Izabela Bondecka-Krzykowska

O związkach informatyki z matematyką

Filozofia Nauki 18/1, 77-89

2010

Filozofia Nauki
Rok XVIII, 2010, Nr 1(69)

Izabela Bondecka-Krzykowska

O związkach informatyki z matematyką*

Wśród centralnych zagadnień filozofii poszczególnych nauk bardzo często znaleźć można pytania, które można by nazwać redukcjonistycznymi. Na przykład w filozofii matematyki rozważa się możliwość redukcji matematyki do logiki (lub do logiki i teorii mnogości). Co więcej bardzo często filozofia danej dyscypliny nauki pojawia się lub zaczyna intensywnie rozwijać wraz z pojawieniem się w jej obrębie koncepcji redukcjonistycznych. Na przykład rozważania filozoficzne związane z psychologią pojawiły się, gdy starły się dwa poglądy: behawioryzm, postulujący zredukowanie człowieka jako istoty do badania jego reakcji na bodźce, oraz psychologia introspekcyjna zakładająca, że każdy człowiek jest unikalny, nieredukowalny i nie można go analizować bez introspekcji.

Jednym z centralnych pytań, które spowodowało wzrost zainteresowania zagadnieniami filozoficznymi związanymi z informatyką, jest pytanie o to, czy informatyka może być zredukowana do kolejnej gałęzi matematyki.[1] Z tym pytaniem wiąże się wiele innych bardziej szczegółowych problemów: Jaka jest rola modeli formalnych w informatyce? Jakie związki zachodzą pomiędzy algorytmem, specyfikacją a programem komputerowym? Jaka jest natura abstrakcji w matematyce, a jaka w informatyce?

Odpowiedź na pytanie o to, czy informatyka jest kolejną gałęzią matematyki ma również swoje praktyczne konsekwencje, między innymi determinuje wybór paradygmatu odpowiedniego dla informatyki oraz określa, jakie jest jej miejsce wśród innych nauk.

---

* Praca napisana przy wsparciu finansowym Fundacji na Rzecz Nauki Polskiej (subsydium prof. Romana Murawskiego).

[1] Por. np. (Colburn 2000).

1. METODY MATEMATYCZNE W PROCESIE TWORZENIA PROGRAMÓW

Podstawową działalnością informatyków jest szeroko rozumiane tworzenie programów: od sformułowania problemu do wdrożenia aplikacji służącej do rozwiązania tego problemu. Proces tworzenia oprogramowania składa się z kilku etapów. Pierwszy z nich to stworzenie tzw. specyfikacji programu, to znaczy opisanie wymogów, jakie stawia się przed programem. Kolejny etap to tworzenie (implementowanie) programu w oparciu o specyfikację. Współcześnie większość programów implementowana jest z użyciem języków programowania wysokiego poziomu, pozwalających programiście na abstrahowanie od fizycznej realizacji zaprogramowanych instrukcji.[2] W ten sposób powstaje kod źródłowy, który poddawany jest kompilacji za pomocą specjalnych programów (kompilatorów, ang. compiler) służących do automatycznego tłumaczenia kodu napisanego w języku programowania przyjaznym dla programisty na równoważny mu kod w języku maszynowym (na operacje zero-jedynkowe). Ostatnim interesującym nas etapem tworzenia oprogramowania jest sprawdzenie jego poprawności, czyli weryfikacja.[3]

Metody matematyczne obecne są w różny sposób na wszystkich etapach tworzenia oprogramowania: od specyfikacji aż do weryfikacji. Rozpocznijmy od analizy zastosowania metod formalnych w tworzeniu specyfikacji programów.

1.1. Specyfikacja

Wielu badaczy uważa, że dziedziną, w której można z powodzeniem korzystać z metod matematycznych, jest tworzenie specyfikacji programów. Ponieważ język naturalny jest niejasny i wieloznaczny, więc nie nadaje się do formułowania specyfikacji i powinien być zastąpiony językiem formalnym. W roku 1985 Bertrand Meyer zaproponował system formalny do specyfikacji programu formatującego fragment tekstu. W jego podejściu do zastąpienia języka potocznego językiem formalnym potrzebna była tylko znajomość pojęć zbioru, ciągu, relacji, funkcji oraz umiejętność posługiwania się rachunkiem predykatów.

Anthony Hall (Hall 1990) zwraca uwagę, że wbrew twierdzeniu wielu przeciwników metod formalnych w informatyce, specyfikacje formalne są wykorzystywane w praktyce przy tworzeniu dużych aplikacji między innymi dla przemysłu oraz że

[2] W momencie przejścia w programowaniu od asemblera do języków wysokiego poziomu, informatycy zostali zwolnieni z konieczności posługiwania się terminami zorientowanymi maszynowo, takimi jak rejestry, na rzecz bardziej abstrakcyjnych wyrażeń takich jak zmienna. Języki wysokiego poziomu umożliwiają programistom opis działania maszyny bez odwołania się do jakiegoś konkretnego typu sprzętu.

[3] Pomijamy tu pozostałe etapy tworzenia oprogramowania (m.in. wdrożenie i utrzymanie aplikacji), ponieważ ich analiza nic nie wnosi do rozważań na temat związków informatyki z matematyką.

dzięki nim powstają łatwiejsze do zrozumienia i krótsze dokumentacje programów. Twierdzi on, powołując się na własne doświadczenia projektowe i programistyczne, że zastąpienie specyfikacji napisanej w języku naturalnym przez specyfikację formalną wiąże się z wieloma praktycznymi korzyściami.

Po pierwsze, specyfikacja formalna pomaga sformułować i uściślić wymagania stawiane przed programem oraz ułatwia ich zrozumienie i to zarówno twórcom oprogramowania, jak i jego przyszłym użytkownikom. Po drugie, pozwala na wczesne wykrycie i poprawienie błędów i nieścisłości w koncepcji działania programu, a co za tym idzie na podjęcie decyzji o zmianach dotyczących funkcjonalności przed rozpoczęciem pisania programu, co z kolei prowadzi do zmniejszenia kosztów jego tworzenia. Po trzecie, specyfikacja formalna pozwala na uzasadnienie pewnych własności programu, których nie można udowodnić innym metodami niż formalne. Po czwarte, formalna specyfikacja pomaga również w kolejnym etapie tworzenia programu, a mianowicie w jego implementacji: „Nasze doświadczenie pokazuje, że łatwiej jest zbudować system w oparciu o formalną specyfikację, niż innymi metodami”.[4]

1.2. Kompilacja

Bliski związek matematyki i informatyki widać też w przypadku kompilatorów tłumaczących kod źródłowy na język maszynowy. Poprawność takiego tłumaczenia nie jest oczywista. I tu znajduje swoje miejsce matematyka, która wykorzystywana jest dwojako: (1) metod matematycznych używa się podczas samego procesu tłumaczenia w celu uzyskania bardziej niezawodnego kodu maszynowego oraz (2) wykorzystuje się matematykę w celu udowodnienia, że program w języku maszynowym działa tak samo, jak program źródłowy. Różnica w obu podejściach polega na używaniu matematyki jako narzędzia inżynieryjnego i używaniu matematyki w celu formalnego udowodnienia pewnych własności programów. Drugie z tych zastosowań wiąże się z zagadnieniem najczęściej dyskutowanym w literaturze dotyczącej wykorzystania metod formalnych w informatyce, a mianowicie z weryfikacją programów i urządzeń. Jest to również zagadnienie budzące najwięcej kontrowersji zarówno wśród matematyków i informatyków, jak i wśród filozofów.[5]

1.3. Weryfikacja programów

Zwolennicy matematycznego podejścia do programowania uważają, że dla pokazania poprawnego działania programu należy stworzyć formalny dowód jego zgodności ze specyfikacją. Przy czym „dowód” rozumie się tu, podobnie jak dowód for-

[4] (Hall 1990, s. 16).

[5] Metody badania poprawności programów są najczęściej dyskutowanym w literaturze problemem filozoficznym związanym z programowaniem.

malny w matematyce, jako ciąg następujących po sobie formuł otrzymany przez zastosowanie pewnych reguł formalnych. Oczywiście, aby taki dowód mógł powstać konieczne jest wcześniejsze stworzenie specyfikacji programu w postaci formalnej. Nie twierdzi się przy tym, że dowody takie muszą być tworzone przez człowieka, lecz dopuszcza się stosowanie programów weryfikujących, które przez zastosowanie odpowiednich systemów formalnych tworzą tzw. weryfikacje. Zdaniem zwolenników takiej metody badania poprawności programów, tylko stworzenie formalnego dowodu gwarantuje poprawność weryfikowanego programu.

Wielu znanych matematyków i informatyków pracowało nad metodami formalnego dowodzenia poprawności programów. W wyniku tych prac powstały techniki i narzędzia wykorzystywane również dzisiaj do opisu i badania pewnych własności programów. Przytoczmy tylko niektóre z początkowych prac. John McCarthy (McCarthy 1962) stworzył język do mówienia o abstrakcyjnie pojmowanym obliczaniu, rozszerzając teorię funkcji rekurencyjnych o mechanizmy pozwalające na rozważania dotyczące instrukcji warunkowych, które są nieodłączną częścią języków programowania. Peter Naur oraz Robert W. Floyd dali początek współczesnym badaniom własności programów z użyciem semantyki (Naur 1962 oraz Floyd 1967). Z kolei Charles A. R. Hoare (Hoare 1969) zaprezentował początki systemu aksjomatycznego, który miał służyć do badania własności programów, podając aksjomaty i reguły dowodowe tego systemu. Pomimo pojawiającej się raz po raz krytyki metod formalnych, informatyka teoretyczna, zajmująca się między innymi formalnym badaniem własności programów, jest do dzisiaj jedną z najprężniej rozwijających się gałęzi informatyki.

Powab metod formalnych w odniesieniu do weryfikacji rozszerzył się również poza rzeczywistość programów (software) w obszar urządzeń (hardware) i systemów komputerowych rozumianych jako całość złożona z urządzeń i działających na nich programów.

1.4. Weryfikacja urządzeń

Formalna weryfikacja systemów stała się w ostatnich latach bardzo atrakcyjnym polem zastosowań dla formalnych metod dowodzenia z kilku powodów (por. Cohn 1989). Po pierwsze, problem weryfikacji urządzeń jest w wielu przypadkach łatwiejszy do rozwiązania niż problem weryfikacji programów. Po drugie, istnieją przyczyny natury ekonomicznej, dla których warto weryfikować urządzenia przed ich wytworzeniem (przebudowywanie chipów jest procesem bardzo drogim). Po trzecie, coraz ważniejsze staje się sprawdzanie urządzeń, które mają zostać użyte w aplikacjach, w których szczególnie ważne jest bezpieczeństwo. Przykładami takich systemów są aplikacje związane z kontrolą lotów, systemy medyczne czy też systemy kontroli i ograniczania dostępu w bankach.[6]

[6] W latach 80. XX wieku National Security Agency finansowała badania nad komputerami

2. PARADYGMAT MATEMATYCZNY W INFORMATYCE

Powszechne wykorzystywanie metod formalnych w informatyce doprowadziło do sformułowania paradygmatu matematycznego. Paradygmat matematyczny można rozumieć jako twierdzenie, że informatyka jest gałęzią matematyki, pisanie programów jest działalnością matematyczną, a rozumowanie dedukcyjne to jedyna akceptowalna metoda badania programów.

Oczywiście właściwe rozumienie paradygmatu matematycznego wymaga wyraźnego określenia pewnych związanych z nim terminów.

Po pierwsze, trzeba określić, co rozumie się pod pojęciem „program komputerowy". Należy w szczególności rozróżnić programy jako napisy (ciągi zapisanych w języku programowania instrukcji) i programy wykonywane na konkretnych maszynach. Na rozróżnienie to zwrócił uwagę Fetzer, twierdząc, że program można rozumieć albo jako ciąg instrukcji wykonywanych na maszynie abstrakcyjnej, która jako byt abstrakcyjny (nieistniejący w czasie i przestrzeni, a charakteryzowany tylko za pomocą relacji logicznych) nie ma odpowiedników fizycznych lub jako program nieodłącznie związany z maszyną fizyczną. Takie rozróżnienie nazywane jest w literaturze *dwuznacznością Fetzera.* [7]

Programy rozumiane jako napisy można traktować jak wyrażenia matematyczne i argumentować, że są one obiektami matematycznymi. Eden pisze w (Eden 2007, s. 145):

> Programy-napisy s_p są wyrażeniami matematycznymi. Wyrażenia matematyczne reprezentują obiekty matematyczne. Program p jest w pełni i dokładnie charakteryzowany przez programy-napisy s_p. Zatem program jest obiektem matematycznym.

Metody formalne w zupełności wystarczą do sprawdzenia poprawności tak rozumianego programu. Jednak jeśli rozumiemy programy jako obiekty wykonywane na rzeczywistych urządzeniach fizycznych, to w procesie ich weryfikacji trzeba wyjść poza metody matematyczne.

Po drugie, należy sprecyzować znaczenie terminu „poprawny" w odniesieniu do programów komputerowych. Najczęściej przez poprawny program rozumie się albo program zgodny ze specyfikacją, albo program działający w zamierzony sposób

---

„*provable secure*", co oznaczało komputery, których bezpieczeństwo byłoby zagwarantowane matematycznie. Celem nie było udowodnienie zgodności specyfikacji i programów, jak w przypadku weryfikacji programów, ale pokazanie, że w systemie bezpieczeństwa tkwiącym w projekcie nie ma luk. Jednym z efektów tych badań było zbudowanie wojskowego mikroprocesora o nazwie VIPER (**V**erifiable **I**ntegrated **P**rocesor for **E**nhanced **R**eliability), który był reklamowany jako pierwszy dostępny komercyjnie mikroprocesor z formalną specyfikacją i dowodem, że chip jest z nią zgodny. Po przeczytaniu materiałów promocyjnych dotyczących VIPER-a można odnieść mylne wrażenie, że udowodnienie bezpieczeństwa projektu pociąga za sobą twierdzenie, że wszystkie komputery zbudowane zgodnie z tym projektem będą bezpieczne (por. par. 3.4.).

[7] Fetzer zwraca uwagę, że istnieje analogia pomiędzy rozróżnieniem dwóch rozumień programu a rozróżnieniem matematyki czystej i stosowanej.

(rozwiązujący problem, dla którego został stworzony). Brian C. Smith sugeruje wprowadzenie rozróżnienia pomiędzy tymi dwoma znaczeniami poprzez używanie różnych terminów. *Poprawność*, w wąskim znaczeniu technicznym, można rozumieć jako relację pomiędzy programem a modelem problemu w terminach formalnej specyfikacji, natomiast *wiarygodność* może określać relację pomiędzy programem i jego funkcjonowaniem w rzeczywistym świecie. Twierdzi się nawet (por. DeMillo 1979), że powinniśmy dokonać ostrego rozróżnienia pomiędzy *wiarygodnością* programu a jego *doskonałością* i skoncentrować się na wiarygodności, podobnie jak to czynią w swojej pracy inżynierowie.

Zwolennicy paradygmatu matematycznego w informatyce najczęściej rozumieją poprawność programu jako jego zgodność ze specyfikacją i twierdzą, że jedyną dopuszczalną metodą badania tej poprawności jest przeprowadzanie dowodów formalnych. Odrzucają przy tym wszelki eksperyment, w tym testowanie, jako metodę dochodzenia do wiedzy na temat programów. Przy takim podejściu wiedza informatyczna jest wiedzą *a priori*, dochodzimy do niej bowiem na drodze czystego rozumowania. Odpowiada to racjonalizmowi w epistemologii, który zakłada, że jedyną metodą dochodzenia do wiedzy pewnej jest metoda rozumowa, a wiedza *a priori* ma pierwszeństwo przed wiedzą *a posteriori*. Dlatego też paradygmat matematyczny w informatyce jest czasami nazywany paradygmatem racjonalnym.

Wielu specjalistów związanych z informatyką teoretyczną opowiada się za uznaniem, że jedyną drogą do uzyskania wiedzy na temat poprawności programów jest zastosowanie metod formalnych. E. W. Dijkstra twierdzi nawet, że programy są obiektami abstrakcyjnymi, do których należy dostosować fizyczne maszyny:

> Program to operowanie na symbolach abstrakcyjnych, które mogą zmienić się w konkretne przez zastosowanie do nich komputerów. Ostatecznie, nie jest celem programów instruowanie naszych maszyn, obecnie celem maszyn jest wykonywanie naszych programów.[8]

Takie podejście do związku pomiędzy komputerem a programem jest odpowiednie dla paradygmatu matematycznego. Można nawet powiedzieć, że jest to *argument ontologiczny* na rzecz tego paradygmatu (por. Fetzer 1991), wskazuje bowiem na pewne ontologiczne pierwszeństwo programów (bytów abstrakcyjnych, formalnych) przed komputerami (bytami fizycznymi).[9]

Drugi, często cytowany argument na rzecz paradygmatu matematycznego pochodzi również od Dijkstry i jest to *argument epistemologiczny*:

> […] testowanie programu może przekonująco pokazać obecność błędów, ale nigdy nie pokaże, że ich nie ma.[10]

---

[8] (Dijkstra 1989, s. 1401).

[9] Nie jest to jednak podejście typowe, ponieważ zazwyczaj programy rozumiane są jako instrukcje wykonywane przez komputery.

[10] […] *program testing may convincingly demonstrate the presence of bugs but can never demonstrate their absence* (Dijkstra 1989, s. 1401).

Dijkstra sugeruje przez to, że jedyną drogą do uzyskania wiedzy na temat poprawności programów jest zastosowanie metod formalnych. Twierdzi on również, że informatyka jest i zawsze będzie połączeniem „obliczania" (wykonywanego przez komputery) oraz „programowania" (operacji na symbolach wykonywanych przez człowieka). Twierdzi on również, że „programowanie automatyczne" jest pojęciem wewnętrznie sprzecznym, ponieważ sugeruje możliwość eliminacji człowieka z procesu programowania. Takie podejście do informatyki pozwala jednoznacznie stwierdzić, że w klasyfikacji wszystkich nauk miejsce informatyki znajduje się tuż obok matematyki i logiki formalnej. Dijkstra proponuje nawet nazywanie informatyki skrótem VLSAL od słów **V**ery **L**arge **S**cale **A**pplication of **L**ogic.

Trzeci argument na rzecz paradygmatu matematycznego związany jest z praktyczną realizacją formalnej weryfikacji programów i nazywany jest zazwyczaj *argumentem ze zwiększania skali* (ang. scaling-up argument). Można go opisać jako rozumowanie złożone z dwóch części:

1. Weryfikacja jest dziedziną stosunkowo młodą („w swoim niemowlęctwie"). Z czasem będziemy w stanie weryfikować coraz bardziej złożone algorytmy i coraz bardzie skomplikowane programy.

2. Produkowane w rzeczywistości duże systemy nie są niczym innym niż złożeniem prostych algorytmów i modeli. Ponadto raz zweryfikowany algorytm czy model programu będzie można powiększać aż do rozmiarów rzeczywistego systemu. Zatem weryfikacja dużego systemu będzie sumą wielu małych weryfikacji jego składowych.

Tak więc powstanie formalnych weryfikatorów rzeczywistych programów jest tylko kwestią czasu.

Argument ze zwiększania skali jest jednym z najczęściej dyskutowanych argumentów na rzecz paradygmatu matematycznego w informatyce.

3. KRYTYKA PARADYGMATU MATEMATYCZNEGO W INFORMATYCE

Krytyka paradygmatu matematycznego w informatyce rozpoczęła się od dyskusji nad tym, czy formalna weryfikacja jest prawdziwym dowodem matematycznym. Debata ta została zapoczątkowana przez DeMillo, Liptona i Perlisa, którzy w pracy z roku 1979 (DeMillo 1979) argumentowali, że mechaniczne stworzone weryfikacje programów, pomimo tego, że są ciągami formuł logicznych, nie są jednak dowodami matematycznymi. Akceptacja wyników matematycznych jest bowiem procesem społecznym prowadzącym do przekonania o poprawności wyniku, w którym tylko jedną z części jest dowód, co więcej, bardzo rzadko jest to dowód formalny (rozumiany jako łańcuch następujących po sobie formuł). Ponieważ w formalnym sprawdzaniu programów nie ma porównywalnego procesu, więc weryfikacje nie są dowodami matematycznymi.

Przyjrzyjmy się teraz temu procesowi, tak jak go opisują DeMillo, Lipton i Perlis. Na początku dowód twierdzenia jest wiadomością ustną lub naszkicowaną na tablicy lub skrawku papieru, a nie istniejącym niezależnie obiektem abstrakcyjnym. Taka właśnie wiadomość jest poddawana pierwszym ocenom poprzez dyskusje wśród kolegów na seminariach lub w prywatnych rozmowach. Jeśli wynik nie wzbudza zaufania lub nawet zainteresowania wśród współpracowników, mądry matematyk rozważa go ponownie. Ale jeśli zostanie przyjęty, choćby z umiarkowanym zainteresowaniem i zaufaniem, autor spisuje dowód. Zapis dowodu krąży przez jakiś czas wśród zainteresowanych w formie maszynopisu i jeśli nadal wydaje się wiarygodny, autor zgłasza dopracowaną wersję dowodu do publikacji. Jeśli recenzenci również uznają go za przekonujący, to dowód jest publikowany i wtedy może być czytany przez szerszą społeczność. Jeśli czytający dowód matematyk uwierzy („na pierwszy rzut oka") w jego poprawność, to następuje proces internalizacji tego wyniku. Matematyk, który czyta dowód i jest przekonany o jego poprawności, podejmuje próbę przeformułowania go tak, by wyrazić go „własnymi słowami". Ponieważ nie ma dwóch matematyków, u których internalizacja przebiegałaby dokładnie w ten sam sposób, więc w wyniku takiego procesu powstaje zazwyczaj wiele wersji tego samego twierdzenia, każda zwiększająca wiarę społeczności matematyków w to, że oryginalne twierdzenie jest prawdziwe. Wiarygodne twierdzenia często znajdują zastosowanie jako lematy w większych twierdzeniach albo w inżynierii, przenikają również do pokrewnych gałęzi matematyki. Jeśli twierdzenie lub technika dowodowa sprawdzają się w innej dziedzinie niż początkowa, to zwiększa się przekonanie co do poprawności wyniku.

Po takiej internalizacji, transformacji, generalizacji, użyciu i badaniu twierdzenia pod różnymi kątami i w różnych działach matematyki, społeczność specjalistów uznaje wynik za poprawny i o twierdzeniu mówi się, że jest prawdziwe w klasycznym sensie, tzn. że poprawność ta może być pokazana formalnie, w sposób dedukcyjny. Oczywiście dla większości dowodów matematycznych takiego dowodu formalnego się nie przeprowadza.

Dowody matematyczne zwiększają nasze przekonanie co do prawdziwości stwierdzeń matematycznych tylko wtedy, gdy zostaną poddane opisanym powyżej mechanizmom społecznym. W przypadku automatycznie wygenerowanych weryfikacji programów nie ma porównywalnych procesów społecznych prowadzących do ich akceptacji. Weryfikacje nie są wiadomościami. Co więcej, weryfikacje nie mogą być przeczytane. Jako niemożliwe do przeczytania i wysłowienia nie mogą zostać zinternalizowane, przeformułowane, uogólnione lub włączone do innych dyscyplin. Nie zyskują one wiarygodności stopniowo, tak jak dowody matematyczne, można w nie wierzyć albo ślepo, albo wcale.

W literaturze znaleźć można również wiele innych argumentów przeciw formalnej weryfikacji programów, a co za tym idzie przeciw paradygmatowi matematycznemu w informatyce. Przeanalizujmy te, które pojawiają się najczęściej.

Jak już wspomniano w par. 2 zwolennicy stosowania metod formalnych w informatyce rozumieją poprawność programu jako jego zgodność ze specyfikacją. Takie podejście implikuje wiele problemów, których część dotyczy specyfikacji formalnych.

3.1. Problemy formalnej specyfikacji

Wielu zwolenników metod formalnych twierdzi, że specyfikacja programu powinna być wyrażona w języku formalnym, ponieważ tylko takie podejście zabezpieczy ją przed błędami i niespójnością (por. np. Meyer 1985). Jednak takie twierdzenie można łatwo obalić. Peter Naur (Naur 1982) podał prosty przykład formalnej specyfikacji, który pokazuje, że jej niekompletność może prowadzić do uznania formalnie zweryfikowanego programu za niepoprawny. Specyfikacje programów bardzo często bywają niekompletne, na przykład specyfikacja dla dowolnego systemu operacyjnego albo kompilatora zajmuje tomy i nikt nie wierzy, że jest ona kompletna. Ponadto, ze względu na złożoność i potencjalną niekompletność, specyfikacje formalne są w praktyce tworzone bardzo rzadko (podobnie jak dowody formalne w matematyce).

Zwolennicy paradygmatu matematycznego w programowaniu twierdzą, że specyfikacje formalne mają pomóc programistom i ułatwić im tworzenie oprogramowania (por. par. 1.1.). Peter Naur (Naur 1982) zwraca uwagę, że ściśle formalne podejście do programowania nie tylko nie pomaga, ale wręcz może szkodzić zmniejszając efektywność pracy programistów. Zgodnie bowiem z paradygmatem formalnym programista powinien najpierw wyrazić rozwiązanie postawionego przed nim problemu w formalnym języku specyfikacji, później zaprogramować to samo rozwiązanie w języku programowania, a na koniec udowodnić, że oba te opisy są w jakimś sensie równoważne. W takim podejściu programista musi stworzyć nie jeden, ale dwa formalne opisy tego samego problemu i udowodnić ich równoważność, podczas gdy problemy związane z adekwatnością systemów formalizmów do wyrażania rzeczywistych problemów nadal pozostają nierozwiązane.

Warto również zwrócić uwagę na wartość metody nieformalnej w tworzeniu oprogramowania. Pozwala ona bowiem na dyskutowanie, krytykowanie i definiowanie pojęć intuicyjnych. Oczywiście formalizacja może być w tym procesie pomocna, tak samo jak ma to miejsce w dowodzeniu twierdzeń matematycznych, ale powinna być jedynie narzędziem pomocniczym intuicji, a nie ją zastępować. Naur pisze (Naur 1982, s. 458):

> […] program powinien być wspomagany i specyfikowany przez dokumentację dowolnego rodzaju, sprawą nadrzędną w tworzeniu jego dokumentacji jest czytelność dla ludzi, którzy mają z nią do czynienia. Dla osiągnięcia tej czytelności (jasności) powinno się używać dowolnego systemu formalnego, nie jako celu samego w sobie, ale wtedy, gdy wydaje się on pomocny autorom i czytelnikom.

Kolejne argumenty skierowane są przeciw możliwości formalnej weryfikacji programów.

3.2. Argument ze zmienności

Formalne podejście do weryfikacji programów jako zgodności programu ze specyfikacją, ma sens jedynie przy założeniu, że powstały one niezależnie oraz że zarówno program, jak i jego specyfikacja nie ulegają zmianom. Oba te założenia można łatwo obalić. Podczas procesu tworzenia oprogramowania wzajemne oddziaływania pomiędzy specyfikacją a programem są bardzo częste. Na przykład błędy znalezione w programie wymuszają zmiany w specyfikacji, z kolei błędy w specyfikacji prowadzą do zmian programu uwzględniających zmiany w tej specyfikacji. Ponadto rzeczywiste programy muszą być serwisowane i modyfikowane, a więc zmieniane i nie ma powodu, by twierdzić, że weryfikacja zmodyfikowanego programu jest łatwiejsza niż programu przed modyfikacją, a to podważa sens weryfikacji jako takiej.

Jeśli nawet przyjmiemy, że zarówno program, jak i specyfikacja powstają niezależnie i nie ulegają zmianom, to nie świadczy to jeszcze, że formalna weryfikacja rzeczywistych programów jest w ogóle możliwa.

3.3. Argument ze złożoności

Twierdzi się, że tworzenie formalnych weryfikacji jest w praktyce niemożliwe, obalając przy tym wspomniany wcześniej argument ze zwiększania skali głoszący, iż z czasem możliwe będzie stworzenie programów dokonujących formalnej weryfikacji rzeczywistych, dużych systemów komputerowych. Takie systemy nie są bowiem niczym innym niż złożeniem prostych algorytmów i modeli, które można łatwo zweryfikować formalnie, a więc weryfikacja dużego systemu będzie sumą wielu małych weryfikacji jego składowych. Raz zweryfikowany algorytm czy model programu będzie można powiększać aż do rozmiarów rzeczywistego systemu. Należy jednak zauważyć, że nawet dla bardzo prostych teorii matematycznych dowody formalne są bardzo długie, a podobnie rzecz ma się w przypadku formalnych weryfikacji rzeczywistych programów. Programy te są często tak złożone, że ich weryfikacje musiałyby być niewiarygodnie długie, co więcej, sam argument ze zwiększania skali jest zwyczajnie fałszywy (por. DeMillo 1979).

Po pierwsze, żaden programista nie zgodzi się z twierdzeniem, że duży system jest niczym więcej niż tylko sumą swoich składowych, ponieważ istnieje wiele metod i trików w łączeniu tych składowych, co w efekcie prowadzi do zupełnie nowego obiektu, który ma pewne cechy dodatkowe. Ponadto oszacowano, że około połowy kodu działających systemów przypada na tak zwany *inferface* i obsługę komunikatów o błędach — są to struktury *ad hoc*, które są z definicji nieweryfikowalne.

Po drugie, w argumencie ze zwiększania skali można zauważyć całkowite utożsamienie programu z algorytmem, a wystarczy porównać ich specyfikacje, żeby stwierdzić, że jest to błędem. Specyfikacje algorytmów są zwięzłe i jasno określone, podczas gdy specyfikacje rzeczywistych systemów są ogromne, często tego samego

poziomu komplikacji co same systemy. Specyfikacje algorytmów są stabilne, czasami nie zmieniają się przez stulecia, specyfikacje rzeczywistych programów zmieniają się z dnia na dzień, a nawet z godziny na godzinę. Nie jest to zatem różnica w skali, ale w rodzaju. Rozwój jednych nigdy nie doprowadzi wprost do drugich. Problemy są zasadniczo różne.

3.4. Argument z relatywności

Rozumowanie matematyczne nie może również pokazać absolutnej poprawności programu czy też poprawnego działania urządzenia. To, czego dowodzimy w sposób matematyczny, to tylko poprawność matematycznego modelu programu lub sprzętu, a nie właściwe działanie fizycznego urządzenia lub programu wykonywanego na fizycznej maszynie. Argumentacja matematyczna nie może pokazać odpowiedniości pomiędzy modelem matematycznym a rzeczywistością fizyczną (Barwise 1989). Na przykład w procesie weryfikacji sprzętu mówimy o zachowaniu abstrakcyjnych miejsc w pamięci i elementów wykonawczych, a nie o rzeczywistym urządzeniu jako takim. Błędem jest zatem twierdzenie, że dowody sprzętowe stwierdzają poprawność działania fizycznych urządzeń ponad wszelką wątpliwość (por. materiały reklamowe dotyczące procesora VIPER, par. 1.4.). Cohn wyraża to następująco:

> Urządzenie materialne może być tylko obserwowane i mierzone; nie może być zweryfikowane. Urządzenie może być opisane w sposób formalny a opis ten zweryfikowany, ale nie ma sposobu na zapewnienie trafności (dokładności) tego opisu.[11]

Stwierdza on, że istnieje zasadnicza różnica pomiędzy naturą obiektów abstrakcyjnych i naturą obiektów fizycznych i że nie można wniosków dotyczących obiektów fizycznych uważać za tak samo pewne, jak wnioski dotyczące obiektów abstrakcyjnych. Zatem, nawet jeśli założymy możliwość udowodnienia poprawności działania skomplikowanego systemu oprogramowania, równie skomplikowanego programu dowodzącego twierdzenia oraz projektu sprzętu na poziomie bramek obwodów jakiegoś urządzenia, na którym działa program, to nadal nie mamy matematycznego dowodu fizycznego zachowania się żadnego danego programu. Każdy taki dowód pokazuje tylko relatywną poprawność programu, zakłada bowiem implicite adekwatność modelu matematycznego użytego do zaprojektowania tego programu oraz poprawne działanie struktury programowo-sprzętowej, na której jest on wykonywany.

3.5. Argument z nieprzewidywalności

Warto zauważyć, że nie zawsze można przewidzieć działanie programu w sposób czysto analityczny. Każdy programista wie, że zmiana linii lub czasami tylko jednego bitu w kodzie źródłowym może zupełnie zmienić sposób wykonania programu

[11] (Cohn 1989, s. 131).

w sposób, którego nie rozumiemy i nie jesteśmy w stanie przewidzieć. Również wiele wirusów zmienia działanie programów, a w dobie Internetu prawie każdy komputer narażony jest na ich ataki i podlega ryzyku modyfikacji znajdującego się na nim oprogramowania.

W systemach chaotycznych nawet drobne zmiany danych początkowych mogą skutkować nieprzewidywalnymi zmianami wyników — efekt taki określa się często mianem „efektu motyla". Można go zaobserwować nie tylko w programowaniu, ale również w przypadku zachowań mikroprocesorów (por. Berry 2006). Co więcej, wielu programistów zauważa, że zachowanie się napisanych przez nich programów bywa zaskakujące, a nawet nieprzewidywalne.[12] Istnienie takich zjawisk przemawia wyraźnie przeciw ograniczeniu się w informatyce do metod formalnych i uzasadnia poszukiwanie innego paradygmatu informatyki niż paradygmat matematyczny.

Przyjęcie paradygmatu matematycznego wiąże się z ograniczeniem metod informatyki do metod formalnych. Jeśli jednak zgodzimy się, że nie można w sposób czysto formalny udowodnić, czy też nawet przewidzieć zachowania rzeczywistych systemów komputerowych, to musimy dopuścić w informatyce inne metody badania poprawności programów na przykład testowanie (uruchomienie programów na rzeczywistych urządzeniach).

4. PODSUMOWANIE

Podstawową działalnością informatyków jest tworzenie programów, a metody matematyczne obecne są w różny sposób na wszystkich etapach procesu tworzenia oprogramowania: od specyfikacji aż do weryfikacji (por. par. 1). Przemawia to na rzecz tezy, że informatyka jest kolejną gałęzią matematyki, a tworzenie programów jest działalnością matematyczną, czyli do paradygmatu matematycznego w informatyce. Oczywiście paradygmat ten był i nadal jest szeroko dyskutowany i krytykowany (por. par. 2 i 3). Rozważania dotyczące zastosowań metod formalnych w informatyce oraz ich ograniczeń doprowadziły do sformułowania innych niż matematyczny paradygmatów informatyki.

Współcześnie w informatyce wyróżnić można trzy podstawowe paradygmaty: (1) *paradygmat matematyczny* (racjonalny), zakładający, że informatyka jest gałęzią matematyki, (2) *paradygmat technokratyczny*, traktujący informatykę jako dziedzinę inżynieryjną, (3) *paradygmat empiryczny* (naukowy), definiujący informatykę jako naukę przyrodniczą, opartą na eksperymencie. [13]

Te różne podejścia do informatyki wynikają z różnego rozumienia relacji pomiędzy matematyką a informatyką. Zatem określenie relacji pomiędzy matematyką a infor-

[12] Efekt zaskoczenia pracą programu podkreślali Appel i Haken (Appel 1983) — autorzy komputerowego dowodu twierdzenia o czterech barwach.

[13] Więcej na temat paradygmatów informatyki znaleźć można w pracy (Eden 2007) oraz w (Izabela Bondecka-Krzykowska „Paradygmaty informatyki", w przygotowaniu).

matyką jest problemem bardzo ważnym dla filozofii informatyki i to nie tylko w kontekście historycznym (ponieważ był on jednym z pierwszych rozważanych w literaturze problemów filozoficznych związanych z informatyką), ale również w kontekście wyboru właściwej metody uprawiania informatyki oraz określenia jej miejsca pośród innych nauk.

LITERATURA CYTOWANA

Appel K., Haken W. (1983), *Zagadnienie czterech barw*, w: *Matematyka współczesna. Dwanaście esejów*, Wydawnictwa Naukowo-Techniczne, Warszawa, s. 170-199.

Barwise J. (1989), Mathematical Proofs of Computer System Correctness, *Notices of the American Mathematical Society* 36, s. 844-851.

Berry H., Perez D. G., Temam O. (2006), Chaos in computer performance, *CHAOS*, **16**:013110. arXiv:nlin.AO/0506030.

Cohn A. (1989), The Notion of Proof in Hardware Verification, *Journal of Automated Reasoning* 5, no. 2, s. 127-139.

Colburn T. R. **(**2000), *Philosophy and Computer Science*, M. E. Sharpe.

DeMillo R., Lipton R., Perlis A. (1979), Social Processes and Proofs of Theorems and Programs, *Communications of Association for Computing Machinery* 22 (May 1979), s. 271-280.

Dijkstra E.W. (1989), On the Cruelty of Really teaching Computing Science, *Communications of the Association for Computing Machinery* 32, s. 1398-1404.

Eden A. H. (2007), Three Paradigms of Computer Science, *Minds and Machines* 17, s. 135-167.

Fetzer J. (1988), Program Verification: The Very Idea, *Communications of Association for Computing Machinery* 31, no. 9, s. 271-280.

Fetzer J. (1991), Philosophical Aspects of Program Verification, *Minds and Machines* 1: 197-216.

Floyd R. (1967), Assigning Meanings to Programs, *Proceedings of Symposia in Applied Mathematics* 19, s. 19-32.

Hall A. (1990), Seven myths of formal methods, *IEEE Software* 7(5), s. 11-19.

Hoare C. A. R. (1969), An Axiomatic Basic for Computer Programming**,** *Communications of the Association for Computing Machinery* , Vol. 12, No. 10, s. 576-580.

McCarthy J. (1962), Towards a Mathematical Science of Computation, [w:] *Proceedings of the IFIP Congress 62*, pp. 21-28.

Meyer B. (1985), On formalism in specifications, *IEEE Software*, 2(1):6-26, January 1985.

Naur P. (1966), Proof of Algorithms by General Snapshots, *BIT Numerical Mathematics*, Volume 6, No. 4.

Naur P. (1982), Formalization in Program Development. BIT *Numerical Mathematics* 22(4), s. 437-453.

Newell A., Simon H. A. (1976), Computer science as empirical inquiry, *Communications of the Association for Computing Machinery*, 19 (3), s. 113-126.

Smith B. C. (1985), Limits of Correctness in Computers, Report No. CSLI-85-36. Center for Study of Language and Information, October 1985.