Wykorzystanie metod formalnych do specyfikacji struktur wskaźnikowych na przykładzie systemu PVS

Sławomir Poreda*
Instytut Informatyki, Uniwersytet Warszawski
Slawomir.Poreda@zodiac.mimuw.edu.pl

Streszczenie

W niniejszej pracy omawiamy podejście do formalnego opisu wskaźników. Przedstawiona została tutaj próba stworzenia specyfikacji wskaźnikowych struktur danych. Specyfikacja opisuje zarówno właściwości statyczne (podstawowe niezmienniki struktur danych) jak i dynamiczne (operacje na tychże strukturach). Jako narzędzie do opracowania i weryfikacji tej specyfikacji wykorzystany został system PVS.

1. Wstęp

Zastosowanie metod formalnych do specyfikowania i weryfikowania pojęć związanych z tworzeniem oprogramowania było i jest kontrowersyjne. Uważa się, że koszty stosowania takich metod znacznie przekraczają koszty wdrożenia niedoskonałego systemu i wykrycia błędów podczas fazy jego wdrożenia, a później rzeczywistego działania.

Można się zgadzać z takim poglądem do momentu, gdy błędy takie są akceptowalne. Wbrew pozorom podejście takie może być zaakceptowane nie tylko przy małych aplikacjach – pomyłki na nieduże kwoty w dużych systemach informatycznych są wliczone w koszty wdrożenia i zazwyczaj są możliwe do wyjaśnienia i anulowania post facto. Sprawy wyglądają inaczej, gdy rozważymy

* Praca finansowana z grantu KBN 8T 11C 01515

systemy podtrzymujące ludzkie życie, system automatyczne sterujący pracą metra czy tez oprogramowanie stacji orbitalnej Alfa, która jak wiadomo ostatnio omal nie wymknęła się spod kontroli właśnie na skutek błędów w aplikacjach pokładowych.

Uświadomiwszy sobie to, można już zrozumieć wsparcie NASA dla rozwoju takich systemów jak PVS – pozwalających na specyfikowanie i dowodzenie stworzonych w nim teorii, opisujących w sposób formalny wybrane zagadnienie. Formalne udowodnienie staje się o tyle istotne, że jest w stanie wykryć wszystkie luki, nieścisłości oraz przypadki nieuwzględnione, których człowiek tworzący oprogramowanie nie dostrzeże.

Wykorzystanie metod formalnych do zagadnień programowania było poświęconych wiele prac, między innymi: [1], [2], [3], [4], [5]. Wśród wielu narzędzi na uwagę zasługuje system PVS, który został zastosowany w opisywanym podejściu [17], jak również w innych pracach [15], [16].

1.1. System PVS

System PVS służy do opracowania i weryfikacji specyfikacji opisujących różne zagadnienia. PVS posiada rozbudowaną składnię i umożliwia operowanie w logice wyższego rzędu, definiowanie własnych typów i podtypów danych oraz tworzenie teorii parametryzowanych.

Podstawowym atutem PVS jest półautomatyczne dowodzenie opisanych specyfikacji. Wykorzystywany jest do tego rachunek sekwentów Gentzena. System umożliwia korzystanie z zestawu podstawowych twierdzeń oraz udostępnia szereg narzędzi wspierających sam proces dowodzenia. Zaletą PVS jest automatyczne generowanie warunków TCC zapewniających spójność typowania dla wprowadzanych twierdzeń i definicji.

Więcej na temat systemu PVS można znaleźć w [6], [7].

2. Dynamiczne struktury danych

Zagadnienia informatyczne są często w pewien sposób próbą opisu zjawisk znanych nam ze świata realnego. Takie pojęcia jak stos czy kolejka istniały już dużo przed tym, nim człowiek wprowadził do słownika nowy wyraz: komputer. Wraz z upowszechnieniem się komputera pojawiła się nauka zwaną informatyką, próbująca przenosić opis zjawisk rzeczywistych do świata maszyny. Jako, że zmiana jest immanentną cechą otaczającej nas rzeczywistości, wobec tego również w informatyce musiały pojawić się obiekty, które nie były już tylko statyczne – o ustalonej z góry raz na zawsze wielkości czy strukturze, ale były

bardziej elastyczne, dynamicznie – dając tym samym większe możliwości wyrażania i opisywania. W ten oto sposób wprowadzono dynamiczne struktury danych. I tak informatyka zaadoptowała pojęcia stosu, listy, drzewa na własne potrzeby.

Do implementacji struktur dynamicznych w językach programowania służą najczęściej typy wskaźnikowe. Wiele efektywnych algorytmów w nieodzowny sposób wymaga użycia wskaźników: [8], [9], [10], [11]. Tym samym ich obecność w programach przestaje być fanaberią, a staje się koniecznością. Programy używające takich dynamicznych struktur opartych o wskaźniki są dużo trudniejsze do analizy i przysparzają nie mało kłopotów początkującym programistom. Ale i przy bardziej zaawansowanych oraz naukowych podejściach wskaźniki przysparzają mnóstwo problemów. Nie poddają się one łatwemu badaniu poprzez metody formalne jak na przykład niezmienniki, ciężko jest je też specyfikować w postaci ogólnej i formalnie udowadniać ich własności. Tematyce tej zostały poświęcone np. prace [12], [13], [14].

Problem specyfikowania dynamicznych struktur danych i wskaźników jest złożony – logika pierwszego rzędu jest zbyt słaba do opisu tych pojęć, równocześnie te struktury są w informatyce istotne. Dlatego podjęliśmy się próby formalnego ich opisania przy użyciu systemu PVS [17], który pozwala operować w logice wyższego rzędu.

3. Specyfikacja wskaźnikowych struktur danych

W niniejszym rozdziale omawiamy kolejne etapy zastosowane przy tworzeniu specyfikacji dynamicznych struktur danych w PVS oraz problemy jakie wiązały się z każdym etapów. Wydaje się, że poniżej opisana metodologia postępowania jaka została użyta przy tym zagadnieniu może być również wykorzystana do innych.

Etapy tworzenia specyfikacji w omawianym przypadku były następujące:

- model wskaźnikowych struktur danych,
- specyfikacja niezmiennika struktury danych,
- dowody poprawności typowania (*type correctness conditions*) specyfikacji niezmiennika,
- określenie i udowodnienie twierdzeń opisujących proste własności,
- specyfikacja operacji,
- dowód zachowania niezmiennika struktur danych przez operacje.

Ustalenie i udowodnienie niezmienników operacji na strukturach wskaźnikowych jest ważne z tego względu, że problemy związane ze

wskaźnikami wynikają zazwyczaj z innego rezultatu działania operacji niż intuicyjnie oczekiwany przez programistę.

Wszystkie powyższe etapy zostały praktycznie zrealizowane w PVS, wiązały się one z dużym nakładem pracy, gdyż system nie pozwala na żadne nieścisłości i braki w dowodach. Poszczególne etapy są zaprezentowane w kolejnych rozdziałach.

3.1. Model wskaźnikowych struktur danych

Pierwszym elementem jaki musiał powstać, było zdefiniowanie samego środowiska wskaźnikowego. Zostało ono określone jako funkcja z typu wskaźnikowego w typ danych. Typ wskaźnikowy został rozszerzony o dodatkowy element *nil*, natomiast typ danych o element typu *undefined*, tak aby umożliwić wskaźnikom posiadanie wartości nieokreślonych. Tym sposobem stan pamięci został określony jako tak oto rozszerzona funkcja ze wskaźników w dane. Teoria została stworzona parametrycznie, tak aby była elastyczna i opisywała środowisko wskaźników na konkretny typ, podany jako parametr. Dzięki temu ta specyfikacja mogła być wykorzystana przy każdym typie danych dynamicznych – zarówno przy liście jednokierunkowej, drzewie binarnym jak i digrafie. Parametrem jest wtedy pojedynczy węzeł danej struktury reprezentowany jako zwyczajny rekord. Jest to więc dokładne odzwierciedlenie rzeczywistej reprezentacji takiego węzła w programie.

Przykład 1 Parametryzacja środowiska wskaźnikowego

- dodanie wartości nil do nośnika wskaźników

```
pointer [P : TYPE] : DATATYPE
BEGIN
  nil : nil?
  value(ptr:P) : value?
END pointer
```

- definicja uniwersalnego środowiska wskaźnikowego

```
pointer_def [P : TYPE,T : TYPE] : THEORY
```

- typ rekordowy dla listy

```
pointer_list_record[P : TYPE, T : TYPE] : THEORY
BEGIN
   IMPORTING pointer_adt[P];
   plist_element : TYPE = [# elem : T, next : pointer[P] #];
END pointer_list_record
```

- użycie środowiska wskaźnikowego w teorii list

```
pointer_finseq_list  [P : TYPE ,T : TYPE,
     (IMPORTING pointer_list_record[P,T],pointer_def[P,plist_element])
     env : pointer_env] : THEORY
```

3.2. Niezmienniki struktury danych

Podjęliśmy próbę zbudowania specyfikacji opisujących wybrane wskaźnikowe struktury danych takie jak lista, drzewo, graf. Specyfikacje tych struktur omówimy szerzej na przykładzie listy jednokierunkowej.

3.2.1 Podejście rekurencyjne a ciągowe

Specyfikacja danej struktury dynamicznej może być wykonana co najmniej na dwa niezależne sposoby:

- w sposób rekurencyjny,
- poprzez ciągi skończone,

W przypadku list, odzwierciedla to dwa różne, powszechnie stosowane spojrzenia na listy: rekurencyjne – jako głowa i ogon, oraz ciągowe – czyli ciąg elementów.

Model listy przyjęty został w sposób naturalny, realizowany w programach, to jest poprzez rekordy, z których każdy wskazuje następny, a pierwszy z rekordów jest utożsamiany z listą. Wobec tego specyfikacja w sposób rekurencyjny jest prostsza i bardziej intuicyjna. Niestety jak się później okazało dużo trudniejsza do wykazywania własności. Jest tak z tego względu, że przy konkretnych własnościach patrzymy na listę w sposób bardziej całościowy, a nie na poziomie pojedynczych rekordów. Dodatkową wadą tego modelu było założenie o skończoności środowiska wskaźnikowego jakie musieliśmy przyjąć.

Wobec tego stworzono drugi model. W tym ujęciu dla każdej poprawnej listy odpowiednia funkcja określa skończony ciąg jej węzłów. Następnie wszelkie własności i ich dowody były przeprowadzane już na takie reprezentacji.

Wniosek jaki stąd płynie jest następujący – na tym etapie trudno jest a priori przewidzieć jaki będzie najlepszy sposób specyfikacji. Elementem, który zweryfikuje „poręczność” danego modelu są dopiero dowody twierdzeń.

3.2.2 Specyfikacja niezmiennika struktury danych (na przykładzie list)

Dla każdej dynamicznej struktury danych możemy wyodrębnić zespół cech, statycznych własności, charakterystycznych dla niej. W ten sposób określimy niezmienniki struktury danych. Chcielibyśmy żeby one zawsze zachodziły, w

szczególności, aby również były zachowane po wykonaniu dowolnej operacji na strukturze – tylko taką sytuację uznamy za poprawną.

Zdefiniowane niezmienniki dla listy jednokierunkowej:

- Każda udostępniona lista jest dobrze zdefiniowana.
- Żadne dwie udostępnione listy się nie przecinają.
- Każdy zdefiniowany wskaźnik jest elementem jakiejś udostępnianej listy.
- Żaden inny wskaźnik nie wskazuje na udostępnioną listę.

Poprzez listy dobrze zdefiniowane rozumieliśmy listę będącą poprawnym ciągiem przejść między kolejnymi węzłami, zakończonych przez *nil*, nie mający pętli i nie wskazujący nigdzie na wartość niezdefiniowaną. Jako udostępnione listy rozumieliśmy wyodrębniony zbiór wskaźników, który jest interpretowany jako głowy list dostępnych w programie.

Określenie tych niezmienników nie było w przypadku listy skomplikowaną częścią – wymagało jednak zdefiniowania dodatkowych predykatów w specyfikacji listy takich jak należenie węzła do listy, nie przecinanie się list itp.

Ponadto prawdziwą weryfikacją czy nie popełniliśmy błędów w definicji tych niezmienników jest dopiero próba udowodnienia, że są zachowywane przez operacje.

Przykład 2 Predykaty tworzące niezmiennik dla listy

- Żadne dwie rożne udostępniane listy się nie przecinają:

```
accessed_disjoint?(accessed? : pred[pointer[P]])  : boolean =
     FORALL(vl1,vl2 : (valid_finseq_list?)) :
          (accessed?(vl1) AND accessed?(vl2) AND vl1 /= vl2 =>
               FORALL(l : pointer[P]) : value?(l) =>
                    NOT list_member?(l,vl1) OR NOT list_member?(l,vl2));
```

- Każdy alokowany rekord jest elementem udostępnianej listy:

```
accessed_start_everything(accessed? : pred[pointer[P]]) : boolean =
     FORALL(l : pointer[P]) : (not nil?(l) AND defined?(^(ptr(l))) =>
          EXISTS(vl : (valid_finseq_list?)) :
               (accessed?(vl) AND list_member?(l,vl)));
```

3.3. Poprawność typowania

Dużą zaletą systemu PVS jest sprawdzanie poprawności typowania. Gdy stworzymy w PVS jakąkolwiek definicję albo twierdzenie, to system automatycznie wygeneruje zestaw dodatkowych lematów wymaganych dla

zapewniania spójności typowania. I tak na przykład jeśli jakaś funkcja jest określona tylko dla list dobrze zdefiniowanych to PVS automatycznie będzie kontrolował każdy użyty argument i w razie potrzeby jeśli to nie jest oczywiste generował twierdzenie wymagane do zachowania poprawności typowania.

Nasze doświadczenia wskazują, że jest to bardzo przydatny element, gdyż tworząc rozwiązanie na poziomie ogólnym, łatwo pominąć istotne szczegóły sprawiające, że całość nie jest spójna, a przez to nieprawdziwa.

3.4. Twierdzenia opisujące proste własności

Ten etap jest formą weryfikacji specyfikacji względem intuicyjnych, niesformalizowanych wyobrażeń o tym co opisujemy. Sformułowanie i udowodnienie prostych twierdzeń opisujących własności jest niezbędne do właściwego wyspecyfikowania struktury danych (np. konieczne do udowodnienia TCC). Reszta takich intuicyjnych twierdzeń okazuje się potrzebna do dowodów w kolejnych etapach. Ponadto np. przejście do udowadniana zachowania niezmienników operacji ujawniło konieczność udowodnienia jeszcze kilku twierdzeń opisujących własności statyczne, a niezdefiniowanych i nie udowodnionych uprzednio.

W przypadku naszych list jednokierunkowych przydatne okazało się udowodnienie szeregu własności dotyczących list dobrze zdefiniowanych (predykat *valid_finseq_list?*). Takich jak różne twierdzenia związane z długością listy, należeniem węzła do listy, podlistą, ostatnim elementem itp. W specyfikacji poprzez ciągi skończone bardzo przydatny okazał się lemat, że istnieje dokładnie jedna reprezentacja ciągowa dla danej listy.

Przykład 3 Lematy opisujące proste własności

```
member_last : LEMMA list_member?(last(vl),vl)

list_member_valid : LEMMA list_member?(l,vl) => valid_finseq_list?(l)

not_nil_last : LEMMA NOT nil?(vl) => NOT nil?(last(vl))

only_one_finseq_list : THEOREM
     valid_sequence?(l)(s1) AND valid_sequence?(l)(s2)
          IMPLIES s1 = s2;
```

3.5. Specyfikacja operacji

Kolejnym elementem, który weryfikuje poprawność i wygodę przyjętych rozwiązań na poziomie specyfikacji struktury, są operacje na tej strukturze. Operacje można wyspecyfikować przy użyciu predykatów określających jak

zmienia się środowisko wskaźnikowe podczas wykonania konkretnej operacji oraz jaki powinien być wynik, jeśli operacja ma zwracać jakąś wartość.

W przypadku listy jednokierunkowej zdefiniowane zostały predykaty określające poprawność operacji wstawiania na koniec listy oraz wyszukania pierwszego węzła zawierającego zadaną wartość.

Predykaty takie można zdefiniować na kilka sposobów – o tym czy zostały zdefiniowane w sposób poprawny i wygodny, można się dopiero przekonać próbując udowodnić zachowanie niezmiennika przez operacje.

Przykład 4 Predykat opisujący operację wstawiania do listy niepustej

```
finseq_list_insert(env1,env2:pointer_env)(v:T,p:pointer[P]) : boolean =
IF not nil?(p) THEN
     accessed_before? = accessed_after? AND
     valid_finseq_list?[P,T,env1](p) AND
     valid_finseq_list?[P,T,env2](p) AND
     accessed_before?(p) AND accessed_after?(p) AND
     EXISTS(q : pointer[P]) : value?(q) AND env1(ptr(q)) = undefined AND
          env2 = env1 WITH [(ptr(last[P,T,env1](p))) :=
          defined((# elem := elem(val(env1(ptr(last[P,T,env1](p))))),
          next := q #)),
          (ptr(q)) := defined((# elem := v, next := nil #))]
ELSE
%...
ENDIF
```

3.6. Dowód zachowania niezmiennika przez operacje

Dopiero ten etap tak naprawdę spina nam wszystkie wcześniejsze razem i potrafi ujawnić błędy w przyjętych założeniach i definicjach w każdym z poprzednich etapów.

Każdy niezmiennik jest związany ze statycznym modelem opisującym dane zagadnienie. Udowodnienie zachowania niezmiennika polega na wykazaniu, że z koniunkcji niezmiennika w środowisku przed operacją oraz predykatu opisującego operację wynika niezmiennik zanurzony w środowisku po wykonaniu operacji.

Okazuje się to najbardziej czasochłonną i trudną częścią. Sprawę dodatkowo komplikuje operowanie na tych samych teoriach tylko inaczej sparametryzowanych (raz środowiskiem początkowym, a raz środowiskiem

końcowym). PVS nie ułatwia nam tutaj zadania podczas dowodzenia i nie zawsze jawnie pokazuje w jakiej teorii zapisane są dane formuły występujące w dowodzie.

Pracochłonność tej części można określić tak: o ile dowód prostej własności struktury mogło zająć maksymalnie kilka godzin, o tyle udowodnienie pojedynczego predykatu należącego do niezmiennika zajmuje do tygodnia.

4. Podsumowanie

Zastosowanie PVS do opisywania struktur wskaźnikowych okazało się trudne i skomplikowane, niemniej jednak możliwe do wykonania. Udało się udowodnić spójność i prawdziwość teorii specyfikującej listy. W przygotowaniu są drzewa binarne i digrafy. Pracochłonność w przypadku listy okazała się na tyle duża, że nie wiadomo, czy uda się udowodnić zachowanie niezmiennika podczas operacji na drzewie i grafie. Plany badań na przyszłość to wskazanie ogólnych schematów dowodzenia różnych faktów dotyczących wskaźników (na razie to się nie udało).

Przyjęte etapy działania i sposób podejścia nadaje się do specyfikowania innych zagadnień związanych z tworzeniem oprogramowania.

Bibliografia

[1]. K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431-483, październik 1981.

[2]. O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[3]. E. W. Dijkstra. *Umiejętność programowania.* Wydawnictwa Naukowo-Techniczne, 1985.

[4]. J. V. Guttag, J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Text and Monographs in Computer Science. Springer-Verlag, 1993.

[5]. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM,* 12:576-583, 1969.

[6]. J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas. A tutorial introduction to PVS. http://www.csl.sri.com/sri-csl-fm.html, kwiecień 1995.

[7]. S. Owre, N. Shankar, J. Rushby. *PVS System Guide. PVS User Guide. PVS Language Reference. Version 2.3.* Computer Science Laboratory, SRI International, Menlo Park, CA, USA, 1999.

[8]. W. Lipski. *Kombinatoryka dla programistów.* Wydawnictwa Naukowo-Techniczne, 1989.

[9]. T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Wprowadzenie do algorytmów.* Wydawnictwa Naukowo-Techniczne, 1997.

[10]. L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych.* Wydawnictwa Naukowo-Techniczne, 1996.

[11]. L. Banachowski, K. Kreczmar, W. Rytter. *Analiza algorytmów i struktur danych.* Wydawnictwa Naukowo-Techniczne, 1987.

[12]. R. Cartwright, D. Oppen,. The logic of aliasing. *Acta Informatica,* 15:365-384, 1981.

[13]. H. Bickel, W. Struckmann. The Hoare logic of data types. Raport techniczny 95-04, Technische Universität Braunschweig, Deutschland, 1995.

[14]. M. Kubica. Formalna specyfikacja wskaźnikowych struktur danych. Praca doktorska, Instytut informatyki UW, 1999.

[15]. A. Petryk. Automatyzacja specyfikacji tropowych. Praca magisterska, Instytut Informatyki UW, 2000.

[16]. M. Lipczyński. Komputerowo wspomagana weryfikacja poprawności programów. Praca magisterska, Instytut Informatyki UW, 2000.

[17]. S. Poreda. Specyfikacja dynamicznych struktur danych w systemie PVS. Praca magisterska, Instytut informatyki UW, (w przygotowaniu).