Metody formalne w projektowaniu oprogramowania: mity i rzeczywistość*

Piotr Dembiński
Instytut Podstaw Informatyki Polskiej Akademii Nauk
ul. Ordona 21, Warszawa
e-mail: piotrd@ipipan.waw.pl

1. Wstęp

Metody formalne obecne są w informatyce od jej początku, ale termin ten w inżynierii oprogramowania pojawiaja się dopiero, gdy jego burzliwy rozwój i rosnąca złożoność wywołały wyraźną potrzebę bardziej uporządkowanego i zorganizowanego podejścia do całego procesu produkcji softwaru. Już w latach sześćdziesiątych podjęto próby wskazania reguł i konstrukcji, które programista (zespół programistów) powinien stosować, a które omijać. Pojawiły się propozycje metod znanych pod chwytliwymi nazwami, jak: programowanie strukturalne (*structured programming*), systematyczne (rygorystyczne) programowanie (*systematic (rigorous) programming*), projektowanie poprzez stopniowe przekształcania opisu problemu do jego rozwiązania programistycznego (*step-wise program refinement, top-down design*) itp. Metody te próbowano coraz bardziej formalizować, łącząc naturalną konieczność pewnej systematyzacji inżynierskiego postępowania z pokusą wykorzystania rosnących możliwości komputerów do automatyzacji tego postępowania.

Akademickie rozważania na temat formalnych specyfikacji i weryfikacji zadań programistycznych zderzyły się z rzeczywistością końca lat siedemdziesiątych, które zwykło się uważać za lata postępującego kryzysu w dziedzinie projektowania softwaru. Właśnie wtedy zauważono ogromnie szybko wzrastające koszty oprogramowania (przy taniejącym ciągle sprzęcie) szczególnie odczuwalne w zestawieniu z jego rosnącą zawodnością. Niektóre z wielkich projektów nie zostały ukończone, mnożyły sie spektakularne przykłady błędów, które wymagały nadmiernych nakładów na ich poprawianie, a to z kolei często wprowadzało nowe usterki, nad którymi, po pewnym czasie, nikt nie panował. Szybko obliczono, że w cyklu życia oprogramowania najwięcej kosztują błędy powstające w początkowej fazie projektowania. Dlatego właśnie zainteresowano się bardzo poważnie metodami formalnymi. Wydawało się niektórym, że ich zastosowanie w praktyce może być remedium na powstałą, kryzysową sytuację. Te nadzieje były, niestety, podtrzymywne przez wiele nieodpowiedzialnych obietnic, które tworzyły niebezpieczny mit nie mający pokrycia ani w rzeczywistych możliwościach metod formalnych, ani w możliwościach dostępnej wtedy technologii komputerowej tam, gdzie metody te mogłyby mieć zastosowanie. Po kilku latach nastąpiło zniechęcenie, które utrwaliło poglądy zdecydowanie sceptyczne (nawet jeśli głośno nie wyrażane), co do praktycznej przydatności metod formalnych. Powstał zatem drugi niebezpieczny mit głoszący, że metodami tymi nie warto się poważnie zajmować.

W poniższym szkicu spróbujemy przekonać czytelnika, że odpowiednio rozważnie rozumiane metody formalne zawsze miały i mają swoje własne miejsce w całym złożonym procesie wytwarzania oprogramowania. Wskażemy na konieczne warunki, które muszą zaistnieć, żeby ich użyteczność była dostrzeżona, a w szczególności zwrócimy uwagę na potrzebę istnienia efektywnych narzędzi automatyzujących wiele żmudnych czynności w systematycznym

---

* Niniejsza praca stanowi zmodyfikowaną wersję wykładu wygłoszonego na plenarnej sesji Krajowego Sympozjum Telekomunikacji, które odbywało się w dniach 9-11 września 1998 r. w Bydgoszczy..

projektowaniu. W tym kontekście pokażemy też, jak postęp technologiczny umożliwia realizację niektórych pomysłów, których (pozorna) prostota i spektakularność otrzymywanych wyników może pomóc w przełamywaniu obiegowych poglądów. Zwrócimy też uwagę na to, że metody "specjalizowane", ograniczone do danej dziedziny zastosowań wydają się spełniać łatwiej oczekiwania odbiorców. Tę tezę zilustrujemy ogólną charakterystyką metod formalnych i stosowanych narzędzi w projektowaniu protokołów komunikacyjnych oraz wnioskami z realizacji projektu europejskiego, którego jednym z zadań było wspomagana komputerowo specyfikacja i implementacja protokołu XTP (*EXpress Transport Protocol*).

2. Inżynieria oprogramowania i jej formalne metody

Inżynieria oprogramowania operuje terminem cyklu życia systemu informatycznego, obejmującego jego różne etapy (fazy). Liczba tych etapów, ich nazwy i powiązania różnią się w zależności od autora, ale zasadniczo panuje zgoda, że cykl taki zawiera interesujące nas tutaj etapy:

- specyfikacji,
- projektowania,
- implementacji.

Każdy z nich można podzielić na podetapy. Zakłada się zwykle, że wyniki uzyskane w kolejnym (pod)etapie mogą spowodować konieczność powrotu do etapów poprzednich. W tym sensie, na przykład, uzyskujemy w kolejnych iteracjach cyklu prototypową, a nie ostateczną implementację.

Przejście z etapu na etap może być mniej lub bardziej sformalizowane i w odpowiednim zakresie wspomagane przez specjalistyczne narzędzia programistyczne. Na każdym etapie pracujemy z jakąś formą opisu projektowanego systemu, przy jednym założeniu niezmiennym: w wyniku fazy implementacji otrzymany opis jest gotowy do wykonania przez komputer lub szerzej - w środowisku komputerowym, np. w sieci. Znaczy to, że jego kształt jest całkowicie sformalizowany. Jeżeli taki sformalizowany opis otrzymujemy automatycznie z opisu powstałego na wcześniejszym etapie, to warto pamiętać, że również ten wcześniejszy opis musi być całkowicie formalny. Tak więc stopień automatyzacji procesu wytwarzania systemu informatycznego (oprogramowania) w bezpośredni sposób zależy od stopnia jego sformalizowania. W tym sensie powiedzieć możemy, że metody formalne towarzyszą informatyce (inżynierii oprogramowania) od jej początku, chociaż zakres ich zainteresowań zmieniał się, w miarę jak uzyskiwano zadowalającą automatyzację pewnych etapów projektowania.

W bardzo wczesnym etapie rozwoju informatyki problemy inżynierii oprogramowania w tym schematycznym ujęciu koncentrowały się wokół relacji pomiędzy:

- programami w językach programowania,
- programami asemblerowymi,
- bezpośrednim kodem komputerowym (kto dziś odróżnia te dwie ostatnie kategorie!).

Przez lata zajmowano się składnią i semantyką języków formalnych w poszukiwaniu efektywnych metod automatyzacji przejścia od opisu programu (w językach takich jak Algol, PL1, Pascal, Algol 68 i wielu dziesiątkach innych) do jego kodu komputerowego. Te często bardzo matematycznie wyrafinowane rozważania zaowocowały tym, że dzisiaj istnieją efektywne kompilatory, a języki programowania, takie jak C, spełniają rolę asemblerów z przeszłości.

Z biegiem lat programowanie zmieniło swój charakter, ponieważ skala i stopień złożoności problemów, które rozwiązuje się za pomocą komputerów, wymagają zaangażowania i współdziałania wielu ludzi, bogatego i często rozproszonego terytorialnie środowiska obliczeniowego oraz wyrafinowanych narzędzi, wspomagających cały proces projektowania, uruchamiania i utrzymywania systemów informatycznych. Jednym z przejawów tych zmian, a także wyników wcześniejszych prób jest fakt, że w schemacie, którym posługujemy się powyżej, uwaga skoncentrowana jest na etapach poprzedzających kodowanie w wybranym języku

programowania. Okazuje się bowiem, że wybory dokonane w tych fazach są decydujące dla powodzenia dużych przedsięwzięć programistycznych. W tym sensie niesłychanie istotna jest faza sformułowania wymagań projektowych (*requirement specification*), a następnie przełożenie tych wymagań na specyfikację systemową (*system level specification*), tzn. taką, która w znacznie większej mierze zajmuje się "logiką" i "architekturą" proponowanego rozwiązania niż oprogramowaniem poszczególnych funkcji i ich efektywnością. Tak więc współczesna inżynieria programowania, w schematycznym ujęciu fragmentu jej zainteresowań, który przedstawiliśmy powyżej, koncentruje się wokół relacji pomiędzy:

- specyfikacją wymagań,
- specyfikacją systemową,
- programami w językach programowania.

Stworzenie metod i narzędzi wspomagających proces wytwarzania systemów informatycznych na tych etapach - wymaga precyzyjnego określenia, czego oczekujemy i jak rozumiemy specyfikację wymagań, a jak specyfikację na poziomie systemowym. Nieuchronnie więc mówimy o formalizacji i metodach formalnych, co najmniej w stopniu i części, w której zakładamy pomoce automatyzujące w pewnym zakresie konieczną do wykonania pracę.

W tym ogólnym wstępie nie będziemy nawet próbowali dokonywać przeglądu proponowanych formalizmów, opartych o nie metod i związanych z nimi narzędzi. Odsyłamy zainteresowanego czytelnika np. do [5] lub popularnego artykułu [11] i podanych tam referencji. Bardziej istotne jest wyjaśnienie, czego po tych metodach i narzędziach oczekujemy i na jakie trudności w ich stosowaniu napotykamy.

Specyfikacja wymagań, to zbiór tego, czego spodziewamy się po projektowanym systemie, jakie funkcje ma on spełnić i jakie wymierne efekty przynieść. Jest to więc, z grubsza rzecz biorąc, określenie problemu, który chcemy rozwiązać, poprzez zdefiniowanie jego najważniejszych atrybutów. Rozwiązań w takim sformułowaniu może nie być wcale (tzn. problem jest źle sformułowany), może ich być kilka istotnie różnych, a nawet nieskończenie wiele. Ważne jest, żeby sformułowanie wymagań nie ograniczało nadmiernie możliwości wyboru rozwiązania, ale też by wykluczało rozwiązania, w sposób oczywisty, nierozsądne. Jak się okazuje, jest to trudne w potocznym rozumieniu, czego dowodzą liczne przypadki nieporozumień pomiędzy odbiorcami, a projektantami systemów (nie tylko) informatycznych. Sformalizowanie wymagań - to problem znacznie trudniejszy.

Specyfikacja systemowa to, jak powiedzieliśmy wyżej, określenie zasadniczej koncepcji rozwiązania problemu, zadanego poprzez specyfikacje wymagań. Opis rozwiązania zależy w ogromnym stopniu od języka, którym się posługujemy. Powiedzmy więc tylko, że taki opis chcielibyśmy traktować poniekąd, jak program wysokiego poziomu, w którym - w pierwszym przybliżeniu - nie zwracamy uwagi na szczegóły i rygory realizacyjne, natomiast przywiązujemy szczególną wagę do wypełnienia wszystkich wymagań. Podejmujemy także zasadnicze decyzje ustalające: strukturę i hierarchie elementów projektowanego systemu, rodzaj komunikacji pomiędzy nimi itp.

Program (programy) w języku programowania (takim, jak np. C lub C++), to konkretna implementacja specyfikacji systemowej, biorąca pod uwagę docelowy system komputerowy i inne wymogi realizacyjne. Mamy więc tutaj do czynienia ze zbiorem wymagań implementacyjnych, np. dotyczących ograniczeń czasowych, różnych od tych z poziomu specyfikacji problemu. Niespełnienie tych wymagań przez implementację może powodować konieczność zmiany specyfikacji systemowej, a nawet niekiedy i wstępnych wymagań, dotyczących definicji problemu. Stąd iteracja etapów projektowania.

Mając określone "produkty" poszczególnych etapów, łatwo jest przewidzieć to, w czym oczekujemy pomocy i co chcielibyśmy automatyzować. O ile trudno wyobrazić sobie, że sformułowanie wymagań mogłoby automatycznie generować specyfikację systemową - konieczne są tutaj zasadnicze wybory i koncepcje, które nie mogą się obyć bez interwencji projektanta - to przejście od specyfikacji systemowej do konkretnej realizacji programistycznej wydaje się krokiem bliskim takiej (wspomaganej i, być może, niekompletnej) automatyzacji.

Podsumowując, metody i narzędzia, których oczekujemy to:

- bogate, a jednocześnie łatwo czytelne i "przyjazne", formalizmy (techniki) definiowania specyfikacji wymagań i specyfikacji systemowych oraz narzędzia wspomagające formułowanie poprawnych i spójnych specyfikacji (edytory tekstowe i graficzne, analizatory składni i semantyki, systemy wspomagające uzyskiwanie i dokumentację kolejnych wersji specyfikacji itp.);

- metody i narzędzia wspomagające weryfikację wymagań (sformułowanych na poziomie specyfikacji wymagań) przez specyfikacje systemowe;

- metody i narzędzia generowania kodu w docelowym języku programowania bezpośrednio z zadanej specyfikacji systemowej;

- metody i narzędzia wspomagające instalację, optymalizację i testowanie otrzymanego kodu w konkretnej konfiguracji komputerowej i programowej (system operacyjny, oprogramowanie komunikacyjne itp.). W dużym zakresie tego rodzaju narzędzia istnieją i są związane z konkretnym językiem programowania (kompilatory, debuggery, itp.) - w fazie uruchamiania potrzebne są jednak na ogół dodatkowe zabiegi (często wymagające interwencji człowieka, który system uruchamia) i specjalizowane biblioteki procedur potrafiące skompilowane części specyfikacji systemowej połączyć w jedną wykonywalną całość, włączoną do istniejącego środowiska.

Jeżeli mówimy o skomplikowanym oprogramowaniu systemowym (protokole komunikacyjnym, poważnej aplikacji itd.), to dodatkowe zabiegi - o których mowa w ostatnim punkcie powyżej - oraz w miarę kompletne testowanie w rzeczywistym środowisku, mogą poważnie wydłużyć i podrożyć proces uzyskania zadowalającej implementacji. Spełnienie wymagań implementacyjnych, np. wydajnościowych, jest, na ogół, bardzo trudne. Wymaga często wielokrotnego powrotu i zmian we wcześniejszych fazach projektowania. Wydaje się więc zasadne dążenie do skrócenia każdego cyklu, poprzez możliwość uzyskania estymacji parametrów implementacyjnych jeszcze przed właściwą implementacją (wykonywalną instalacją) oprogramowania. Myśl to nie nowa, a osiągnięcie tego celu można uzyskać modelując (specyfikując) projektowany system razem ze środowiskiem, w którym przewidywane jest jego wykonanie, a następnie analizując wymagania implementacyjne bezpośrednio w tym modelu za pomocą środków analitycznych lub symulacyjnych. Jeżeli przyjmiemy takie rozwiązanie za istotny element wyposażenia projektanta, to do powyżej wymienionych punktów dodamy:

- metody i narzędzia analizy wymagań implementacyjnych, np. czasowych i wydajnościowych, na poziomie specyfikacji systemowych, lub na poziomie pośrednim pomiędzy taką specyfikacją, a implementacją.

Współczesna inżynieria oprogramowania próbuje sprostać oczekiwaniom krótko scharakteryzowanym powyżej. Istnieje wiele modeli, formalizmów, metod i narzędzi. Technologiczny postęp, jaki dokonał się w ostanim dziesięcioleciu, pozwala realizować zamysły, o których nie można było marzyć jeszcze niedawno. Przykładem na to może być postęp jaki dokonał się w zastosowaniu tzw. metody sprawdzania (w) modelu (*model checking*), która służy automatycznej weryfikacji własności projektowanego systemu zadanych w postaci pewnych formuł logicznych. Metoda ta, do pewnego stopnia, spełnia naiwne marzenie, które można wysłowić następująco:

- "gdyby istniały niesłychanie szybkie i pojemne (w sensie pamięci) komputery, to dla każdego systemu (programu) można by sprawdzić zadaną własność, przeglądając jego pełny graf stanów".

Taka rzeczywiście jest zasadnicza myśl tej metody. Modelami, o których tutaj mowa, są zbiory (lub drzewa) obliczeń definiowane przez taki graf stanów. Skończoność zbioru stanów daje się (częściowo) uzasadnić, jeśli pamiętamy, że zbiór wartości, które możemy reprezentować w komputerze, jest zawsze ograniczony. Łatwo jednak zauważyć, że liczba stanów rośnie bardzo

szybko (wykładniczo) w stosunku do rozmiaru systemu, szczególnie jeśli mówimy o modelach dla systemów rozproszonych i współbieżnych. Czy więc metoda jest zupełnie nierealna? Okazuje się, że nie. Stosując wyrafinowane techniki reprezentacji danych i redukcji przestrzeni stanów, w połączeniu z niewiarygodnym postępem technologicznym, obserwowanym ostatnio, okazuje się, że można mieć nadzieję na praktyczne stosowanie tej metody. Dość powiedzieć, że udaje się weryfikować, w rozsądnym czasie, systemy o liczbie stanów sięgającej 10^{20} (zob. np. [2]). Dla metody weryfikacji w modelu naturalnymi przykładami testującymi były układy logiczne i proste protokoły komunikacyjne. Własności w postaci formuł logicznych (temporalnych, czy ogólniej modalnych) stosunkowo dobrze pasują do definicji wymagań w takich przykładach.

Nie będziemy tego tematu tutaj rozwijać i zainteresowanego czytelnika odsyłamy np. do książki [7], gdzie znajdzie potrzebne referencje i przykłady dotyczące protokołów. Takie przykłady mogą napawać pewnym optymizmem, chociaż metody formalne inżynierii oprogramowania dalekie są od powszechnego stosowania.

3. Metody formalne w projektowaniu protokołów komunikacyjnych

W krótkiej charakterystyce oczekiwanych metod i narzędzi, wspomagających wytwarzanie oprogramowania na początkowych jego etapach, skupiono uwagę na tych, które mają charakter uniwersalny, niezależny od dziedziny zastosowań. Jak słusznie zauważają różni autorzy (np. P. Zave w [4]), wydaje się dzisiaj zupełnie niemożliwe stworzenie metod i narzędzi równie pomocnych w produkcji oprogramowania we wszystkich dziedzinach. Ogólne zasady, powszechnie słuszne, nie zastąpią wiedzy i doświadczenia inżynierskiego, związanego z wybranemi zastosowaniami. Projektowanie protokołów potwierdza, naszym zdaniem, ten pogląd. Wiele czynników miało wpływ na to, że metody formalne dość wcześnie pojawiły się w projektowaniu protokołów. Nie bez znaczenia jest tutaj fakt, że ta dziedzina wymaga, jak mało która, kodyfikacji i normalizacji. Nie do przecenienia w tym kontekście jest rola bazowego modelu (*Basic Reference Model*) OSI (*Open Systems Interconnection*) [10], nad którym pracę w ISO (*International Standards Organization*) rozpoczęto w połowie lat siedemdziesiątych, a ostatecznie upowszechniono w formie normy w roku 1979. Za tym modelem nastąpiło szybkie normalizowanie protokołów różnego poziomu.

Wysiłek normalizacyjny spowodował, że w gronie specjalistów wytworzył się w miarę wspólny zestaw pojęć, terminów, notacji i sposobów na wyrażanie podobnych własności. W standardach komunikacyjnych stosuje się zestaw tych znanych środków, dobrze (na ogół) dostosowanych do sytuacji, co powoduje, że zapis norm, w dużej części przecież nie formalny, stanowi dobry punkt odniesienia z chwilą przejścia do definicji sformalizowanych. Model OSI ma tę właściwość, że sytuuje i definiuje protokół komunikacyjny - centralny obiekt swego zainteresowania - w kontekście otoczenia (warstwowego środowiska), w którym on będzie działał i jednocześnie określa jego wyabstrahowaną rolę w tym kontekście, a mianowicie usługę, którą protokół danej warstwy, wraz z kontekstem niższych warstw, oferuje użytkownikowi warstwy wyższej. Mamy więc w tym przypadku sytuację (z punktu widzenia metod inżynierii oprogramowania, w tym metod formalnych) koncepcyjnie niesłychanie klarowną: specyfikacja usługi protokolarnej pełni rolę specyfikacji wymagań, a sam protokół, sprzężony z niższymi warstwami, ma stanowić rozwiązanie spełniające te wymagania. Dodatkowym ułatwieniem na tym poziomie koncepcyjnym jest fakt, że protokoły skupiają uwagę na samej komunikacji (i jej jakości), nie wnikając w treść przesyłanej informacji. Obrazowo rzecz ujmując, w przesyłanych wiadomościach pomiędzy użytkownikami i w interfejsach międzywarstwowych interesuje nas, jak postępować, sprawdzać i uzupełniać informacje, podawane na kopercie listu, a nie to, co jest zawartością koperty. Wymagania, dotyczące treści i formy informacji na kopercie, mogą być skodyfikowane i znormalizowane, ponieważ ich interpretacja nie zależy od poszczególnego użytkownika. Protokół, a także usługa protokolarna, określone zostają w zasadzie poprzez wymianę tak znormalizowanych komunikatów.

Opisy protokołów i ich usług - znormalizowanych lub nie - w języku naturalnym (angielskim), stanowią podstawę specyfikacji wymagań. Można powiedzieć, że nic nie może ich całkowicie zastąpić na tym poziomie i że zawsze takie opisy będą dla człowieka (a więc projektanta) punktem odniesienia i źródłem rozumienia jakichkolwiek formalizmów. Stosuje się

tutaj także pewne specyficzne notacje, tablice i diagramy stanów i przejść oraz przepływu wiadomości w czasie MSC (*Message Sequence Charts*) [6]. Całkowita formalizacja wymagań potrzebna jest wtedy, gdy mamy możliwość ich weryfikacji (najchętniej automatycznie - np. poprzez sprawdzanie modelu, o którym wspomnieliśmy wcześniej) w tworzonym oprogramowaniu. Próby takiej formalizacji są czynione i najczęściej wykorzystuje się tutaj języki dla różnych odmian logiki temporalnej, tzn. takiej, w której można wyrazić własności wiążące zdarzenia obliczeń w ich przeszłości, teraźniejszości i przyszłości. Trzeba od razu jednak zaznaczyć, że formalizmy logik temporalnych tylko w pewnym stopniu wyrazić mogą ograniczenia czasowe, które chcielibyśmy móc reprezentować. Jednakże doświadczenia zdobyte przy wykorzystaniu np. systemu weryfikacji SPIN [2] są wielce obiecujące.

Prace normalizacyjne i pojawiające się przy nich trudności sprawiły, że stosunkowo wcześnie powstał w ISO pomysł zaprojektowania specjalnych języków dla formułowania standardów usług i protokołów - języków wolnych od niejednoznaczności i niejednorodności, dotychczas używanej mieszaniny języka naturalnego z tworzonymi ad hoc specjalnymi notacjami. Około 1980 roku powstała w ISO, w której podjęto prace nad dwoma takimi językami, reprezentującymi dwa różne podejścia do problemu specyfikacji: jednym (Estelle) opartym na rozszerzonym pojęciu (komunikujących się) automatów i drugim (Lotos), którego podstawę stanowi algebra (komunikujących się) procesów. Wcześniej znany był język SDL - znormalizowany przez CITT - którego model miał tę samą podstawę, co Estelle. Wszystkie trzy razem występują często pod wspólną nazwą FDTs (*Formal Description Techniques*) [13], chociaż ostatnio ta nazwa staje się generyczną dla wielu innych, nie znormalizowanych formalizmów.

W końcu lat osiemdziesiątych zakończono prace normalizacyjne nad tymi trzema językami (ostatnia wersja SDL w roku 1992). Równolegle, wokół tych języków, powstawały mniej lub bardziej prototypowe narzędzia, ułatwiające ich używanie. Okazało się jednak, że przypisywany początkowo cel powstania FDTs polegający na chęci stopniowego eliminowania języka naturalnego z opisów norm, nie da sie łatwo zrealizować, albo też nie jest to w istocie cel podstawowy. W coraz większym stopniu eksperymentowano z tymi językami, jak niegdyś z językami programowania wysokiego poziomu. Traktowano je jako takie, na które wygodnie jest tłumaczyć nieformalne zapisy norm i jednocześnie wykorzystać, jako podstawę do otrzymania prototypowych implementacji, których działanie można sprawdzić w środowisku rzeczywistym lub symulacyjnym. Stąd też postulowane przez nas usytuowanie tej klasy języków w kategorii specyfikacji systemowej dla odróżnienia zarówno od języków specyfikacji wymagań, jak też od powszechnie używanych języków programowania.

Jak powiedzieliśmy uprzednio, języki poziomu systemowego tylko wtedy będą praktycznie stosowane jeżeli dają sie przekształcić (skompilować) w wyrażenia języka programowania, takiego jak C. Taka kompilacja jest potrzebna nie tylko, gdy chcemy protokół rzeczywiście uruchomić w węźle sieci, ale także gdy chcemy go wykonywać, poprzez symulację w środowisku modelowanym. Jednocześnie, nie można wymagać, by taka kompilacja prowadziła do efektywnych imlementacji, ponieważ mechanizmy definicyjne na poziomie specyfikacji systemu nastawione są raczej na modelowanie, a nie wykonanie. Przy założeniu, że kompilator działa poprawnie, otrzymujemy jednak - w dużej części automatycznie - działający poprawnie prototyp protokołu.

Na ogół, o efektywności implementacji przekonać się możemy dopiero po zainstalowaniu otrzymanego oprogramowania w konkretnym środowisku komputerowym (sieciowym). Co można zrobić, by przewidywać jego efektywność wcześniej, na etapie projektowania? Klasyczną metodą w takim przypadku jest stworzenie modelu kolejkowego (*queuing networks*) dla środowiska implementacyjnego, zawierającego projektowany moduł (np. protokół) i poddanie go analizie istniejącymi na rynku narzędziami, specjalnie dostosowanymi do tego modelu. To podejście ma swoje zalety, ponieważ model kolejkowy i stworzone dla niego metody analizy mają ugruntowaną pozycję. Ma jednak i tę wadę, że nie wykorzystuje modelu, w którym powstała specyfikacja systemowa. Zmusza więc do pracy z dwoma modelami, których wzajemną relację ustala, na ogół, intuicja i doświadczenie projektanta(ów). Istnieją więc próby by istniejące narzędzia pracujące bezpośrednio na specyfikacjach w konkretnym języku (Estelle, SDL, Lotos itp.), takie jak symulator, uzupełnić specjalnymi funkcjami, na wzór tych przystosowanych do

modelu kolejkowego, które pozwalają na różne oceny wydajnościowe symulowanego systemu. Takie narzędzia zaprojektowano dla Estelle (EDT) [1] i w pewnym stopniu dla SDL (SDT) (zob. [6]). W tym pierwszym przypadku stworzone były i następnie używane w projekcie COP 62, o którym piszemy więcej poniżej.

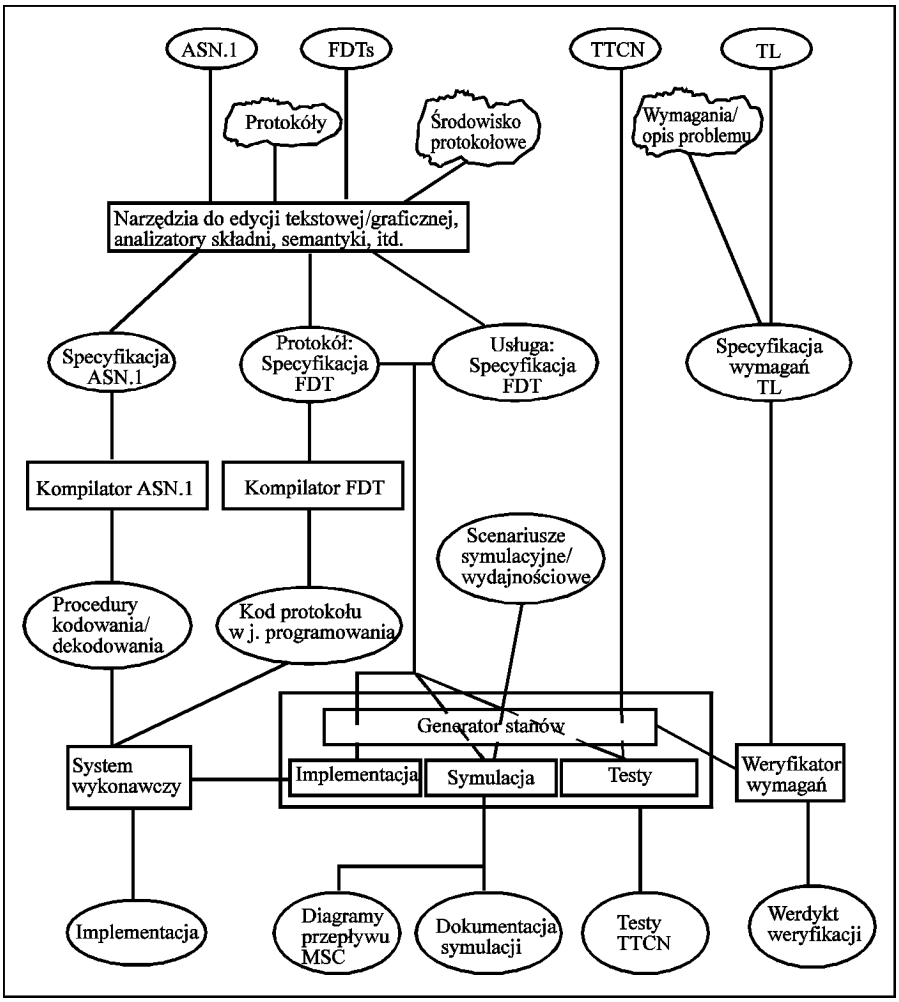
Proces projektowania i implementacji protokołów zawiera specyficzne zadania, których rozwiązanie wymaga zarówno formalizacji, jak i automatyzacji. Dwa takie specyficzne zadania charakteryzujemy poniżej.

Protokoły komunikacyjne służą wymianie informacji zrozumiałej dla obu stron nawet wtedy, gdy mówią one - stosując terminologię potoczną - innymi językami. Aby to osiągnąć, trzeba pewnej umowy, precyzującej sposób, w którym informacje są zapisywane i jednoznacznego systemu ich kodowania (dekodowania) w ciągi bitów, które ostatecznie przesyłane będą poprzez łącza fizyczne. Te wymagania wymuszają ustalenie ogólnie przyjmowanego standardu obu tych elementów: zapisu na poziomie struktury danych (*abstract syntax*) i na poziomie kodu binarnego (*concrete syntax*). Elementem łączącym te zapisy jest jednoznaczny systemu kodujący. Istnieje taki standard dla definiowania struktur danych, tzn. język ASN.1 (*Abstract Syntax Notation One* - cyfra na końcu ma wyrażać możliwość powstawania różnych, znormalizowanych notacji) [8] i sposób kodowania jego wyrażeń (*Basic Encoding Rules*) [9]. Jeżeli myślimy o pewnym środowisku wspomagającym projektowanie protokołów, to nie może zabraknąć narzędzi właściwego kodowania i dekodowania. Istnieją próby integracji takich narzędzi z narzędziami związanymi z implementacją języków specyfikacji systemowej.

Inną specyfiką definicji protokołów i usług telekomunikacyjnych jest dobre rozgraniczenie pomiędzy tym, co jest w każdej imlementacji danego protokołu konieczne, a tym, co pozostawione jest do decyzji projektanta konkretnej jego wersji (tzw. profilu). Testowanie protokołu, na poziomie jego specyfikacji systemowej, dotyczy sprawdzenia semantycznej poprawności przekształcenia normy, zapisanej nieformalnie w wyrażenie sformalizowane. Na poziomie implementacji mamy do czynienia z testami, które wiążą się z normalną praktyką programistyczną i wymagają sprawdzenia jakości kodu wyprodukowanego przez kompilator oraz tymi (*conformance tests*), które sprawdzają czy implementacja jest zgodna z deklarowanym profilem. Z naszego punktu widzenia ważne jest, że zaproponowano sformalizowaną notację TTCN (*Tree and Tabular Combined Notation*), służącą opisowi testów tego rodzaju i określono wymagania na ich powstawanie i stosowanie. Istnieją także poważne próby generacji testów w notacji TTCN bezpośrednio ze specyfikacji protokołu w wyżej wymienionych językach poziomu systemowego. Podobnie jak w poprzednim przypadku, stwarzając środowisko projektowania protokołów komunikacyjnych nie można pominąć zadania automatyzacji powstawania, reprezentacji i generacji testów zgodności (zob. np.[12]).

Wymienić można jeszcze inne specyficzne cechy i wymagania na specjalistyczne oprogramowanie w interesującej nas dziedzinie. Wydaje się jednak, że krótka charakterystyka, którą przeprowadziliśmy, przekonuje do tego, że poprzez ograniczenie się do projektowania protokołów komunikacyjnych otrzymujemy istotne informacje szczegółowe, które pozwalają mieć nadzieje, że metody formalne komputerowo wspomagane, mogą dać efekty, na które trudno jest liczyć, jeżeli mówimy o oprogramowaniu w ogóle, nie odnosząc się do specyficznej dziedziny zastosowań. Trzeba jednak jasno powiedzieć, że odległa jest jeszcze perspektywa stosowania takiego systematycznego podejścia w powszechnej praktyce.

Na zakończenie tego paragrafu pokazujemy na Rys. 3.1 schemat hipotetycznego, zintegrowanego systemu, wspomagającego projektowanie i implementację protokołów komunikacyjnych, który byłby niejako próbą połączenia formalizmów, metod i narzędzi, opisanych krótko powyżej. Ze znanych autorowi systemów jedynie kilka zawiera zaledwie fragmenty tego schematu o znacznym stopniu zintegrowania. Są to przede wszystkim narzędzia wokół języka SDL, np. SDT oferowany przez Telelogic ze Szwecji i francuski GEODE firmy Verilog (o narzędziach SDL zob. np. [6]) a także, w pewnym stopniu, EDT (*Estelle Development Toolset*) [1]. W wielu innych systemach rozwinięte są tylko pojedyncze elementy prezentowanego na rysunku schematu.



Rys. 3.1. Schemat zintegrowanego systemu projektowania protokołów.

4. Przykład zastosowania

Chcielibyśmy zakończyć te z natury rzeczy ogólne uwagi krótkim opisem etapów i wyników zakończonego już projektu, mającego na celu otrzymanie prototypowej implementacji protokołu XTP (*EXpress Transfer Protocol*), w wersji 4.0 [14], bezpośrednio i (prawie) automatycznie otrzymanej z jego formalnej specyfikacji systemowej, również stworzonej w ramach tego projektu. Mówimy tutaj o projekcie Komisji Europejskiej Copernicus 62 pod nazwą "*High Speed Communication Systems Supporting Multimedia Applications - Selected Issues*" i realizowanym przez sześciu partnerów z czterech krajów (IT PW, IPI PAN i Alcatel z Polski, INT z Francji, PUB z Rumunii i UTC ze Słowacji). Projekt trwał dwa lata i zakończył się w kwietniu 1997 roku. Podajemy tutaj odwołanie tylko do raportu końcowego [3] tego projektu, w którym można znaleźć dokładny opis jego wyników, listę raportów i ogólnie dostępnych publikacji na jego temat.

Pierwszym z zadań projektu było zainstalowanie na Politechnice Warszawskiej i Politechnice w Żilinie (Słowacja) szybkiej sieci typu FDDI, jako rzeczywistego środowiska, w którym przeprowadzono by końcowe eksperymenty z przesyłaniem informacji multimedialnych pomiędzy węzłami sieci z zainstalowaną w nich prototypową implementacją XTP, o której była mowa powyżej.

Specyfikacje wyjściowe protokołu powstały zarówno w Estelle jak i SDL. Tworzone były za pomocą narzędzi z pakietu EDT (częściowo uzupełnianych w trakcie trwania projektu) i SDT. Przewagą specyfikacji w Estelle był fakt, że jej pierwsza wersja istniała wcześniej. Zaletą

specyfikowania w SDL były rozbudowane pomoce pakietu SDT dla tworzenia opisu w formie graficznej i jego dokumentacji. Z powodu zbyt krótkiego czasu trwania projektu eksperyment, aż do uzyskania implementacji, kontynuowany był jedynie dla specyfikacji w wersji Estelle.

Specyfikacje w SDL, i w znacznie poważniejszym stopniu w Estelle, poddane zostały weryfikacji metodami symulacyjnymi (a więc poniekąd testowaniu na poziomie specyfikacji). Punktem odniesienia był dokument na temat protokołu [14] oraz równolegle prowadzone prace studyjne nad zaproponowaniem definicji usług XTP (*XTP service*). Na tym etapie struktura specyfikacji bardziej odpowiadała logicznej i funkcjonalnej konstrukcji samego protokołu niż potrzebom implementacji.

Prototypowa implementacja XTP została wygenerowana przy użyciu narzędzi EDT dla środowiska UNIX-owego. Eksperymenty przeprowadzono na stacjach roboczych typu SPARC 5/110.

Podstawowym założeniem projektowym była teza, że otrzymana (prawie) automatycznie implementacja z oryginalnej specyfikacji będzie cakowicie nieefektywna, w szczególności dla potrzeb ruchu multimedialnego. Stąd zrodziło się drugie ważne zadanie projektu: zaproponować i przetestować metodologię analizy wydajnościowej protokołu na poziomie jego specyfikacji, tak by jeszcze przed implementacją przewidzieć jego zachowanie, przy różnej intensywności ruchu w danym węźle i przy różnych obciążeniach pozostałych części sieci. Takie podejście wymagało wymodelowania całego środowiska, w którym implementacja protokołu jest przewidziana (w tym przypadku w Estelle). W szczególności w tym projekcie obejmowało to wymodelownie strumieni ruchu multimedialnego po stronie aplikacji i ruchu w sieci pomiędzy jej węzłami. Jako model ruchu multimedialnego przyjęto kombinację strumieni audio i video, przy czym dla tego drugiego wzorzec stanowił znany format MPEG.

Ogólne warunki metodologiczne, dotyczące powyższych wymagań, zostały sformułowane najpierw maksymalnie niezależnie od rodzaju FDT i jego narzędzi. Bardziej szczegółowy ich opis oraz dokumentacja przeprowadzonych analiz dotyczyła specyfikacji XTP w Estelle.

Ważne w tym eksperymencie było przekonanie się, że przeprowadzone symulacje można stosować (choć nie bez trudności) do rzeczywistych protokołów i uzyskać pewne informacje, dotyczące zachowania się implementacji we wczesnym etapie projektowania. W wyniku przeprowadzonych symulacji (oraz wcześniej znanych cech języka Estelle), pierwotna specyfikacja XTP w kolejnych iteracjach była zmieniana i dostosowywana do wymogów implementacyjnych. Ostatecznie otrzymano prototypową implemetację znacznie efektywniejszą (co nie znaczy, że efektywną) od tej, którą można by uzyskać ze specyfikacji w wersji oryginalnej. Co więcej, okazała się ona porównywalna efektywnością z dostępną implementacją XTP otrzymaną poprzez bezpośrednie programowanie w C. Uzyskano także cenne wskazówki, dotyczące stosowanej metodologii i jakości używanych narzędzi.

Podsumowując, wydaje się, że wyniki tego międzynarodowego projektu są zachęcające i wskazują, że praktyczne rezultaty, które można uzyskać, stosując formalne metody inżynierii programowania, są warte zainteresowania.

5. Uwagi końcowe

Niniejszy artykuł próbował usytuować metody formalne zarówno w kontekście historycznym, jak i w odniesieniu do współczesnych uwarunkowań. Celem tych uwag było wskazanie właściwego miejsca tych metod i rozważnej oceny możliwości ich stosowania. Skupienie uwagi na wykorzystaniu metod formalnych w praktyce projektowania protokołów komunikacyjnych pokazuje, że szczególne potrzeby każdej dziedziny i uwzględnie wiedzy zgromadzonej latami przez specjalistów prowadzi do metod bardziej przydatnych i efektywnych. Z drugiej strony, obserwacja rozwoju inżynierii oprogramowania w tej szczególnej dziedzinie potwierdza ogólne tendencje. W coraz większym stopniu uwaga projektantów dużych systemów skupia się na początkowych fazach projektowania - one są bowiem decydujące dla jego żywotności i jakości - a więc elementach, które decydują także o efektach ekonomicznych.

Nawiązując na koniec do tytułu tej pracy, można powiedzieć, że mity, jak zwykle, nie znajdują potwierdzenia w rzeczywistości, ale jednocześnie trzeba jeszcze wiele wysiłku, żeby rzeczywistość nie ulegała mitologii.

Literatura

[1]. S. Budkowski, Estelle Development Toolset, Computer Networks and ISDN Systems, Vol. 25, No 1, 1992

[2]. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic Model Checking: 10^{20} states and beyond, Information and Computation, 98(2), 1992

[3]. COP 62, High Speed Communication Systems Supporting Multimedia Applications - Selected Issues, Final Project Report COP 62/5, EC INCO Programme, 1997

[4]. J-P. Courtiat, P. Dembiński, G.J. Holzmann, L. Logrippo, H. Rudin, P. Zave, Formal methods after 15 years: Status and trends, Computer Network and ISDN Systems, Vol. 28, No 13, 1996

[5]. M. Flasiński, Wstęp do analitycznych metod projektowania systemów informatycznych, WNT Warszawa, 1997

[6]. O. Haugen (ed.), SDL and MSC, Special Issue, Computer Networks and ISDN Systems, Vol. 29, No 13, 1996

[7]. G. J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall International, 1991

[8]. ISO 8824, Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), Geneva 1987

[9]. ISO 8825, Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), Geneva 1987

[10]. ISO 7498, Information processing systems - Open Systems Interconnection - Basic Reference Model, Geneva 1984

[11]. M. Piechówka, S. Szejko, CASE, jaki jest, każdy widzi, Informatyka 4, 1998

[12]. O. Rafiq, A. R. Cavalli (eds.), Protocol Testing, Special Issue, Computer Networks and ISDN Systems, Vol. 28, No 1, 1996

[13]. K. J. Turner (ed.) Using Formal Description Techniques. An Introduction to Estelle, Lotos and SDL, Wiley 1993

[14]. XTP Forum, Xpress Transport Protocol Specification, XTP Revision 4.0, 1995