POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akad. 2002/2003

Praca magisterska

Maciej Mochol

Z-notacja jako narzędzie specyfikacji formalnej

Kierownik pracy
dr. inż. Henryk Dobrowolski

Ocena:

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Streszczenie

Praca ta przedstawia w przekrojowy sposób zastosowanie metod formalnych do projektowania i weryfikacji oprogramowania, ze szczególnym uwzględnieniem notacji Z jako języka specyfikacji. Zostały w niej zawarte informacje na temat dowodzenia poprawności, metod i narzędzi do automatycznej analizy specyfikacji, oraz zarysu standardu ISO.

Słowa kluczowe: Z, Z-notacja, metody formalne

Z-notation as a tool for a formal specification

This thesis describes using the formal methods in a software development and verification, with special allowance of the Z notation as a specification language. There is information included showing the proof, methods and tools for an automatic specification analysis and an outline of ISO standard.

Keywords: Z, Z-notation, formal methods

Spis treści

Rozdział 1

Wstęp

Współczesne oprogramowanie staje się coraz większe i bardziej skomplikowane. W przypadku niewielkich systemów, możliwe jest aby jeden człowiek rozumiał wszystkie aspekty tworzonego kodu, jednak dotyczy to projektów składających się z 5 lub 6 uczestników (a i tu można by polemizować, że zależy to od wielu czynników i nawet ta granica może się okazać zbyt duża). Dla większych implementacji staje się to niemożliwe. Większość profesjonalnie projektowanego oprogramowania posiada dokumentację projektową, z dokładną specyfikacją dokumentującą działanie systemu, funkcjonalność, jego zachowanie w sytuacjach wyjątkowych i awaryjnych, itp. Te specyfikacje pisane są w fazie poprzedzającej implementację i mają służyć jako źródło informacji dla programistów w fazie pisania kodu, zbiór referencji dla ekipy testującej nowo powstałe oprogramowanie czy też punkt odniesienia dla osób piszących instrukcję lub też negocjujących z klientem. Specyfikacja jest wspólnym obszarem danych wszystkich osób zaangażowanych w projekt. Podkreślić należy, że opisuje ona co ma system robić, a nie w jaki sposób, jest też całkowicie niezależna od kodu, przez co może być ukończona we wczesnym etapie projektu.

Zależnie od przyjętej strategii projektowej specyfikacje mają różny kształt, zawierają również różną treść. Nie ma uniwersalnego sposobu na tworzenie dobrych specyfikacji, zależy to często również od wykorzystywanych technologii, rodzaju projektu lub też od innych czynników - np. konieczność współpracy z innymi systemami. Niemniej jednak wszystkie specyfikacje mają wspólny cel - w sposób jasny i zrozumiały powinny wyjaśniać wszystkie aspekty danego systemu, włączając w to jego funkcjonalność, format danych, scenariusze użycia, opis użytkowników i ich uprawnienia, sposób radzenia sobie z sytuacjami wyjątkowymi.

Zazwyczaj taka specyfikacja powstaje przy użyciu języka naturalnego, wszystkie jej elementy są dokładnie opisane językiem zrozumiałym dla człowieka. Niestety często prowadzi to do nieporozumień, bowiem okazuje się, iż w wielu sytuacjach programista odbiera tekst napisany przez projektanta niekoniecznie tak, jak życzyłby sobie tego ten pierwszy, a to z powodu faktu, że język ludzki obfituje w pułapki i wieloznaczności.

Dlatego od kilkunastu lat obserwujemy rosnące zainteresowanie formalnymi metodami specyfikacji i projektowania oprogramowania. Metody takie mają główną zaletę: dzięki formalizmowi jest dokładnie wiadomo, co dany zapis przedstawia, nie ma miejsca na wieloznaczności lub nieporozumienia. Zapisy formalne mogą być poddane automatycznej analizie w celu wykrycia błędów i nieścisłości, a nawet mogą posłużyć

do matematycznego udowodnienia poprawności danego projektu.

Celem tej pracy było zbadanie dostępnych metod i narzędzi, które nadają się do praktycznego wykorzystania w projektowaniu i implementacji systemów komputerowych. Pomimo tego, że jest ich wiele, niektóre nadal są w fazie badań, a inne z kolei wystarczają do pracy z jedną klasą problemów, będąc bezużytecznymi dla innej.

W niniejszym dokumencie przedstawię metody i narzędzia do formalnego opisu systemów, ze szczególnym naciskiem na **notację** $\mathbb{Z}$, która jako jedna z metod ścisłego projektowania specyfikacji z wykorzystaniem aparatu matematycznego, wyróżnia się funkcjonalnością i uniwersalnością.

Praca składa się z części teoretycznej i praktycznej.

W części teoretycznej czytelnik znajdzie wprowadzenie do notacji $\mathbb{Z}$ oraz matematycznych podstaw - logiki pierwszego rzędu, rachunku predykatów i teorii mnogości (w ujęciu teorii zbiorów Zermelo-Fraenkel), niezbędnych do tworzenia formalnej specyfikacji. Opisanym elementom notacji towarzyszą przykłady zastosowań, nawiązujące do tradycyjnych metod zapisu algorytmów i struktur danych.

Oprócz tego podane są metody dowodzenia poprawności specyfikacji - wraz z podstawowymi technikami wyprowadzania dowodów.

Część praktyczna pomoże potencjalnym projektantom oprogramowania w zapoznaniu się z dostępnymi narzędziami, wspomagającymi sprawdzanie poprawności zapisów oraz weryfikacji formalnej specyfikacji.

Część praktyczna zawiera formalny opis rzeczywistego systemu StockPack; użycie metod formalnych ułatwiło jego projekt i implementację.

Autor zakłada, że czytelnikowi znany jest aparat matematyczny leżący u podstaw notacji.

Rozdział 2

Wprowadzenie do notacji $\mathbb{Z}$

2.1. Podstawy notacji $\mathbb{Z}$

$\mathbb{Z}$ jest notacją formalnej specyfikacji, opartą na teorii zbiorów Zermelo-Fraenkel, logice pierwszego rzędu oraz rachunku predykatów. Została zaprojektowana na uniwersytecie w Oksfordzie (Programming Research Group at OUCL) w późnych latach siedemdziesiątych. Jest wykorzystywana jako część procesu produkcji oprogramowania w Europie oraz w USA. W 2002 powstał oficjalny standard ISO/IEC 13568 [2].

Jest to język wykorzystywany w specyfikacji, a nie implementacji. Nie ma więc narzędzia pełniącego rolę kompilatora lub linkera. Są projekty, które badają wykorzystanie $\mathbb{Z}$ do szybkiego tworzenia prototypów funkcji, jednak idea tego języka podąża w zupełnie innym kierunku. Notacja ta została zaprojektowana w celu podwyższenia czytelności specyfikacji projektu przez ludzi, oraz dostarczenia metod formalnej specyfikacji.

Dlatego potencjalny projektant ma do dyspozycji właściwość niedostępną językom programowania, a mianowicie niedeterminizm. Specyfikacja mówi nam bowiem, CO program ma robić, a nie w JAKI sposób. Dzięki temu programista tworzący konkretny kod ma pewną wolność, ponieważ projekt nie narzuca mu sposobu implementacji. Formalna specyfikacja określa działanie programu, które można osiągnąć różnymi metodami, zatem dopiero w fazie implementacji trzeba określić, która z nich jest najbardziej efektywna.

Co więcej niedeterminizm ten pozwala na stopniowe budowanie specyfikacji, poprzez stopniowe uszczegółowianie jej. Formalnie poprawny jest początkowy projekt wyznaczający główne ramy systemu, jak również końcowy, który zawiera informacje o wszystkich detalach związanych z jego działaniem.

W praktyce użycie tej specyfikacji polega na przeplataniu tekstu napisanego w języku naturalnym zapisów formalnych, przez co projektant od razu wyjaśnia przyświecające mu idee, popierając je odpowiednimi wzorami. Najczęściej używa się w tym celu systemu LaTeX, ponieważ pozwala on na automatyczną obróbkę i złożenie tekstu oraz zapisów, a do tego postać elementów języka $\mathbb{Z}$ jako makr LaTeX jest właściwie standardem i służy jako zamiennik ASCII notacji w formie symboli matematycznych. Makra te mogą być wprowadzane na wejście narzędzi sprawdzających poprawność zapisu **(type-checkers)** lub innych.

Notacja $\mathbb{Z}$ składa się właściwie z dwóch języków: matematycznego oraz schematów.

2.2. Przykłady rzeczywistych projektów, w których zastosowano $\mathbb{Z}$

Najsłynniejszym projektem, który został zaprojektowany z użyciem notacji Z jest wspólne przedsięwzięcie firmy IBM Labolatories at Hursley (Wielka Brytania) i Programming Research Group na Oxford University Computing Labolatory, znane jako projekt CICS. Projekt rozpoczęto w 1981 i w ramach niego opracowano specyfikację kilkunastu części wchodzących w skład systemu przetwarzania transakcji **IBM CICS**, szeroko wykorzystywanego w komputerach klasy *mainframe*. Szersze informacje można znaleźć w [12].

Inne znane projekty:

- System komputerowej kontroli urządzenia medycznego, służącego do klinicznej terapii neutronowej opartej na cyklotronie, które jest używane na Uniwersytecie Waszyngton w Seattle do leczenia raka. Funkcjonalność systemu została opisana z użyciem zbioru narzędzi do projektowania w $\mathbb{Z}$ systemów sterowania o podwyższonym poziomie bezpieczeństwa [13].

- Projekt formalnego opisu reguł bezpieczeństwa dla systemu **NATO ACCS** (ang. *Air Command and Control System*). W ramach niego sporządzono formalną specyfikację, łącznie z atestacją części specyfikacji wykonaną z użyciem tradycyjnych metod. Dostępny jest raport [14] zawierający wyniki.

- Specyfikacja reguł sygnalizacji Kolei Brytyjskich, jako część dokumentu opisującego wymagania dla systemu przełączania trakcji kolejowych. Specyfikacja została napisana w $\mathbb{Z}$ przez firmę Praxis Systems dla British Rail's Network SouthEast (obecnie Railtrack). Ich doświadczenia opisane są w [15].

- Implementacja mechanizmu transakcji dla **SWORD** - systemu relacyjnych baz danych. Jego celem jest kontrola wielu transakcji dostępu do bazy, bez niepotrzebnego blokowania danych. Podsumowanie projektu [16] zawiera również specyfikację $\mathbb{Z}$ tego mechanizmu.

2.3. Język matematyczny notacji

Język matematyczny wykorzystywany w notacji służy do opisu obiektów, oraz relacji zachodzących pomiędzy nimi. Składa się z zestawu typów i operatorów, co można porównać do biblioteki standardowej języka programowania, złożonej z typów i funkcji.

Wszystkie operatory $\mathbb{Z}$ bazują na fundamentalnych koncepcjach zaczerpniętych z logiki oraz teorii mnogości. Część z nich to anatomiczne pojęcia matematyczne takie jak zbiór, iloczyn kartezjański, albo relacja większości. Pozostałe są pochodnymi tych pojęć, wyprowadzonymi zgodnie z regułami zastosowanego aparatu matematycznego.

Ponieważ jak zostało to opisane poprzednio, notacja nie jest językiem programowania, lecz narzędziem do budowania specyfikacji; formalne zapisy nie muszą się składać jedynie z zapisów matematycznych. Często w przypadkach, kiedy trudno jest zdefiniować dane pojęcie, stosuje się opisy w języku naturalnym, łatwo zrozumiałym dla człowieka (np. zbiór skończony). W dalszych etapach specyfikowania i dokumentowania systemu zapisy te mogą być sformalizowane.

Część matematyczna notacji służy do określenia poziomu abstrakcji, dzięki czemu zachowania systemu mogą być opisane za pomocą bogatej kolekcji praw matematycznych. Nie jest ona zorientowana na typowo komputerowe podejście określenia formatu danych czy opisu algorytmów, ale na maksymalne wykorzystanie jej mechanizmów i logiki do stworzenia spójnego i formalnego opisu.

W rozdziale tym zostanie krótko omówione wykorzystanie elementów notacji, natomiast zestawienie operatorów i definicji znajduje się w dodatku A. $\mathbb{Z}$ to bardzo bogata notacja, zatem nie wszystkie jej elementy tutaj opisano, koncentrując się tylko na podstawowych i najczęściej wykorzystywanych zapisach. Dla tworzenia rzeczywistych specyfikacji może się okazać niezbędne skorzystanie z referencji języka, jaką jest książka Spivey'a [1], pełniąca przez długi czas funkcję standaryzacji $\mathbb{Z}$, oraz z opisu międzynarodowego standardu ISO [2].

2.3.1. Zbiory

Pojęcia zbiorów znane z teorii mnogości są kluczowym elementem $\mathbb{Z}$, ponieważ wykorzystywane są do modelowania danych. Ich zastosowanie w specyfikacji jest dość intuicyjne, podobnie jak innych pokrewnych pojęć takich jak: suma, różnica, iloczyn, potęga zbiorów, zbiór pusty, oraz relacje zawierania się i niezawierania.

Korzystając z tych elementów można wygodnie opisywać struktury danych. Nadają się one też do projektowania baz danych [1]. W zestawieniu z kwantyfikatorami i sekwencjami możemy strukturalizować dane oraz specyfikować ich format.

2.3.2. Pary i relacje binarne

Jeśli $x \in X$ i $y \in Y$, to zbiorem wszystkich par uporządkowanych (x, y) jest iloczyn zbiorów (produkt kartezjański) $X \times Y$. Relacją binarną R nazywamy dowolny podzbiór $X \times Y$. Jeżeli para $(x, y) \in R$, to mówimy, że elementy x, y są ze sobą w relacji i zapisujemy $x\ R\ y$.

Praktycznym zastosowaniem tych pojęć jest możliwość powiązania ze sobą dwóch zbiorów w/g określonych reguł. Oto przykłady.

Mamy dwa zbiory: użytkowników i uprawnień.

$$[UZYTKOWNICY, UPRAWNIENIA]$$

Teraz wiążemy te dwa zbiory, użytkownik jest uprawniony do czegoś, jeśli jest w poniższej relacji. Relacja ta opisuje uprawnienia użytkowników.

$$RELACJA : UZYTKOWNICY \leftrightarrow UPRAWNIENIA$$

[1] praca Barrosa [17] jest próbą formalnego opisu relacyjnych baz danych, wprowadzającego definicje podstawowych pojęć jak tabela, indeks, klucz obcy, lub transakcja.

Przykładowa para, będąca elementem relacji. Adam może zakładać konta.

$(ADAM, ZAKLADANIEKONT)$

lub

$ADAM \mapsto ZAKLADANIEKONT$

Zbiór pierwszych elementów wszystkich par, czyli dziedzina. W tym przypadku zbiór wszystkich użytkowników uprawnionych do czegoś:

dom $RELACJA$

Zbiór drugich elementów - czyli zbiór wszystkich uprawnień posiadanych przez użytkowników:

ran $RELACJA$

Zdefiniujmy jakiś zbiór użytkowników:

$U : \mathbb{P}\, UZYTKOWNICY$

Przydatny może być obraz relacyjny. Poniższy zapis oznacza zbiór wszystkich uprawnień nadanym użytkownikom ze zbioru U.

$RELACJA(\!|\ U\ |\!)$

2.3.3. Liczby i arytmetyka

W notacji $\mathbb{Z}$ są zdefiniowane liczby całkowite $\mathbb{Z}$ i naturalne $\mathbb{N}$. Ponadto określone są typowe operacje arytmetyczne, arytmetyka modulo, liczność zbioru, oraz minimum i maksimum zbiorów (ze zdefiniowanymi relacjami porównania).

W $\mathbb{Z}$ nie ma pojęcia liczb rzeczywistych [2]. Przykład zdefiniowania zbioru liczb (którego podzbiorem byłyby liczby naturalne) jest w [9]. Praca [18] przedstawia szczegółowy model liczb rzeczywistych $\mathbb{R}$ wyrażony w $\mathbb{Z}$ za pomocą sekwencji cyfr, wraz z częściowym dowodem ich właściwości.

2.3.4. Funkcje

Funkcje mają podobne zastosowanie jak relacje, kiedy chcemy powiązać ze sobą dwa zbiory danych. W/g przykładu z punktu 2.3.2 moglibyśmy zdefiniować funkcję, która zwraca uprawnienia użytkownika. W przykładzie zastosowano potęgę, ponieważ funkcja oczywiście nie może zwracać wielu uprawnień, ale może zwracać podzbiór zbioru *UPRAWNIENIA*.

$UZYTKOWNICY \nrightarrow \mathbb{P}\, UPRAWNIENIA$

[2] W stosunku do ZRM [1] pojęcie liczb zostało jednak nieco rozbudowane w standardzie ISO [2]. Został zdefiniowany specjalny typ $\mathbb{A}$ 'arithmos', na którym oparto definicję liczb całkowitych. W ten sam sposób można zdefiniować liczby rzeczywiste i inne, nie uczyniono tego jednak w standardzie.

Jest to funkcja częściowa, bo nie wszyscy użytkownicy muszą mieć uprawnienia (tzn. niektóre elementy zbioru *UZYTKOWNICY* są w parze z elementem ze zbioru *UPRAWNIENIA*). Gdybyśmy chcieli zapewnić, aby wszystkim użytkownikom przypisano uprawnienia, moglibyśmy napisać:

$$UZYTKOWNICY \to \mathbb{P}\, UPRAWNIENIA$$

Do dyspozycji projektanta dostępne są również injekcje i bijekcje. Przykładowo zakładając, że uprawnień jest tyle samo co użytkowników (tzn. liczność zbioru *UPRAWNIENIA* jest równa liczności zbioru *UZYTKOWNICY*), oraz że każdy użytkownik może mieć jedno uprawnienie, inne niż pozostałych użytkowników, to możemy napisać bijekcję:

$$UZYTKOWNICY \rightarrowtail\!\!\!\!\!\twoheadrightarrow UPRAWNIENIA$$

2.3.5. Ciągi (sekwencje)

Ciągi są wykorzystywane do opisywania struktur danych, w których ważna jest kolejność. Za pomocą ciągów można opisywać m. in. kolejki, strumienie danych lub pakiety w sieciach komputerowych. W języku matematycznym $\mathbb{Z}$ zdefiniowane jest pojęcie ciągów skończonych, wraz z zestawem operacji na nich.

Wyrażenie $\operatorname{seq} X$ oznacza zbiór wszystkich ciągów skończonych z X. Jego definicją jest

$$\operatorname{seq} X == \{\, f : \mathbb{N} \pfun X \mid \operatorname{dom} f = 1 \mathinner{..} \# f \,\}$$

Weźmy przykład:

$$KOLEJKA : \operatorname{seq} \mathbb{N}$$

Powyższa definicja mówi zatem o ciągu liczb naturalnych.

Ponieważ sekwencje zdefiniowane są jako skończone funkcje częściowe, czyli zbiory par, zawartość kolejki może być zapisana jako:

$$\{\, 1 \mapsto x_1, \ldots, n \mapsto x_n \,\}$$

lub w skrócie

$$\langle x_1, \ldots, x_n \rangle$$

i-ty element w kolejce to $KOLEJKA(i)$.

KOLEJKA może być pusta ($\langle\rangle$), to znaczy może nie zawierać żadnego elementu. Aby zagwarantować występowanie co najmniej jednego elementu w *KOLEJKA*, należy zapisać:

$$KOLEJKA : \operatorname{seq}_1 \mathbb{N}$$

Przykładowo, zapis

$$LICZBA : \operatorname{seq}_1 \{\, 0, 1 \}$$

oznacza liczby zapisane w postaci binarnej.
Kolejne pojęcie to ciąg pokrywający *(ang. injective sequence)* . Jego definicją jest:

$$\mathrm{iseq}\, X == \mathrm{seq}\, X \cap (\mathbb{N} \rightarrowtail X)$$

Jest to zatem zbiór skończonych ciągów z X, które nie zawierają powtarzających się elementów.
Analogicznie do powyższego przykładu:

$$LICZBA : \mathrm{iseq}\{\ 0, 1\}$$

Wedle tej definicji, prawdą jest, że:

$$LICZBA \in \{\ \langle\rangle, \langle 0\rangle, \langle 1\rangle, \langle 0, 1\rangle, \langle 1, 0\rangle\}$$

Na ciągach można wykonywać operację konkatenacji ($\frown$), można też stwierdzić czy A jest podciągiem B (A in B). W $\mathbb{Z}$ zdefiniowano kilkanaście innych operacji i operatorów na sekwencjach (np. *rev*, *tail*, $\uparrow$, $\restriction$, $\frown/$), których opis pomijamy.

2.3.6. Wielozbiory

Wielozbiór jest pochodną pojęć zbioru i sekwencji. Może on zawierać wiele elementów (tak jak w sekwencji), ale nie jest na nich określona kolejność (tak jak w zbiorze). Definicja:

$$\mathrm{bag}\, X == X \nrightarrow \mathbb{N}_1$$

Oznacza to, że niektórym elementom ze zbioru X jest przyporządkowana liczba wystąpień w wielozbiorze (> 1). Aby podać elementy wielozbioru, stosowana jest notacja:

$$[\![a, b, b, c]\!]$$

Wielozbiór może być pusty ($[\![]\!]$). Są na nim zdefiniowane:

- liczność (# lub *count*), która definiuje ilość wystąpień elementu x w wielozbiorze B, co zapisuje się jako $B\#x$ (lub *count* B x)
- operator skalowania ($\otimes$). Jeśli n jest liczbą naturalną, a B wielozbiorem, to $n \otimes B$ jest wielozbiorem, w którym każdy element x występuje $n-$krotną ilość razy częściej niż w B.
- relacja przynależności (in). x in B jest spełniona, jeśli element x występuje w B co najmniej raz.
- relacja zawierania 'podwielozbioru' ($\sqsubseteq$). Jeśli $B \sqsubseteq C$, to każdy element wielozbioru B występuje w nim nie częściej, niż w C.
- suma ($\uplus$). Każdy element x wielozbioru $B \uplus C$ występuje w nim tyle razy, ile łącznie w B i C.

- różnica ($\uplus$). Każdy element x wielozbioru $B \uplus C$ występuje w nim tyle razy ile B minus ilość wystąpień w C. Jeśli element występuje częściej w C niż w B, to nie występuje on w różnicy $B \uplus C$.

Jeśli nie interesuje nas kolejność elementów w sekwencji, to łatwo możemy ją przekonwertować na wielozbiór za pomocą operatora *idem* . Jeśli s jest sekwencją, to

$$idem\ s$$

jest wielozbiorem, w którym każdy element x występuje dokładnie tyle samo razy, w sekwencji s.

Użytecznym przykładem zastosowania wielozbioru jest specyfikacja koszyka z zakupami. Mamy zatem:

Zakupy

$$Koszyk : \text{bag}\ TOWARY$$

Typowe operacje na koszyku:

$$Dodaj \mathrel{\widehat=} [\ \Delta Zakupy;\ co? : TOWARY \mid Koszyk' = Koszyk \uplus [\![co?]\!]\ ]$$

$$Wyjmij \mathrel{\widehat=} [\ \Delta Zakupy;\ co? : TOWARY \mid co? \text{ in } Koszyk \land Koszyk' = Koszyk \uplus [\![co?]\!]\ ]$$

$$Ilosc \mathrel{\widehat=} [\ \Xi Zakupy;\ co? : TOWARY;\ ilosc! : \mathbb{N} \mid ilosc! = Koszyk\#co?\ ]$$

Ilosc może też być zdefiniowana jako funkcja, a nie operacja:

$$ilosc = (\lambda\, x : \text{dom}\ Zakupy.Koszyk \bullet Zakupy.Koszyk\#x)$$

2.3.7. Predykaty

Notacja $\mathbb{Z}$ jest oparta na logice pierwszego rzędu (ang. *FOL - First-Order Logic*). Oprócz dobrze znanego rachunku zdań (m.in. operatory $\land$, $\lor$, $\Rightarrow$, $\Leftrightarrow$, etc.), wykorzystuje się rachunek predykatów. Są one postaci:

$$Qx : a \mid p \bullet q$$

gdzie:

Q	-	kwantyfikator
x	-	zmienna wiązana (ang. *bound variable*)
a	-	zakres x
p	-	ograniczenie (ang. *constraint*)
q	-	predykat

Właściwości predykatów:

$$(\exists\, x : a \mid p \bullet q) \Leftrightarrow (\exists\, x : a \bullet p \wedge q)$$
$$(\forall\, x : a \mid p \bullet q) \Leftrightarrow (\forall\, x : a \bullet p \Rightarrow q)$$
$$(\exists\, x : a \bullet \exists\, y : b \bullet p) \Leftrightarrow (\exists\, x : a;\ y : b \bullet p)$$

Predykat może mieć wartość *true* lub *false*, co może być bezpośrednio zapisane w notacji (nie może mieć 'niezdefiniowanej' wartości).

Należy pamiętać, że predykaty nie mogą być używane do modelowania relacji lub funkcji (co jest częstym błędem) tylko raczej do opisu ich właściwości. Nie ma predefiniowanego typu *Boolean*; chociaż można zdefiniować typ zmiennej o wartościach binarnych, to lepiej tego unikać.

Podczas przekształcania predykatów często występuje problem redukcji kwantyfikatorów. Mając wyrażenie postaci:

$$(\exists\, x : X \bullet x = E \wedge \ldots x \ldots)$$

możemy pozbyć się kwantyfikatora egzystencjalnego przez zastosowanie 'reguły jednego punktu' (ang. *one-point rule*), która polega na zapisaniu równoważnego predykatu, w którym usuwamy kwantyfikator i równość $x = E$, a w pozostałej części zamieniamy wszystkie wystąpienia x na E.

2.3.8. $\mu-$notacja

Mając dany predykat/ograniczenie - aby stwierdzić, że istnieje tylko jeden zestaw wartości zmiennych, które go spełniają - możemy użyć $\mu-$notacji:

$$(\mu\, x : a \mid p)$$

Zapis taki określa unikalny element x należący do a, taki że p.

Przykładowo:

$$2 = (\mu\, n : \mathbb{N} \mid 4 + n = 6)$$

Jedyną liczbą naturalną, która po dodaniu do 4 daje 6 - jest 2.

2.3.9. $\lambda-$notacja

Notacja taka jest używana do skrócenia definicji funkcji. Mając daną funkcję:

$$f = \{\ x : X \mid p \bullet x \to e\}$$

Możemy zapisać to samo:

$$f = (\lambda\, x : X \mid p \bullet e)$$

Zbiorem argumentów funkcji są elementy z X, które spełniają wymaganie p. Każdemu x jest wtedy przyporządkowane e.

Przykład - funkcja kwadratowa:

$$sqr = (\lambda\, x : \mathbb{Z} \bullet x * x)$$

Funkcja *min*:

$$\begin{array}{|l} min : \mathbb{P}\,\mathbb{N} \pfun \mathbb{N} \\ \hline min = (\lambda\, s : \mathbb{P}\,\mathbb{N} \mid s \neq \emptyset \bullet \\ \qquad (\mu\, x : s \mid \forall\, y : s \mid y \neq x \bullet y > x)) \end{array}$$

2.3.10. Wyrażenia warunkowe - if then else

Wyrażenie takie ma postać:

$$\textbf{if } P \textbf{ then } E1 \textbf{ else } E2$$

gdzie P jest predykatem, natomiast $E1$ i $E2$ to wyrażenia. Warunkiem poprawności jest to, że $E1$ i $E2$ muszą być tego samego typu, który też jest typem całego wyrażenia. Jeżeli $P = true$ to jego wartością jest $E1$, w przeciwnym przypadku $E2$.

Przykład - wartość bezwzględna:

$$abs = (\lambda\, x : \mathbb{Z} \bullet \textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x)$$

2.4. Język schematyczny notacji i rachunek schematów

Notacja $\mathbb{Z}$ jest oparta na pojęciach matematycznych. Język matematyczny dostarcza spójnych i precyzyjnych mechanizmów opisu, co pozwala na zdefiniowanie mechanizmów systemu. Jednakże opis taki bez jasnej struktury byłby trudno zrozumiały, co uniemożliwiałoby stosowanie $\mathbb{Z}$ jako języka specyfikacji. Problem ten rozwiązuje język schematów $\mathbb{Z}$.

2.4.1. Definicje typów bazowych

Jest to zapis o postaci:

$$[TYP1, \ldots, TYPn]$$

Typy mogą być traktowane jako zbiory i w stosunku do nich mają sens np. operatory relacji lub zawierania. Jest to definicja wprowadzająca nazwę typu do specyfikacji, ale nie mówiąca nic o jego właściwościach.

Definicje te (podobnie jak definicje aksjomatyczne i schematy) są widoczne globalnie w specyfikacji od momentu zdefiniowania.

2.4.2. Aksjomaty

Aksjomat umożliwia wyspecyfikowanie zmiennych oraz warunków, jakie muszą spełniać ich wartości. Poprawna definicja składa się z deklaracji i ew. predykatów; ma postać:

$$n : \mathbb{Z}$$

lub

$$n : \mathbb{Z} \mid n > 4$$

2.4.3. Schematy

Jest to podstawowy element notacji $\mathbb{Z}$. Podobnie jak aksjomat, składa się z deklaracji i predykatów, ale jest nazwany, co pozwala odwoływać się do niego w innych częściach specyfikacji. Można tworzyć nowe za pomocą działań rachunku schematów. Przykład z punktu 2.7.2:

Rejestr

$$zarejestrowany : SAMOCHOD \pfun OSOBA$$

$$zarejestrowany \subseteq pozwolenie$$

Dostępna jest też równoważna definicja:

$$Rejestr \mathrel{\widehat=} [\, zarejestrowany : SAMOCHOD \pfun OSOBA \mid zarejestrowany \subseteq pozwolenie \,]$$

Schemat jest złożony (podobnie jak aksjomat) z dwóch części - deklaracji (tzw. sygnatury) i predykatów. Pomijając ich matematyczną definicję, możemy je przedstawić jako deklarację zmiennych wraz z opisem typu, oraz wyrażenia opisujące możliwe wartości tych zmiennych (czyli predykaty). Kolejność zmiennych nie jest określona - nie ma na nich określonej relacji porządku. Jeżeli część predykatu składa się z kilku wyrażeń w oddzielnych wierszach, to domyślnie są one połączone koniunkcją ($P \Leftrightarrow P_1 \land \ldots \land P_n$).

W definicji schematu można włączać referencje do innych schematów, np.

$$Urzad \mathrel{\widehat=} [\, Rejestr;\ \ldots \mid \ldots \,]$$

Do komponentów schematu można odwoływać się poprzez mechanizm selekcji:

$$Rejestr.zarejestrowany \notin \emptyset$$

Weźmy schemat:

Punkt

$$x, y : \mathbb{N}$$

$$x > 0 \land y > 0$$

Notacja $\theta Punkt$ oznacza sygnaturę schematu - czyli x, y. Predykat schematu określa możliwe wartości, jakie mogą przyjmować zadeklarowane zmienne. Stosując podejście obiektowe, możemy powiedzieć że jest to pewna klasa obiektów reprezentujących punkty. Aby opisać konkretny punkt - czyli instancję klasy, możemy napisać wiązanie [3] $\langle\!| x \rightsquigarrow 1, y \rightsquigarrow 1 |\!\rangle$. Prawidłowe jest przypisanie $\theta Punkt = \langle\!| x \rightsquigarrow 1, y \rightsquigarrow 1 |\!\rangle$.

Deklaracja $a : S$ jest skróconą formą zapisu $a : \{\ S \bullet \theta S\}$.

2.4.4. Przemianowywanie i dekorowanie

Przemianowywanie (ang. *renaming*) polega na utworzeniu nowego schematu, za pomocą starej definicji. Mając schemat S, to zapis

$$S[\ y_1/x_1, \ldots, y_n/x_n\ ]$$

tworzy nowy schemat, powstały przez zamianę każdego komponentu x_i na odpowiadający y_i. Aby to miało sens, odpowiadające sobie komponenty muszą być tego samego typu, poza tym nazwy x_i nie mogą się powtarzać (y_i mogą). Sygnatura powstaje przez podstawienie, a jeśli po tym okaże się, że występują dwa komponenty o tej samej nazwie, to są one łączone w jeden (pod warunkiem że są tego samego typu). Takie łączenie może zajść, gdy w schemacie S znajduje się komponent o tej samej nazwie co y_i, który nie uczestniczy w zamianie.

Dekorowanie jest operacją, która pozwala na przemianowanie wszystkich komponentów schematu poprzez dodanie przyrostka. Przykładowo jeśli S jest schematem, to S' jest takim samym schematem, w którym nazwy komponentów mają dodany przyrostek $'$. Jeśli sygnatura S zawiera element x, to S' zawiera x'.

Dekoracje mają fundamentalne znaczenie w opisie systemów sekwencyjnych (patrz 2.5). Standardowymi dekoracjami w $\mathbb{Z}$ są $'$ (które oznaczają stan po wykonaniu operacji), ? (które oznaczają wejścia operacji) oraz ! (wyjścia). Dopuszczalnymi dekoracjami są też indeksy (np. x_1), podwójne cudzysłowy (x''), lub kombinacje ($x?!$, $x'?$, $x_1!$).

2.4.5. Rachunek schematów

Schematy są dobrze zdefiniowanymi elementami $\mathbb{Z}$ notacji. Z ich właściwości wynika, że można na nich przeprowadzać różne działania, takie jak złożenie, albo aplikowanie standardowych operatorów logicznych, tworząc w ten sposób nowe schematy. Ma to zasadnicze znaczenie praktyczne i pozwala na łatwe dzielenie specyfikacji na bloki, które potem można składać ze sobą.

Operatory logiczne

Wspomniano już, że schematy składają się z sygnatury oraz predykatu. Sygnatury można łączyć ze sobą, pod warunkiem że są *kompatybilne pod względem typów*, tzn. że jeśli występują w nich deklaracje tych samych zmiennych, to muszą być tego samego typu. Ilustruje to przykład dwóch sygnatur:

[3] Należy zwrócić uwagę, że wiązania nie są częścią standardowej $\mathbb{Z}$ notacji w rozumieniu **ZRM** [1] - chociaż konstrukcja taka jest używana w tej książce. Wiązania są natomiast częścią standardu $\mathbb{Z}$ (p. 5.3.4). Narzędzia wspomagające projektowanie $\mathbb{Z}$ mogą zatem różnie je traktować.

$$a \in \mathbb{P}\, X;\ b \in X \times Y$$

i

$$b \in X \times Y;\ c \in Z$$

Są one *kompatybilne pod względem typów*, ponieważ typ wspólnej zmiennej b jest jednakowy dla obu sygnatur. Zatem ich połączenie to:

$$a \in \mathbb{P}\, X;\ b \in X \times Y;\ c \in Z$$

Zatem, mając dwa schematy spełniające powyższą definicję, możemy zdefiniować operacje logiczne. Weźmy:

$S1$

$a : A$
$b : B$

$P1$

i

$S2$

$b : B$
$c : C$

$P2$

Negacja schematu $\neg\ S1$ określona jest następująco (uwaga: schematy nienazwane nie są częścią standardowej notacji):

$a : A$
$b : B$

$\neg\ P1$

Koniunkcja:

$a : A$
$b : B$
$c : C$

$P1 \wedge P2$

W taki sam sposób zdefiniować można alternatywę schematów ($\vee$), implikację ($\Rightarrow$) i równoważność ($\Leftrightarrow$).

Kwantyfikacja

Nowy schemat możemy również utworzyć poprzez zastosowanie kwantyfikacji po wybranych elementach. Jeśli D jest deklaracją, P predykatem a S schematem, to

$$\forall D \mid P \bullet S$$

również jest schematem. Sygnatura zawiera wszystkie komponenty schematu S z wyjątkiem tych, które znajdują się również w D. Rezultat można opisać następująco: dla każdego wiązania z rezultatu weź wszystkie rozszerzenia z' do sygnatury S. Jeżeli dla każdego wiązania z', które spełnia warunki S spełnione są jednocześnie warunki D i predykat P, to wtedy oryginalne wiązanie z spełnia warunki $\forall D \mid P \bullet S$.

Mając

Tabliczka

$x : 0 \mathinner{..} 9$
$y : \mathbb{Z}$

$y = x * x$

to schemat $\forall x : \mathbb{Z} \mid x > 5 \bullet$ *Tabliczka* jest określony następująco:

$y : \mathbb{Z}$

$\forall x : \mathbb{Z} \mid x > 5 \bullet x \in 0 \mathinner{..} 9 \land y = x * x$

Należy zauważyć, że element x przeszedł z sygnatury do predykatu.

Podobne działania możemy wykonać z użyciem kwantyfikatorów $\exists$ i $\exists_1$ [4].

Ukrywanie

Ukrywanie (ang. *hiding*) nosi też nazwę egzystencjalnej kwantyfikacji. Pozwala on na pozbywanie się elementów z części deklaracyjnej schematu (podobnie jak kwantyfikowanie opisane w poprzednim punkcie). Zapis ten ma postać

$$S \setminus (x_1, \ldots, x_n)$$

i jest równoważny do

$$(\exists\, x_1 : t_1;\ \ldots;\ x_n : t_n \bullet S)$$

gdzie $x_1 \ldots x_n$ są typu $t_1 \ldots t_n$ w S.

Mając deklarację *Tabliczka*, schemat *Tabliczka* $\setminus (x)$ jest określony:

$y : \mathbb{Z}$

$\exists x : 0 \mathinner{..} 9 \bullet y = x * x$

[4] $\exists_1$ oznacza 'istnieje dokładnie jeden'

Można ukryć wszystkie komponenty schematu. Rezultatem jest schemat z pustą sygnaturą i predykatem, który ma wartość *true* lub *false* dla unikalnego pustego wiązania $\langle\!|\rangle$, zależnie od tego czy istnieją wiązania spełniające warunki oryginalnego schematu.

Projekcja

Projekcja jest rozszerzeniem ukrywania. Operator projekcji $S \upharpoonright T$ ukrywa wszystkie elementy z S, które także występują w T. S, T muszą być *kompatybilne pod względem typów*. Wyrażenie $S \upharpoonright T$ jest równoważne do

$$(S \wedge T) \setminus (x_1, \ldots, x_n)$$

gdzie $(x_1, \ldots, x_n)$ są wszystkimi komponentami S, których nie ma w T.

2.5. Systemy sekwencyjne

Schematy pozwalają na opisanie zarówno statycznych, jak i dynamicznych aspektów systemu. Część statyczna obejmuje:

- możliwe stany systemu
- relacje, które są zachowywane podczas przechodzenia systemu ze stanu do stanu

W tej definicji mieszczą się również deklaracje podstawowych typów, struktur danych, używanych zmiennych i zasobów. Modelowanie komponentów systemu również ma charakter statyczny o ile nie opisujemy interakcji pomiędzy nimi (co najwyżej wspólne zasoby). Charakterystyka zachowań systemu oddawana jest przez część dynamiczną:

- operacje na systemie
- możliwe zmiany stanu
- relacje pomiędzy wejściem a wyjściem stanów

Za pomocą dynamicznych elementów notacji możemy opisać interakcje pomiędzy komponentami systemu w sposób formalny i właśnie to stanowi o największej zalecie projektowania z użyciem notacji $\mathbb{Z}$. Zapisy te nie są już sekwencją instrukcji, którą trudno analizować, ale zbiorem założeń i twierdzeń dotyczących działania oprogramowania w formie schematów. Takie podejście umożliwia stosowanie logiki i rachunku predykatów, dzięki czemu projektant może udowodnić słuszność specyfikacji, wykazując że nie zachodzi sprzeczność pomiędzy poszczególnymi częściami projektu i że całość jest spójna w sensie formalnym.

2.5.1. Przestrzeń stanów

Weźmy schemat postaci:

StateSpace

$$x_1 : S_1;\ \ldots;\ x_n : S_n$$

$$Inv(x_1, \ldots, x_n)$$

Schemat ten definiuje pewną przestrzeń stanów. $x_1, \ldots, x_n$ to są zmienne stanu, natomiast $Inv(x_1, \ldots, x_n)$ jest *niezmiennikiem* stanu, tj. predykatem zawierającym zmienne $x_1, \ldots, x_n$, zawsze spełnionym przez możliwe wiązania schematu, niezależnie od operacji przeprowadzanych w systemie.

Klasyczny przykład z referencji Spivey'a [1]:

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

$$known = \mathrm{dom}\, birthday$$

Niezmiennikiem jest fakt, że każda znana osoba posiada datę urodzenia.

2.5.2. Operacje

Do zdefiniowania operacji wykorzystuje się dekorowanie (p. 2.4.4). Przyjmuje się, że schemat S opisuje stan systemu przed wykonaniem operacji, natomiast S' - po. Rozważmy *StateSpace* z poprzedniego punktu, wtedy *StateSpace*$'$:

StateSpace$'$

$$x_1' : S_1;\ \ldots;\ x_n' : S_n$$

$$Inv(x_1', \ldots, x_n')$$

wyraża przestrzeń stanów po wykonaniu operacji. Wystarczy teraz połączyć je ze sobą, aby otrzymać definicję operacji:

Operation

StateSpace
StateSpace$'$

$\ldots$

lub po rozwinięciu:

Operation

$x_1 : S_1;\ \ldots;\ x_n : S_n$
$x'_1 : S_1;\ \ldots;\ x'_n : S_n$

$Inv(x_1, \ldots, x_n)$
$Inv(x'_1, \ldots, x'_n)$

Aby skrócić ten zapis, używamy notacji Δ:

$\Delta StateSpace$

$StateSpace$
$StateSpace'$

Umieszczenie $\Delta StateSpace$ w części deklaracyjnej oznacza włączenie dwóch kopii tego schematu w celu opisania systemu sekwencyjnego.

Operację zmiany stanów można rozszerzyć o włączenie deklaracji wejść i wyjść (ang. *inputs, outputs*). Otrzymuje się je również poprzez dekorowanie, tym razem z użyciem symboli odpowiednio ? i !. Wersja rozszerzona wygląda teraz tak:

Operation

$x_1 : S_1;\ \ldots;\ x_n : S_n$
$x'_1 : S_1;\ \ldots;\ x'_n : S_n$
$i_1? : TI_1;\ \ldots;\ i_m? : TI_m$
$o_1! : TO_1;\ \ldots;\ o_p! : TO_p$

$Inv(x_1, \ldots, x_n)$
$Inv(x'_1, \ldots, x'_n)$
$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$
$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$

lub krócej

Operation

$\Delta StateSpace$
$i_1? : TI_1;\ \ldots;\ i_m? : TI_m$
$o_1! : TO_1;\ \ldots;\ o_p! : TO_p$

$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$
$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$

$(i_1?, \ldots, i_m?)$ to są wejścia operacji, $(o_1!, \ldots, o_p!)$ - wyjścia.

Warunkami wstępnymi operacji są predykaty zawierające zmienne wejść operacji $i_1?, \ldots, i_m?$ oraz zmienne stanu $x_1, \ldots, x_n$, co ilustruje wyrażenie $Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$.

Warunki wstępne operacji są ważne z punktu widzenia badania poprawności systemu (patrz p. 3). Notacja $\mathbb{Z}$ wprowadza zapis pre *Operation*, który jest schematem określonym następująco:

$$\exists\, StateSpace';\ o_1! : TO_1;\ \ldots;\ o_p! : TO_p \bullet Operation$$

Warunki wstępne są określone przez predykat P, jeżeli spełnione jest

$$\forall\, StateSpace;\ in? : IN \mid P \bullet \text{pre}\ Operation$$

Operacje nie muszą wiązać się ze zmianą stanu. W tym celu język schematów $\mathbb{Z}$ wprowadza następny zapis, $\Xi StateSpace$, określony jako

$\Xi StateSpace$

$StateSpace$
$StateSpace'$

$\theta StateSpace = \theta StateSpace'$

czyli

$$\Xi StateSpace \mathrel{\widehat=} [\ \Delta StateSpace \mid \theta StateSpace' = \theta StateSpace\ ]$$

2.5.3. Stan początkowy

W systemie sekwencyjnym, oprócz zdefiniowania przestrzeni stanów oraz operacji przejścia pomiędzy nimi, potrzebna jest też informacja o stanie początkowym.

Może ona być zdefiniowana następująco:

$Init$

$StateSpace'$

$Op(x'_1, \ldots, x'_n)$

Warunkiem poprawności takiej operacji jest

$$\exists\, StateSpace' \bullet Init$$

Kompletny dowód poprawności specyfikacji zawiera zarówno teorematy potrzebne do stwierdzenia, czy opisane operacje są możliwe do przeprowadzenia na danej przestrzeni stanów, jak również dowód osiągalności stanu początkowego.

2.5.4. Operator złożenia sekwencyjnego

Złożenie jest sposobem opisania dwóch następujących po sobie operacji. Mając operacje S i T, ich złożenie $S \fatsemi T$ to schemat zawierający wszystkie komponenty S i T z wyjątkiem x' schematu S i x schematu T, gdzie x jest odpowiednią zmienną stanu. Jest on określony:

$$\begin{aligned}&\exists\, State'' \bullet \\ &\quad (\exists\, State' \bullet [S;\ State'' \mid \theta State' = \theta State'']) \wedge \\ &\quad (\exists\, State \bullet [T;\ State'' \mid \theta State = \theta State''])\end{aligned}$$

$State''$ jest ukrytym stanem, na którym kończy S, a zaczyna się T.
Np. mając operacje:

$Op1$

$a, a' : Ab, b' : B$

P

i

$Op2$

$a, a' : Ab, b' : B$

P

ich złożenie to

$$Op1 \fatsemi Op2 = (Op1[\ a''/a', b''/b'\ ] \land Op2[\ a''/a, b''/b\ ]) \setminus (a'', b'')$$

co daje następujący schemat:

$a, a' : A$
$b, b' : B$

$$\exists\, a'', b'' \bullet P[\ a''/a', b''/b'\ ] \land Q[\ a''/a, b''/b\ ]$$

Należy podkreślić, że taka definicja sekwencji operacji różni się od tego, co znamy z języków programowania. Wystarczy bowiem by istniał *jakikolwiek* stan końcowy operacji $Op1$, który spełnia warunki wstępne operacji $Op2$. W rzeczywistym programie zazwyczaj każdy stan, w jakim kończy się operacja $Op1$ musi spełniać warunki wstępne $Op2$. Taki warunek poprawności, który nakłada to ograniczenie w celu zagwarantowania poprawnej implementacji $Op1$; $Op2$ złożenia $Op1 \fatsemi Op2$ można zapisać:

$$\forall\, State'' \bullet (\exists\, Op1 \bullet \theta State' = \theta State'') \Rightarrow (\exists\, Op2 \bullet \theta State = \theta State'')$$

Oznacza to, że jakikolwiek stan, w jakim kończy się operacja $Op1$, spełnia warunki wstępne operacji $Op2$. Oczywiście warunkiem poprawności złożenia jest też fakt, że obie operacje są określone na komponentach tego samego typu.

2.5.5. Operator potoku

Operator potoku jest używany do połączenia dwóch operacji, które są niezależne od siebie - tzn. są określone na innych danych i nie muszą (chociaż mogą) mieć współdzielonych komponentów. W $Op1 \gg Op2$ wyjścia operacji $Op1$ (oznaczone przez !) są

podawane na wejście operacji $Op2$ (czyli komponenty dekorowane przez ?). W nowym schemacie komponenty uczestniczące w łączeniu są ukrywane, natomiast pozostałe są łączone tak jak dla koniunkcji $Op1 \wedge Op2$.

Mając schemat

Wplata

$\Delta Konto$
$kwota? : \mathbb{N}$
$nowe\_saldo! : \mathbb{N}$

$saldo' = saldo + kwota?$
$limit' = limit$
$nowe\_saldo! = saldo'$

oraz operację bez zmiennych stanu

Podwojenie

$x?, y! : \mathbb{N}$

$y! = x? + x?$

możemy zapisać

$$WplataPodwojna \mathrel{\widehat=} Podwojenie \gg Wplata[\ y?/kwota?\ ]$$

Stosując dekorowanie przyporządkowujemy właściwe wyjścia *Podwojenie* do odpowiednich danych wejściowych operacji *Wplata*. Otrzymujemy wtedy schemat:

WplataPodwojna

$\Delta Konto$
$x? : \mathbb{N}$
$nowe\_saldo! : \mathbb{N}$

$saldo' = saldo + x? + x?$
$limit' = limit$
$nowe\_saldo! = saldo'$

2.6. Zaawansowane elementy notacji

2.6.1. Definicje uogólnione

Definicje uogólnione (ang. *generic definitions*) wzbogacają $\mathbb{Z}$ o możliwość szablonowego specyfikowania elementów specyfikacji, które się powtarzają i są niezależne od pozostałych. Ma to swoje odbicie w rzeczywistych systemach komputerowych, w których często występują składowe elementy o analogicznych właściwościach. Przykładowo możemy za pomocą tych definicji opisać system zarządzania zasobami, niezależnie od ich

typu (pliki, bloki pamięci, komputery w klastrze). W ten sposób da się również opisać funkcję spełniającą określone założenia (np. funkcję haszującą) - niezależnie od zbiorów, na których operuje.

Pojęcie definicji uogólnionych nasuwa skojarzenia z szablonami *(ang. templates)*, znanymi z języka programowania C++, choć oczywiście są to odmienne pojęcia. Pozwalają one na dalszą strukturalizację i uproszczenie specyfikacji.

Definicja uogólniona ma postać:

$[X_1, \ldots, X_n]$

D

P

Identyfikatory X_i mogą być używane w części deklaracyjnej, jak i w predykacie. Oczywiście definicję taką można nazwać, otrzymując wtedy uogólniony schemat:

$GenericSchema[X_1, \ldots, X_n]$

D

P

Z tak zdefiniowanego schematu można korzystać tak samo jak ze zwykłego schematu, pod warunkiem że odwołując się do niego podamy wartości dla identyfikatorów X_i, np. $S[\mathbb{Z}, \mathbb{P}\,\mathbb{N}]$ [5].

Możliwe jest pominięcie notacji schematycznej $\mathbb{Z}$ i stosowanie zwykłych wyrażeń, dopuszczonych przez składnię notacji i widocznych globalnie. Przykład: relacja częściowego porządku:

$$
\begin{array}{l}
partial\_order[\,X\,] \;== \\
\quad \{\, \_\underline{R}\_ : X \leftrightarrow X \mid (\forall\, x, y, z : X \bullet \\
\qquad x \underline{R} x \land \\
\qquad (x \underline{R} y \land y \underline{R} x) \Rightarrow x = y \land \\
\qquad (x \underline{R} y \land y \underline{R} z) \Rightarrow x \underline{R} z)\}
\end{array}
$$

Relacja porządku:

$$
\begin{array}{l}
total\_order[\,X\,] \;== \\
\quad \{\, \_\underline{R}\_ : partial\_order[\,X\,] \;\mid \\
\qquad (\forall\, x, y : X \bullet x \underline{R} y \lor y \underline{R} x)\}
\end{array}
$$

Oczywiście dowód przeprowadzony na definicji uogólnionej, jest również uogólniony, co skraca też proces dowodzenia poprawności.

[5] w $\mathbb{Z}$ dostępny jest mechanizm automatycznej asercji właściwych wartości, co pozwala na uniknięcie ich podawania. Reguły na to podane są w książce Spivey'a [1], określone też są w standardzie ISO [2]. Praktyka pokazuje jednak, że jest to niebezpieczne, szczególnie może sprawiać kłopoty podczas dowodzenia poprawności.

2.6.2. Typy wolne

Notacja dla definicji typów wolnych została stworzona, aby ułatwić zapis struktur rekursywnych, takich jak listy i drzewa. W/g [1] składnia definicji typu wolnego wygląda następująco:

$$\begin{array}{ll} \mathit{Paragraph} & ::= \mathit{Ident} ::= \mathit{Branch} \mid \ldots \mid \mathit{Branch} \\ \mathit{Branch} & ::= \mathit{Ident}[\langle\langle \mathit{Expression} \rangle\rangle] \end{array}$$

Definicja

$$T ::= c_1 \mid \ldots \mid c_m \mid d_1\langle\langle E_1[T]\rangle\rangle \mid \ldots \mid d_n\langle\langle E_n[T]\rangle\rangle$$

definiuje nowy typ podstawowy T, oraz $m + n$ zmiennych $c_1, \ldots, c_m$ i $d_1, \ldots, d_n$, co jest równoważne deklaracji:

$$\begin{array}{|l} c_1, \ldots, c_m : T \\ d_1 : E_1[T] \rightarrowtail T \\ . \\ . \\ . \\ d_n : E_n[T] \rightarrowtail T \\ \hline \ldots \text{patrz nizej} \ldots \end{array}$$

Zmienne c_i są stałymi typu T, natomiast d_j nazywane są konstruktorami i są injekcjami ze zbiorów $E_j[T]$ na T. W wyrażeniach $E_j[T]$ mogą znajdować się deklaracje typu T, co pozwala na definicje rekursywne.

Aby definicja typu wolnego nie była cykliczna, stałe c_i występują przed zmiennymi d_j, co powoduje, że typ T jest już zadeklarowany jako typ podstawowy, zanim następuje deklaracja wyrażeń d_j.

Poprawność definicji warunkuje spełnienie dwóch aksjomatów. Pierwszy mówi, że zarówno stałe c_i muszą być rożne, jak również zakresy zbiorów wartości konstruktorów muszą być rozłączne. Opisuje to:

$$\mathsf{disjoint}\ \langle \{c_1\}, \ldots, \{c_m\}, \operatorname{ran} d_1, \ldots, \operatorname{ran} d_n \rangle$$

Po drugie najmniejszy podzbiór T, który zawiera wszystkie stałe i jest zamknięty pod konstruktorami, to jest T [6]. Opisuje to reguła indukcyjna:

$$\begin{array}{l} \forall\, W : \mathbb{P}\, T \bullet \\ \quad \{c_1, \ldots, c_m\} \cup d_1(\!|\, E_1[W] \,|\!) \cup \ldots \cup d_n(\!|\, E_n[W] \,|\!) \subseteq W \\ \quad \Rightarrow T = W \end{array}$$

$E_j[T]$ to wyrażenie, które powstaje jako rezultat zamiany wszystkich wolnych wystąpień T w $E_j[T]$ na W. Jeżeli konstruktory E_j są skończone, wtedy ta reguła indukcyjna implikuje, że stałe i konstruktory razem pokrywają cały zbiór T, więc

$$\langle \{c_1\}, \ldots, \{c_m\}, \operatorname{ran} d_1, \ldots, \operatorname{ran} d_n \rangle\ \mathsf{partition}\ T$$

[6] Zbiór jest zamknięty pod konstruktorami jeśli zastosowanie któregokolwiek z nich na jakimś elemencie zbioru generuje jedynie inny element z tego zbioru

Przykład 1 Najprostszym przykładem definicji typu wolnego jest przypadek, gdy występują tylko stałe, czyli konstruktory bez argumentów.

$$BARWY ::= CZERWONA \mid ZIELONA \mid NIEBIESKA$$

W/g powyższego, definicja ta jest równoważna do:

$$[BARWY]$$

$$CZERWONA, ZIELONA, NIEBIESKA : BARWY$$

$$\mathsf{disjoint}\ \langle \{CZERWONA\}, \{ZIELONA\}, \{NIEBIESKA\} \rangle \land$$
$$\forall W : \mathbb{P}\, BARWY \bullet$$
$$\quad \{CZERWONA, ZIELONA, NIEBIESKA\} \subseteq W \Rightarrow BARWY = W$$

Przykład 2 Klasycznym przykładem typu wolnego jest zbiór drzew binarnych oznaczonych liczbami naturalnymi.

$$TREE ::= tip \mid fork\langle\langle \mathbb{N} \times TREE \times TREE \rangle\rangle$$

Stała *tip* jest pustym drzewem, a jeśli n to liczba, a t_1 i t_2 to drzewa, zatem $fork(n, t1, t2)$ to również jest drzewo. Definicja ta jest równoważna do:

$$[TREE]$$

$$tip : TREE$$
$$fork : \mathbb{N} \times TREE \times TREE \rightarrowtail TREE$$

$$\mathsf{disjoint}\ \langle \{tip\}, \operatorname{ran} fork \rangle$$
$$\forall W : \mathbb{P}\, TREE \bullet$$
$$\quad \{tip\} \cup fork(\!| \mathbb{N} \times W \times W |\!) \subseteq W$$
$$\quad \Rightarrow TREE = W$$

Konstruktor *fork* jest injekcją, a zatem dając mu jako argument dwa różne drzewa, lub drzewa oznaczone innym numerem, otrzymujemy różne rezultaty. Zakres *fork* jest rozłączny ze zbiorem $\{tip\}$:

$$tip \notin \operatorname{ran} fork$$

więc *tip* nie może być rezultatem konstruktora *fork*.

Właściwości typów wolnych można dowodzić za pomocą indukcji i podanej reguły indukcyjnej.

2.7. Przykłady zastosowań $\mathbb{Z}$

2.7.1. Przetwarzanie tekstu

W przykładzie tym chciałbym przedstawić model $\mathbb{Z}$, podany za [11], pokazujący specyfikację programów służących do przetwarzania tekstu. Analogiczne specyfikacje można również znaleźć w [10].

Dzielenie tekstu na słowa

Zacznę od zdefiniowania zbioru znaków. Tekst jest sekwencją znaków. Niektóre znaki są znakami 'pustymi' (spacje, tabulatory, końce linii). Odstęp jest sekwencją takich znaków, natomiast słowo jest sekwencją, która ich nie zawiera. Słowa mogą być oddzielone odstępem.

$$[ZNAK]$$

$$pusty : \mathbb{P}\, ZNAK$$

$$TEKST == \mathrm{seq}\, ZNAK$$
$$ODSTEP == \mathrm{seq}_1\, pusty$$
$$SLOWO == \mathrm{seq}_1(ZNAK \setminus pusty)$$

TEKST może być ciągiem pustym, ale *ODSTEP* and *SLOWO* muszą mieć co najmniej jeden znak, zatem deklarujemy je jako seq_1 (niepuste sekwencje).

Przedstawiane programy do zliczania i formatowania bazują na funkcji *slowa*. Funkcja ta zwraca sekwencję wszystkich słów w tekście. Przykładowo:

$$slowa\ \langle A, l, a, \ , m, a, \ , k, o, t, k, a\rangle = \langle\langle A, l, a\rangle, \langle m, a\rangle, \langle k, o, t, k, a\rangle\rangle$$

Zatem *words* to funkcja *TEKST* do sekwencji *SLOWO*. Aby ją zdefiniować, określę wszystkie możliwe wzorce słów oraz odstępów, dla każdej z nich pisząc równanie.

$$slowa : TEKST \to \mathrm{seq}\, SLOWO$$

$$\begin{aligned}
&\forall\, o : ODSTEP;\ s : SLOWO;\ l, p : TEKST \bullet \\
&\quad slowa\ \langle\rangle = \langle\rangle \land \\
&\quad slowa\ o = \langle\rangle \land \\
&\quad slowa\ s = \langle s\rangle \land \\
&\quad slowa\ (o \frown p) = slowa\ p \land \\
&\quad slowa\ (l \frown o) = slowa\ l \land \\
&\quad slowa\ (l \frown o \frown p) = (slowa\ l) \frown (slowa\ p)
\end{aligned}$$

Jak widać wzorców tych nie jest zbyt wiele. Kiedy tekst jest pusty lub jest odstępem, rezultat również jest pusty. Kiedy tekst składa się z jednego słowa, wynikiem jest to słowo. Kiedy tekst zaczyna się lub kończy odstępami, można je wyciąć. Kiedy złożenie tekstu zawiera odstęp, można go podzielić na dwa, ignorując odstęp.

Ten przykład ilustruje techniki $\mathbb{Z}$, które potrafią sprawić, że definicje są krótsze i bardziej przejrzyste niż kod. Funkcja składa się z wyrażeń, które ukrywają wewnętrzną strukturę argumentów. Definicja jest rekursywna. Ostatnie wyrażenie jest niedeterministyczne - nie ustala początku ani końca tekstu. Zatem tekst może być skanowany od początku do końca tekstu, albo odwrotnie, nie określamy tego pozwalając uczynić to programiście.

Zliczanie słów

Liczba słów w tekście t to $\#(slowa\ t)$. Można zdefiniować funkcję podobną do *slowa*, która dzieli tekst na linie. Zakładam, że koniec linii wyznaczany jest specjalnym znakiem pustym *nl*.

$$linie : TEKST \rightarrow \mathrm{seq}\, LINIA$$

$\ldots$ definicja pominięta $\ldots$

Teraz jest wszystko co trzeba, aby napisać formalną specyfikację dla Unixowego polecenia *wc*. To popularne narzędzie to właściwie funkcja, której argumentem jest nazwa pliku, a wynikiem trójka liczb: liczba linii, słów oraz znaków w pliku. Typowe wywołanie:

```
% wc struktura.tex
110 559 4509
```

Oto definicja *wc*:

$$wc : TEKST \rightarrow (\mathbb{N} \times \mathbb{N} \times \mathbb{N})$$

$$\forall\, plik : TEKST \bullet$$
$$wc\ plik = (\#(linie\ plik), \#(slowa\ plik), \#plik)$$

Lub krócej

$$wc == (\lambda\, plik : TEKST \bullet (\#(linie\ plik), \#(slowa\ plik), \#plik))$$

Wypełnianie paragrafów

Prawie każdy edytor tekstu zapewnia operację *wypełnienie*. Operacja ta zamienia nieuporządkowany tekst z liniami o różnej długości w sformatowany tekst, którego wszystkie linie mają prawie tę samą długość.

Zdefiniujmy operację wypełnienia w $\mathbb{Z}$. Można zauważyć, że wypełnienia to przykład operacji formatowania tekstu, który zmienia jego wygląd poprzez łamanie linii w różnych miejscach, oraz poszerzając odstępy pomiędzy słowami, zakładając że linie nie wychodzą poza obszar strony. Co więcej, operacja formatowania nie może zmienić zawartości tekstu: zachowuje te same słowa w oryginalnej kolejności.

$$szerokosc : \mathbb{N}$$

Format

$$t, t' : TEKST$$

$$slowa\ t' = slowa\ t$$
$$\forall\, l : \mathrm{ran}\ (linie\ t') \bullet \#l \leq szerokosc$$

Operacja wypełnienia to operacja, która spełnia dodatkowy warunek - linie powinny być wypełnione jak tylko jest to możliwe. Jest wiele różnych metod aby to zapewnić, chyba najprostszą zasadą jest wymaganie, że wypełniony tekst zawiera jak najmniej linii.

Wypelnienie

$$Format$$

$$\#(linie\ t') = min\ \{t' : TEKST \mid Format \bullet \#(linie\ t')\}$$

Ta definicja mówi, że *Wypelnienie* jest operacją *minimalizacji*, jest to specjalizacja *Format*, która minimalizuje liczbę linii w tekście. Funkcja *Format* w definicji *Wypelnienie* użyta jest jako predykat.

Definicja ta jest niedeterministyczna, jest zwykle wiele dróg do tego, aby zapewnić tę samą najmniejszą liczbę linii.

2.7.2. System rejestracji samochodów

Chciałbym przedstawić model $\mathbb{Z}$ systemu rejestracji samochodów. Osoby mogą zarejestrować samochód pod warunkiem posiadania sprawnego przeglądu technicznego. Nie może zostać wydana rejestracja, jeśli samochód nie posiada ważnego przeglądu. Ten sam samochód nie może zostać zarejestrowany przez dwie osoby, może być jednak niezarejestrowany. Zdefiniuję zatem procedurę rejestracji samochodu.

Oto model $\mathbb{Z}$. Najpierw wprowadzić trzeba zbiory, które zawierają wszystkie dane potrzebne w naszym systemie. Zbiory te oprócz samochodów i właścicieli określają bazę przeglądów.

$$[OSOBA, SAMOCHOD, PRZEGLAD]$$

Następnie wyspecyfikować trzeba relacje użyteczne do procesu rejestracji.

Pierwsza łączy samochody z właścicielami. Osoba jest w relacji z samochodem, jeśli jest jego właścicielem.

$$pozwolenie : SAMOCHOD \leftrightarrow OSOBA$$

Druga mówi o związku samochodu z przeglądem.

$$sprawnosc : SAMOCHOD \leftrightarrow PRZEGLAD$$

Stan systemu jest relacją tego samego typu co *pozwolenie*, co określa na kogo zarejestrowany jest dany samochód. Wymaganiem jest, że samochód może należeć tylko do jednej osoby, zatem w tym przypadku mamy do czynienia z funkcją.

Rejestr

$$zarejestrowany : SAMOCHOD \nrightarrow OSOBA$$

$$zarejestrowany \subseteq pozwolenie$$

Trzeba zauważyć, że *zarejestrowany* jest częściową funkcją, co oznacza że niektóre samochody mogą nie być zarejestrowane na nikogo. Predykat mówi o konieczności bycia właścicielem samochodu.

Formalna specyfikacja bazy przeglądów:

Przeglady

$$zatwierdzony : SAMOCHOD \pfun PRZEGLAD$$

$$zatwierdzony \subseteq sprawnosc$$

Potrzeba teraz dwóch operacji do zmiany stanu systemu: *Zarejestruj* i *Wyrejestruj*. Oto *Zarejestruj*, które ma dwa parametry: osoba $o?$ i samochód $s?$.

Zarejestruj

$$\Delta Rejestr$$
$$\Xi Przeglady$$
$$o? : OSOBA$$
$$s? : SAMOCHOD :$$

$$s? \notin \dom zarejestrowany$$
$$(s?, o?) \in pozwolenie$$
$$s? \in \dom zatwierdzony$$
$$zarejestrowany' = zarejestrowany \cup \{(s?, o?)\}$$

Zarejestruj ma trzy warunki. Samochód nie może już być zarejestrowany (nie może być w dziedzinie *zarejestrowany*), osoba musi być właścicielem i samochód musi posiadać ważny przegląd. Jeśli te warunki są spełnione operacja dodaje parę $(o?, s?)$ do relacji *zarejestrowany*.

Analogiczna operacja *Wyrejestruj*:

Wyrejestruj

$$\Delta Rejestr$$
$$o? : OSOBA$$
$$s? : SAMOCHOD$$

$$s? \in \dom zarejestrowany$$
$$zarejestrowany' = zarejestrowany \setminus \{(s?, o?)\}$$

Samochód można wyrejestrować, jeśli jest zarejestrowany.

Trzeba teraz określić definicje dla przypadków, dla których warunki nie są spełnione do zarejestrowania. Są trzy warunki, więc muszą być trzy przypadki.

Pierwszy, że samochód nie jest zarejestrowany, skoro para $(o?, s?)$ znajduje się już w relacji *zarejestrowany*, to znaczy, że samochód jest już dopuszczony do ruchu:

$$Dopuszczony \mathrel{\widehat=} [\ \Xi Rejestr;\ s? : SAMOCHOD \mid s? \in \dom zarejestrowany\ ]$$

Drugi, że osoba nie jest upoważniona do rejestracji.

Nieupowazniony

$$\Xi Rejestr$$
$$o? : OSOBA$$
$$s? : SAMOCHOD$$
$$(s?, o?) \notin pozwolenie$$

I trzeci, że samochód nie ma ważnego przeglądu technicznego.

$$Niesprawdzony \mathrel{\widehat=} [\ \Xi Rejestr;\ \Xi Przeglady;\ s? : SAMOCHOD \mid s? \in \operatorname{dom} zatwierdzony\ ]$$

Całkowita operacja rejestracji obejmuje wszystkie możliwości.

$$T\_Zarejestruj \mathrel{\widehat=} Zarejestruj \lor Dopuszczony \lor Nieupowazniony \lor Niesprawdzony$$

2.7.3. Projekt komunikacji między radiomodemami SATEL

Tym razem przedstawię formalną specyfikację rzeczywistego projektu komunikacji pomiędzy radiomodemami SATEL. Protokół opisany w tej sekcji został zaimplementowany i uruchomiony. Założeniem systemu było opracowanie komunikacji w systemie składającym się ze stacji nadrzędnej oraz jednej lub kilku stacji podrzędnych. Ze względu na architekturę modemów podstawą komunikacji było rozgłaszanie - tzn. jeden węzeł wysyłał komunikat, a wszystkie pozostałe go odbierały.

Zakłada się, że architektura systemu jest znana i niepotrzebna jest ani elekcja jednostki nadrzędnej, ani dynamiczne konfigurowanie jednostek podrzędnych (włączanie do transmisji, itp).

Zaczniemy od wprowadzenia zbioru węzłów i komunikatów:

$$[NODES, MESSAGES]$$

Maksymalna liczba węzłów:

$$max\_nodes : \mathbb{N}$$

Węzły są adresowane liczbą naturalną. Adresy nie mogą się powtarzać. Można zatem zapisać sieć węzłów jako injekcję zbioru węzłów na zbiór liczb naturalnych:

$$network : NODES \rightarrowtail \mathbb{N}$$

Zdefiniujmy teraz zbiór pakietów. Każdy pakiet składa się z adresu węzła oraz z komunikatu.

$$PACKETS : \operatorname{ran} network \times MESSAGES$$

Przyda się nam funkcja *address*, która dla podanego węzła w przyporządkuje jego adres:

$$address : NODES \to \mathbb{N}$$

$$\forall w : NODES \bullet$$
$$(\exists l : network \mid first\ l = w \bullet address(w) = second\ l) \land$$
$$(w \notin \operatorname{dom} network \Leftrightarrow address(w) = 0)$$

Następnym użytecznym pojęciem będzie potwierdzenie komunikatu, które też jest komunikatem. Specyfikujemy więc funkcję ACK:

$$ACK : MESSAGES \to MESSAGES$$

Zdefiniujmy zatem system, który składa się z jednej stacji nadrzędnej m i jednej lub kilku podrzędnych. System może być w stanie SEND lub RECEIVE (co zostanie opisane później). Z każdym węzłem powiązana jest kolejka komunikatów wejściowych, co jest wyrażone za pomocą funkcji *queues*.

System

$$m : network$$
$$slaves : \mathbb{P}\, network$$
$$queues : NODES \to \operatorname{seq} MESSAGES$$
$$state : \{SEND, RECEIVE\}$$

$$\#slaves \in \{1 \mathinner{..} max\_nodes - 1\}$$
$$\#queues = 1 + \#slaves$$
$$\operatorname{dom} queues = \operatorname{dom} slaves \cup \{\ first\ m\ \}$$

W powyższym schemacie wprowadzona została również definicja stanu.

Mamy teraz wszystko, aby opisać operację wysłania pakietu z węzła nadrzędnego do podrzędnego - *TransmitMtoS*. Aby operacja zakończyła się sukcesem, wymagane jest potwierdzenie otrzymania pakietu przez węzeł podrzędny. Możemy więc wyodrębnić trzy przypadki.

TransmitMtoS_Success

$$\Delta System$$
$$p? : PACKETS$$

$$\forall s \in slaves \mid address(s) = first\ p? \bullet$$
$$queues' = queues \oplus$$
$$\{\ s \mapsto (queues(s) \frown \langle second\ p? \rangle),\ m \mapsto (queues(m) \frown \langle ACK(second\ p?) \rangle)\ \}$$

Sukces tej operacji wynika z faktu, że zarówno do kolejki komunikatów docelowego węzła podrzędnego został dodany komunikat z pakietu, jak również że do kolejki komunikatów nadawcy zostało dodane potwierdzenie.

Może się jednak zdarzyć sytuacja, kiedy nie dojdzie potwierdzenie do nadawcy.

TransmitMtoS_FailReceive

$$\Delta System$$
$$p? : PACKETS$$

$$\forall s \in slaves \mid address(s) = first\ p? \bullet$$
$$queues' = queues \oplus \{ s \mapsto (queues(s) \frown \langle second\ p? \rangle) \}$$

Pozostaje jeszcze sytuacja, kiedy w ogóle komunikat nie zostanie dostarczony odbiorcy:

TransmitMtoS_FailSend

$$\Xi System$$

Całkowita operacja *TransmitMtoS* obejmuje wszystkie trzy możliwości.

$$TransmitMtoS \mathrel{\widehat=} TransmitMtoS\_Success \lor TransmitMtoS\_FailSend \lor TransmitMtoS\_FailReceive$$

W systemie przewidziana jest również komunikacja pomiędzy węzłami podrzędnymi. W analogiczny sposób można zdefiniować operację *TransmitStoS*, która zawiera operacje składowe (ich definicje pomijam).

$$TransmitStoS \mathrel{\widehat=} TransmitStoS\_Success \lor TransmitStoS\_FailSend \lor TransmitStoS\_FailReceive$$

Transmisję z węzła podrzędnego do nadrzędnego (operacja *TransmitStoM*) opisuje się w ten sam sposób. Należy jednak zauważyć, że dotychczas zdefiniowaliśmy jedynie funkcjonalność transmisji komunikatu, a dokładniej fakt, że po udanym przesłaniu pakietu stacja otrzymuje potwierdzenie, a dany pakiet jest dodawany do kolejki stacji docelowej. Przejdziemy teraz do definicji w jaki sposób może to być zrealizowane.

Jak już zostało wspomniane, system może się znajdować w stanie SEND lub RECEIVE.

SEND

Stan ten wykorzystywany jest podczas transmisji danych z węzła nadrzędnego do podrzędnego. Jeśli system znajduje się w tym stanie, to wysłanie pakietu od nadrzędnego do podrzędnego wymaga jedynie wykonania operacji *TransmitMtoS_Success*. Możemy więc zapisać:

$$SendMtoS\_Success \mathrel{\widehat=} [\ \Delta System;\ p? : PACKETS \mid (state = SEND) \land (TransmitMtoS\_Success(p?))\ ]$$

RECEIVE

W tym stanie w systemie może być przesłany komunikat od węzła podrzędnego (który normalnie nie może zainicjować transmisji) do nadrzędnego lub do innych podrzędnych. Jest to zrealizowane poprzez cykliczne odpytywanie węzłów podrzędnych przez nadrzędny.

Węzeł nadrzędny wysyła pakiet startu z adresem jednostki podrzędnej. Następnie podrzędny ma obowiązek wysłać potwierdzenie (jest to operacja *TransmitMtoS*), a potem pakiet przeznaczony dla innych jednostek. Pakiety te są adresowane, a ponieważ wszystkie jednostki (łącznie z nadrzędną) nasłuchują, mogą wtedy odebrać dane dla nich przeznaczone. Po wysłanym pakiecie adresowana jednostka musi 'odpowiedzieć' - czyli potwierdzić pakiet. Są to operacje *TransmitStoM* oraz *TransmitStoS*. Następnie z powrotem prawo nadawania przyznawane jest stacji nadrzędnej.

Proces ten składa się zatem z trzech etapów:

1. inicjalizacji transmisji, co odbywa się poprzez wysłanie pakietu startu od stacji nadrzędnej do podrzędnej i przejście do stanu RECEIVE,
2. przesłania komunikatu od stacji podrzędnej,
3. zakończenia transmisji - powrotu do stanu SEND.

Zakładamy, że transmisję rozpoczynamy w stanie SEND. Mamy więc początek transmisji:

$$SendStart \mathrel{\widehat=} [\ TransmitMtoS \mid p? = start\_packet \land state = SEND \land state' = RECEIVE\ ] \setminus (p?)$$

i koniec:

$$SendEnd \mathrel{\widehat=} [\ \Delta System \mid state' = SEND\ ]$$

Definiujemy zatem operacje *SendStoSSuccess*.

$$SendStoS\_Success \mathrel{\widehat=} SendStart \fatsemi TransmitStoS\_Success \fatsemi SendEnd$$

Oraz *SendStoMSuccess*.

$$SendStoM\_Success \mathrel{\widehat=} SendStart \fatsemi TransmitStoM\_Success \fatsemi SendEnd$$

Przykład ten ilustruje wykorzystanie różnych narzędzi matematycznych $\mathbb{Z}$. Specyfikacji tej można zarzucić też brak niektórych detali, np. nie ma tu opisu zależności czasowych. Wykorzystano niedeterminizm $\mathbb{Z}$, który stwarza różne możliwości implementacji podanego systemu. Przykładowo operacja *TransmitMtoS_FailReceive* 'zakończy się sukcesem', jeśli jej wynikiem będzie to, że do zbioru kolejek w systemie zostanie dodany pakiet wysłany, ale nie odebrany.

Można to zaimplementować na kilka sposobów, np. ustalić limit czasowy (ang. *timeout*), albo sprawdzić sumę kontrolną CRC odebranego potwierdzenia (nawet gdy pakiet dojdzie, ale z błędem, to nie zostanie to uznane jako potwierdzenie) - w sytuacjach awaryjnych będziemy mieli do czynienia z tą operacją.

Zależnie od wymaganego poziomu szczegółowości specyfikacja taka mogłaby być rozwijana; decyzja taka jest podejmowana przez projektanta, np. w oparciu o posiadane doświadczenie zespołu lub dostępność gotowych i przetestowanych komponentów realizujących opisywane funkcje.

Rozdział 3

Dowodzenie poprawności

3.1. Dowodzenie jako element metodologii projektowania

Błędy w programach komputerowych są plagą od początku ich istnienia. Ich źródła bywają różne, w większości przypadków są jednak powodowane przez pomyłkę człowieka. Mogą powstawać na różnych etapach powstawania oprogramowania, zarówno w fazie projektu, jak i implementacji. Z tego wszystkiego najbardziej niebezpieczne są nieprawidłowości i nieścisłości projektu. Koszt naniesienia poprawek do implementacji jest stosunkowo niewielki, jeśli jednak okaże się, że błędy wkradły się do projektu i zostanie to wykryte w końcowej fazie produkcji, to konsekwencje tego mogą być bardzo kosztowne. Nie tylko koszty są jednak istotne. Wiele współczesnych urządzeń, których awaria mogłaby kosztować życie ludzkie, jest obecnie budowanych w oparciu o systemy mikroprocesorowe. W takich przypadkach poprawne zaprojektowanie oprogramowania sterującego jest sprawą nadrzędną.

Wtedy przychodzą z pomocą metody formalne, ponieważ nie tylko umożliwiają precyzyjne i jednoznaczne przekazanie zamierzeń projektanta programiście, ale również analizę poprawności i weryfikację założeń na etapie budowania specyfikacji. Ich stosowanie umożliwia badanie wielu aspektów przyszłego systemu - przewidywanie stanu systemu po wykonaniu jakiejś operacji, lub udowodnienie poprawnej obsługi sytuacji wyjątkowych.

Rozdział ten pokazuje typowe techniki i przykłady towarzyszące weryfikacji $\mathbb{Z}$.

3.2. Zastosowanie rachunku predykatów do dowodzenia poprawności

Formalne opisywanie procesów, systemów, algorytmów i struktur danych umożliwia ich analizę za pomocą dostępnych metod matematycznych. Jedną z nich jest rachunek predykatów, który dostarcza metod połączenia celów przyświecających projektowi, założeń co do funkcjonalności - z detalami dotyczącymi szczegółów projektowych.

Możemy założyć, że specyfikacja jest poprawna w sensie formalnym, jeśli poszczególne jej elementy nie wykluczają się wzajemnie, są spójne, oraz jest spełniony zbiór

założeń, często znanych od początku tworzenia systemu - czyli teorematów. Tworząc np. system obsługujący transakcje bankowe podstawowym teorematem jest, że pieniądze nie giną, tzn. suma środków zgromadzonych na rachunkach przed transakcją jest sumą wynikającą z bilansu zaksięgowanych operacji. Projektując system określamy szczegółowo, co to jest rachunek, co znaczy wpłata i wypłata, a następnie za pomocą przekształceń matematycznych wynikających z logiki oraz innych znanych twierdzeń (również już udowodnionych teorematów specyfikacji) możemy udowodnić prawdziwość danego twierdzenia, pokazując w ten sposób, że system zaprojektowany zgodnie z tą specyfikacją będzie działał w/g dokładnie określonych założeń.

Są dostępne narzędzia automatyzujące ten proces, co zostanie pokazane w dalszej części pracy.

3.3. Typowe błędy specyfikacji

Matematyczny zapis założeń projektowych ma niewątpliwie wiele zalet, może się jednak okazać, że projektant popełnił błędy, które powodują, że specyfikacja jest wewnętrznie sprzeczna lub niespójna, albo po prostu źle zapisana. Sam fakt formalnego wyrażenia problemu nie oznacza poprawności projektu. Przedstawimy teraz klasy problemów, z jakimi może się spotkać twórca tekstu w $\mathbb{Z}$.

3.3.1. Błędy składni i typów

Język $\mathbb{Z}$ ma bardzo bogatą, a przez to skomplikowaną składnię. Ze względu na mnogość elementów matematycznych i operatorów jeszcze bardziej złożony jest aparat matematyczny $\mathbb{Z}$. Powoduje to, że bardzo łatwo jest pomylić się już w samym zapisie.

$\mathbb{Z}$ jest wprawdzie notacją beztypową, nie znaczy to jednak, że nie można tych typów w niej wyrazić, lub że ich się nie stosuje. System typów w $\mathbb{Z}$ jest zbudowany na bazie teorii zbiorów oraz ograniczeń stosowania elementów aparatu matematycznego. Przykładowo jeśli próbujemy użyć operatora relacji do czegoś, co nie jest relacją, albo znaków działań arytmetycznych w odniesieniu do danych, które nie są liczbami. Pomyłki tego typu są dość oczywiste i łatwe do zidentyfikowania.

Sprawa jest bardziej skomplikowana, gdy spróbujemy zaaplikować argument, którego typ jest niekompatybilny z danym zapisem, tzn. należy on do zbioru, który jest rozłączny ze zbiorem dopuszczalnych wartości. Przykładowo:

Sklep

$towary : TOWAR \rightarrow CENA$

NowyTowar

$\Delta Sklep$

$t? : TOWAR$

$towary' = t? \lhd towary$

Argument $t?$ powinien być parą $(TOWAR, CENA)$, czyli uogólniając należeć do zbioru $TOWAR \times CENA$, a nie do $TOWAR$. Zbiory te są rozłączne, więc taka definicja operacji *NowyTowar* jest nieprawidłowa.

Błędy związane ze składnią i systemem typów są najłatwiejsze do wykrycia.

3.3.2. Błędy dziedziny

W $\mathbb{Z}$ możliwe jest zapisanie wyrażenia, które ma nieokreślone znaczenie. Może to zdarzyć się w dwóch przypadkach:

1. podczas przekazania funkcji argumentu, który nie znajduje się w jej dziedzinie (np. $\#\mathbb{N}$, $max\ \mathbb{Z}$),

2. w przypadku użycia $\mu-$wyrażenia, dla którego nie istnieje wartość spełniająca jego predykat (np. $\mu\, x : \mathbb{Z} \mid x \neq x$), lub istnieje wiele takich wartości (np. $\mu\, x : \mathbb{N} \mid 0 * x = 0$).

Predykat zawierający takie nieokreślone wyrażenie może być różnie interpretowany. Praca Spivey'a [1] nie precyzuje jak powinna być traktowana nieokreśloność, jednak standard $\mathbb{Z}$ mówi, że taki predykat ma wartość *false*.

Błędy dziedziny nie zawsze mogą być wykryte za pomocą prostej kontroli typów. Do dowiedzenia ich poprawności można użyć następującego mechanizmu, który jest zaimplementowany w systemie **Z/EVES**. Mając dany schemat

$$S \mathrel{\widehat=} [\ D \mid P\ ]$$

poprawność w sensie dziedzin jest zagwarantowana, jeśli spełnione jest

$$S \wedge quantifiers(S) \Rightarrow domain\_check(S)$$

$domain\_check(S)$ jest predykatem, który tworzymy z P za pomocą algorytmu:

1. wybieramy z P wszystkie odwołania do funkcji, np. predykat

   $$\forall\, x : X \bullet fun(x) < fun(gfun(x))$$

   tworzy listę: $fun(x)$, $gfun(x)$, $fun(gfun(x))$,

2. dla każdego odwołania postaci $f(x)$ otrzymujemy warunek $x \in \operatorname{dom} f$. W podanym przykładzie mamy listę warunków:

   $$x \in \operatorname{dom} fun,\ x \in \operatorname{dom} gfun,\ gfun(x) \in \operatorname{dom} fun$$

3. z otrzymanej listy warunków $p_1, \ldots, p_n$ powstaje predykat[1]

   $$domain\_check(S) \Leftrightarrow p_1 \wedge \ldots \wedge p_n$$

[1] Należy zauważyć, że w podanym przypadku występuje użycie operatora $<$, który również posiada ograniczenia jego stosowania (a konkretnie argumenty po obu stronach muszą należeć do zbiorów, pomiędzy którymi określona jest relacja porządku) - można to jednak złożyć na karb kontroli typów $\mathbb{Z}$.

Następnie bierzemy z P wszystkie zmienne kwantyfikowane, których typ również tworzy warunek, tu $x \in X$. Koniunkcja tych warunków jest predykatem $quantifiers(S)$.

Weźmy przykład z [4]:

Personnel

$$employees : \mathbb{P}\, PERSON$$
$$boss\_of : PERSON \pfun PERSON$$
$$salary : PERSON \pfun \mathbb{N}$$

$$\mathrm{dom}\, salary = employees$$
$$\forall\, e : employees \bullet salary(e) < salary(boss\_of\ e)$$

Predykat $domain\_check(Personnel)$ ma postać:

$$e \in \mathrm{dom}\, salary \land e \in \mathrm{dom}\, boss\_of \land boss\_of\ e \in \mathrm{dom}\, salary$$

Po analizie tego wyrażenia okaże się, że ta definicja jest błędna, gdyż nic nie jest powiedziane na temat dziedziny, ani zbioru wartości funkcji częściowej $boss\_of$, zatem nie da się np. dowieść, że $boss\_of\ e \in \mathrm{dom}\, salary$.

Poprawnym schematem jest więc

Personnel

$$employees : \mathbb{P}\, PERSON$$
$$boss\_of : PERSON \pfun PERSON$$
$$salary : PERSON \pfun \mathbb{N}$$

$$\mathrm{dom}\, salary = employees$$
$$\mathrm{dom}\, boss\_of \subseteq employees$$
$$\mathrm{ran}\, boss\_of \subseteq employees$$
$$\forall\, e : employees \bullet salary(e) < salary(boss\_of\ e)$$

3.3.3. Niespójność

Specyfikacja jest rozumiana jako opis możliwego systemu. Jeśli nie istnieje taki model systemu, który spełniałby jej wymagania, to znaczy że specyfikacja jest *niespójna*.

Lokalna niespójność

Niespójność lokalna występuje w przypadku, gdy dany schemat S posiada predykat niemożliwy do spełnienia. Przykładowo:

S

$$n : \mathbb{Z}$$

$$n \neq n$$

Jeżeli taki schemat jest użyty do opisania stanu systemu, to stan ten jest nieosiągalny, a system niemożliwy do zrealizowania. Jeśli byłaby to operacja, to nie mogłaby być ona wykonana pomyślnie.

Ogólnym warunkiem spójności schematu jest predykat:

$$\exists S \bullet true$$

W punkcie 2.5.3 opisano pojęcie stanu początkowego. W takim wypadku powyższy predykat może być rozszerzony do postaci:

$$\exists S \bullet Init\_S$$

Dowód takiego teorematu mówi, że istnieją możliwe wartości sygnatury S, oraz że osiągalny jest również ich stan początkowy.

Globalna niespójność

Opisana niespójność z poprzedniego punktu ma charakter lokalny, gdyż dotyczy składowych elementów specyfikacji. Możliwa jest jednak sytuacja, że poszczególne paragrafy specyfikacji są prawidłowe, jednak ich połączenie wyklucza istnienie modelu systemu. Wtedy mamy do czynienia z niespójnością globalną, która sprawia, że specyfikacja jako całość jest nieprawidłowa.

Błędy mogą kryć się w aksjomatach, definicjach uogólnionych (patrz p. 2.6.1), lub predykatach.

Spójność definicji aksjomatycznej postaci

$$\begin{array}{|l} D \\ \hline P \end{array}$$

może być udowodniona za pomocą predykatu:

$$\exists D \bullet P$$

Trudniejsze przypadki pojawiają się, gdy grupa paragrafów jest niespójna, np.

$$\begin{array}{|l} x, y : \mathbb{N} \end{array}$$

$$x \in 1..3$$
$$\ldots$$
$$y < 1 - 3 * x$$

W rzeczywistych specyfikacjach paragrafy te mogą być rozdzielone kilkunastoma innymi definicjami, a są one niespójne dopiero w połączeniu, może zatem to być trudne do wykrycia. Dobrą praktyką jest grupowanie odwołujących się do siebie deklaracji, a jeśli jest to niemożliwe - wprowadzanie dodatkowych konstrukcji wiążących właściwości grupy paragrafów.

3.3.4. Błędy z punktu widzenia logiki specyfikacji

Ze względu na bogactwo składni $\mathbb{Z}$ i skomplikowanie aparatu matematycznego, nie jest trudno skonstruować poprawną w sensie formalnym specyfikację, ale niezgodną z zamierzeniami projektanta. W takim wypadku, pomimo że wprowadzone schematy, twierdzenia i teorematy są spójne i prawidłowe pod względem matematycznym, to nie opisują one oczekiwanego systemu.

Takie błędy są najtrudniejsze do wykrycia i są niezauważalne dla automatycznych narzędzi sprawdzających i wspomagających specyfikację w $\mathbb{Z}$. Dlatego wskazane jest przemyślenie strategii testowania projektu w postaci przygotowania jak największej ilości teorematów testowych opisujących poprawny system (patrz p. 3.4.4).

3.4. Techniki dowodzenia poprawności

3.4.1. Indukcja

Indukcja jest klasyczną metodą dowodzenia, która może być zastosowana w stosunku do ciągów, zbiorów, sekwencji oraz typów wolnych.

Ciągi

Aby dowieść, że właściwość $P(n)$ odnosi się do wszystkich elementów ciągu n, wystarczy pokazać, że:

1. $P(0)$ jest spełnione,
2. jeśli spełnione jest $P(n)$ dla pewnego $n : \mathbb{N}$, to wynika z tego również $P(n+1)$:

$$\forall\, n : \mathbb{N} \bullet P(n) \Rightarrow P(n+1)$$

Jeśli trudno jest udowodnić prawdziwość $P(n)$ tylko z właściwości poprzednika, można posłużyć się potężniejszą wersją indukcji: z faktu, że wszystkie elementy poprzedzające n spełniają $P(k)$ (dla $k < n$) wynika prawdziwość $P(n)$. Odpowiada to zapisowi:

$$\forall\, n : \mathbb{N} \bullet (\forall\, k : \mathbb{N} \mid k < n \bullet P(k)) \Rightarrow P(n)$$

Zbiory

Indukcja pomocna jest w dowodzeniu właściwości zbiorów skończonych. Korzystając z faktu, że każdy zbiór skończony może zostać 'skonstruowany' ze zbioru pustego poprzez stopniowe dodawanie nowych elementów, łatwo formułuje się zasady indukcji. $P(S)$ jest zachowane dla wszystkich zbiorów skończonych $S : \mathbb{F}\, X$, gdy:

1. $P(\emptyset)$ jest spełnione,

2. jeśli spełnione jest $P(S)$ dla pewnego zbioru S, to wynika z tego również $P(\ S \cup \{x\}\ )$:

$$\forall\, S : \mathbb{F}\, X;\ x : X \bullet P(S) \Rightarrow P(\ S \cup \{x\}\ )$$

Analogicznie do pierwszego punktu dostępna jest potężniejsza wersja, którą można wyrazić: pewna właściwość charakteryzuje zbiór S, jeśli wynika ona ze spełnienia jej dla wszystkich podzbiorów S. Zatem $\forall\, S : \mathbb{F}\, X \bullet P(S)$ jeśli:

$$\forall\, S : \mathbb{F}\, X \bullet (\forall\, T : \mathbb{F}\, X \mid T \subset S \bullet P(T)) \Rightarrow P(S)$$

Sekwencje

Sekwencje mają zdefiniowane pojęcie konkatenacji, $\mathbb{Z}$ dopuszcza też sekwencje puste. Analogicznie zatem do zbioru skończonego, sekwencja może zostać 'skonstruowana' z sekwencji pustej poprzez kolejne dodawanie elementów. Elementy te mogą być jednak wstawiane na kilka sposobów.

Pierwszy warunek indukcji jest standardowy:

(abc1) $P(\langle\rangle)$ jest spełnione.

Kolejne warunki zależą od sposobu budowania sekwencji w regule indukcyjnej. Jeśli elementy dodawane są na początek, to reguła ma postać:

(a2) jeśli spełnione jest $P(s)$ dla pewnej sekwencji s, to wynika z tego $P(\langle x\rangle \frown s)$:

$$\forall\, s : \mathrm{seq}\, X;\ x : X \bullet P(s) \Rightarrow P(\langle x\rangle \frown s)$$

Elementy dodawane na koniec:

(b2) jeśli spełnione jest $P(s)$ dla pewnej sekwencji s, to wynika z tego $P(s \frown \langle x\rangle)$:

$$\forall\, s : \mathrm{seq}\, X;\ x : X \bullet P(s) \Rightarrow P(s \frown \langle x\rangle)$$

Istnieje jeszcze jedna strategia tworzenia sekwencji. Zamiast zaczynać od sekwencji pustej, można dowieść również właściwości poszczególnych elementów $\langle x\rangle$ i sekwencje dłuższe otrzymywać poprzez konkatenację krótszych. Daje to dwa warunki:

(c2) Dla każdego $x : X$ spełnione jest $P(\langle x\rangle)$,

(c3) jeśli spełnione są warunki $P(s), P(t)$ dla pewnych sekwencji s, t, to wynika z tego $P(s \frown t)$:

$$\forall\, s, t : \mathrm{seq}\, X \bullet P(s) \wedge P(t) \Rightarrow P(s \frown t)$$

Ostatni sposób może wydawać się bardziej skomplikowany ze względu na konieczność dowodu trzech warunków zamiast dwóch, ale często okazuje się że jest on łatwiejszy do przeprowadzenia.

Typy wolne

Indukcja ma zastosowanie w sprawdzaniu spójności dla typów wolnych. Reguły indukcyjne podane są w 2.6.2.

3.4.2. Test warunków wstępnych

Warunki wstępne, jak wspomniano już w punkcie 2.5.2, są predykatami określającymi możliwe wartości argumentów operacji, które gwarantują jej pomyślne wykonanie. Aby dowieść, że P jest poprawnym warunkiem wstępnym, należy pokazać, że:

$$\forall S;\ in? : IN \mid P \bullet \text{pre } Op$$

Lista takich teorematów skutecznie opisuje ograniczenia danego systemu. Jeżeli operacja może wykonać się dla dla dowolnych argumentów, w dowolnym stanie, to teoremat

$$\forall S;\ in? : IN \bullet \text{pre } Op$$

jest prawdziwy.

Stosowanie testu warunków wstępnych konieczne jest podczas dowodzenia poprawności złożenia operacji (patrz definicje w 2.5.4).

3.4.3. Niezmienniki

Niezmiennik jest predykatem, który jest spełniony niezależnie od wykonywanych operacji na systemie. Jeśli umieścimy taki predykat w schemacie postaci

Inv

System
P

to jest spełnione:

$$(Init \Rightarrow Inv) \land (Op \land Inv \Rightarrow Inv')$$

przy założeniu, że Op jest operacją, a $Init$ opisuje stan początkowy.

Przykładem systemu z niezmiennikiem może być implementacja bufora, zawierającego wskaźnik r.

Buffer

$r : \mathbb{Z}$

Stan początkowy (bufor pusty):

Init

Buffer

$r = 0$

I operacja dodawania elementów do bufora:

Add

Δ*Buffer*

$r' = r + 1$

Niezmiennikiem jest fakt, że wskaźnik bufora jest liczbą naturalną:

Invariant

Buffer

$r \in \mathbb{N}$

Zatem $(Init \Rightarrow Invariant) \wedge (Add \wedge Invariant \Rightarrow Invariant') \Leftrightarrow true$.
Niezmienniki są użyteczne, gdy

- chcemy zagwarantować zachowanie pewnych ważnych ograniczeń, kluczowych dla bezpieczeństwa systemu (np. że temperatura spalania nie przekroczy progu),
- niezmienność jakiegoś faktu jest przydatna dla programisty w zastosowaniu bardziej efektywnego algorytmu.

Należy zauważyć, że predykaty wewnątrz schematu również mogą być niezmiennikami, ale bywają sytuacje, kiedy potrzebne jest zadeklarowanie nowego, dedykowanego niezmiennikowi schematu. Jest tak gdy mając dwie operacje, chcemy określić stałą właściwość, niezależnie od ich wykonania. Ponieważ przed zadeklarowaniem niezmiennika obie operacje muszą być wprowadzone do specyfikacji, konieczny jest trzeci schemat.

Jeżeli system jest zdefiniowany przez operacje $Op_1 \ldots Op_n$ i pre $Op_1 \wedge \ldots \wedge$ pre Op_n jest niezmiennikiem, to w systemie nie występują sytuacje wyjątkowe (wykonanie operacji $Op_1 \ldots Op_n$ zawsze jest możliwe).

3.4.4. Testowe teorematy

Poprawność specyfikacji może też być rozumiana jako zachowanie prawdziwości teorematów, przyjętych jako założenia przed jej powstaniem. Takie teorematy mogą być użytecznym sposobem wyrażenia kluczowych warunków poprawności danego systemu. Przykładowo w programie sterującym sygnalizacją świetlną takim warunkiem może być wykluczenie sytuacji, w której zielone światło jest zapalone dla dwóch lub więcej przecinających się pasów ruchu. Weźmy schemat:

Skrzyzowanie

$$zapalone : \mathbb{N} \pfun \{ 0, 1\}$$

$$\#(\mathrm{dom}\ zapalone) = n$$

n jest liczbą wykluczających się linii sygnalizacyjnych, które nie powinny być jednocześnie zapalone. Weźmy zatem operację *Zmiana*

Zmiana

$$\Delta Skrzyzowanie$$

$$\forall x_1, x_2 : \mathrm{dom}\ zapalone \mid zapalone(x_1) = 1 \land zapalone'(x_2) = 1 \bullet x_1 \neq x_2$$
$$\ldots$$

Teorematem testowym jest warunek

$$Zmiana \Rightarrow (\#\{ \forall x : \mathrm{dom}\ zapalone \mid zapalone'(x) = 1 \bullet x\} = 1)$$

czyli, że po wykonaniu operacji *Zmiana* będzie zapalona tylko jedna linia sygnalizacyjna (i będzie to inna linia niż przed operacją).

Dowodząc prawdziwości testowych teorematów możemy sprawdzić nie tylko spełnienie założeń wstępnych, ale również badać zachowanie wyspecyfikowanego systemu w sytuacjach wyjątkowych lub krytycznych.

3.5. Zagadnienie uszczegółowienia (*refinement*)

3.5.1. Uszczegółowienie w ujęciu ogólnym

Tworząc program z podanej specyfikacji, programista musi rozwiązywać dwa rodzaje problemów:

- Zapisać struktury danych w odpowiedni sposób zdefiniowany przez język programowania, aby odpowiadały one zbiorom, relacjom i innym definicjom podanym w formalnej specyfikacji $\mathbb{Z}$.
- Zdefiniować algorytmy za pomocą składni i bibliotek środowiska, aby realizowały one operacje na danych w sposób określony w specyfikacji za pomocą języka schematów.

Bazując na różnym stopniu złożoności problemów, twórcy notacji $\mathbb{Z}$ wprowadzili pojęcie uszczegółowienia. Polega ono na tym, że projektant może pracować na różnych poziomach abstrakcji, na początku projektowania zależy mu głównie na stworzeniu ogólnych ram systemu, opisujących w sposób zwięzły dane i operacje na nich. W miarę dalszego rozwijania specyfikacji, może on określać bardziej szczegółowe wersje schematów i operacji, które już zdefiniował, przechodząc w ten sposób na niższy poziom abstrakcji. Pomaga to zarówno projektantom jak i programistom, mogą oni bowiem

korzystać z różnych części specyfikacji o innym stopniu trudności, w miarę potrzeby korzystając odpowiednio z wersji ogólnej i szczegółowej. Na podstawie podanych klas problemów wyróżniamy więc uszczegółowienie danych (ang. *data refinement*) oraz operacji (ang. *operation refinement*).

Z punktu widzenia dowodzenia poprawności formalnej i weryfikacji specyfikacji powstaje w ten sposób problem formalnego dowiedzenia, że dwie wersje tej samej operacji, różniące się poziomem szczegółowości, opisują dokładnie ten sam proces i że spójne wobec siebie.

Matematycznie ten problem można wyrazić w sposób następujący. Mając 'abstrakcyjny' system będący w stanie A, posiadający schemat początkowy AI oraz operację AO, jak również odpowiadający 'konkretny' system w stanie C, ze schematem początkowym CI i operacją CO, można skonstruować pojęcie relacji R łączącej oba te systemy. Zachodzi wtedy:

$$\forall\, CI \bullet \exists\, AI \bullet R$$

Jeśli te dwa systemy rzeczywiście sobie odpowiadają, to muszą być spełnione dwa warunki. Pierwszy mówi, że 'konkretna' operacja CO może być wywołana wtedy, kiedy i 'abstrakcyjna':

$$\forall\, R \mid \text{pre}\ AO \bullet \text{pre}\ CO$$

Drugi określa, że jeśli operacja 'abstrakcyjna' może być wywołana, to rezultat spójny z wynikiem analogicznej operacji 'konkretnej':

$$\forall\, R;\ CO \mid \text{pre}\ AO \bullet \exists\, A' \bullet AO \land R'$$

3.5.2. Uszczegółowienie operacji

Dwa przedstawione warunki dla dowodzenia właściwości uszczegółowienia operacji, mogą być też zapisane w poniższy sposób. Niech $x?$ będzie parametrem operacji.

$$\forall\, State;\ x? : X \bullet \text{pre}\ AO \Rightarrow \text{pre}\ CO$$

Wersja bez operatora pre :

$$\forall\, State;\ x? : X \bullet \\ \quad (\exists\, State';\ y! : Y \bullet AO) \Rightarrow (\exists\, State';\ y! : Y \bullet CO)$$

Drugi warunek: jeśli warunek wstępny operacji AO jest spełniony, to każdy rezultat jaki może zwrócić operacja CO musi być możliwym rezultatem operacji AO:

$$\forall\, State;\ State';\ x? : X;\ y! : Y \bullet \\ \quad \text{pre}\ AO \land CO \Rightarrow AO$$

Wersja bez operatora pre :

$$\forall\, State;\ x? : X \bullet \\ \quad (\exists\, State';\ y! : Y \bullet AO) \\ \quad \Rightarrow (\forall\, State';\ y! : Y \bullet CO \Rightarrow AO)$$

3.5.3. Uszczegółowienie danych

Zagadnienie uszczegółowienia danych rozszerza pojęcie uszczegółowienia operacji w ten sposób, że pozwala, aby przestrzeń stanów abstrakcyjnych operacji różniła się od przestrzeni stanów operacji konkretnych. Problem ten odpowiada sytuacji, gdy programista mając do dyspozycji definicje danych zawarte w specyfikacji za pomocą abstrakcyjnych pojęć, takich jak zbiory lub relacje, musi zapisać je w sposób określony przez język programowania. Pierwszy opis odpowiada więc specyfikacji, drugi - już konkretnemu projektowi. Oczywiście w sensie notacji opis projektu również jest zapisany za pomocą abstrakcyjnych pojęć notacji, różni się jednak poziomem szczegółowości.

Przechodząc do problemu dowodzenia poprawności, dla uszczegółowienia danych musimy również zapisać dwa warunki, analogiczne do tych opisanych w poprzednich podrozdziałach.

Pierwszy mówi, że operacja na 'konkretnym' typie danych kończy się wtedy, kiedy również operacja 'abstrakcyjna' spełnia warunki do zakończenia. Jeśli stany abstrakcyjny i konkretny są związane za pomocą abstrakcyjnego schematu Abs i stan abstrakcyjny spełnia warunki wstępne operacji abstrakcyjnej AO, to również stan konkretny również musi spełniać warunki wstępne operacji konkretnej.

$$\forall\, Astate;\ Cstate;\ x? : X \bullet$$
$$\quad \text{pre}\ AO \land Abs \Rightarrow \text{pre}\ CO$$

Drugi warunek mówi o tym, że stan po wykonaniu operacji 'konkretnej' odpowiada jednemu ze stanów abstrakcyjnych, w jakich mogła się zakończyć operacja abstrakcyjna AO.

$$\forall\, Astate;\ Cstate;\ Cstate';\ x? : X;\ y! : Y \bullet$$
$$\quad \text{pre}\ AO \land Abs \land CO \Rightarrow (\exists\, Astate' \bullet Abs' \land AO)$$

Oprócz tego, możemy zapisać jeszcze jeden warunek dla stanów początkowych.

$$\forall\, Cstate \bullet$$
$$\quad CI \Rightarrow (\exists\, Astate \bullet AI \land Abs)$$

Rozdział 4

System Z/EVES

Z/EVES jest systemem służącym do tworzenia, sprawdzania oraz analizy specyfikacji $\mathbb{Z}$. Może być używany do testów poprawności typów i dziedzin, eksplorowania schematów, sprawdzania warunków wstępnych (patrz p. 2.5.2) oraz dowodzenia teorematów i faktów związanych z uszczegółowieniem *(ang. refinement)*. Obecnie jest to najbardziej zaawansowany system automatycznego przeprowadzania dowodów dla notacji $\mathbb{Z}$.

Jest wyposażony w maszynę dowodzenia *(ang. prover)* automatyzujący proces przeprowadzania dowodów, co czasem jest dokonywane w pełni automatycznie, bez udziału użytkownika. Niemniej jednak aby efektywnie go wykorzystywać niezbędna jest wiedza na temat wewnętrznej reprezentacji elementów specyfikacji oraz stosowanych przez niego technik.

Z/EVES zgodny jest ze standardem ISO oraz biblioteką $\mathbb{Z}$ zawartą w tym standardzie. Jest firmowany przez organizację ORA Canada (*http://www.ora.on.ca*) i został częściowo sfinansowany ze środków Departamentu Obrony USA.

4.1. Koncepcja i możliwości systemu

Z/EVES składa się z jądra systemu, które jest intepreterem języka *EVES* i posiada wiedzę o logice i aparacie matematycznym. Jądro to posiada wbudowany zestaw teorematów *(ang. mathematical toolkit)*, który umożliwia automatyczne sprawdzanie specyfikacji. Oprócz tego dostępny jest interfejs użytkownika *(ang. GUI)*, który w trybie graficznym prezentuje daną specyfikację $\mathbb{Z}$, dając przy tym dostęp do standardowych komend systemu Z/EVES. Specyfikacja może być sprawdzona pod względem poprawności składniowej i typów, jak również poprawności merytorycznej, popartej teorematami i znanymi twierdzeniami z zakresu logiki predykatów, arytmetyki i teorii mnogości.

Językiem specyfikacji jest tekst zawierający instrukcje, których składnia jest zgodna z makrami LaTeX. Z/EVES potrafi importować matematyczne zapisy notacji z dokumentów LaTeX. Mogą one zatem być źródłem teorematów dla systemu **Z/EVES**, jak również wchodzić w skład regularnej dokumentacji systemu.

Z/EVES zawiera zaawansowane mechanizmy do przeprowadzania automatycznych dowodów. Większość oczywistych lub łatwych faktów jest dowodzona automatycznie. Jednakże zazwyczaj **Z/EVES** nie radzi sobie z trudniejszymi dowodami, wtedy użytkownik musi znać dowód, albo chociaż jego zarys.

Specyfikacja jest sprawdzana paragrafami. Sprawdzanie jest zintegrowane z maszyną dowodzenia, oprócz samej kontroli poprawności składni podejmowana jest próba dowodu. Do każdego paragrafu można dołączyć zbiór komend pozwalających na jego dowód (gdyż nie wszystkie dają się przeprowadzić automatycznie). Poszczególne paragrafy są dowodzone po kolei, z wiedzą dostępną na etapie danego paragrafu. Oznacza to, że jeśli do dowodu jakiegoś paragrafu potrzebny jest lemat, to powinien on w specyfikacji poprzedzać dany teoremat.

4.2. Maszyna dowodzenia

Jednym z celów towarzyszącym twórcom systemu było jak najlepsze zautomatyzowanie przeprowadzania dowodów. Okazało się to jednak niemożliwe lub znacznie utrudnione w wielu przypadkach, dlatego proces ten wymaga często ingerencji użytkownika. Następuje to poprzez wydawanie komend, np. określających rodzaj strategii dowodu, lub dołączających do bieżącej wiedzy konkretne twierdzenie z *Mathematical Toolkit*.

Wewnętrznie notacja zamieniana jest na wyrażenia beztypowego rachunku predykatów. Oznacza to, że nie ma 'logiki $\mathbb{Z}$ ' jako takiej, operacje na wyrażeniach przeprowadza się w przestrzeni rachunku predykatów, a następnie zostają one z powrotem zamienione na język zgodny z notacją $\mathbb{Z}$. Może się zdarzyć (co jest bardzo rzadkie), że po przeprowadzeniu przekształceń nie można sprowadzić wyrażeń do formy $\mathbb{Z}$ (należy wtedy użyć jednej z reguł przekształceń).

W notacji jest tylko typ - $\mathbb{Z}$, natomiast rachunek predykatów jest beztypowy. Informacja o typach jest zatem wyrażona i przetwarzana logicznie, za pomocą pojęć zbiorów i zawierania, np. fakt, że liczba i jest typu całkowitoliczbowego, jest wyrażony przez $i \in \mathbb{Z}$. Nie ma w tym zaimplementowanych dodatkowych mechanizmów, które w specjalny sposób traktowałyby tą informację - używany jest ten sam zestaw reguł dla predykatów co zwykle.

Z każdym dowodem związany jest tzw. kontekst, który reprezentuje aktualną wiedzę. Każda specyfikacja rozpoczyna od kontekstu **Mathematical Toolkit**. Kolejne paragrafy powiększają tę wiedzę dodając nowe teorematy. Każdy teoremat posiada nazwę, która może być później użyta w dowodzie. Jeżeli użytkownik nie podał nazwy, zostaje ona przydzielona przez system automatycznie.

Przeprowadzenie dowodu wiąże się z tzw. docelowym predykatem. Wyrażenie przekształcane jest w równoważne, za pomocą znanych reguł logicznych i komend maszyny dowodzenia. Dowód jest zakończony, jeśli uda się doprowadzić do wyrażenia *true*. Dla łatwiejszych faktów komendy automatycznego dowodzenia, jak np. **prove by reduce**, często kończą dowód w jednym kroku, jednak jeśli tak się nie dzieje, użytkownik musi wspomóc automat poprzez podanie mu wskazówek dla dalszego przekształcania.

4.3. Aparat matematyczny $\mathbb{Z}$

Przekształcanie predykatu do możliwie najprostszej postaci w systemie **Z/EVES** wykorzystuje wbudowany w maszynę dowodzenia rachunek zdań. W niej również zaimplementowane są niektóre podstawowe twierdzenia i mechanizmy (np. reguły równo-

ważności lub działania arytmetyczne). Jednakże w większości przypadków sama logika nie jest wystarczająca do pomyślnego przeprowadzenia dowodu, potrzebna jest również wiedza na temat właściwości wykorzystywanych konstrukcji matematycznych, takich jak relacje, zbiory lub sekwencje. Ponieważ wiedza ta jest obszerna i ich implementacja w rdzeniu systemu nastręczałaby wiele trudności, jednocześnie uniemożliwiając jej stopniową rozbudowę, autorzy systemu postanowili umieścić pozostałe twierdzenia i teorematy w postaci zbioru reguł, zwanym też aparatem matematycznym systemu (ang. *mathematical toolkit*). Jest on szczegółowo udokumentowany w [5].

Najważniejszą rzeczą, z której powinien zdawać sobie sprawę użytkownik jest fakt, że nie wszystkie twierdzenia i właściwości dotyczące konkretnych elementów aparatu matematycznego mogą być przydatne do przeprowadzenia dowodu. Dzieje się tak z różnych powodów, najczęściej zastosowanie niewłaściwych twierdzeń powoduje niepotrzebne skomplikowanie predykatu docelowego, które prowadzi donikąd, natomiast zaciemnia istotne fakty mogące naprowadzić użytkownika na właściwy plan dowodu. Dlatego część twierdzeń jest oznaczona w *mathematical toolkit* jako reguły wyłączone (ang. *disabled rules*). Są one znane **Z/EVES** jako prawdziwe, ale nie są wykorzystywane w automatycznej strategii dowodzenia. Niemniej znajomość tych faktów jest często niezbędna dla pomyślnego dowodu w przypadku, kiedy system nie może zakończyć go automatycznie. Przykładem może być sprawdzanie równości zbiorów i reguła *extensionality*:

theorem disabled rule extensionality
$X = Y \Leftrightarrow (\forall x : X \bullet x \in Y) \land (\forall y : Y \bullet y \in X)$

theorem disabled rule extensionality2
$X = Y \Leftrightarrow X \in \mathbb{P}\, Y \land Y \in \mathbb{P}\, X$

Gdyby jedna z tych reguł była włączona automatycznie, to system zamieniałby każde wystąpienie znaku równości na predykat po prawej stronie równoważności ($\Leftrightarrow$), co oczywiście jest w większości wypadków niepożądane.

4.4. Metody i algorytmy automatycznego przeprowadzania dowodów

4.4.1. Analiza wyrażenia i normalizacja

W **Z/EVES** dostępne są trzy komendy (**simplify, rewrite, reduce**), które potrafią zredukować wyrażenie. Analizują je one od lewej do prawej strony, z góry do dołu. Każda część jest rozważana i ewentualnie zastępowana przez nowe wyrażenie. Podczas analizy utrzymywany jest *kontekst*, który jest zbiorem predykatów zakładanych jako prawdziwe. Przykładowo, podczas analizowania predykatu w formie $P \cap Q \Rightarrow R$:

- P jest rozważane i ew. zastępowane przez P',
- z założeniem, że P jest prawdziwe, rozważane jest Q i ew. zastępowane przez Q'

- z założeniem, że P i Q są prawdziwe, rozważane jest R ew. zastępowane przez R'
- wynikiem jest $P' \cap Q' \Rightarrow R'$. Jeżeli jeden lub więcej nowych predykatów to *true* lub *false*, to mogą być wtedy wyeliminowane, lub nawet całe wyrażenie jest redukowany do *true* lub *false*.

Te trzy komendy różnią się miedzy sobą sposobem 'rozważania' wyrażenia.

Czasami domyślna metoda wnioskowania **Z/EVES** nie jest w stanie przetworzyć wyrażenia. Przykładowo dla predykatu

$$(P \lor Q) \\ \land (P \Rightarrow R) \\ \land (Q \Rightarrow R)$$

system nie jest w stanie skonkludować predykatu R.

W takich wypadkach dostępne jest bardziej potężne narzędzie wnioskujące, czyli normalizacja. Jeśli jest włączona i system napotka w wyrażeniu relację logiczną zagłębioną w innej, to rozważane są wtedy wszystkie przypadki. Przykładowo predykat $(P \lor Q) \land R$ jest zamieniany do postaci **if** P **then** R **else** $(Q \land R)$. Predykat R jest rozważany dwukrotnie, raz z założonym P, drugi raz z wykluczonym P i założonym Q. Normalizacja jest domyślnie wyłączona, ponieważ poprzez powtarzanie części predykatu, jej użycie prowadzi często do bardzo długich wyrażeń docelowych. Jednakże zwiększa ona moc maszyny dowodzenia.

4.4.2. Upraszczanie

Dla upraszczania wyrażeń, w połączeniu z analizą opisaną w poprzednim punkcie, stosuje się kilka metod, jak np. sprawdzanie równości arytmetycznych, kontrolowanie tautologii i równoważności w sensie rachunku zdań, oraz aplikowanie reguł przekazujących *(ang. forward rule)* i reguł założeń *(ang. assumption rule)*.

Rachunek zdań

Twierdzenia i zależności rachunku zdań *(ang. propositional logic)* są wykorzystywane w różnych częściach maszyny dowodzenia, tzn. w analizie wyrażenia, normalizacji i przekształcaniu.

Jak już było wspomniane wcześniej, analizie wyrażenia towarzyszy *kontekst*. Jeśli dany predykat jest w kontekście założony jako *true* lub *false*, to wszystkie jego wystąpienia w badanym wyrażeniu są zamieniane odpowiednio na *true* lub *false*, a następnie nowe wyrażenie jest przekształcane poprzez usunięcie redundantnych części. Np. $x = 1 \land true$ jest zamieniane na $x = 1$, a $x > 2 \Rightarrow true$ na *true*. Predykat $x \in S \Rightarrow x \in S$ jest rozpatrywany w dwóch krokach: najpierw $x \in S$ jest dodawane do *kontekstu* i predykat jest sprowadzany do postaci $x \in S \Rightarrow true$, a potem z rachunku zdań do *true*.

Mechanizm ten pozwala na rozpoznanie większości tautologii, jeśli włączona jest normalizacja, **Z/EVES** obsługuje wszystkie tautologie.

Równoważność

Równoważności *(ang. equalities)* są traktowane w specjalny sposób, zaimplementowany automat decyzyjny wyrażenia takie jak $a = a$, $a = b \Rightarrow b = a$, $X = Y \Rightarrow \mathbb{P}\, X = \mathbb{P}\, Y$ rozpoznaje jako prawdziwe.

Reguły arytmetyczne

Algorytm upraszczający potrafi sobie radzić z podstawowymi działaniami i relacjami arytmetycznymi, np. wie że wyrażenie $1 < 2$ jest prawdziwe, a $2 * 2 > 3 + 4$ - nie. Reguły arytmetyczne są włączane dla wyrażenia t, jeśli w *kontekście* znajduje się predykat $t \in \mathbb{Z}$. Oprócz liczb obsługiwane są również liniowe wyrażenia algebraiczne, np. $\forall\, x, y : \mathbb{Z} \mid 1 \leq x \land 2 + 3 * x \leq y \bullet 5 \leq y$ jest sprowadzane do *true*.

Reguły założeń

System **Z/EVES** pozwala na definiowanie tzw. reguł założeń *(ang. assumption rule)*, które są teorematami oznaczonymi słowem kluczowym `grule`. Jest to reguła 'wyzwalana' *(ang. triggered)* predykatem. Podczas analizy, jeśli system podczas rozważania jakiegoś wyrażenia napotka na wystąpienie predykatu 'wyzwalającego' jakiejś reguły, to reguła zostanie zastosowana.

'Stosowanie' reguły zależy od jej postaci. Jeśli to nie jest implikacja, jej predykat zostanie dodany do *kontekstu*. Jeśli ma ona postać warunkową, to wtedy badane jest spełnienie tego warunku, poprzez sprawdzenie, czy nie występuje zaprzeczenie zakładając że reguła jest nieprawdziwa (ma wartość *false*). Po pozytywnym zweryfikowaniu warunku, predykat reguły dodawany jest do kontekstu.

Z/EVES automatycznie generuje założenia, podczas sprawdzania kolejnych paragrafów. Reguły te są użyteczne we wspomaganiu dowodzenia arytmetycznych właściwości zmiennych. Przykładowo:

$$\mid \mathit{maxLength} : \mathbb{N}_1$$

Po zdefiniowaniu takiego aksjomatu automatycznie jest generowana reguła $\mathit{maxLength\$declaration}$, zawierająca warunek $\mathit{maxLength} \in \mathbb{N}_1$. Podczas przekształcania predykat ten współdziała z tzw. regułami uogólniającymi z aparatu matematycznego systemu, które m.in. stanowią, że $\mathbb{N}_1 \in \mathbb{P}\,\mathbb{N}$, co pozwala na stwierdzenie, że np. $\mathit{maxLength} \in \mathbb{N}$ lub $\mathit{maxLength} \in \mathbb{Z}$ są prawdziwe. Proces 'upraszczania' nie ma jednak danych na temat liczb naturalnych, zatem próba udowodnienia

$$\forall\, x : \mathbb{Z} \bullet x + \mathit{maxLength} > x$$

zakończy się niepowodzeniem. Możemy zatem pomóc poprzez dodanie odpowiedniego teorematu:

theorem grule maxLengthBound
$\mathit{maxLength} \geq 1$

System napotkawszy definicję *maxLength* automatycznie doda fakt, że $\mathit{maxLength} \geq 1$, co sprawia, że upraszczanie jest w stanie udowodnić proste fakty arytmetyczne, jak np. $\mathit{maxLength} + \mathit{maxLength} \geq 2$.

Reguły przekazujące

Wspomniany wcześniej *kontekst* jest zbiorem predykatów zakładanych jako prawdziwe. Kiedy predykat jest dodawany do kontekstu, może spowodować to zastosowanie kolejnej 'wyzwalanej' reguły - czyli reguły przekazującej *(ang. forward rule)*.

Jest to teoremat w formie $P \Rightarrow Q$, który jest oznaczony słowem kluczowym `frule`. Jeśli dodawany do *kontekstu* predykat występuje dla jakiejś reguły jako instancja P, wtedy również Q jest dodawane do kontekstu.

Podobnie jak reguły założeń, reguły przekazujące są automatycznie generowane przez system podczas deklarowania schematów. Dla schematu $S \mathrel{\widehat=} [x, y : \mathbb{Z} \mid x < y]$ ta reguła to $S \Rightarrow x \in \mathbb{Z} \land y \in \mathbb{Z}$. Zatem, kiedy zakładana jest prawdziwość S, typy jego komponentów x, y są znane i dodawane do *kontekstu*.

4.4.3. Przekształcanie

Przekształcanie *(ang. rewriting)* to następna z trzech metod operacji na predykatach przez system **Z/EVES**. Jest to osiągane poprzez upraszczanie wyrażenia oraz stosowanie reguł przekształceń *(ang. rewriting rules)*. Reguła taka ma postać *Warunek* $\Rightarrow$ *Wzorzec* $=$ *Zamiennik* (lub *Warunek* $\Rightarrow$ *Wzorzec* $\Leftrightarrow$ *Zamiennik*). Do jej wywołania służy komenda `reduce`. Przykładowe reguły przekształceń:

theorem rule inRange
$\forall\, a, b : \mathbb{Z} \bullet x \in a \mathinner{..} b \Leftrightarrow a \leq x \leq b$

(warunek to $a \in \mathbb{Z} \land b \in \mathbb{Z}$, wzorzec $x \in a \mathinner{..} b$, zamiennik $a \leq x \leq b$),

theorem rule ranSeqInPower [X]
$\forall\, s : \operatorname{seq} X \bullet \operatorname{ran} s \in \mathbb{P}\, Y \Leftrightarrow s \in \operatorname{seq} Y$

(warunek - $s \in \operatorname{seq} X$, wzorzec - $\operatorname{ran} s \in \mathbb{P}\, Y$, zamiennik $s \in \operatorname{seq} Y$).

Reguły przekształceń są najłatwiejsze w użyciu, ponieważ nie mają takich składniowych ograniczeń jak reguły założeń lub przekazujące. Mogą być one zastosowane świadomie przez użytkownika za pomocą komendy `apply`. Są one często domyślnie wyłączone, jak np.:

theorem disabled rule extensionality
$X = Y \Leftrightarrow (\forall\, x : X \bullet x \in Y) \land (\forall\, y : Y \bullet y \in X)$

Gdyby ta reguła była domyślnie włączona, to każda równość w wyrażeniu byłaby zastępowana sprawdzaniem zawierania się elementów. Jest ona jednak potrzebna do udowodnienia $(1 \mathinner{..} 5) \cap (3 \mathinner{..} 7) = (3 \mathinner{..} 5)$, ponieważ redukcja nie potrafi sobie z tym poradzić.

Interfejs graficzny programu pozwala na informowanie użytkownika o wyłączonych regułach przekształceń, jeżeli warunek jakiejś reguły jest spełniony dla danego predykatu, to jej nazwa pojawia się po jego zaznaczeniu.

4.4.4. Redukcja

Redukcja *(ang. reduction)*, wywoływana komendą `reduce`, wykorzystuje dwie poprzednie metody - tzn. upraszczanie i przekształcanie. Jej działanie polega na sprawdzaniu podczas analizy, czy pod-predykat lub pod-wyrażenie nie jest odwołaniem do innej definicji lub schematu. Jeśli jest, to referencja zamieniana jest na oryginalną definicję, która zostaje wstawiona do badanej formuły. Całe wyrażenie jest wtedy upraszczane i redukowane od początku, proces ten powtarza się do momentu, kiedy nie ma już referencji do innych schematów. W kroku pętli stosowane są również reguły przekształceń, o ile ich warunek jest spełniony.

Jest to wygodna metoda, jeśli chcemy pozbyć się wszelkich zewnętrznych definicji i skupić się na samym wyrażeniu docelowym. Często kończy się powodzeniem, ale nie zawsze jest polecana. Szczególnie w wypadku, kiedy mamy do czynienia z wieloma schematami, długimi definicjami, których substytucja w miejsce referencji nie zawsze jest pomocna i nie ma wpływu na dowód. W takim wypadku zastosowanie redukcji ograniczy znacznie czytelność dowodu, ponieważ w miejsce krótkich wyrażeń identyfikujących inne schematy zostaną wstawione ich rozwinięcia (definicje), czego rezultatem jest długie i nieczytelne wyrażenie, trudne do zrozumienia.

Aby tego uniknąc, dostępne są dwie metody. Pierwsza polega na podzieleniu dowodu na kilka mniejszych, łatwo czytelnych teorematów, które udowodnione wcześniej spowodują większą czytelność i łatwiejsze zarządzanie procesem dowodzenia. Druga, to stosowanie komendy `invoke`, która polega na zamianie wybranego odwołania na jego definicję, co pozwala na efektywne sterowanie rozwinięciami wyrażeń.

4.4.5. Pętle dowodu

Komendy **Z/EVES**, które pozwalają na największy poziom automatyzacji dowodzenia to `prove` i `prove by reduce`. Ich wywołanie powoduje wykonanie przez system typowych sekwencji innych komend. Uruchomiona pętla przerywana zostaje, kiedy dowód jest zakończony, lub nie ma dalszych postępów.

Inne użyteczne komendy, to `prenex`, które w miarę możliwości eliminuje kwantyfikatory, `rearrange` używane w celu przeorganizowania predykatu prostsze fakty przesuwając na początek wyrażenia, oraz `equality substitution` aplikujące specjalne mechanizmy wykrywania równości i równoważności.

4.4.6. Podstawienie

W niektórych wypadkach zdarza się, że w zdaniu logicznym występuje kwantyfikator $\exists$ i system nie jest w stanie przekształcić takiego predykatu, chociaż oczywisty jest przypadek spełniający dane założenia. Jeśli komenda `prenex` nie potrafi zlikwidować kwantyfikatorów, do dyspozycji użytkownika dostępna jest jeszcze komenda podstawienia *(ang. instantiation)*, która umożliwia wyspecyfikowanie i podstawienie jakiejś wartości pod zmienną lub część wyrażenia. Odbywa się to w/g następującej reguły - mając dany predykat:

$$\exists x : S \bullet P(x)$$

podstawienie x do e przekształca go do

$$(e \in S \land P(e)) \lor \exists\, x : S \bullet P(x)$$

Ta reguła jest stosowana w celu zapewnienia równoważności początkowego wyrażenia docelowego z końcowym. Przykładowo

$$\begin{array}{l} \quad k \in \mathbb{Z} \land k \geq 0 \\ \Rightarrow (\exists\, n : \mathbb{N} \bullet k = n \lor k < 1 \land n < 1) \end{array}$$

Z/EVES nie potrafi znaleźć właściwej wartości dla n, choć oczywiste jest że $n = 0$ spełnia warunek kwantyfikatora. Możemy więc pomóc mu komendą **instantiate n == 0**. Prowadzi to do

$$\begin{array}{l} \quad k \in \mathbb{Z} \land k \geq 0 \\ \quad \land \neg\, (0 \in \mathbb{N} \land (k = 0 \lor k < 1 \land 0 < 1)) \\ \Rightarrow (\exists\, n : \mathbb{Z} \bullet k = n \lor k < 1 \land n < 1) \end{array}$$

co może już być udowodnione komendą **prove**. Dodatkowo system przekształcił zdanie typu $P \Rightarrow Q \lor R$ do $P \land \neg\, Q \Rightarrow R$.

4.5. Planowanie dowodu

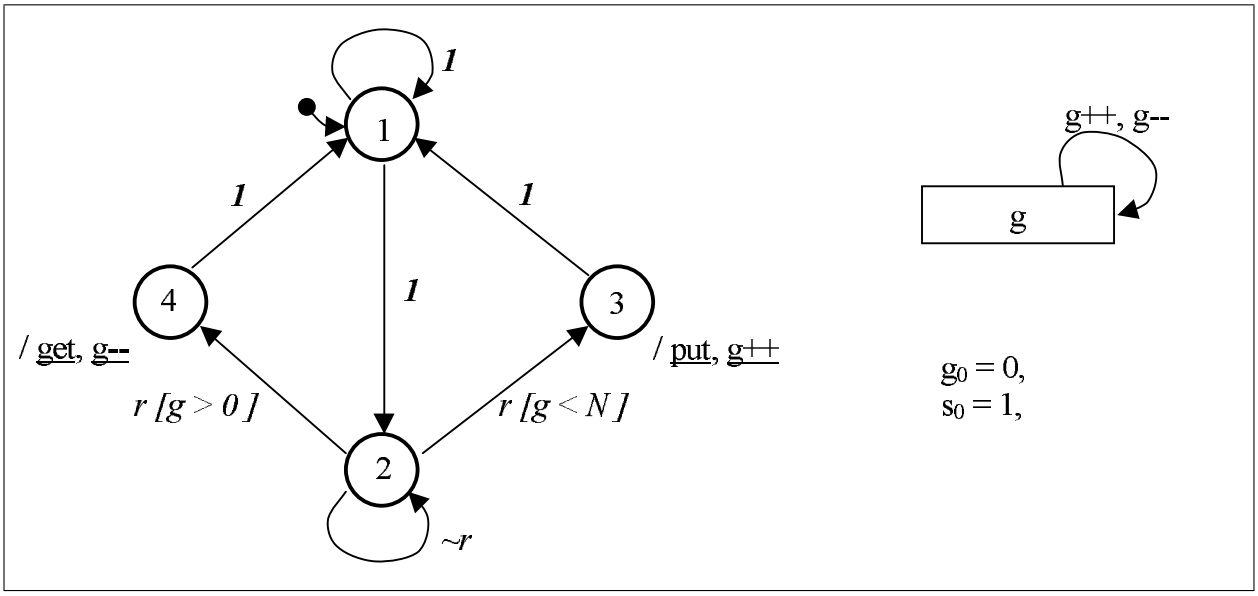
Pomimo faktu, że **Z/EVES** jest w chwili obecnej najbardziej zautomatyzowanym narzędziem wspomagającym przeprowadzanie dowodów, w wypadku bardziej skomplikowanych teorematów musimy mu pomóc, poprzez podanie zarysu dowodu. Interwencja użytkownika potrzebna też jest jeśli mechanizmy maszyny dowodzenia przekształcają predykat w wyrażenie trudno zrozumiałe i nie mogą doprowadzić go do końca, podczas gdy wystarczy zmienić jeden z kroków początkowych, aby dowód stał się prosty. Niestety wymaga to pewnego doświadczenia oraz dobrej znajomości aparatu matematycznego i jego twierdzeń, dostarczanego razem z systemem (patrz [5]).

Weźmy następujący przykład:

theorem example $[X, Y, Z]$
$\forall\, Q : X \leftrightarrow Y;\ R : Y \leftrightarrow Z;\ S : \mathbb{P}\, Y \bullet$
$(Q \rhd S) \fatsemi R = Q \fatsemi (S \lhd R)$

Na dowód tego teorematu składają się następujące kroki:

1. $Q \rhd S = Q \fatsemi \mathrm{id}\, S$ - ze znanego teorematu *compIdRight* z aparatu matematycznego pakietu.
2. Po podstawieniu mamy: $(Q \rhd S) \fatsemi R = (Q \fatsemi \mathrm{id}\, S) \fatsemi R$.
3. $(Q \fatsemi \mathrm{id}\, S) \fatsemi R = Q \fatsemi (\mathrm{id}\, S \fatsemi R)$ - ze znanego teorematu *compAssociates*.
4. $\mathrm{id}\, S \fatsemi R = S \lhd R$ - ze znanego teorematu *compIdLeft*.
5. Po podstawieniu mamy: $Q \fatsemi (\mathrm{id}\, S \fatsemi R) = Q \fatsemi (S \lhd R)$.



Rysunek 4.1: Model ECSM generatora komunikatów **put** i **get**

6. $(Q \rhd S) \fatsemi R = Q \fatsemi (S \lhd R)$ z (2), (3) i (5).

Podstawienia (2) i (5) mogą być przeprowadzone automatycznie przez system dzięki regułom równości mechanizmu upraszczania. Użytkownik znając ten dowód może wskazać systemowi użycie właściwych teorematów z aparatu matematycznego. Służy do tego komenda `use`. Plan dowodu **Z/EVES** wygląda zatem następująco:

proof
use compIdRight$[X,\ Y][R := Q]$;
use compAssociates$[X,\ Y,\ Y,\ Z][P := Q,\ Q :=$ id $S]$;
use compIdLeft$[Y,\ Z]$;
prove;
■

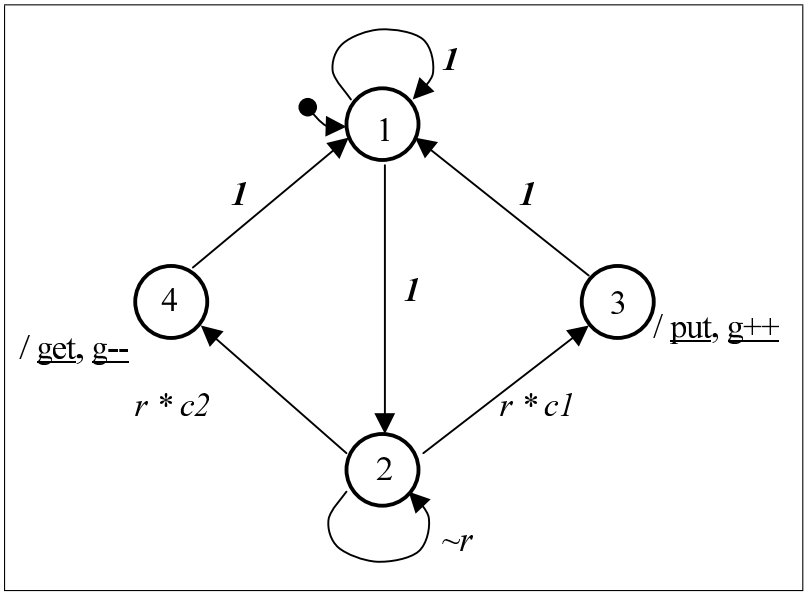
4.6. Przykład - dowodzenie właściwości ECSM

Przedstawię teraz przykład praktycznego użycia systemu **Z/EVES** do udowodnienia właściwości niedeterministycznego automatu skończonego, w tym przypadku ECSM. Podany sposób może być wykorzystany do opisu dowolnego automatu skończonego, oraz jako przykład wejściowy do próby udowodnienia jego własności.

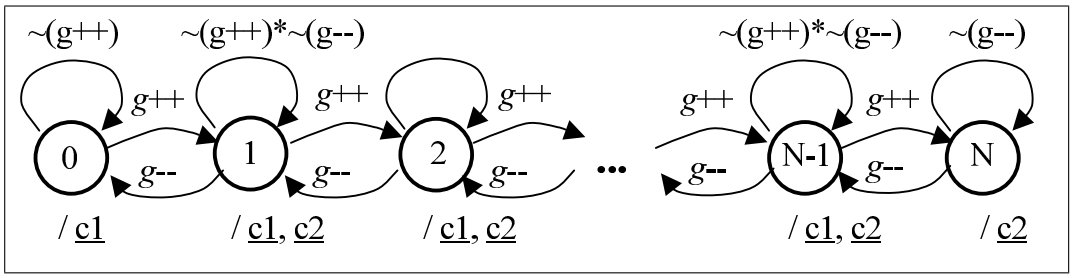
Rysunek 4.1 przedstawia generator odwołań do bufora, którego pojemność wynosi N.

Generator jest skonstruowany tak, aby nie przepełnił bufora, ani nie próbował pobierać z bufora pustego, dla znanego N. Komunikaty *put* i *get* są generowane niedeterministycznie, ale tylko wtedy, gdy bufor jest gotowy do przyjęcia zlecenia, czyli znacznik zajętości r nie jest ustawiony ($\neq 1$).

Warunkiem poprawności jest więc, że 'nigdy nie jest tak, że $g > N$'. 'Nigdy' można rozumieć jako 'na żadnej ścieżce w grafie osiągalności (RG)' lub 'w żadnym stanie RG'.



Rysunek 4.2: Model ECSM sprowadzony do CSM



Rysunek 4.3: Model CSM zachowań zmiennej **g**

Na gruncie modeli skończenie-stanowych, jeśli N jest *konkretne i znane*, badanie tego warunku poprawności można przeprowadzić w grafie RG, utworzonym jako produkt grafu sterowania CSM jak na rys. 4.2 i grafu CSM zachowań danej - jak na rys. 4.3.

Model zachowania z rys. 4.1 sprowadzono do postaci CSM, zastępując warunki $g > 0$ i $g < N$ przez występowanie lub nie występowanie sygnałów $c1$, $c2$, generowanych przez model zmiennej g.

Spróbujemy podejść do problemu, opisując graf za pomocą logiki $\mathbb{Z}$ i wprowadzając do tego opisu narzędzia rachunku predykatów. Musimy więc udowodnić, że dla dowolnego $N \geq 1$ wartość g nie przekracza N. Nie zakładamy już, że N jest *konkretne i znane*, ale przyjmujemy że $N \in \mathbb{N}$, a zatem automat jest skończony.

Zdefiniujmy bufor *Buffer* o pojemności N, wraz ze wskaźnikiem zajętości bufora r.

Buffer

$N, r : \mathbb{N}$

$N > 0$

Mając bufor, potrzebna jest nam przestrzeń stanów generatora i związany z nią

znacznik g bufora. Definiujemy 4 stany - zmienna *state* może znajdować się w zakresie $1 \mathinner{..} 4$.

States

$$state, g : \mathbb{N}$$

$$state \in 1 \mathinner{..} 4$$

Łączymy oba te schematy w jedną całość otrzymując definicję *Automat*. Jest to przykład koniunkcji dwóch schematów.

$$Automat \mathrel{\widehat=} Buffer \land States$$

Operacja *AutomatInit* określa początkowy stan automatu.

AutomatInit

$$Automat$$

$$state = 1$$
$$g = 0$$
$$r = 1$$
$$N = 10$$

W rzeczywistym systemie bufor ma zawsze ograniczony rozmiar, w *AutomatInit* ustalamy $N = 10$, chociaż N może być dowolną liczbą dodatnią.

Kolejne operacje będą określały możliwe przejścia z poszczególnych stanów automatu. W ten sposób faktycznie opisujemy automat za pomocą modelu $\mathbb{Z}$. Wskaźnik zajętości r jest nieistotny z punktu widzenia właściwości, którą chcemy udowodnić, pomijamy go więc w opisie. Operacja przejścia ze stanu 1 do 1:

*Op*1*to*1

$$\Delta Automat$$

$$state' = 1$$
$$g' = g$$

oraz z 1 do 2:

*Op*1*to*2

$$\Delta Automat$$

$$state' = 2$$
$$g' = g$$

Operacja *GoFromState*1 określa warunek przejścia ze stanu 1. Z tego że automat jest w stanie 1, wynika że przejdzie on do stanu 1 lub 2. Należy zauważyć, że użyta

jest tu alternatywa, a więc nie jest pewne czy po przejściu automat będzie w stanie 2, czy też nadal w stanie 1 - opisujemy więc w ten sposób niedeterminizm operacji.

*GoFromState*1

$\Delta Automat$

$state = 1 \Rightarrow Op1to1 \lor Op1to2$

Analogicznie operacje przejść ze stanu 2:

*Op*2*to*2

$\Delta Automat$

$r \neq 1 \Rightarrow state' = 2$
$g' = g$

*Op*2*to*3

$\Delta Automat$

$g < N \land state' = 3 \land g' = g + 1$

*Op*2*to*4

$\Delta Automat$

$g > 0 \land state' = 4 \land g' = g - 1$

Operacja *GoFromState*2 opisuje warunki wyjścia ze stanu 2, analogicznie do poprzedniego stanu:

*GoFromState*2

$\Delta Automat$

$state = 2 \Rightarrow Op2to2 \lor Op2to3 \lor Op2to4$

Operacje przejścia ze stanu 3:

*Op*3*to*1

$\Delta Automat$

$state' = 1$
$g' = g$

*GoFromState*3
$\Delta Automat$

$state = 3 \Rightarrow Op3to1$

I ze stanu 4:

*Op*4*to*1
$\Delta Automat$

$state' = 1$
$g' = g$

*GoFromState*4
$\Delta Automat$

$state = 4 \Rightarrow Op4to1$

Opisaliśmy już cały automat, teraz potrzebujemy formalnego połączenia tych opisów. Schemat *GoState* sumuje znane fakty, określając możliwe operacje na przestrzeni stanów automatu. Operacja ta symbolizuje zatem jedno przejście automatu.

GoState
$\Delta Automat$

$state = 1 \Rightarrow GoFromState1$
$\lor\ state = 2 \Rightarrow GoFromState2$
$\lor\ state = 3 \Rightarrow GoFromState3$
$\lor\ state = 4 \Rightarrow GoFromState4$

Teraz możemy przejść do dowodzenia właściwości generatora. Najpierw standardowy teoremat dowodzący osiągalność początkowego stanu generatora:

theorem canInitAutomat
$\exists\, Automat \bullet AutomatInit$

Dowód jego właściwości **Z/EVES** potrafi przeprowadzić automatycznie:

proof
prove by reduce
■

I końcowy teoremat reprezentujący właściwość, którą chcieliśmy dowieść na początku. *gNoMoreThanN* mówi, że ani operacja początkowa *AutomatInit*, ani przejście z dowolnego stanu nie spowodują, że $g > N$. Czyli 'nigdy g nie przekracza N'.

theorem gNoMoreThanN
$(AutomatInit \Rightarrow \neg\ g > N) \land (GoState \land \neg\ g > N \Rightarrow \neg\ g' > N)$

Dowód może być przeprowadzony automatycznie, za pomocą redukcji z włączoną normalizacją.

proof
with normalization reduce
■

Wykazaliśmy zatem właściwość automatu za pomocą pojęć dostępnych w notacji Z, oraz maszyny dowodzenia systemu **Z/EVES**. Przykład jest dosyć prosty, jego wadą jest jednak ograniczenie do skończonej przestrzeni stanów, może też razić opisem całego automatu i wszystkich jego przejść. W rzeczywistej specyfikacji powinny być umieszczone fakty, na których nam zależy żeby je opisać lub udowodnić, natomiast rzeczy mniej ważne i nieistotne z punktu widzenia funkcjonalności powinny być pominięte. Można w ten sposób zapewnić pole manewru programistom, nie narzucając im sztywnych ram działań, a jednocześnie zapewniając niezbędny formalizm całego przedsięwzięcia.

4.7. Uwagi praktyczne

Z/EVES jest bardzo rozbudowanym programem, z niespotykanymi możliwościami automatycznego przeprowadzania dowodów. Niestety jest to jednak narzędzie przydatne w tej chwili bardziej naukowcom, aniżeli projektantom komercyjnych projektów. Dzieje się tak z kilku powodów:

- Brakuje bardziej zaawansowanego zarządzania projektem, ponieważ wszystkie fakty muszą być zawarte w jednym dokumencie. Nie można rozproszyć specyfikacji na niezależne sekcje, co w przypadku większych przedsięwzięć stanowi oczywistą trudność.

- Automatyczne dowodzenie, choć bardzo dobrze rozwiązane, działa niestety tylko dla prostszych dowodów. Zazwyczaj konieczna jest interwencja projektanta, który musi przygotować odpowiedni /it proof script. Ustalanie strategii dowodu jest zajęciem niebanalnym, oprócz doskonałej znajomości logiki i aparatu matematycznego wymagane jest zrozumienie konstrukcji wbudowanego /it toolkitu $\mathbb{Z}$, a także wiedza, w jaki sposób działa /it Z/EVES. W praktyce projektant musi mieć duże doświadczenie, nawet gdy stosowane zapisy /zet są na średnim poziomie trudności.

- Toolkit i wiedza matematyczna zakodowana w /bf Z/EVES ograniczają się jedynie do podstawowych pojęć notacji. W praktyce większość problemów, z jakimi borykają się projektanci jest typowa, brakuje więc czegoś, co pełniło by rolę biblioteki standardowej, opisującej klasy problemów. Przykładami takich klas problemów są liczby zmiennoprzecinkowe, definicje znane z baz danych (tabela, indeks, klucz), pliki lub przetwarzanie równoległe (synchronizacja, semafor, wątek).

Rozdział 5

Standard ISO

W lipcu 2002 roku, po ponad 8 latach prac, powstała pierwsza edycja międzynarodowego standardu ISO/IEC 13568.

Historię standaryzacji notacji $\mathbb{Z}$ rozpoczyna ukazanie się w 1989 pierwszego wydania referencji języka J. M. Spivey'a ([1]). Książka była przełomem w ustaleniu postaci notacji, w której mniej więcej przetrwała do dzisiaj. Praca ta stała się nieformalnym standardem, na którym bazuje większość publikacji, jak również narzędzi do sprawdzania składni i typów.

Następnym krokiem było rozpoczęcie procesu standaryzacji $\mathbb{Z}$ zgodnie z zaleceniami organizacji ISO. Proces ten przewiduje cztery fazy ewoluowania dokumentów. Pierwsza, czyli *Working Drafts* obejmuje dokumenty, które są tworzone oraz sprawdzane przez członków panelu. Następuje po tym faza *Committee Drafts* - które są podane do publicznej wiadomości. Jest to czas debaty, w wyniku której powstaje finalna wersja dokumentu. Końcowy CD dla $\mathbb{Z}$ został zatwierdzony 25.08.1999. Trzeci etap to są już wstępne wersje standardu, do których wprowadza się tylko zmiany korygujące wykryte błędy i nieścisłości. Dokument o nazwie *Draft International Standard* został zaaprobowany 28.04.2002, w stosunku do finalnego CD zawierał tylko drobne zmiany dotyczące kodowania Unicode. Wersja ta oczekiwała na akceptację do 04.07.2002, czego wynikiem jest gotowy i opublikowany *International Standard*, dostępny poprzez ciało standaryzacyjne.

5.1. Struktura standardu

Standard wprowadza regulacje w następujących dziedzinach:

- składnia $\mathbb{Z}$
- system typów
- semantyka $\mathbb{Z}$
- aparat matematyczny i stosowane operatory
- język typu *mark-up* do zapisu notacji w trybie `ASCII`

W jego skład nie wchodzi opis żadnej metodologii stosowania $\mathbb{Z}$. Założeniem jego twórców miało być jak najpełniejsze wykorzystanie faktu, że referencja języka określona w [1] jest najszerzej znanym dokumentem opisującym tę tematykę. Standard wykorzystuje zatem dorobek tej pracy powstałej 10 lat wcześniej, wprowadzając jednak szereg innowacji i poprawek w miejscach, które nie były do końca zbadane lub przemyślane. Pomimo faktu, że większa waga przywiązywana jest do formalizacji zapisu i składni języka, to utrzymany jest duch książki Spivey'a, przykładowo w skład standardu wchodzi wprowadzenie (ang. *tutorial*) czytelnika w podstawy $\mathbb{Z}$.

Podstawowym rozszerzeniem wprowadzonym w ISO jest dalsza strukturalizacja specyfikacji poprzez istnienie *sekcji*. Mają one nazwy i możliwe jest podanie sekcji nadrzędnej, zawierającej definicje przydatne w danym momencie.

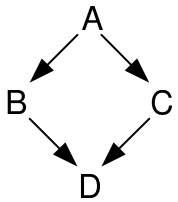
section *myspec* parents *toolkit*
$paragraph_1$
...
$paragraph_n$

Jak już było to wspomniane, w podstawowej wersji $\mathbb{Z}$ każda specyfikacja zaczyna się w zasięgu aparatu matematycznego $\mathbb{Z}$, tzn. dostępne są stałe, nazwy i typy przez niego wprowadzone. Sekcje natomiast identyfikują *otoczenie*, które może być dziedziczone, co pozwala na projektowanie własnych bibliotek $\mathbb{Z}$ i dzielenie dużych specyfikacji na komponenty możliwe do ponownego wykorzystania.

Weźmy deklaracje czterech sekcji:

section A
section B parents A
section C parents A
section D parents B,C

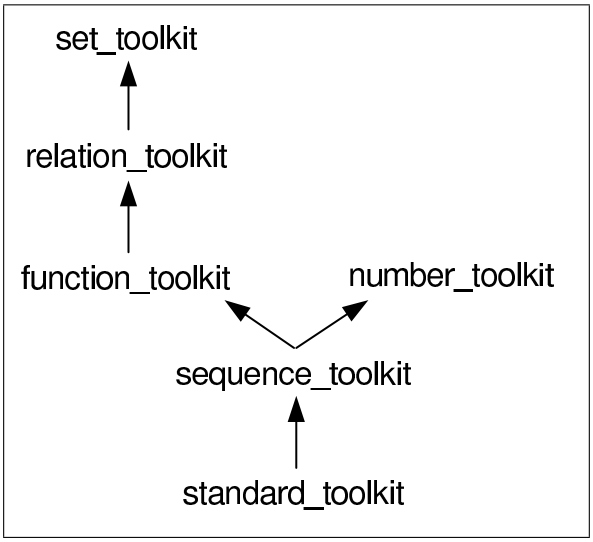
Dziedziczenie deklaracji po rodzicach przedstawia graf:



W sekcjach B i C można używać definicji z sekcji A, natomiast w D - definicji z A, B, C. W każdym punkcie specyfikacji ważne są globalne deklaracje z bieżącego kontekstu, oraz z sekcji-przodków.

5.2. Aparat matematyczny

Aparat matematyczny pełni funkcję biblioteki standardowej w klasycznych językach programowania. Rzeczywiste specyfikacje korzystają nie tylko ze standardowej składni



Rysunek 5.1: Hierarchia sekcji aparatu matematycznego **Standard** $\mathbb{Z}$

$\mathbb{Z}$, ale również z matematycznych rozszerzeń, twierdzeń i faktów towarzyszących poszczególnym konstrukcjom. Dlatego oprócz części standardu opisujących składnie i semantykę notacji (np. reguły leksykalne czy znaki charakterystyczne $\mathbb{Z}$) zdefiniowane są matematyczne elementy notacji (ang. *mathematical toolkit*). Podobnie jak w zwykłym języku programowania korzystamy z funkcji, które służą np. do operacji na łańcuchach znakowych lub liczbach, tak w **Standard** $\mathbb{Z}$ dostępne są operatory i zapisy, pozwalające na korzystanie z sekwencji czy wielozbiorów.

Aparat matematyczny opisany jest w sekcjach, których hierarchia przedstawiona jest na rys. 5.1.

Podział aparatu matematycznego na sekcje umożliwia ich redefiniowanie lub nawet usuwanie. Przykładowo *function_toolkit* może zostać pominięte bez szkody dla wiedzy zdefiniowanej w *number_toolkit*. Jest to informacja szczególnie przydatna dla narzędzi analizujących specyfikację $\mathbb{Z}$.

5.3. Rozszerzenia $\mathbb{Z}$ w stosunku do ZRM

Jak już wspomniano, standard jest rozszerzeniem pracy Spivey'a [1]. W pewnych miejscach jest niekompatybilny z ZRM, w większości przypadków jednak zawiera udogodnienia i innowacje, które usuwają nieścisłości i określają brakujące fakty. Punkt ten zawiera najważniejsze różnice pomiędzy standardem a [1]; w większości jest oparty na pracy Toyna [19].

5.3.1. Wzajemnie rekursywne typy wolne

W punkcie 2.6.2 opisano pojęcie typów wolnych oraz zestawienie narzędzi matematycznych potrzebnych do ich definicji. To użyteczne narzędzie do opisu danych, mających charakter rekurencyjny. Jednak dotychczasowy kształt $\mathbb{Z}$ w rozumieniu Spivey'a nie

dopuszczał jednoczesnego referowania się przez dwa różne typy, czyli wykluczał ich wzajemną rekurencyjność. Standard ISO $\mathbb{Z}$ usuwa tę lukę.

Przykładowa definicja

$$
\begin{array}{ll}
dec & ::= Dec\langle\langle name \times exp\rangle\rangle \\
\& & \\
exp & ::= Let\langle\langle \mathrm{seq}\, dec \times exp\rangle\rangle \\
 & \mid Num\langle\langle \mathbb{N}\rangle\rangle
\end{array}
$$

ilustruje wzajemną rekurencyjność typów wolnych. Definicje typów wzajemnie rekursywnych muszą być opisane w jednym akapicie i połączone znakiem &.

Nowa właściwość notacji jest szczególnie przydatna w zastosowaniu $\mathbb{Z}$ do definicji składni języków.

5.3.2. Operatory

Operator jest jednostką leksykalną, która zapisana w specyfikacji musi spełniać pewne reguły, np. poprzedzać i być poprzedzana przez parametry. $\mathbb{Z}$ definiuje wiele operatorów, a ZRM wprowadza nawet jego pojęcie. Jednakże składnia ich opisu nie była określona; pojęcie i definicja operatora musiały być de facto zaimplementowane w parserze notacji. Przed standaryzacją zakładano, że czytelnik zna już znaczenie danego operatora i ich definicje nie mogły być częścią specyfikacji. Standard $\mathbb{Z}$ to zmienia. W dokumencie ISO dostępna jest notacja, która pozwala na definiowanie operatorów za pomocą tzw. szablonów. Ich użycie musi poprzedzać określenie szablonu, który podaje jeden z trzech jego rodzajów: *relation, function, generic.*

Przykłady z toolkitu:

relation (_ ≠ _)
function 30 leftassoc (_ ∪ _)
generic 5 rightassoc (_ ⇔ _)
relation (disjoint _)

Szablon rozpoczyna się nazwą rodzaju operatora, a kończy jego wzorcem. W środku, w zależności od typu może znajdować się liczba oznaczająca pierwszeństwo, oraz informacja o jego łączności.

Za pomocą nowej składni w toolkicie zdefiniowano w ten sposób wiele operatorów $\mathbb{Z}$.

5.3.3. Hipotezy

Notacja ZRM wprowadzała wiele praw, które jednak nie były formalnie określone w sensie notacji. Narzędzia dowodu często wprowadzają swoje elementy, które umożliwiają opisanie nie tylko praw, ale również hipotez, docelowych predykatów, teorematów, lematów i aksjomatów. Ich składnia jest oczywiście zmienna. Autorzy standardu doszli do wniosku, że warto sformalizować hipotezy *(ang. conjectures)*, gdyż są to elementy bardzo przydatne do przeprowadzania dowodów. Do tego celu służy symbol $\models$, po

którym następuje pojedynczy predykat. Pozwala to również na sformalizowanie praw zawartych w aparacie matematycznym notacji, np.

$$\models \forall\, a : \mathbb{Z} \bullet a..a = \{a\}$$

Oczywiście hipoteza może być błędna, chociaż poprawna w sensie formalnym zapisu. Przykładem może być:

$$\models 42 \in \{1, 2, 3\}$$

Hipotezy najczęściej wykorzystuje się jako punkty startowe dowodu.

5.3.4. Wiązania i selekcje elementów

Standardowa notacja $\mathbb{Z}$ dopuszczała definiowanie typów danych będącymi iloczynami zbiorów. Składowe elementu takiego zbioru mogły być nazwane (tzn. istniał identyfikator, za pomocą którego można było określić składową) lub nienazwane (krotka). Aby w pełni wykorzystać te pojęcia, potrzebne są dwa rodzaje wyrażeń: konstruktory, za pomocą których specyfikuje się dany zbiór, oraz selektory, które określają wybraną składową elementu. Notacja ZRM nie definiowała w pełni składni realizującej te dwa typy wyrażeń. W dokumentacji notacji posługiwano się wprawdzie wyrażeniami, aby opisać wiązania elementów z poszczególnymi składowymi (używając notacji $\langle p_1 \rightsquigarrow r_1, \ldots, p_n \rightsquigarrow r_n \rangle$), ale zaznaczono, że tego typu wyrażeń nie można stosować w specyfikacjach.

Standard $\mathbb{Z}$ rozwiązuje ten problem. Od tej pory można używać wiązań dla elementów nazwanych (wiązanie to konstrukcja, która łączy identyfikator elementu z jego wartością):

$$\langle\!| p_1 == r_1, \ldots, p_n == r_n |\!\rangle$$

oraz selekcji elementów krotki:

$$e.b$$

gdzie e odwołuje się do nazwy oznaczającą krotkę, a b to dodatnia liczba całkowita w zakresie od 1 do $n-$krotności iloczynu kartezjańskiego.

Ilustruje to poniższa tabela:

	ZRM		**Standard $\mathbb{Z}$**	
	Konstruktor	**Selektor**	**Konstruktor**	**Selektor**
Krotki	(x, y, z)		(x, y, z)	*trojka*.3
Wiązania		*wiazanie.nazwa*	$\langle\!\| x == 1, y == 2 |\!\rangle$	*wiazanie.nazwa*

5.3.5. Schematy w wyrażeniach

Wyrażenie ma wartość określonego typu. Schemat również posiada wartość - jest to zbiór jego wiązań. W dotychczas obowiązującej notacji ZRM schematy nie mogły być jednakże użyte jako zwyczajne wyrażenie. ZRM dopuszczała wprawdzie użycie schematów w wyrażeniach rachunku schematów, ale było to ograniczone, ponieważ dla schematów obowiązywała oddzielna kategoria syntaktyczna, jako wyrażenia schematyczne (ang. *schema expressions*). Te rozgraniczenie pomiędzy wyrażeniem, a wyrażeniem schematycznym stwarzało ograniczenia w stosowaniu schematów do zapisywania wyrażeń, np. nie można było go użyć jako parametru operatora θ.

Standard $\mathbb{Z}$ ujednolica notację likwidując wyrażenia schematyczne jako oddzielną kategorię syntaktyczną - można teraz ich używać jako elementów zwykłych wyrażeń. Przykładowo możliwy jest następujący zapis:

S

$$x : \mathbb{Z}$$
$$y : \mathbb{N}$$

ΔSx

$$S;\ S'$$

$$\theta(S \setminus (x)) = \theta(S \setminus (x))'$$

5.3.6. Schematy bez sygnatur

Schemat bez sygnatury, czyli schemat pusty, może powstać np. w wyniku operacji ukrywania (opisanej w 2.4.5). Jest to schemat bez deklaracji, zawierający jedynie predykat. Jednakże notacja ZRM nie pozwalała na bezpośrednie zapisanie takiego schematu, co zostało zmienione w standardowym $\mathbb{Z}$. Standard ISO pozwala na deklarowanie schematów bez części deklaracyjnej, np.:

$$Schema \mathrel{\widehat=} [\ |\ \exists\, x, y : \mathbb{Z} \bullet x \neq y\ ]$$

Schemat taki powstaje w wyniku ukrycia x i y, jednak taki wynik jest niemożliwy do zapisania w ZRM:

$$Schema \mathrel{\widehat=} [\ x, y : \mathbb{Z} \mid x \neq y\ ] \setminus (x, y)$$

Schematy puste są szczególnie użyteczne, szczególnie w połączeniu z nową właściwością standardu $\mathbb{Z}$ sprowadzającą schemat do zwykłego wyrażenia. Schemat pusty użyty jako predykat ma wartość *true* jeśli jego predykat również jest *true* i odpowiednio *false* gdy predykat jest *false*. Ujednolica to definicje syntaktyczne notacji i pozwala na bardziej intuicyjne konstruowanie wyrażeń.

5.4. Reprezentacja znakowa notacji

$\mathbb{Z}$ wyrażone jest w symbolach matematycznych. Jest kilka sposobów ich kodowania do postaci akceptowalnej przez współczesne systemy komputerowe. Standard definiuje trzy:

- Unicode
- Znaczniki zgodne z LaTeX
- Znaczniki typu *e-mail*

Obecnie prowadzone są też prace nad sposobem reprezentacji $\mathbb{Z}$ w **XML**, nie jest to jednak część standardu.

5.4.1. Unicode

Powszechnie zaakceptowany standard Unicode zakłada wykorzystanie dwóch bajtów do opisu jednego znaku, zamiast jednego (ASCII). Większość znaków $\mathbb{Z}$ jest już zdefiniowana w tym standardzie, ale istnieją wyjątki. Znaki, których nie ma w obecnej wersji standardu znajdują się w AMS/STIX - zbioru znaków matematycznych proponowanych w następnej wersji Unicode.

5.4.2. Znaczniki LaTeX

Dla potrzeb nadal szeroko stosowanego ASCII i 8-bitowej reprezentacji znakowej, oraz ze względu na upowszechnienie się zapisów $\mathbb{Z}$ w formacie akceptowanym przez LaTeX, w skład standardu wszedł zbiór znaczników, które reprezentują elementy notacji. Przykładowo:

Δ	`\Delta`
Ξ	`\Xi`
$\Leftrightarrow$	`\iff`

Identycznie jak w dokumentach LaTeX, standard definiuje sposób zapisu paragrafów, schematów, aksjomatów i definicji uogólnionych. Schemat wygląda więc następująco:

```
\begin{schema}{NAME}
...
\where
...
\end{schema}
```

Język znaczników LaTeX jest obecnie najbardziej rozpowszechnionym sposobem zapisu dokumentów $\mathbb{Z}$ i najwięcej narzędzi potrafi go obsługiwać.

5.4.3. Znaczniki *e-mail*

Ostatni sposób zapisu $\mathbb{Z}$ jest przewidziany jako prosty język znaczników przeznaczony do czytania przez człowieka. Nie przewiduje się czytania tego formatu przez narzędzia. Znajduje on zastosowanie w korespondencji e-mail, lub w wyświetlaniu specyfikacji przez urządzenia o małej rozdzielczości, takie jak terminale ASCII.

Podstawową cechą tego zapisu jest wyodrębnienie "tekstów specjalnych", oznaczonych przez znak %. Np. tekst `%x` jest w ten sposób odróżnione od nazwy `x`. Standard zaleca implementację przez urządzenie małego zbioru liter greckich, takich jak `%Delta%` (Δ) albo `%lambda%` (λ). Pozostałe operatory i zapisy w przeważającej większości mogą być zastąpione przez proste kombinacje znaków ASCII.

Oto przykładowy zapis w konwencji e-mail:

```
+== [X] ===
     _ %u _ :  %P X %x %P X -> %P X
|-
     %A a, b:  %P X @ a %u b = { x :  X | x %e a \/  x %e b }
-==
```

5.5. Podsumowanie

Standard $\mathbb{Z}$ jako międzynarodowy standard ISO jest ważnym krokiem naprzód w upowszechnieniu się notacji jako metody projektowania oprogramowania, oraz ogólnie szerszego zastosowania metod formalnych. Oprócz wielu ważnych innowacji w celu usunięcia niejasności i niejednoznaczności mają jednak miejsce zmiany, które są niekompatybilne z wcześniejszymi wersjami notacji, a co za tym idzie ze stosowanymi obecnie narzędziami.

Wydaje się więc, że rola standardu jako czynnika promującego formalną specyfikację będzie się zwiększać w miarę powstawania narzędzi go obsługujących. Niestety, obserwując tempo prac i rozwoju nad $\mathbb{Z}$ oraz zważywszy na stopień skomplikowania tej technologii nie należy oczekiwać, aby powszechne zastosowanie $\mathbb{Z}$ w produkcji oprogramowania było kwestią najbliższych lat. Najprawdopodobniej ograniczy się ono tymczasem do projektów o zwiększonych wymaganiach co do stabilności i niezawodności systemu.

Rozdział 6

Object-Z

Object-Z jest rozszerzeniem standardowej notacji $\mathbb{Z}$, wzbogacającym ją o elementy programowania zorientowanego obiektowo (OOP). Paradygmat obiektowy stał się *de facto* podstawową metodologią tworzenia oprogramowania od połowy lat 80-tych, głównie dzięki doskonałej strukturalizacji kodu, która szczególnie ułatwia rozwijanie i zarządzanie dużymi projektami.

Specyfikacja $\mathbb{Z}$ składa się ze schematów opisującym stany systemu oraz operacje przejść pomiędzy nimi. Analiza jednej operacji wymaga analizowania wszystkich pozostałych schematów, co jest niepraktyczne w przypadku większych specyfikacji.

Podstawowym pojęciem wprowadzonym Object-Z jest więc *klasa*. Jest to fragment specyfikacji, który może być analizowany niezależnie od reszty. Powiązane ze sobą stany systemu i operacje na nich, enkapsulowane są w jedną grupę, która tworzy klasę. Uproszczenie struktury projektu poprzez te mechanizmy jest przydatne na każdym etapie tworzenia specyfikacji, począwszy od budowania schematycznych diagramów, aż do sprawdzania i dowodzenia poprawności. Klasy mogą wywodzić się z innych klas, dzięki dziedziczeniu i zawieraniu instancji, co umożliwia rozbijanie skomplikowanych opisów obiektów na szereg mniejszych, powiązanych ze sobą relacjami obiektowymi.

Notacja Object-Z powstała w wyniku pracy badawczej Software Verification Research Centre przy Uniwersytecie w Queensland (Australia). Informacje na jej temat dostępne są w [20] i [21]. Oprócz Object-Z istnieje kilka innych rozszerzeń obiektowych $\mathbb{Z}$, jak np. Hall, ZERO, MooZ, OOZE, Schuman&Pitt, Z++ i ZEST.

6.1. Klasa

Klasa w Object-Z jest nazwanym schematem, który ma następującą postać:

Klasa[*Parametry*]

lista dostępności
dziedziczone klasy
definicje typów
definicje stałych
schematy stanów
stan początkowy
schematy operacji

---

niezmiennik historii

Aby opisać poszczególne składowe klasy, posłużmy się przykładem. Poniższy schemat przedstawia abstrakcyjną klasę figur, które posiadają współrzędne, rozmiar, kolor i mogą być przesuwane.

Figure

$\uparrow(x, y, colour, max\_x, max\_y, bound\_x, bound\_y, \textsc{Init}, Move)$

$colour : Colour$

$max\_x : \mathbb{Z}$	$max\_y : \mathbb{Z}$
$max\_x = 1000$	$max\_y = 1000$

$x, y : \mathbb{R}$

$x > -max\_x \land x < max\_x$
$y > -max\_y \land y < max\_y$

$bound\_x, bound\_y : \mathbb{R}$

$bound\_x > 0 \land bound\_y > 0$
$x + bound\_x < max\_x \land y + bound\_y < max\_y$

Init

$x = y = 0$
$bound\_x = bound\_y = 0$

Move

$\Delta(x, y)$
$dx?, dy? : \mathbb{Z}$

$x' = x + dx?$
$y' = y + dy?$

6.1.1. Właściwości klasy i lista dostępności

Właściwościami klasy (ang. *features*) są jej atrybuty (stałe i zmienne stanów), operacje oraz stan początkowy. Dla naszej klasy są nimi: $x, y, colour, max\_x, max\_y, bound\_x,$

bound_y, *INIT*, *Move*.

Definicja klasy rozpoczyna się od *listy dostępności*, oznaczonej operatorem $\uparrow$. Wszystkie elementy na niej umieszczone są bezpośrednio dostępne przez operacje klasy i jej instancje. Jeżeli lista jest pominięta, dostępne są wszystkie komponenty klasy.

6.1.2. Definicje typów i stałych

Wewnątrz klasy mogą znajdować się definicje typów, aksjomaty oraz definicje aksjomatyczne.

$$colour : Colour$$

$$max\_x : \mathbb{Z}$$
$$\hline$$
$$max\_x = 1000$$

$$max\_y : \mathbb{Z}$$
$$\hline$$
$$max\_y = 1000$$

Definicje te obowiązują wewnątrz klasy. Jeśli znajdują się na liście dostępności, to mogą z nich korzystać operacje.

Za pomocą definicji aksjomatycznych zdefiniowaliśmy kolor figury, oraz stałe max_x, max_y, które określają możliwy przedział wartości współrzędnych do $-1000 \,..\, 1000$.

6.1.3. Stany klasy

Podstawową różnicą pomiędzy schematem opisującym stan w Object-Z, a zwykłym $\mathbb{Z}$ jest to, że jest on nienazwany (w standardowym $\mathbb{Z}$ schematy nienazwane nie są częścią języka).

Zmienne, które w schematach stanowych klasy wprowadzane są części deklaracyjnej, są zmiennymi, na których operuje klasa. Predykaty tych schematów są *niezmiennikami klasy*. Niezmienniki są zawsze spełnione; każda operacja musi zagwarantować, że po jej wykonaniu niezmienniki klas pozostaną prawdziwe.

Dowolny zestaw danych, który jest zgodny z predykatami stanów klasy, jest nazywany *instancją klasy*.

W klasie *Figure* schematy stanów wprowadzają zmienne $x, y, bound\_x, bound\_y$. Każda figura znajduje się w obszarze prostokąta ograniczonego punktami (x, y) oraz $(x + bound\_x, y + bound\_y)$. Zmienne $bound\_x, bound\_y$ to rozmiar tego prostokąta, są one dodatnie.

Niezmiennikiem klasy jest to, że niezależnie od operacji wykonywanych na figurze, będzie się ona mieścić wewnątrz zadeklarowanego układu współrzędnych, czyli współrzędne każdego wierzchołka opisującego ją prostokąta znajdują się w przedziałach odpowiednio $-max\_x \,..\, max\_x$ i $-max\_y \,..\, max\_y$.

6.1.4. Stan początkowy

W Object-Z schemat, który warunkuje stan początkowy, zawsze nazywany jest *INIT*. Jest to pojęcie, które występuje również w $\mathbb{Z}$ (p. 2.5.3), występują tu jednak istotne różnice.

Po pierwsze, stan początkowy jest lokalny dla danej klasy i opisuje tylko elementy znajdujące się na liście dostępności. Niepotrzebna jest więc ich ponowna deklaracja, zatem sygnatura schematu *INIT* jest pusta.

Po drugie, w Object-Z stan początkowy nie jest operacją. Nie zmienia on wartości zmiennych istniejącego obiektu, ale jego predykat definiuje warunek. Warunek ten wraz z niezmiennikami klas definiują warunek początkowy obiektu (ang. *initial condition*). Jeśli dane obiektu spełniają warunek początkowy, to znaczy że znajduje się on w *konfiguracji początkowej*.

W naszym przypadku instancja klasy *Figures* jest w konfiguracji początkowej, jeśli figura znajduje się na początku układu współrzędnych, oraz długość i szerokość opisującego ją prostokąta są równe 0.

6.1.5. Operacje

Operacje w Object-Z są zdefiniowane podobnie jak w $\mathbb{Z}$, z tą różnicą (poza oczywiście zasięgiem danych, na których są określone - wewnątrz klasy) że zamiast odwołania do istniejącego schematu, występuje w ich sygnaturze Δ-lista. Ma ona postać $\Delta(v_1, \ldots, v_n)$, gdzie $v_1, \ldots, v_n$ to są zmienne, których wartość będzie zmieniona w wyniku operacji.

Inaczej niż w $\mathbb{Z}$, o ile operacja nie zmienia wartości zmiennych, nie ma potrzeby jawnego deklarowania ich w sygnaturze (nie ma więc odpowiednika zapisu ΞS), ani pisania predykatu postaci $x' = x$.

Operacja może mieć warunki wstępne, w postaci predykatów określonych na niedekorowanych zmiennych. Jest możliwe, że dla poprawnej instancji klasy (dane spełniają niezmienniki klasy) nie będzie możliwe zastosowanie danej operacji, gdyż nie będą spełnione jej warunki wstępne.

W klasie *Figures* występuje jedna operacja *Move*, która 'przesuwa' obiekt w oparciu o podane parametry $dx?$, $dy?$, zmieniając w ten sposób zmienne x, y. Nie ma potrzeby podawania dodatkowych warunków na $dx?$, $dy?$, gdyż operacja i tak będzie poprawna tylko wtedy, gdy zostaną zachowane niezmienniki klasy.

6.2. Instancje

Tworzenie instancji jest podstawową techniką łączenia klas. Typowy system składa się z wielu klas i ich instancji, Object-Z umożliwia nie tylko opisanie jak się zachowują indywidualne obiekty, ale również interakcje zachodzące pomiędzy nimi.

Podstawowym zapisem instancji obiektu jest:

$f : Figure$

f jest referencją do instancji klasy *Figure*. Należy zauważyć, że jest to tylko referencja, nie mówi ona nic o stanie danego obiektu. Jego stan może się zmieniać, ale f pozostaje niezmienne.

Dwie referencje można porównać ze sobą, aby ustalić, czy mamy do czynienia z tym samym obiektem. Zapis:

$$f1, f2 : Figure$$

nie musi oznaczać deklaracji dwóch różnych instancji *Figure*. Wymusić to można dodatkowym predykatem $f1 \neq f2$.

Jak już było wspomniane, instancja obiektu oznacza zestaw danych zgodny z niezmiennikami klasy. Niepotrzebne są więc dodatkowe predykaty na f, ponieważ referencja ta odnosi się do prawidłowej instancji, spełniającej niezmienniki klasy *Figure*.

Do komponentów klasy (znajdujących się na liście dostępności) można odwoływać się za pomocą typowej notacji z kropką. Przykładowo

$$f.x$$

jest odwołaniem się do zmiennej x klasy *Figure*, natomiast

$$f.Move$$

jest odniesieniem się do operacji *Move*.

Do zilustrowania tych pojęć posłużymy się przykładem klasy, zawierającej dwie figury.

FigurePair

$\upharpoonright(\textsc{Init}, MoveBoth, SameBoundingBox)$

$f1, f1 : Figure$

$f1 \neq f2$

Init

$f1.\textsc{Init}$

$f2.\textsc{Init}$

$MoveBoth \mathrel{\widehat=} f1.Move \land f2.Move$

SameBoundingBox

$f1.bound\_x = f2.bound\_x$

$f1.bound\_y = f2.bound\_y$

W klasie tej, która zawiera dwie instancje klasy *Figure* definiujemy najpierw schemat stanu, który mówi, że referencje $f1, f2$ mają odnosić się do dwóch różnych obiektów (co nie znaczy że obiekty te nie mogą być takie same, np. identyczne figury położone w tym samym miejscu na układzie współrzędnych).

Stan początkowy *INIT* określa, że obiekt *FigurePair* jest w konfiguracji początkowej, jeśli również są w niej instancje $f1, f2$. Możemy zapisać $f1.INIT$, gdyż *INIT* występuje na liście dostępności klasy *Figure*.

Operacja *MoveBoth* jest zadeklarowana jako koniunkcja operacji *Move* obu figur ($f1$ i $f2$). Jest to koniunkcja schematów, opisana już w rozdziale o rachunku schematów (2.4.5).

SameBoundingBox nie zmienia nic w żadnym obiekcie, nie ma potrzeby zatem deklarowania Δ-listy. Operacja będzie wykonana pomyślnie, jeżeli figury $f1, f2$ będą miały ten sam rozmiar (prostokąta opisującego).

W Object-Z określone są standardowe operatory $\mathbb{Z}$ łączące operacje, jak np. złożenie ($\fatsemi$) czy potok ($\gg$), a oprócz tego dwa dodatkowe: operator równoległy ($a.Op1 \parallel b.Op2$) i zasięgu ($Op1 \bullet Op2$). Więcej informacji na ten temat w [20].

6.3. Dziedziczenie

Dziedziczenie jest drugą techniką łączenia klas. Za jego pomocą można definiować skomplikowane klasy poprzez włączanie właściwości prostszych klas, bardziej złożone struktury tworzymy więc w sposób przyrostowy.

W klasie pochodnej, wszystkie definicje stałych i typów są łączone z definicjami z klasy nadrzędnej. Podobnie są łączone schematy - te pochodzące z klasy nadrzędnej i te zadeklarowane w klasie pochodnej. W wypadku kolizji nazw, schematy są spajane w jeden (za pomocą koniunkcji schematów), tak samo niezmienniki historii. Dotyczy to również nienazwanych schematów stanu.

Aby zadeklarować klasę nadrzędną, wystarczy włączyć jej nazwę do części deklaracyjnej.

ShadowedFigure

$\upharpoonright(x, y, colour, max\_x, max\_y, bound\_x, bound\_y,$
$shadow\_x, shadow\_y, INIT, Move)$

Figure

$shadow\_x, shadow\_y : \mathbb{R}$

INIT

$shadow\_x = x + 2$
$shadow\_y = y + 2$

Move

$\Delta(shadow\_x, shadow\_y)$
$dx?, dy? : \mathbb{Z}$

$shadow\_x' = shadow\_x + dx?$
$shadow\_y' = shadow\_y + dy?$

Taka klasa jest równoważna następującej wersji rozszerzonej:

ShadowedFigure

$\upharpoonright(x, y, colour, max\_x, max\_y, bound\_x, bound\_y,$
$shadow\_x, shadow\_y, \textsc{Init}, Move)$

$colour : Colour$

$max\_x : \mathbb{Z}$ | $max\_y : \mathbb{Z}$

$max\_x = 1000$ | $max\_y = 1000$

$x, y : \mathbb{R}$
$bound\_x, bound\_y : \mathbb{R}$
$shadow\_x, shadow\_y : \mathbb{R}$

$x > -max\_x \land x < max\_x$
$y > -max\_y \land y < max\_y$
$bound\_x > 0 \land bound\_y > 0$
$x + bound\_x < max\_x \land y + bound\_y < max\_y$
$shadow\_x > x \land shadow\_y > y$

Init

$x = y = 0$
$bound\_x = bound\_y = 0$
$shadow\_x = x + 2$
$shadow\_y = y + 2$

Move

$\Delta(x, y, shadow\_x, shadow\_y)$
$dx?, dy? : \mathbb{Z}$

$x' = x + dx?$
$y' = y + dy?$
$shadow\_x' = shadow\_x + dx?$
$shadow\_y' = shadow\_y + dy?$

Object-Z daje możliwość redefiniowania operacji w klasie pochodnej (z wyłączeniem mechanizmu łączenia schematów), lub nawet usunięcia jej. Do tego służą słowa kluczowe **redef** i **remove**. Przykładowo, definicja klasy nie zawierającej operacji *Move*:

ImmovableFigure

Figure[**remove** *Move*]

Deklaracja klas figur przesuwalnych tylko poziomo:

FigureMovedInX

Figure[**redef** *Move*]

Move

$\Delta(x)$
$dx? : \mathbb{Z}$

$x' = x + dx?$

6.4. Polimorfizm

Poliformizm jest to mechanizm, który pozwala na zadeklarowanie zmiennej, której wartość może być obiektem jednej z wielu możliwych klas. W programowaniu zorientowanym obiektowo użyteczne jest założenie, że zmienna może być instancją danej klasy, lub dowolnej pochodnej tej klasy (dziedziczącej po niej).

W Object-Z, jeśli C jest klasą, to notacja

$c_1 : C$

oznacza obiekt klasy C, w odróżnieniu od

$c_2 : \downarrow C$

gdzie c_2 może być instancją C lub dowolnej klasy pochodnej.

Odwołanie się do operacji $c_2.Op$ może, ale nie musi, oznaczać odwołanie się do operacji Op zdefiniowanej w klasie, której typu jest instancja. Jeżeli definicja Op znajduje się w którejś z klas nadrzędnych, a nie w klasie danej instancji, to oczywiście $c_2.Op$ nawiązuje do tej definicji. Dzięki mechanizmowi dziedziczenia klasy mogą redefiniować operację, dostarczając ich specjalizowane wersje.

6.5. Historia obiektu

Stan obiektu może się zmieniać, poprzez wykonywanie na nim operacji. Jak wiemy, obiekt rozpoczyna swoją egzystencję od stanu początkowego, zgodnego z warunkiem *Init*. Potem mogą być na nim wykonywane operacje.

Powiązanie stanów obiektu st_i z operacjami op_i jest przedstawione następująco:

$$st_1 \xrightarrow{op_1} st_2 \xrightarrow{op_2} \ldots st_{n-1} \xrightarrow{op_{n-1}} st_n$$

Lista par st_i, op_i tworzy historię operacji obiektu. W opisie klasy mogą zawierać się *niezmienniki historii*. W Object-Z dodano operatory wywodzące się z logiki temporalnej, przydatnej do ich opisu.

$\Box P$	P jest spełnione na każdym etapie w historii
$\Diamond P$	jest możliwe, że istnieje etap, w którym zachodzi P
$\bigcirc P$	P jest spełnione na następnym etapie

Oprócz tego

op Nazwa ostatniej operacji, która wystąpiła na obiekcie

Przykładem zastosowania tych operatorów, może być zadeklarowanie klasy figur, które są przesuwalne tylko w prawo.

FigureMovedRight

Figure

$\Box(\mathbf{op} = Move \Rightarrow ((\bigcirc x) > x))$

Rozdział 7

System STOCKPACK - przykładowa specyfikacja

7.1. Założenia systemu

Celem systemu STOCKPACK jest zapewnienie użytkownikom urządzeń przenośnych PDA platformy do obserwacji i monitorowania danych giełdowych, narzędzi analizy technicznej, oraz symulacji portfela inwestycyjnego. Użytkownicy są w stanie pobierać dane giełdowe z Internetu i wyświetlać wykresy spółek wraz z naniesionymi typowymi wskaźnikami (takimi jak RSI, MACD). Oprócz tego aplikacja zawiera implementację portfela, w którym dla poszczególnych spółek użytkownicy mogą zapisywać przeprowadzone transakcje w różnych walutach.

STOCKPACK jest rzeczywistym systemem, zaimplementowanym na platformie PalmOS. W jego skład wchodzi główny program przeznaczony do uruchomienia na urządzeniu PDA, oraz programy pomocnicze, realizujące synchronizację danych z różnych źródeł (Internet, pliki z danymi giełdowymi w formacie Metastock na komputerze desktop).

Przedmiotem niniejszego rozdziału jest główny moduł systemu, czyli aplikacja systemu operacyjnego Palm OS. Jej rzeczywistą implementację tworzy 20.000 linii kodu w języku programowania C.

Autorami oprogramowania są Maciej Mochol i Jarosław Nowisz. System jest dostępny komercyjnie, dystrybutorem jest firma Aeronic ([22]).

7.2. Specyfika programowania aplikacji mobilnych w systemie Palm OS

Zastosowanie notacji $\mathbb{Z}$ do opisu aplikacji mobilnej ma szereg zalet, głównie ze względu wymagań, jakie stawia środowisko systemowe wobec programisty. Spotyka się on bowiem z wieloma trudnościami, które wymagają odpowiedniej metodologii już na etapie projektu. W przeciwieństwie bowiem do wielu systemów, które usprawniają tworzenie, szczególnie bazodanowych aplikacji, niejednokrotnie dostarczając własnej metodologii projektowania odpowiedniej dla zastosowania, tworzenie aplikacji dla Palm OS opiera

się na bardzo dokładnym i szczegółowym programowaniu.

Do najważniejszych trudności należą:

- język C nie jest intuicyjny i może wydawać się niejednoznaczny, wiele jego konstrukcji stwarza niebezpieczeństwo błędnej interpretacji przez programistę i popełnienia błędu,
- system Palm OS wprowadza wprawdzie pewne mechanizmy bezpieczeństwa i ochrony danych, jest to jednak mechanizm nieporównywalny ze współczesnymi systemami operacyjnymi. W szczególności możliwe jest nadpisanie danych, pamięci i kodu innych aplikacji, poprzez błędne operacje na wskaźnikach,
- biblioteka funkcji standardowych dostępnych programiście nie jest zgodna ze standardem C. Wiele ułatwień przyjętych jako standard nie jest zaimplementowanych. Często brak typowych funkcji znanych z systemów stacjonarnych (np. atoi()),
- urządzenie PDA posiada zazwyczaj niewielkie zasoby pamięci operacyjnej, oraz małą moc procesora. Są sytuacje, gdy brak dbałości podczas projektowania i programowania, które w systemach stacjonarnych co najwyżej doprowadziłyby do niepotrzebnego zużycia zasobów, na palmtopie zazwyczaj uniemożliwia poprawne funkcjonowanie aplikacji. Często wręcz niemożliwe jest dokończenie projektu, bez uprzedniej weryfikacji założeń i wprowadzenia niezbędnych poprawek do stworzonego już kodu,
- brak implementacji relacyjnej bazy danych, a nawet systemu plików powoduje, że programista pracuje na poziomie obszarów pamięci i wskaźników, co znacznie zwiększa ryzyko pomyłki i rozbieżności implementacji z założeniami projektowymi.

Dobrym rozwiązaniem jest więc wprowadzenie metod formalnych do fazy projektowej. Oprócz samych korzyści płynących ze strukturalizacji projektu, notacja $\mathbb{Z}$ dostarcza wiele narzędzi pomocnych przy weryfikacji specyfikacji, zmniejszając ryzyko pomyłki projektanta lub programisty.

7.3. Specyfikacja

Niniejsza specyfikacja powstała przy użyciu narzędzia Z/EVES, opisanego już w tej pracy. Ze względu na charakter aplikacji, głównym celem autora było zapewnienie spójności danych po wykonaniu różnych operacji. Wiele błędów, które może popełnić programista wynika z jego pomyłek, a nie na skutek nieprawidłowej interpretacji założeń projektowych, jednak w przypadku aplikacji mobilnej bez formalnego podejścia do fazy projektowej bardzo prawdopodobne są błędy projektanta. Jednym z newralgicznych punktów systemu jest obsługa bazy danych. W wyniku braku na urządzeniach mobilnych standardu relacyjnego dostępu do danych, trudno jest zagwarantować poprawność wykonywanych operacji, gdyż cała praca potrzebna do utrzymania założonej struktury danych musi zostać wykonana po stronie aplikacji.

Notacja $\mathbb{Z}$ jest dobrym narzędziem do modelowania relacji pomiędzy poszczególnymi tabelami. Pojęcia znane z teorii mnogości, rachunek zbiorów i relacji oraz operatory $\mathbb{Z}$ pozwalają w łatwy sposób modelować relacje 1-1, 1-n, itd., znane ze świata baz danych.

W specyfikacji systemu StockPack główny nacisk położono na założenie, że wszystkie operacje na danych w systemie nie zmieniają z góry założonej struktury danych. Z tego powodu skoncentrowano się na weryfikowaniu i dowodzeniu poprawności tych faktów. System Z/EVES, poprzez maszynę dowodzenia, dostarcza narzędzi do weryfikacji relacji głównie poprzez mechanizm sprawdzania dziedzin *(ang. domain check)*. Sprawdzenie dziedziny jest dokonywane podczas weryfikacji paragrafu specyfikacji, w tym celu jest tworzony specjalny teoremat, np. $SESSIONREC\$domainCheck$, który zawiera wszystkie predykaty niezbędne dla poprawności definicji paragrafu i wynikające z zadeklarowanych struktur. Przykładem takiego automatycznie generowanego predykatu jest zadeklarowanie wywołania funkcji:

$$y = funkcja\ x$$

Aby ten zapis był prawidłowy, musi być spełniony warunek $x \in \mathrm{dom}\, funkcja$. Predykat ten jest automatycznie dodawany do teorematu sprawdzającego poprawność dziedzin.

Dzięki takiej konstrukcji weryfikacja i dowodzenie poprawności odbywa się niejako automatycznie, uwalniając użytkownika od żmudnego dowodzenia założeń projektowych poprzez testowe teorematy, co w przypadku większych specyfikacji jest bardzo skomplikowane.

7.3.1. Deklaracje struktur danych

Potrzebna jest deklaracja łańcucha znaków, używanego do przechowywania nazw, identyfikatorów i innych danych. Łańcuch znaków jest sekwencją liczb naturalnych. Niepotrzebne jest dla celów tej specyfikacji komplikowanie tej definicji poprzez wprowadzanie dodatkowych założeń.

$$CHARS == \mathrm{seq}\,\mathbb{N}$$

Drugim podstawowym typem danych są liczby zmiennoprzecinkowe typu FLOAT, które służą do przechowywania danych nt. cen spółek i wartości transakcji. Ponieważ notacja $\mathbb{Z}$ nie zawiera definicji liczb rzeczywistych, możemy przyjąć dla uproszczenia, że typ $FLOAT$ to po prostu liczby całkowite (mają podobne właściwości).

$$FLOAT == \mathbb{Z}$$

Analogiczna definicja daty:

$$DATE == \mathbb{N}$$

i czasu:

$$TIME == \mathbb{N}$$

W wielu przypadkach potrzebny jest aktualny czas i data, np. do zaksięgowania transakcji. Wprowadzamy zatem dwie wartości, *today* i *now*, przyjmujemy, że zawierają one aktualną datę i czas.

$$today : DATE$$

$$now : TIME$$

Notowania spółek mogą pochodzić z różnych źródeł (np. z różnych giełd lub rynków kapitałowych). Zbiór *SOURCE* jest zbiorem możliwych źródeł notowań spółek.

$$[SOURCE]$$

Oprócz tego potrzebne nam jest domyślne źródło. W dalszej części nie wykorzystujemy faktu, że notowania mogą pochodzić z różnych miejsc, można to rozbudować w miarę potrzeb później. Jest to przykład niedeterminizmu $\mathbb{Z}$.

$$default\_source : SOURCE$$

Sesja jest pojęciem rynku kapitałowego, które oznacza dzień, w którym dokonuje się transakcji, ustalając w ten sposób ceny spółek. Zazwyczaj posiada numer identyfikacyjny. W tej specyfikacji jest zamodelowana przy użyciu schematu, który zawiera datę sesji i jej numer. W dalszej części będzie wykorzystywana instancja *SESSIONREC*, która jest zbiorem wiązań identyfikatorów *session_number* i *session_date* z ich wartościami (czyli $\theta SESSIONREC$). Wykorzystywanie zapisu θ nie jest dozwolone w ZRM, ale Z/EVES pozwala na stosowanie specjalnej notacji, która pozwala przypisać zbiór wiązań do instancji schematu. Dzięki takiemu rozwiązaniu możliwe jest wygodne strukturalizowanie danych, poprzez stosowanie podejścia podobnego do projektowania obiektowego.

SESSIONREC

$$session\_date : DATE$$
$$session\_number : \mathbb{N}$$

Schemat *SESSIONS* jest w zasadzie zbiorem poszczególnych sesji. Można by go zdefiniować jako alias typu, ale ze względów strukturalnych wygodniej przedstawić główne definicje danych w postaci schematów (które zawierają informacje o stanach).

$$SESSIONS \mathrel{\widehat=} [\ sessions : \mathbb{P}\, SESSIONREC\ ]$$

Podobnie definiujemy pojęcie waluty, w skład której wchodzi jej cena w stosunku do dolara oraz skrót w standardzie ISO.

CURRENCY

$$iso\_code : CHARS$$
$$usd\_ratio : FLOAT$$

Ponieważ przechowujemy ceny walut relatywnie w stosunku do dolara, definiujemy specjalną instancję waluty, oznaczającej dolar amerykański.

$$usd\_currency : CURRENCY$$

Łączymy wprowadzone pojęcia walut w schemat, wprowadzając predykaty jako niezmienniki. Predykaty te mówią, że zbiór walut przechowywanych przechowywanych systemie nigdy nie jest pusty i zawsze zawiera definicję dolara. Oprócz tego nie mogą się powtarzać elementy z identycznym kodem ISO ? to znaczy gwarantujemy, że kod ISO jest w systemie jednoznaczny i unikalny.

CURRENCYDATA

$$currencies : \mathbb{P}\, CURRENCY$$

$$currencies \neq \emptyset$$
$$usd\_currency \in currencies$$
$$\forall\, x, y : currencies \bullet x.iso\_code = y.iso\_code \Rightarrow x = y$$

Analogicznie definiujemy pojęcie spółki. Jest to rekord, który zawiera informację o walucie, źródle pochodzenia notowań oraz czas i datę ostatniej modyfikacji.

STOCKREC

$$name, symbol : CHARS$$
$$currency : CURRENCY$$
$$source\_id : SOURCE$$
$$mod\_date : DATE$$
$$mod\_time : TIME$$

Schemat *STOCKS* kumuluje dane o wszystkich spółkach w systemie oraz o walutach (poprzez włączenie schematu *CURRENCYDATA*). Jego niezmiennikiem jest predykat, który gwarantuje, że wszystkie kody walut zawarte w rekordach spółki muszą znajdować się również w zmiennej *currencies* schematu *CURRENCYDATA*.

STOCKS

$$CURRENCYDATA$$
$$stock\_ids : \mathbb{P}\, STOCKREC$$

$$\forall\, x : STOCKREC \bullet x \in stock\_ids \Rightarrow x.currency \in currencies$$

Typ *VALTYPE* określa możliwe typy wartości notowania spółki. System STOCKPACK definiuje ceny otwarcia, zamknięcia, maksymalną, minimalną oraz obrót.

$$VALTYPE ::= OPEN \mid CLOSE \mid MAX \mid MIN \mid VOL$$

Notowania mogą być dwojakiego typu: dzienne lub ciągłe (intraday). Notowania dzienne podawane są pod koniec sesji i obliczane są na podstawie jej przebiegu. Notowania zmienne stanowią zapis przebiegu sesji.

$$STOCKTYPE ::= DAILY \mid INTRADAY$$

Typ *VALUES* definiuje jeden punkt wartości. Zdefiniowany jako całkowita funkcja, każdemu typowi wartości przyporządkowuje jedną wartość zmiennoprzecinkową. Razem zbiór definiuje jeden chwilowy stan wartości spółki.

$$VALUES == VALTYPE \rightarrow FLOAT$$

Typ *STOCKDAILYPOINTS* wiąże dane sesji z notowaniami. Niektórym sesjom (funkcja częściowa) przyporządkowane są wartości *daily*. Oznacza to, że w systemie nie dla każdej sesji musi być zdefiniowana jej wartość (może być nieznana albo nie istnieć).

$$STOCKDAILYPOINTS == SESSIONREC \nrightarrow VALUES$$

Definicja zbioru notowań ciągłych *STOCKINTRAPOINTS* jest podobna do poprzedniej, z tym, że jednej sesji przyporządkowuje się zbiór par zawierających czas notowania i wartość. W ten sposób dla jednej sesji może być wiele notowań chwilowych (czyli relacja $1 \mapsto N$ znana z relacyjnych baz danych).

$$STOCKINTRAPOINTS == SESSIONREC \nrightarrow TIME \times VALUES$$

Zdefiniowane struktury, podobnie jak waluty i spółki zamykamy w jeden schemat *STOCKDATA*. Każdy stan tego schematu zawiera informacje o walutach, spółkach i ich wartościach wartościach systemie. Definiujemy przy tym niezmienniki zapewniające spójność danych ? dwa pierwsze predykaty mówią o tym, że wszystkie identyfikatory spółek zawarte w typach *STOCKDAILYPOINTS* i *STOCKINTRAPOINTS* są spójne z danymi schematu *STOCK* (czyli przechowywaną tabelą spółek). Następne dwa predykaty łączą tabelę sesji ze zbiorami punktów ? niemożliwa w ten sposób jest sytuacja, że w zbiorach punktów notowań jest zawarta informacja o jakiejś sesji, o której nie ma informacji w głównej tabeli.

STOCKDATA

$$\begin{array}{l} STOCKS \\ SESSIONS \\ daily\_data : STOCKREC \nrightarrow STOCKDAILYPOINTS \\ intra\_data : STOCKREC \nrightarrow STOCKINTRAPOINTS \\ \hline \mathrm{dom}\ daily\_data \subseteq stock\_ids \\ \mathrm{dom}\ intra\_data \subseteq stock\_ids \\ \forall x : \mathrm{ran}\ daily\_data \bullet \mathrm{dom}\ x \subseteq sessions \\ \forall x : \mathrm{ran}\ intra\_data \bullet \mathrm{dom}\ x \subseteq sessions \end{array}$$

Alarm służy do zawiadomienia użytkownika o sytuacji, w której wartość jakiegoś waloru przekracza pewien określony próg. Może to być sygnał sprzedaży lub kupna akcji.

ALARMTYPE definiuje 5 rodzajów progów: alarm nastąpi wtedy, gdy wartość jest większa, większa lub równa, równa, mniejsza, mniejsza lub równa od wartości progowej.

$$ALARMTYPE ::= MORE \mid MOREEQ \mid EQ \mid LESS \mid LESSEQ$$

Schemat *ALARMREC* zawiera wartość progową, rodzaj progu, oraz wartości waloru służące do określenia stanu alarmu.

ALARMREC

$$trigger\_val : ALARMTYPE \times VALTYPE \times FLOAT$$

Schemat *ALARMDATA* łączy zbiór alarmów i dane o spółkach. Predykat gwarantuje, że nie istnieją alarmy dla nieokreślonych spółek.

ALARMDATA

$$STOCKS$$
$$alarms : STOCKREC \pfun \mathbb{P}\, ALARMREC$$

$$\mathrm{dom}\, alarms \subseteq stock\_ids$$

Transakcja jest podstawowym pojęciem, używanym do zbudowania pojęcia portfela. Założeniem funkcjonalnym programu jest, aby użytkownik miał możliwość archiwizowania swoich transakcji, z których może potem obliczać różne dane statystyczne (takie jak np. ilość zgromadzonego kapitału).

Transakcja może być dwojakiego typu, kupna lub sprzedaży.

$$TRANSTYPE ::= BUY \mid SELL$$

Schemat *TRANSACTIONREC* opisuje jedną transakcję. W jej skład wchodzą informacje o cenie zakupu lub sprzedaży, prowizja od transakcji, zysk i ilość akcji w transakcji. Transakcja przechowuje też dane czasie jej zawarcia.

TRANSACTIONREC

$$trans\_date : DATE$$
$$trans\_price, trans\_fee, profit, buy\_price : FLOAT$$
$$trans\_amount : \mathbb{N}$$
$$mod\_date : DATE$$
$$mod\_time : TIME$$

Schemat *TRANSDATA* wiąże wszystkie transakcje w systemie ze spółkami. Eliminuje istnienie transakcji dla nieokreślonych spółek.

TRANSDATA

$$STOCKS$$
$$transactions : STOCKREC \pfun \mathbb{P}(TRANSTYPE \times TRANSACTIONREC)$$

$$\mathrm{dom}\, transactions \subseteq stock\_ids$$

Na końcu wprowadzamy pojęcie portfolio, które zawiera wszystkie jak dotąd zdefiniowane struktury. W obrębie jednego portfolio są informacje o walutach, spółkach, notowaniach, alarmach i transakcjach.

Schemat *PORTFOLIOREC* zawiera informacje specyficzne dla jednego portfolio, w tym przypadku jego nazwę, którą przypisuje mu użytkownik. STOCKPACK obsługuje wiele portfolii, ale ze względu na rozmiar pracy nie są celowe dalsze definicje. W obrębie jednego portfolio można opisać całą funkcjonalność systemu.

PORTFOLIOREC

$$portfolio\_name : CHARS$$

Schemat *PORTFOLIO* łączy wszystkie rodzaje danych systemu STOCKPACK. Na nim określone są operacje w systemie.

$$PORTFOLIO \mathrel{\widehat=} PORTFOLIOREC \land STOCKDATA \land ALARMDATA \land TRANSDATA$$

7.3.2. Stan początkowy

W podrozdziale tym zdefiniowany jest stan początkowy systemu.

Zbiór walut początkowo zawiera tylko informacje o dolarze.

$$CurrenciesInit \mathrel{\widehat=} [\ CURRENCYDATA' \mid currencies' = \{usd\_currency\}\ ]$$

Zbiory spółek, sesji i punktów są puste.

$$StockDataInit \mathrel{\widehat=} [\ STOCKDATA' \mid stock\_ids' = \emptyset \land sessions' = \emptyset \land daily\_data' = \emptyset \land intra\_data' = \emptyset\ ]$$

Zbiór alarmów jest pusty.

$$AlarmDataInit \mathrel{\widehat=} [\ ALARMDATA' \mid alarms' = \emptyset\ ]$$

W systemie nie istnieją żadne transakcje.

$$TransDataInit \mathrel{\widehat=} [\ TRANSDATA' \mid transactions' = \emptyset\ ]$$

Nazwa portfolio jest ciągiem pustym.

$$PortfolioRecInit \mathrel{\widehat=} [\ PORTFOLIOREC' \mid portfolio\_name' = \langle\rangle\ ]$$

Za pomocą koniunkcji schematów definiujemy całkowitą operację *Init* na schemacie *PORTFOLIO*.

$$PortfolioInit \mathrel{\widehat=} CurrenciesInit \land StockDataInit \land AlarmDataInit \land TransDataInit \land PortfolioRecInit$$

Poniższy teoremat dowodzi, że tak określony stan początkowy nie jest sprzeczny z żadnym niezmiennikiem w systemie, to znaczy że istnieje taki zbiór wiązań $\theta PORTFOLIO$ dla którego możliwy jest taki stan początkowy.

theorem PortfolioInitPossible
$\exists\, PORTFOLIO' \bullet PortfolioInit$

Aby dowieść poprawności tego teorematu, wystarczy skorzystać z automatycznego Dowodzenia przy użyciu redukcji.

proof
prove by reduce
■

7.3.3. Operacje na walutach

Podrozdział ten określa możliwe operacje na danych o walutach w systemie.

Pierwszą operacją jest *AddCurrency*, która na podstawie kodu ISO oraz wartości w stosunku do dolara dodaje do systemu definicję waluty. Korzystamy tutaj z konstrukcji niezgodnej z ZRM, ale zgodnej z Z/EVES i standardem ISO (standard ISO określa ją inną składnią).

Warunkiem wstępnym operacji jest, że nie istnieje taka waluta w systemie, która ma identyczny kod ISO.

AddCurrency

$$\Delta PORTFOLIO$$
$$isocode? : CHARS$$
$$usdratio? : FLOAT$$

$$\neg\, (\exists\, x : CURRENCY \bullet x.iso\_code = isocode?)$$
$$currencies' = currencies \cup \{\theta CURRENCY[iso\_code := \langle\rangle, usd\_ratio := usdratio?]\}$$

Operacja *FindCurrency* nie zmienia stanu systemu. Jej zadaniem jest znalezienie waluty w systemie, której kod ISO jest zgodny z zadanym.

Warunkiem wstępnym operacji jest to, że istnieje dokładnie jedna taka waluta w systemie.

FindCurrency

$$\Xi PORTFOLIO$$
$$isocode? : CHARS$$
$$out! : CURRENCY$$

$$\exists_1\, x : currencies \bullet x.iso\_code = isocode?$$
$$out! = (\mu\, x : currencies \mid x.iso\_code = isocode?)$$

Dla udowodnienia poprawności tej definicji, musimy użyć dowodu przez redukcję.

proof

prove by reduce

■

Na podstawie podanego zbioru wiązań schematu *CURRENCY*, operacja *GetRatio* zwraca wartość waluty.

$$GetRatio \mathrel{\widehat=} [\ currency? : CURRENCY;\ ratio! : FLOAT \mid ratio! = currency?.usd\_ratio\ ]$$

Operacja *GetCurrencyRatio* jest potokiem, który tworzy schemat operacji akceptującej kod ISO i zwracającej wartość waluty w stosunku do dolara.

$$GetCurrencyRatio \mathrel{\widehat=} FindCurrency \gg GetRatio[\ out!/currency?\ ]$$

Operacja *DeleteCurrencyBindings* zmienia stan systemu w ten sposób, że usuwana jest waluta o podanym zbiorze wiązań, pod warunkiem, że żadna spółka nie jest związana z tą walutą. Jeśli istnieje spółka, która ma przypisaną daną walutę, to aby ją usunąć, trzeba również usunąć spółkę. W ten sposób zachowane są niezmienniki struktur.

Warunkiem wstępnym jest też występowanie instancji podanej waluty w systemie.

DeleteCurrencyBindings

$$\Delta PORTFOLIO$$
$$currency? : CURRENCY$$

$$currency? \in currencies$$
$$\forall x : stock\_ids \bullet x.currency \neq currency?$$
$$currencies' = currencies \setminus \{currency?\}$$

Operacja *DeleteCurrency* realizuje to samo co poprzednia, ale wymaga podania tylko kodu ISO.

$$DeleteCurrency \mathrel{\widehat=} FindCurrency \gg DeleteCurrencyBindings[\ out!/currency?\ ]$$

7.3.4. Operacje na punktach danych dla spółek

Zbiory punktów danych przechowują informację o notowaniach.

Mając daną spółkę i sesję, możemy dodać do systemu informację o notowaniach dziennych. Warunkiem wstępnym jest, że spółka *stock*? musi istnieć w systemie.

Aby zdefiniować taką operację, musimy rozpatrzyć dwa przypadki. Pierwszy, że informacja o punktach danych z podanej spółki, na określonej sesji nie została jeszcze wprowadzona do systemu. Możemy wtedy użyć operatora przeciążenia $\oplus$.

Jeśli w systemie nie istnieje informacja o podanej sesji, to jest ona wprowadzana do systemu.

AddDailyData

$\Delta PORTFOLIO$
$stock? : STOCKREC$
$session? : SESSIONREC$
$vals? : VALUES$

$stock? \in stock\_ids$
$stock? \notin \mathrm{dom}\ daily\_data$
$sessions' = sessions \cup \{session?\}$
$daily\_data' = daily\_data \oplus \{(stock? \mapsto \{(session? \mapsto vals?)\})\}$

Drugi przypadek: informacja była już wprowadzona, nie możemy więc użyć bezpośrednio operatora przeciążenia, gdyż skasowalibyśmy stare dane. Łączymy więc stare i nowe dane w jeden zbiór, korzystając z funkcji częściowej *daily_data*.

Warunek wstępny ? że istnieją punkty danych dla podanej sesji i spółki ? jest jednocześnie warunkiem poprawności użycia funkcji *daily_data*, gwarantuje więc poprawność całej definicji.

UpdateDailyData

$\Delta PORTFOLIO$
$stock? : STOCKREC$
$session? : SESSIONREC$
$vals? : VALUES$

$stock? \in stock\_ids$
$stock? \in \mathrm{dom}\ daily\_data$
$sessions' = sessions \cup \{session?\}$
$daily\_data' = daily\_data \oplus$
$\quad \{(stock? \mapsto daily\_datastock? \cup \{(session? \mapsto vals?)\})\}$

Skrypt dowodu:

proof

rewrite

■

Łączymy oba przypadki i otrzymujemy definicję dodania punktu danych *daily* do systemu.

$AddDailyPoint \mathrel{\widehat=} AddDailyData \lor UpdateDailyData$

Analogiczna konstrukcja została wykorzystana w definicji dla danych ciągłych.

AddIntradayData

$\Delta PORTFOLIO$
$stock? : STOCKREC$
$session? : SESSIONREC$
$time? : TIME$
$vals? : VALUES$

$stock? \in stock\_ids$
$stock? \notin \mathrm{dom}\, intra\_data$
$sessions' = sessions \cup \{session?\}$
$intra\_data' = intra\_data \oplus$
$\quad \{(stock? \mapsto \{(session? \mapsto (time? \mapsto vals?))\})\}$

UpdateIntradayData

$\Delta PORTFOLIO$
$stock? : STOCKREC$
$session? : SESSIONREC$
$time? : TIME$
$vals? : VALUES$

$stock? \in stock\_ids$
$stock? \in \mathrm{dom}\, intra\_data$
$sessions' = sessions \cup \{session?\}$
$intra\_data' = intra\_data \oplus$
$\quad \{(stock? \mapsto intra\_data\ stock? \cup \{(session? \mapsto (time? \mapsto vals?))\})\}$

Skrypt dowodu:

proof
rewrite
■

Operacja dodania punktu notowań ciągłych.

$AddIntradayPoint \mathrel{\widehat=} AddIntradayData \lor UpdateIntradayData$

Operacja skasowania wszystkich notowań dla podanej spółki:

DeletePointsForStock

$\Delta PORTFOLIO$
$stock? : STOCKREC$

$daily\_data' = \{stock?\} \lhd daily\_data$
$intra\_data' = \{stock?\} \lhd intra\_data$

Operacja *GetDailyValues* zwraca pojedynczy punkt danych dziennych na podstawie podanej spółki I sesji. Jej warunkiem wstępnym jest istnienie takiej spółki wraz z informacją o podanej sesji.

GetDailyValues

$$\Xi PORTFOLIO$$
$$stock? : STOCKREC$$
$$session? : SESSIONREC$$
$$vals! : VALUES$$

$$stock? \in \mathrm{dom}\ daily\_data$$
$$session? \in \mathrm{dom}(daily\_data\ stock?)$$
$$daily\_data\ stock?\ session? \in VALUES$$
$$vals! = (\mu\, x : VALUES \mid x = daily\_data\ stock?\ session?)$$

Skrypt dowodu:

proof

rewrite

■

Operacja *FindSessions* zwraca instancję schematu *SESSIONREC* na podstawie daty. Jej warunkiem wstępnym jest prawidłowość tej daty (musi istnieć informacja o sesji w danym dniu).

Operacja ta nie zmienia stanu systemu.

FindSession

$$\Xi PORTFOLIO$$
$$date? : DATE$$
$$out! : SESSIONREC$$

$$\exists_1 x : sessions \bullet x.session\_date = date?$$
$$out! = (\mu\, x : sessions \mid x.session\_date = date?)$$

Dowód:

proof

rewrite

■

Operacja *GetDailyValuesByDate* za pomocą operatora potoku $\gg$ definiuje pobranie punktu danych mając daną datę.

$$GetDailyValuesByDate \mathrel{\widehat=} FindSession \gg GetDailyValues[out!/session?]$$

7.3.5. Operacje na alarmach

Aby określić na podstawie danych alarmu i notowań, czy dany alarm powinien być aktywny czy nie, potrzebujemy jakiejś funkcji. W tym celu definiujemy relację *active_alarm_values*. Ta relacja powinna być właściwie funkcją całkowitą, ale wtedy dowód w Z/EVES jest o wiele trudniejszy.

W specyfikacji nie jest określony sposób implementacji tej funkcji, pozostawione jest to programiście.

$$active\_alarm\_values : \mathbb{P}(ALARMREC \times VALUES)$$

W taki sam sposób jak wprowadzanie notowań definiujemy dodawanie nowych alarmów do systemu.

AddAlarmData

$$\Delta PORTFOLIO$$
$$alarm? : ALARMREC$$
$$stock? : STOCKREC$$

$$stock? \in stock\_ids$$
$$stock? \notin \mathrm{dom}\ alarms$$
$$alarms' = alarms \oplus \{(stock? \mapsto \{alarm?\})\}$$

UpdateAlarmData

$$\Delta PORTFOLIO$$
$$alarm? : ALARMREC$$
$$stock? : STOCKREC$$

$$stock? \in stock\_ids$$
$$stock? \in \mathrm{dom}\ alarms$$
$$alarms' = alarms \oplus \{(stock? \mapsto alarms\ stock? \cup \{alarm?\})\}$$

Dowód:

proof

prove by reduce

■

Warunkiem wstępnym operacji łącznej jest występowanie informacji o podanej spółce.

$$AddAlarm \mathrel{\widehat=} AddAlarmData \lor UpdateAlarmData$$

Operacja *IsStockActiveForVals*, na podstawie danej spółki i notowań pozwala na określenie, czy powinien być dla niej aktywny alarm (w programie STOCKPACK zaimplementowano to występowaniem ikonki dzwoneczka obok jej nazwy).

IsStockActiveForVals

$$\Xi PORTFOLIO$$
$$stock? : STOCKREC$$
$$vals? : VALUES$$

$$\exists\, arec : ALARMREC \bullet$$
$$stock? \in \mathrm{dom}\, alarms \land arec \in alarms\ stock?$$
$$\Rightarrow arec \mapsto vals? \in active\_alarm\_values$$

Dowód:

proof

rewrite

■

Operacja potoku *IsStockActive* eliminuje potrzebę podawania danych *VALUES*. Stan alarmu dla danej spółki otrzymujemy podając jej instancję oraz datę.

$$IsStockActive \mathrel{\widehat=} GetDailyValues \gg IsStockActiveForVals[vals!/vals?]$$

Operacja *DeleteAlarmsForStock* usuwa wszystkie alarmy dla podanej spółki.

DeleteAlarmsForStock

$$\Delta PORTFOLIO$$
$$stock? : STOCKREC$$

$$alarms' = \{stock?\} \lhd alarms$$

7.3.6. Operacje na transakcjach

Podrozdział ten specyfikuje funkcjonalność portfela STOCKPACK.

Pierwsza operacja znakuje transakcję stemplem czasowym. Jest określona na schemacie *TRANSACTIONREC*.

MarkTransactionTimestamp

$$\Delta TRANSACTIONREC$$

$$mod\_date' = today$$
$$mod\_time' = now$$

DoTransaction jest operacją ogólnego przeznaczenia, oznaczającą wykonanie jakiejś transakcji. Dzięki zastosowaniu słów kluczowych **if thenelse** wyeliminowana jest konieczność stosowania dwóch przypadków, tak jak w poprzednich paragrafach. Wykonana transakcja dodawana jest do systemowego rejestru transakcji.

DoTransaction

$\Delta PORTFOLIO$
$stock? : STOCKREC$
$record? : TRANSACTIONREC$
$type? : TRANSTYPE$
$merged : \mathbb{P}(TRANSTYPE \times TRANSACTIONREC)$

---

if $stock? \in \operatorname{dom} transactions$
then $merged = transactions\ stock? \cup \{(type? \mapsto record?)\}$
else $merged = \{type? \mapsto record?\}$
$transactions' = transactions \oplus \{(stock? \mapsto merged)\}$

Dowód:

proof

rewrite

■

Definicja transakcji kupna wykorzystuje operator złożenia oraz podstawienie:

$$Buy \mathrel{\widehat=} MarkTransactionTimestamp \fatsemi DoTransaction[type? := BUY]$$

Analogicznie transakcja sprzedaży:

$$Sell \mathrel{\widehat=} MarkTransactionTimestamp \fatsemi DoTransaction[type? := SELL]$$

Operacja usuwania z rejestru transakcji wszystkich należących do danej spółki.

DeleteTransactionsForStock

$\Delta PORTFOLIO$
$stock? : STOCKREC$

$transactions' = \{stock?\} \lhd transactions$

7.3.7. Operacje na spółkach

Po zdefiniowaniu wszystkich ważniejszych operacji składowych pora na operacje, które pozwalają na modyfikację danych z punktu widzenia jednej spółki.

Operacja *AddStock* za pomocą operatora θ dodaje do systemu zbiór wiązań odpowiadający nowej spółce.

AddStock

$\Delta PORTFOLIO$
$myname?, mysymbol? : CHARS$

$stock\_ids' = stock\_ids \cup$
$\quad \{STOCKREC[name := myname?, symbol := mysymbol?,$
$\qquad currency := usd\_currency, source\_id := default\_source,$
$\qquad mod\_date := 0, mod\_time := 0]\}$

Usunięcie definicji spółki z tabeli:

DeleteStockData

$$\Delta PORTFOLIO$$
$$stock? : STOCKREC$$

$$stock\_ids' = stock\_ids \setminus \{stock?\}$$

Operacja *DeleteStock* kasuje wszystkie elementy w systemie związane z daną spółką. Wykluczona jest więc sytuacja, w której w systemie znajdują się jakieś dane dla nieokreślonej spółki, co spełnia warunki odpowiednich niezmienników systemowych.

$$\begin{aligned} &DeleteStock \mathrel{\widehat=} \\ &\quad DeleteStockData \\ &\quad \land\ DeletePointsForStock \\ &\quad \land\ DeleteAlarmsForStock \\ &\quad \land\ DeleteTransactionsForStock \end{aligned}$$

7.3.8. Operacje na portfolio

Ponieważ w niniejszej specyfikacji pojęcie portfolio nie zostało zbytnio przybliżone, a użyte jedynie w celu połączenia ze sobą wszystkich elementów systemu, określona jest tylko jedna operacja zmieniająca nazwę portfolio:

SetStockName

$$\Delta PORTFOLIO$$
$$newname? : CHARS$$

$$portfolio\_name' = newname?$$

7.4. Wnioski końcowe

W systemie STOCKPACK zaimplementowano część funkcjonalności, która nie jest opisana specyfikacją w tym rozdziale, ale ze względów objętościowych przytoczone zostały najważniejsze części oddające ideę wspomagania projektowania metodami formalnymi. Specyfikacja ta formalizuje dostęp do danych, dowodząc przydatności notacji $\mathbb{Z}$ i systemu Z/EVES w zastosowaniach baz danych.

Rozdział 8

Zakończenie

Notacja $\mathbb{Z}$ jest jednym z najbardziej rozbudowanych i uniwersalnych systemów formalnego opisu oprogramowania. Jej stosowanie może stwarzać wiele korzyści, zarówno dla projektantów, jak i programistów. Niestety, jak w wielu innych przypadkach, uniwersalność jest osiągnięta kosztem mniejszej użyteczności dla specyficznych zastosowań. W chwili obecnej, pomimo faktu, że prace nad $\mathbb{Z}$ są intensywnie prowadzone, wydaje się, że nie jest to metoda, która ma szansę sukcesu w komercyjnych przedsięwzięciach. Przede wszystkim, wbrew zapewnieniom autorów używanie notacji jest trudne. Oprócz znajomości samego rozbudowanego aparatu matematycznego, zawierającego logikę, rachunek predykatów, teorię mnogości i wiele elementów z innych dziedzin, projektant musi się często wykazać zdolnością przeprowadzania dowodów, umieć przełożyć pojęcia czasem wydające się oczywistymi (np. liczba rzeczywista) na język notacji, a także posiadać duże doświadczenie. Niestety umiejętności takie o wiele trudniej jest zdobyć i opanować, aniżeli nauczyć się jednego z dostępnych na rynku narzędzi, zazwyczaj przedstawiających zawiłe problemy, np. z tematyki baz danych, w formie graficznej i diagramów, a nie ścisłych symboli matematycznych.

Z ekonomicznego punktu widzenia, pomimo potencjalnych korzyści finansowych, wynikających ze zmniejszenia nakładów na testowanie i usuwanie poprawek, konieczność zatrudnienia specjalistów posiadających wiedzę nt. metod formalnych zwiększa budżet projektu. Zwłaszcza, że notację i formalne zapisy muszą również rozumieć programiści, którzy na podstawie specyfikacji implementują system. Można więc zaryzykować twierdzenie, że zespół projektowy, stosujący metody formalne takie jak $\mathbb{Z}$, musi składać się ze specjalistów wysokiej klasy i z dużym doświadczeniem, co nie jest koniecznością dla każdego projektu informatycznego.

Innym problemem jest niewielka baza doświadczeń, z jakiej może korzystać projektant. Jest wiele zbadanych i opisanych zagadnień, zaliczających się do kanonu współczesnej informatyki, których znajomość jest niezbędna do tworzenia oprogramowania będącego w stanie sprostać przyjętym standardom. Dotyczy to zarówno algorytmów i struktur danych (pojęcia jak złożoność, sortowanie, wyszukiwanie), jak i kryptografii, baz danych, budowy języków, lub przetwarzania równoległego. Często jednakże zagadnienia te nie są opisane za pomocą pojęć znanych z teorii mnogości lub rachunku predykatów. Powoduje to, że opisanie ich za pomocą $\mathbb{Z}$ jest niejednokrotnie bardzo trudne.

Wydaje mi się, że stanowi to barierę w rozwoju metod formalnych. Dopóki nie

zostanie stworzona baza doświadczeń, jednoznacznie opisująca choćby te przykładowe pojęcia, dopóty projektanci i programiści będą wybierać prostsze narzędzia, ale przeznaczone do konkretnych zastosowań. Optymistyczny jest fakt, że zostało wykonane wystarczająco dużo pracy, żeby metody formalne przekroczyły próg labolatoriów i uczelni. Ich zastosowanie w konkretnych projektach jest możliwe już dzisiaj. Sądzę, że w miarę upływu czasu sytuacja się poprawi, będzie to jednak proces długotrwały.

Dodatek A

Zestawienie stosowanych operatorów $\mathbb{Z}$

A.1. Język matematyczny

A.1.1. Zbiory

Zestawienie operatorów

Symbol	Znaczenie
$\emptyset$	Zbiór pusty: zbiór, który nie zawiera żadnego elementu
$x \notin S$	Niezawieranie się: x nie jest elementem S
$S \subseteq T$	Podzbiór: wszystkie elementy z S należą do T
$S \subset T$	Podzbiór właściwy: S jest podzbiorem T i S nie jest równy T
$S \cup T$	Suma zbiorów: zbiór elementów zawierających się w S lub w T
$S \cap T$	Iloczyn zbiorów: zbiór elementów zawierających się zarówno w S jak i w T
$S \setminus T$	Różnica zbiorów: zbiór elementów S, które nie zawierają się w T
$\mathbb{P} X$	Potęga zbioru: zbiór wszystkich podzbiorów zbioru X

Definicje

Oto przykład formalnej definicji tych pojęć w sposób znany z podręczników matematyki. Do wyspecyfikowania podanych operatorów użyty został język schematów notacji Z, który zostanie przedstawiony w następnym rozdziale.

$$\emptyset[X] == \{\ x : X \mid \mathit{false}\ \}$$

$$
\begin{array}{|l}
\hline
[X] \\
\_ \subseteq \_, \_ \subset \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X \\
\_ \cup \_, \_ \cap \_, \_ \setminus \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X \\
\hline
\forall\, x : X;\ y : Y;\ S, T : \mathbb{P}\, X \bullet \\
\quad x \notin S \Leftrightarrow \neg\, (x \in X) \land \\
\quad (S \subseteq T \Leftrightarrow (\forall\, x : X \bullet x \in S \Rightarrow x \in T)) \land \\
\quad (S \subset T \Leftrightarrow S \subseteq T \land S \neq T) \land \\
\quad S \cup T = \{\ x : X \mid x \in S \lor X \in T\ \} \land \\
\quad S \cap T = \{\ x : X \mid x \in S \land X \in T\ \} \land \\
\quad S \setminus T = \{\ x : X \mid x \in S \land X \notin T\ \} \\
\hline
\end{array}
$$

A.1.2. Pary i relacje binarne I

Zestawienie operatorów

Symbol	Znaczenie
(x, y)	Para x,y
$x \mapsto y$	Wskazywanie: x wskazuje y, to samo co (x, y)
first p	Pierwszy element pary p
second p	Drugi element pary p
$\mathrm{id}\, X$	Relacja identyczności
$X \leftrightarrow Y$	Relacja binarna
$\mathrm{dom}\, R$	Dziedzina: zbiór pierwszych elementów wszystkich par z R
$\mathrm{ran}\, R$	Zbiór wartości: zbiór drugich elementów wszystkich par z R

Definicje

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$
$$\mathrm{id}\, X == \{x : X \bullet x \mapsto x\}$$

$$
\begin{array}{|l}
\hline
[X, Y] \\
\mathit{first} : X \times Y \to X \\
\mathit{second} : X \times Y \to Y \\
\_ \mapsto \_ : X \times Y \to X \times Y \\
\mathrm{dom} : (X \leftrightarrow Y) \to \mathbb{P}\, X \\
\mathrm{ran} : (X \leftrightarrow Y) \to \mathbb{P}\, Y \\
\hline
\forall\, x : X;\ y : Y;\ R : X \leftrightarrow Y \bullet \\
\quad \mathit{first}(x, y) = x \land \\
\quad \mathit{second}(x, y) = y \land \\
\quad x \mapsto y = (x, y) \land \\
\quad \mathrm{dom}\, R = \{\ x : X;\ y : Y \mid x\ \underline{R}\ y \bullet x\ \} \land \\
\quad \mathrm{ran}\, R = \{\ x : X;\ y : Y \mid x\ \underline{R}\ y \bullet y\ \} \\
\hline
\end{array}
$$

A.1.3. Pary i relacje binarne II

Zestawienie operatorów

Symbol	Znaczenie
$S \lhd R$	Wszystkie pary z R których pierwszy element należy do S
$R \rhd T$	Wszystkie pary z R których drugi element należy do T
$S \lhd\!\!\!\cdot\, R$	Wszystkie pary z R których pierwszy element nie należy do S
$R \rhd\!\!\!\cdot\, T$	Wszystkie pary z R których drugi element nie należy do T
$R^{\sim}$	Relacyjna odwrotność: pary z R, w których zamieniony jest pierwszy i drugi element
$R(\!\vert\ S\ \vert\!)$	Relacyjny obraz: drugie elementy par z R których pierwszy element należy do S.
$R \oplus Q$	Przeciążanie: wszystkie pary z R lub z Q, oprócz par z R których pierwszy element należy również do Q.
R^{+}	Przechodnie domknięcie R

Definicje

$[X, Y]$

$$
\begin{array}{l}
\_ \lhd \_, \_ \lhd\!\!\!\cdot \_ : \mathbb{P}\, X \times (X \leftrightarrow Y) \to X \leftrightarrow Y \\
\_ \rhd \_, \_ \rhd\!\!\!\cdot \_ : (X \leftrightarrow Y) \times \mathbb{P}\, Y \to X \leftrightarrow Y \\
\_^{\sim} : (X \leftrightarrow Y) \to (Y \leftrightarrow X) \\
\_(\!| \_ |\!) : (X \leftrightarrow Y) \times \mathbb{P}\, X \to \mathbb{P}\, Y \\
\_ \oplus \_ : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \to (X \leftrightarrow Y) \\
\_^{+} : (X \leftrightarrow X) \to (X \leftrightarrow X)
\end{array}
$$

$$
\begin{array}{l}
\forall\, x : X;\ y : Y;\ S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y;\ Q, R : X \leftrightarrow Y \bullet \\
\qquad S \lhd R = \{\ x : X;\ y : Y \mid x \in S \land x\ \underline{R}\ y \bullet x \mapsto y\ \} \land \\
\qquad S \lhd\!\!\!\cdot R = \{\ x : X;\ y : Y \mid x \notin S \land x\ \underline{R}\ y \bullet x \mapsto y\ \} \land \\
\qquad R \rhd T = \{\ x : X;\ y : Y \mid x\ \underline{R}\ y \land y \in T \bullet x \mapsto y\ \} \land \\
\qquad R \rhd\!\!\!\cdot T = \{\ x : X;\ y : Y \mid x\ \underline{R}\ y \land y \notin T \bullet x \mapsto y\ \} \land \\
\qquad R^{\sim} = \{\ x : X;\ y : Y \mid x\ \underline{R}\ y \bullet y \mapsto x\ \} \land \\
\qquad R(\!|\ S\ |\!) = \{\ x : X;\ y : Y \mid x \in S \land x\ \underline{R}\ y \bullet y\ \} \land \\
\qquad Q \oplus R = ((\mathrm{dom}\, R) \lhd\!\!\!\cdot Q) \cup R \\
\qquad \ldots \text{predykat dla } \_^{+} \text{ pominięty} \ldots
\end{array}
$$

A.1.4. Pary i relacje binarne III

Zestawienie operatorów

Symbol	Znaczenie
$Q \fatsemi R$	Złożenie relacji: Q złożona z R
$R \circ Q$	To samo co $Q \fatsemi R$

Definicje

$[X, Y, Z]$

$$
\begin{array}{l}
\_ \fatsemi \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \to (X \leftrightarrow Z) \\
\_ \circ \_ : (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \to (X \leftrightarrow Z)
\end{array}
$$

$$
\begin{array}{l}
\forall\, Q : X \leftrightarrow Y;\ R : Y \leftrightarrow Z \bullet \\
\qquad Q \fatsemi R = R \circ Q = \{\ x : X;\ y : Y;\ z : Z \mid x\ \underline{Q}\ y \land y\ \underline{R}\ z \bullet x \mapsto z\ \}
\end{array}
$$

A.1.5. Liczby i arytmetyka

Zestawienie operatorów

Symbol	Znaczenie
$\mathbb{Z}$	Zbiór liczb całkowitych
$\mathbb{N}$	Zbiór liczb naturalnych wraz z zerem
$\mathbb{N}_1$	Zbiór liczb naturalnych dodatnich (bez zera)
$+, -, *$	Operatory arytmetyczne: suma, różnica, mnożenie
$\mathsf{div}, \mathsf{mod}$	Operatory arytmetyczne: dzielenie całkowite i reszta z dzielenia
$<, \leq$	Porównanie: mniejsze, mniejsze lub równe
$>, \geq$	Porównanie: większe, większe lub równe
$i \mathinner{..} j$	Zakres: zbiór liczb całkowitych od i do j
$min\ S$	Minimum: największy element S (jeśli istnieje)
$max\ S$	Maksimum: największy element S (jeśli istnieje)
$\#S$	Liczność: liczba elementów należących do zbioru S
$\mathbb{P}_1$	Zbiory niepuste
$\mathbb{F}$	Zbiory skończone

Definicje

$$[\mathbb{Z}]$$

$$\mathbb{N} == \{\ n : \mathbb{Z} \mid n \geq 0\ \}$$

$$\mathbb{N}_1 == \mathbb{N} \setminus \{0\}$$

$$\mathbb{P}_1\, X == \{\ S : \mathbb{P}\, X \mid S \neq \emptyset\ \}$$

$$\mathbb{F}\, X == \{\ S : \mathbb{P}\, X \mid \ldots S \text{ jest skończony } \ldots\ \}$$

$$\begin{array}{l} \_ + \_, \_ - \_, \_ * \_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ \_\ \mathsf{div}\ \_, \_\ \mathsf{mod}\ \_ : \mathbb{Z} \to (\mathbb{Z} \setminus \{0\}) \to \mathbb{Z} \\ \_ - : \mathbb{Z} \to \mathbb{Z} \\ \_ < \_, \_ \leq \_, \_ \geq \_, \_ > \_ : \mathbb{Z} \leftrightarrow \mathbb{Z} \\ \_ \mathinner{..} \_ : \mathbb{Z} \leftrightarrow \mathbb{Z} \to \mathbb{P}\, \mathbb{Z} \\ \# : \mathbb{F}\, X \to \mathbb{N} \\ min, max : \mathbb{P}_1\, \mathbb{Z} \nrightarrow \mathbb{Z} \\ \hline \forall\, a, b : \mathbb{Z} \bullet a \mathinner{..} b = \{\ i : \mathbb{Z} \mid a \leq i \leq b\ \} \\ \qquad \ldots \text{predykaty dla innych operatorów pominięte } \ldots \end{array}$$

A.1.6. Funkcje

Zestawienie operatorów

Symbol	Znaczenie
$X \pfun Y$	Funkcja częściowa: niektóre elementy zbioru X są w parze z elementem z Y
$X \fun Y$	Funkcja zupełna: wszystkie elementy zbioru X są w parze z elementem z Y
$X \pinj Y$	Częściowa injekcja: niektóre elementy zbioru X są w parze z innym elementem z Y
$X \inj Y$	Zupełna injekcja: wszystkie elementy zbioru X są w parze z innym elementem z Y
$X \bij Y$	Bijekcja: każdy element zbioru X jest w parze z innym elementem z Y, pokrywając cały zbiór Y

Definicje

$$X \pfun Y == \{\, f : X \rel Y \mid \text{Każdy element z } X \text{ występuje nie więcej niż raz.} \,\}$$
$$X \fun Y == \{\, f : X \pfun Y \mid \dom f = X \,\}$$
$$X \pinj Y == \{\, f : X \pfun Y \mid \text{Każdy element z } X \text{ jest w parze z innym elementem z } Y. \,\}$$
$$X \inj Y == (X \pinj Y) \cap (X \fun Y)$$
$$X \bij Y == \ldots \text{definicja pominięta} \ldots$$

A.1.7. Ciągi (sekwencje)

Zestawienie operatorów

Symbol	Znaczenie
$\seq X$	Ciąg: zbiór wszystkich ciągów z X
$\seq_1 X$	Niepusty ciąg: niepusty zbiór wszystkich ciągów z X
$\iseq X$	Ciąg pokrywający: zbiór wszystkich ciągów z X w których każdy element z X występuje tylko raz
$s \cat t$	Konkatenacja: ciąg s z dołączonym ciągiem t
$head\ s$	Pierwszy element ciągu s
$last\ s$	Ostatni element ciągu s
$front\ s$	Ciąg s bez ostatniego elementu
$tail\ s$	Ciąg s bez pierwszego elementu
$s \mathsf{\ in\ } t$	Relacja zawierania: ciąg s jest podciągiem ciągu t

Definicje

$$\seq X == \{\, f : \mathbb{N} \pfun X \mid \dom f = 1 \upto \# f \,\}$$
$$\seq_1 X == \{\, f : \seq X \mid \# f > 0 \,\}$$
$$\iseq X == \seq X \cap (\mathbb{N} \pinj X)$$

$[X]$

$head, last : \operatorname{seq}_1 X \to X$

$tail, front : \operatorname{seq}_1 X \to \operatorname{seq} X$

$\_ \frown \_ : \operatorname{seq} X \times \operatorname{seq} X \to seqX$

$\_ \mathsf{in} \_ : \operatorname{seq} X \leftrightarrow \operatorname{seq} X$

$\forall s : \operatorname{seq}_1 X;\ u, v : \operatorname{seq} X \bullet$

$head(s) = s(1) \land$

$last(s) = s(\#s) \land$

$u \frown v = u \cup \{\ i : \operatorname{dom} v \bullet i + \#u \mapsto v(i)\ \}$

$u \mathsf{in}\ v \Leftrightarrow (\exists s, t : \operatorname{seq} X \bullet s \frown u \frown t = v)$

... predykaty dla pozostałych operatorów pominięte ...

A.2. Język schematyczny notacji

A.2.1. Notacja schematów

Definicja schematu

S

d

---

p

Definicja aksjomatu

d

---

p

Definicja uogólniona

$[a, \ldots]$

d

---

p

A.2.2. Rachunek schematów

Symbol	Znaczenie
$S \mathrel{\widehat=} [\ X\ ]$	Schemat poziomy
$[\ T; \ldots \mid \ldots\ ]$	Dołączanie referencji schematu
$z.a$	Zaznaczenie komponentu schematu (mając dane $z : S$)
θS	Sygnatura
$\neg\ S$	Negacja schematu
$S \wedge T$	Koniunkcja schematów
$S \vee T$	Alternatywa schematów
$S \setminus (x_1, \ldots, x_n)$	Ukrywanie
$S \fatsemi T$	Złożenie schematów
$S \gg T$	Operator potoku
$S \upharpoonright T$	Projekcja

A.2.3. Systemy sekwencyjne

Symbol	Znaczenie
$a?$	Parametr operacji
$a!$	Rezultat operacji
a	Komponent stanu przed operacją
a'	Komponent stanu po operacji
S	Stan schematu przed operacją
S'	Stan schematu po operacji
ΔS	Zmiana stanu
ΞS	Niezmienność stanu

Bibliografia

[1] J. M. Spivey. *The Z Notation: A Reference Manual.*
Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[2] ISO/IEC 13568:2002 *Information technology - Z formal specification notation, Syntax, type system and semantic.*
International standard.

[3] ISO. *Z notation.* Technical report, ISO/IEC JTC1/SC22 N1970, 1995.
ISO CD 13568; Committee Draft of the proposed Z Standard.

[4] Irwin Meisels and Mark Saaltink. *The Z/EVES 2.0 Reference Manual.*
Technical Report TR-99-5493-03e, ORA Canada, October 1999.

[5] Mark Saaltink. *The Z/EVES 2.0 Mathematical Toolkit.*
Technical Report TR-99-5493-05b, ORA Canada, October 1999.

[6] Mark Saaltink. *The Z/EVES 2.0 User's Guide.*
Technical Report TR-99-5493-06, ORA Canada, October 1999.

[7] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.*
Prentice Hall International Series in Computer Science, 1996.

[8] R.D.Arthan. *On free type definitions in Z.*
http://lemma-one.com/files/18.ps.

[9] R.D.Arthan. *Arithmetic in Z.*
ftp://ftp.comlab.ox.ac.uk/pub/Zforum/ZSTAN/papers/z-188.ps

[10] Jonathan P. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach.*
International Thomson Computer Press (ITCP), 1996.

[11] Jonathan Jacky. *Z examples.*
http://www.randonc.washington.edu/prostaff/jon/

[12] Houston I. and King S. *CICS Project Report: Experiences and Results from the Use of Z in IBM.*
In Prehn S. and Toetenel W.J. (eds.), *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computing Science*, pages 588-596.
Springer-Verlag, October 1991.

[13] Jonathan Jacky. *Specifying a Safety-Critical Control System in Z.*
In Woodcock J.C.P. and Larsen P.G. (eds.), *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computing Science*, pages 388-402.
Springer-Verlag, April 1993. Industrial Usage Report.

[14] Boswell T. *Specification and Validation of a Security Policy Model.*
In Woodcock J.C.P. and Larsen P.G. (eds.), *FME'93: Industrial-Strength Formal Methods*, volume 680 of *Lecture Notes in Computing Science*, pages 42-51.
Springer-Verlag, April 1993. Industrial Usage Report.

[15] King T. *Formalizing British Rail's Signalling Rules.*
In Naftalin M., Denvir T. and Bertran M. (eds.), *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computing Science*, pages 45-54.
Springer-Verlag, October 1994. Industrial Usage Report.

[16] Smith P. and Keighley R. *The Formal Development of a Secure Transaction Mechanism.*
In Prehn S. and Toetenel W.J. (eds.), *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computing Science*, pages 457-476.
Springer-Verlag, October 1991.

[17] R. S. M. Barros. *On the Formal Specification and Derivation of Relational Database Applications.*
PhD Thesis, Department of Computing Science, The University of Glasgow, November, 1994.

[18] W. R. Oliveira and R. S. M. Barros. *The Real Numbers in Z.*
In A. Evans and D. Duke (eds.), *Second BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing.
Springer-Verlag, July 1997.

[19] I. Toyn. *Innovations in the Notation of Standard Z*
ZUM'98: *The Z Formal Specification Notation*, LNCS 1493, pp193-213
Springer-Verlag, Berlin, September 1998

[20] Duke R. and King P. and Rose G. and Smith G. *The Object-Z Specification Language: Version 1*
Software Verification Research Centre, The University of Queensland. Available at *http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?91-01.*

[21] Duke R. and Rose G. and Smith G. *Object-Z: A Specification Language Advocated for the Description of Standards.*
Software Verification Research Centre, The University of Queensland. Available at *http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?94-45.*

[22] Aeronic. *Strona firmy Aeronic.*
http://www.aeronic.com