‌کنند که حالت سیستم را شکل می‌دهند و ثابت نیز محدودیت‌هایی را در چگونگی تکامل حالت اِعمال می‌کند. خلاصه‌ای از نمادگذاری زبان Z در جدول پ۳-۲ آمده است.

1. Schemas

۲. به خاطر داشته باشید که در فصل‌های دیگر، "حالت" برای شناسایی یک مُد از رفتار سیستم است که در خارج از آن قابل مشاهده است.

جدول پ ۳-۲ خلاصه‌ی نمادگذاری Z

Z notation is based on typed set theory and first-order logic. Z provides a construct, called a schema, to describe a specification's state space and operations. A schema groups variable declarations with a list of predicates that constrain the possible value of a variable. In Z, the schema X is defined by the form

X

declarations

---

predicates

Global functions and constants are defined by the form

declarations

---

predicates

The declaration gives the type of the function or constant, while the predicate gives it value. Only an abbreviated set of Z symbols is presented in this table.

Sets:

$S : \mathbb{P} X$	S is declared as a set of Xs.
$X \in S$	x is a member of S.
$x \notin S$	x is not a member of S.
$S \subseteq T$	S is a subset of T: Every member of S is also in T.
$S \cup T$	The union of S and T: It contains every member of S or T or both.
$S \cap T$	The intersection of S and T: It contains every member of both S and T.
$S \setminus T$	The difference of S and T: It contains every member of S except those also in T.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just x.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S : \mathbb{F} X$	S is declared as a finite set of Xs.
max (S)	The maximum of the nonempty set of numbers S.

Functions:

$f : X \rightarrowtail\!\!\!\!\!+ \; Y$	f is declared as a partial injection from X to Y.
dom f	The domain of f: the set of values x for which $f(x)$ is defined.
ran f	The range of f: the set of values taken by $f(x)$ as x varies over the domain of f.
$f \oplus \{x \mapsto y\}$	A function that agrees with f except that x is mapped to y.
$\{x\} \lhd f$	A function like f, except that x is removed from its domain.

Logic:

$P \wedge Q$	P and Q: It is true if both P and Q are true.
$P \Rightarrow Q$	P implies Q: It is true if either Q is true or P is false.
$\theta S' = \theta S$	No components of schema S change in an operation.

مثال زیر از یک شِما، حالت اداره‌کننده‌ی بلوک و ثابت داده‌ای را نشان می‌دهد:

———**BlockHandler**———————————————————————
used, free : ℙ *BLOCKS*
BlockQueue : *seq* ℙ *BLOCKS*
used ∩ *free* = ∅ ∧
used ∪ *free* = *AllBlocks* ∧
∀*i*: **dom** *BlockQueue* • *BlockQueue i* ⊆ *used* ∧
∀*i, j*: dom *BlockQueue* • *i* ≠ *j* => *BlockQueue i* ∩ *BlockQueue j* = ∅
———————————————————————————————————

همان‌طور که گفته شد، شِما شامل دو بخش می‌شود: بخش موجود در بالای خط مرکزی، متغیرهای حالت را نشان می‌دهد، درحالی‌که بخش پایین خط مرکزی، ثابت داده‌ای را توصیف می‌کند. هر وقت شِما اَعمالی را مشخص می‌کند که حالت را تغییر می‌دهد، با نماد Δ نمایش داده می‌شود. مثال زیر از یک شِما عملی را توصیف می‌کند که عنصری را از صف بلوک حذف می‌کند:

———RemoveBlocks———————————————
Δ *BlockHandler*
———————————————————————————

#BlockQueue > 0,
used′ = *used* \ *head BlockQueue* ∧
free′ = *free* ∪ *head BlockQueue* ∧
BlockQueue′ = *tail BlockQueue*
———————————————————————————

گنجاندن Δ BlockHandler موجب می‌شود تمام متغیرهای سازنده‌ی حالت، در دسترس شِمای RemoveBlocks باشند و اطمینان حاصل شود که ثابت داده‌ای، قبل و بعد از اجرای عمل برقرار است.

عمل دوم که یک کلکسیون از بلوک‌ها را به انتهای صف اضافه می‌کند، به صورت زیر نمایش داده می‌شود:

———AddBlocks———————————————————
Δ BlockHandler
Ablocks? : BLOCKS
———————————————————————————

Ablocks? ⊆ *used*
BlockQueue′ = *BlockQueue* ⌢ ⟨Ablocks?⟩ ∧
used′ = *used* ∧
free′ = *free*
———————————————————————————

بر اساس قواعد *Z*، یک متغیر ورودی که خوانده می‌شود ولی از نه از بخشی از حالت، به علامت سوال ختم می‌شود. لذا، Ablock? که به عنوان پارامتر ورودی عمل می‌کند، به علامت سوال ختم می‌شود.