



Formalna weryfikacja w procesie projektowania systemów informatycznych. Projekt COSMA

Jerzy Mieścicki

Instytut Informatyki, Politechnika Warszawska
00 665 Warszawa, ul. Nowowiejska 15/19
email: J.Miescicki@ii.pw.edu.pl

Abstract. W artykule omawia się korzyści i trudności, jakie wiążą się z próbami przzerucenia pomostu między praktyką projektowania systemów a wykorzystaniem metod formalnych (zwłaszcza model checkingu) w procesie projektowania. Jako przykład, przedstawia się pokrótce badania nad środowiskiem programowym COSMA, prowadzone w Instytucie Informatyki Politechniki Warszawskiej.

1 Wstęp

1.1 Metody formalne a proces projektowania

Jednym ze zjawisk, które dają się zaobserwować we współczesnej informatyce jest dysproporcja między rozwojem badań nad formalnymi metodami opisu i analizy systemów a stosunkowo słabym ich przenikaniem do praktyki projektowania.

Istotnie, w środowiskach naukowych i akademickich zainteresowanie badaniami w tej dziedzinie w ostatniej dekadzie wyraźnie wzrosło. Można ocenić, że co roku około dwudziestu międzynarodowych konferencji jest poświęconych w całości lub przynajmniej w znacznej części metodom formalnym. Dość wspomnieć periodyczne konferencje FME (*Formal Methods Europe*), FORTE (*IFIP International Conference on Formal Techniques for Networked and Distributed Systems*), czy multi-konferencję ETAPS (*European Joint Conferences on Theory and Practice of Software*, w 2003 roku w Warszawie). Wydawane są czasopisma naukowe, jak *Formal Methods in System Design* oraz *Formal Aspects of Computing Science*. Obok tych wyspecjalizowanych konferencji i czasopism, tematyka metod formalnych w projektowaniu pojawia się często w informatycznych czasopismach naukowych o szerszym światowym obiegu. Powstają wartościowe monografie na ten temat (np. [5, 2]) oraz narzędzia programowe, przewidziane do formalnej weryfikacji projektów sprzętowych i software'owych. Zainteresowany czytelnik znajdzie więcej danych na ten temat w witrynie [3].

Z drugiej strony, większość rezultatów i narzędzi z dziedziny metod formalnych jest mało znana poza środowiskiem naukowym i akademickim, chociaż wszyscy badacze starają się, by ich rezultaty miały charakter metod o wartości przemysłowej (*industrial strength formal methods*). Metody formalne z kręgu weryfikacji modelowej (*model checking*) są jeszcze stosunkowo często stosowane przy projektowaniu rozwiązań sprzętowych (*hardware*), jednakże praktyczna formalna weryfikacja projektów *software*'owych należy do rzadkości. Z tych zapewne powodów mówi się, że podczas gdy błąd w komercyjnie dostępnym układzie scalonym jest sensacją na światową skalę — błąd w równie powszechnym produkcie software'owym jest nie wartą uwagi codziennością.

Do najszerzej stosowanych w praktyce sposobów zwiększania zaufania co do poprawności projektów software'owych należą: testowanie i próbna eksploatacja rzeczywistej lub symulacyjnej wersji produktu. Jednakże, przy pomocy testów można sprawdzić, czy system w zasadzie *robi to, co powinien* robić, nie można natomiast sprawdzić, czy *nie robi tego, czego nie powinien* (np. zawieszać się). Również próbna, nawet długotrwała, eksploatacja systemu (zwłaszcza współbieżnego) nie gwarantuje, że akurat w jej trakcie pojawią się losowe koincydencje zdarzeń, prowadzące do ujawnienia się błędu synchronizacji czy koordynacji między współbieżnymi komponentami. Metody formalne usiłują *dowieść* istnienia (nieistnienia) błędu, nie zaś wytropić go eksperymentalnie.

Również komercyjnie oferowane narzędzia CASE i środowiska programistyczne, przewidziane do wspomagania produkcji software'u, praktycznie nie zajmują się formalną analizą poprawności zachowań projektowanych systemów. Koncentrują się raczej na składniowej poprawności software'u, zgodności jego konstrukcji z przyjętym paradygmatem programowania (np. strukturalnego czy obiektowego), a także na wspieraniu testowania i zarządzania samym procesem projektowania (praca zespołowa, kontrola wersji, dokumentacja itp.). Oferowane współcześnie sposoby specyfikacji zachowań systemu (np. UML [15]) są

ukierunkowane bardziej na doraźną wygodę użytkownika, niż na formalną poprawność projektu. Jedyne nieliczne (i nie tak powszechnie używane) środowiska tego typu (jak np. EDT, oparte na zastosowaniu języka Estelle [17, 18]), oferują formalnie zdefiniowaną semantykę stosowanych konstrukcji programistycznych.

Znaczna część trudności ma z pewnością źródła w niedoskonałości samych metod i narzędzi do formalnej weryfikacji, tworzonych w środowiskach naukowych i akademickich. Niezależnie od tego, stosowanie ich w praktyce wymaga od projektanta po pierwsze opanowania nowego zestawu pojęć i samej ‘technologii’ weryfikacji, po drugie — opisywania zarówno systemu, jak właściwości podlegających weryfikacji — w formalny sposób wygodny dla danego narzędzia czy metody, lecz obcy dla projektanta-praktyka.

Tym zapewne można tłumaczyć małą praktyczną znajomość metod opartych np. na notacji Z, VDM, czy B-method, a także technik i narzędzi formalnych z kręgu dowodzenia twierdzeń (*theorem proving*), jak np. PVS, HOL czy Larch, które wymagają sporej kultury matematycznej. Większym zainteresowaniem zdają się cieszyć metody skończenie stanowe, w szczególności weryfikacja modelowa (*model checking*, [1, 2, 5, 8]). Jej krótkiej charakterystyce poświęcony jest sekcja 1.2.

1.2 Zasady model checkingu

W najogólniejszym zarysie, weryfikacja modelowa (model checking) polega na tym, że:

- zachowania komponentów systemu opisuje się w postaci pewnego typu automatów skończonych (o skończonej liczbie stanów i etykietowanych przejściach między nimi),
- z opisu zbioru komponentów, przy odpowiednich założeniach co do zasad ich synchronizacji (np. model synchroniczny, asynchroniczny, przepłotowy, ...) otrzymuje się (oczywiście również skończony, choć może bardzo wielki) graf stanów (lub funkcję przejść) całego systemu, opisujący wszystkie możliwe zachowania modelu,
- właściwości podlegające ewaluacji są zadawane formalnie, zazwyczaj w postaci formuł pewnej logiki temporalnej (*temporal model checking*) lub automatu Büchiego.
- ewaluacja wymagań co do poprawności zachowań systemu polega na *wyczerpującej inspekcji* grafu stanów osiągalnych (lub funkcji przejść) systemu.

Podstawową zaletą tego podejścia jest fakt, że wszystkie pytania o zachowania systemu opisanego skończonym modelem są rozstrzygalne, jeśli odsunąć problem złożoności obliczeniowej algorytmów przeszukiwania grafu stanów systemu oraz algorytmu samego otrzymywania tego grafu. Co więcej, algorytmy inspekcji grafu, raz opracowane przez twórców metody — nie wymagają potem inwencji użytkownika, który musi jedynie nauczyć się formułować pytania i odczytać odpowiedź. Ważne, że w przypadku odpowiedzi negatywnej model skończenie-stanowy może dostarczyć kontrprzykładu, to znaczy wymienić ścieżkę w grafie, prowadzącą do miejsca (stanu lub podgrafu), w którym wymogi poprawności zostały naruszone. Analiza kontrprzykładów ułatwia identyfikację i usuwanie błędów.

Inna właściwość, predysponująca model checking do roli forpoczty metod formalnych w praktyce, polega na tym, że każdy komponent jest oddzielnym automatem, a opis jego zachowania (w kategoriach stanów i przejść) odpowiada intuicji projektanta i ułatwia późniejsze przekształcenie go sekwencyjny wątek, proces, automat sprzętowy itd. Tej właściwości nie mają np. systemy tranzycyjne czy sieci Petriego.

Z drugiej strony, modele skończenie-stanowe praktycznie nie nadają się (bez dodatkowych mechanizmów abstrakcji) do opisu abstrakcyjnych typów danych i operacji na nich: dopuszczają właściwie jedynie dane binarne i krótkie liczby całkowite. Żle sobie radzą również z innymi aspektami zachowań systemu, np. z dynamicznie zmienną liczbą procesów (wątków, klientów itp.) czy istnieniem buforów (kolejek) o nieograniczonej długości. Dlatego naturalnym i pierwotnym terenem zastosowania modeli skończenie-stanowych było projektowanie i weryfikacja układów sprzętowych ([4]) oraz *control-dominated systems*, to znaczy takich systemów, w których analizuje się poprawność sterowania i koordynacji między komponentami, nie zaś poprawność operacji na danych.

Problemem, z jakim walczą twórcy wszystkich metod i narzędzi skończenie stanowych jest *wykładnicza eksplozja modelu* wraz ze wzrostem liczby komponentów. Dlatego praktyczny sukces uwarunkowany jest opracowaniem skutecznych metod hierarchizacji opisu automatowego i redukcji modelu, a także metod badania modelu ‘częściami’ (*compositional model checking*).

Dla celów weryfikacji modelowej opracowano wiele narzędzi (model checkerów). Do najczęściej cytowanych w literaturze należą SPIN ([6], [7]), SMV ([8], [9]) i Cospan (lub jego komercyjna wersja FormalCheck, [10]). Kilkadziesiąt innych implementowano dla celów akademickich i badawczych ([3]).

Niżej zostanie opisane środowisko programowe COSMA ([16]) powstałe w Instytucie Informatyki Politechniki Warszawskiej w toku badań nad weryfikacją modelową. Jest ono oparte na modelu automatowym Concurrent State Machines (CSM, [25]). Model CSM wydaje się szczególnie dobrze nadawać do opisu i badania zachowań *współbieżnych systemów reaktywnych*, to znaczy takich systemów, w których szczególnie ważna jest wzajemna kooperacja zarówno między współbieżnie działającymi komponentami systemu, jak między systemem a jego otoczeniem. Pozwala on na wyrażenie dwóch form współbieżności: jednoczesnego występowania zdarzeń komunikacyjnych (formalnie: symboli alfabetu wejściowego) i jednoczesnego wykonywania przez komponenty swoich akcji. Nie zakłada się żadnego mechanizmu przeplatania akcji (*interleaving*) ani szeregowania symboli wejściowych automatu. Komponenty systemu są więc względem siebie asynchroniczne.

2 Automaty współbieżne CSM i środowisko COSMA

2.1 Automaty CSM

Technikę opisu zachowań systemu przy pomocy automatów współbieżnych CSM (Concurrent State Machines, [25]) opiszemy nieformalnie na przykładzie prostego systemu o schemacie strukturalnym jak na Rys. 1. Przyjmijmy, że Nadawca przygotowuje i stopniowo wkłada do bufora kolejne porcje danych (*put*). W chwili, gdy liczba danych w buforze osiągnie N_1 - Nadawca zatrzymuje się i czeka, aż Odbiorca przetworzy całą zawartość bufora, opróżni bufor (*reset*) i potwierdzi koniec swej pracy sygnałem potwierdzenia *ack*. Wówczas Nadawca znów podejmuje zapełnianie bufora itd. Załóżmy, że Odbiorca śledzi stan zapełnienia bufora i samoczynnie podejmuje przetwarzanie, gdy liczba porcji danych staje się równa N_2 . Oczywiście, system działa poprawnie, jeśli $N_1 = N_2$. Dla zilustrowania konsekwencji błędnego zachowania systemu, rozpatrzmy jednak przypadek $N_1 > N_2$, tak, jak gdyby na skutek nieporozumienia między projektantami Nadawcy i Odbiorcy - Odbiorca rozpoczął przetwarzanie zbyt wcześnie.

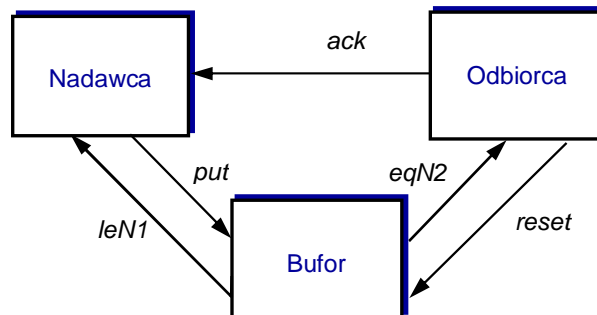


Fig. 1. Schemat strukturalny przykładowego systemu

W każdym momencie, Bufor (początkowo pusty) sygnalizuje Nadawcy, czy liczba danych jest mniejsza od N_1 (wyjście *leN1*), a Odbiorcy, czy liczba przebywających w nim porcji danych jest równa N_2 (wyjście *eqN2*). Otrzymując na swym wejściu symbol *put* (od Nadawcy) Bufor zwiększa swój stan, zaś symbol *reset* (od Odbiorcy) powoduje wyzerowanie bufora.

Zachowanie Nadawcy i Odbiorcy, opisane w postaci automatów współbieżnych CSM, jest przedstawione na Rys. 2. Modelem Nadawcy jest graf, składającym się z czterech węzłów (stanów) i siedmiu skierowanych krawędzi. *Think* jest stanem początkowym. W dolnej części prostokąta stanu widnieje zbiór symboli wyjściowych produkowanych w danym stanie. Tak więc np. w stanie *Write*, Nadawca produkuje jednoelementowy zbiór symboli $\{put\}$, w stanach *Think*, *Decide* i *Wait* produkowany zbiór symboli jest pusty, Odbiorca w stanie *Conclude* produkuje zbiór dwuelementowy: $\{reset, ack\}$. Symbole te są produkowane *jednocześnie i przez cały czas* przebywania w danym stanie.

Krawędzie grafu są etykietowane nie symbolami alfabetu wejściowego (jak to ma miejsce w 'klasycznych' automatach skończonych) lecz *boolowskimi formułami*. Argumentami formuł są symbole produkowane przez inne automaty ('wewnątrz') systemu lub jego otoczenie ('zewnątrz'). Operatory $+$, $*$, $!$ oznaczają (odpowiednio) boolowską sumę, iloczyn i negację. Formuła oznacza warunek konieczny do tego, by dana krawędź była 'potencjalnie aktywna' (*enabled*). Tak więc na przykład formuła $!leN1$ na

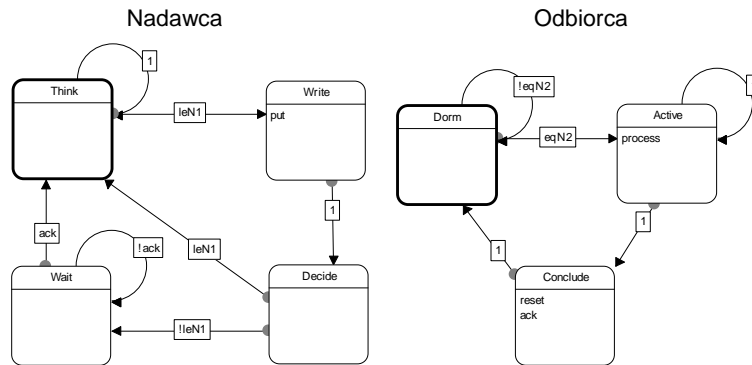


Fig. 2. Modele CSM Nadawcy i Odbiorcy

krawędzi od *Decide* do *Wait* oznacza, że Nadawca może przejść od stanu *Decide* do *Wait* jeśli *nie przyszedł* symbol *leN1* itd.

Formuła **1** reprezentuje funkcję boolowska, która jest *true* niezależnie od aktualnego stanu wejścia maszyny. Krawędzie etykietowane przez **1** opisują zatem zachowania *spontaniczne*. Tak więc na przykład, wchodząc do stanu *Write*, Arbiter produkuje symbol wyjściowy *put*, a następnie spontanicznie przechodzi do stanu *Decide*. Formuła **0** (zawsze *false*) nie występuje w grafie, gdyż krawędzie, które nie są nigdy *enabled* są po prostu z grafu usuwane.

Jeśli dwie krawędzie wychodzące z tego samego stanu są jednocześnie *enabled*, dokonuje się wyboru jednej krawędzi aktywnej (*active*), która wskazuje następny stan. Zakłada się, że wybór jest niedeterministyczny i uczciwy (*fair*), co w praktyce oznacza, że każde potencjalnie aktywne przejście będzie kiedyś wykonane. Na przykład, Odbiorca w stanie *Active* może — niedeterministycznie i spontanicznie — albo pozostać w tym stanie, albo przejść do *Conclude*. Praktycznie oznacza to, że w stanie *Active* Odbiorca przebywa przez nieokreślony długi, ale skończony czas¹. Podobnie niedeterministyczne jest zachowanie Nadawcy w stanie *Think*.

W modelu CSM *przejściami* (tranzycjami) nazywane są jedynie krawędzie łączące dwa *różne* stany. Automat w każdej chwili przebywa więc w dokładnie jednym stanie, a przejścia są natychmiastowe (w zerowym czasie). Natomiast t. zw. ‘ucha’ (krawędzie od stanu z powrotem do tego samego stanu) opisują warunki *trwania* w danym stanie.

Model Bufora (Rys. 3) ilustruje pewną możliwość abstrakcji. Zbiór liczb całkowitych reprezentujących liczbę danych w Buforze można podzielić na pięć klas abstrakcji, odpowiadających stanom *B1...B5*. W każdym ze stanów produkowane są (lub nie) odpowiednie symbole *leN1* lub *eqN2*. Warunkiem przejścia do ‘wyższego’ stanu jest pojawienie się symbolu *put* (od Nadawcy), symbol *reset* (od Odbiorcy) sprowadza Bufor do stanu *B1*.

Zakłada się, że komponenty systemu (tu: Nadawca, Bufor i Odbiorca) współistnieją we wspólnym środowisku komunikacyjnym. Rozpowszechnia ono do wszystkich komponentów (natychmiast i bezbłędnie) mnogościową sumę zarówno zbiorów symboli wyjściowych produkowanych przez wszystkie komponenty, jak zbiorów symboli przychodzących do systemu z jego zewnętrznego otoczenia². Ta mnogościowa suma jest natychmiast odbierana przez wszystkie komponenty, które traktują ją jak aktualny zbiór swoich symboli wejściowych. Każdy z komponentów (asynchronicznie i niezależnie od innych) ewaluje formuły boolowskie na krawędziach wychodzących z jego aktualnego stanu i zachowuje się zgodnie z opisanymi wyżej zasadami. Nie nakłada się żadnego ograniczenia na jednoczesność występowania symboli (zdarzeń, sygnałów itd.) ani jednoczesność przejść (jak np. *interleaving*). Model CSM jest zatem współbieżny, asynchroniczny i koincydencyjny.

Globalne działanie całego systemu jest iloczynem (produktem) zachowań komponentów. Opracowano sprawny algorytm wyznaczania produktu automatów CSM [25], który prowadzi do otrzymania grafu stanów osiągalnych systemu. Operacja mnożenia (\otimes) jest przemienne i łączna. Produkt kilku komponentów jest z powrotem *pojedynczą* maszyną CSM. Ważne, że formuły boolowskie przypisane krawędziom produktu odwołują się wyłącznie do obecności (nieobecności) symboli przychodzących do

¹ W podstawowym modelu CSM nie specyfikuje się ograniczeń czasowych. ‘Czasowa’ wersja modelu, przewidziana do model checkingu, jest w trakcie opracowywania. Do celów *symulacji* (choć nie model checkingu) zachowania systemu w czasie opracowano rozszerzony model ECSM, patrz punkt 2.2

² Omawiany przykładowy system nie odbiera sygnałów z otoczenia

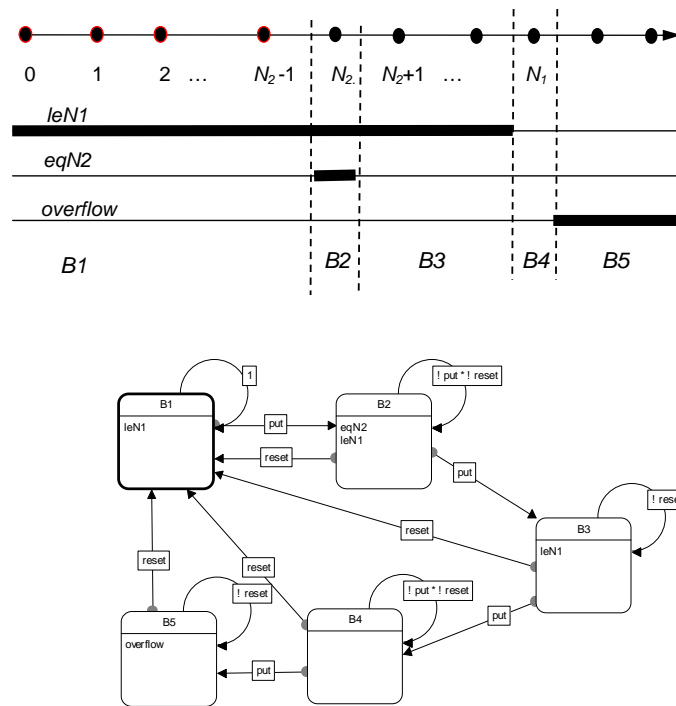


Fig. 3. Model CSM Bufora

systemu z zewnątrz, natomiast symbole, którymi komponenty komunikują się *wewnątrz* systemu — nie występują w tych formułach.

Produkt Nadawca \otimes Bufor \otimes Odbiorca (Rys. 4) ujawnia, iż system działa źle. Szarym kolorem zaznaczone są dwa stany zakleszczenia (*deadlock*), po osiągnięciu których system pozostaje bez możliwości wyjścia. Ponadto, w niektórych stanach symbole *put* i *process* występują jednocześnie, co oznacza, że zachowania Nadawcy i Odbiorcy są źle skoordynowane. Każda ścieżka w grafie stanów osiągalnych, prowadząca od stanu początkowego do takiego stanu niepożądanego jest tzw. kontrprzykładem (*counterexample*). Analiza kontrprzykładu pozwala na identyfikację (i w konsekwencji skorygowanie) błędów projektowego.

Powyższy przykład ma oczywiście charakter poglądowy. Jego celem było nieformalne wprowadzenie w tematykę weryfikacji z użyciem automatów CSM, a zwłaszcza zilustrowanie faktu, że zasady opisywania zachowań przy użyciu automatów CSM, interpretacja pojęć: system, produkt systemu itd. — wydają się naturalne i bliskie intuicji każdego, kto zna pojęcie grafu, stanu, przejścia, symbolu elementarnego i formuły boolowskiej. Graf z Rys. 4 ma 24 stany, można go narysować i zanalizować gołym okiem. W praktyce weryfikacji należy się liczyć z grafami osiągalności o rozmiarach setek tysięcy czy milionów stanów. Do tworzenia i analizy grafu stanów osiągalnych systemu są więc niezbędne odpowiednie algorytmy i narzędzia programowe. W przypadku modelu opartego na automatach CSM rolę tę odgrywa środowisko programowe COSMA.

2.2 Środowisko COSMA

Poglądowy schemat środowiska COSMA ([16]) jest przedstawiony na Rys. 5. Centralną jego część stanowi repozytorium, w którym (w postaci plików tekstowych w języku CXL) przechowywane są opisy modeli CSM. Język CXL został opracowany dla celów reprezentacji modeli CSM oraz ECSM w oparciu o XML. Funkcje sterowania całością narzędzia pełni (nie pokazany na schemacie) nadrzędny moduł sterujący, który umożliwia:

- utworzenie i nazwanie aktualnej przestrzeni roboczej (*workspace*),
- tworzenie, usuwanie i edycję projektów w tej przestrzeni,
- wywoływanie pozostałych modułów i komunikację między nimi.

Funkcje pozostałych podstawowych modułów są następujące:

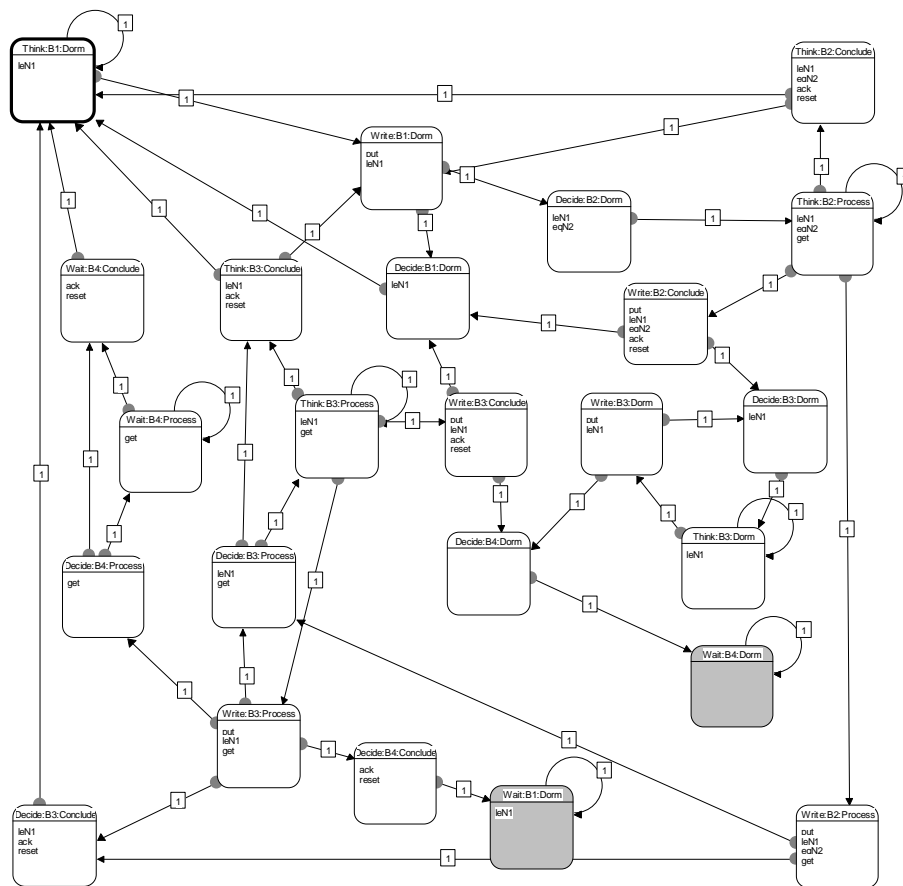


Fig. 4. Graf stanów osiągalnych systemu (Produkt Nadawca \otimes Bufor \otimes Odbiorca)

- *Grapher* umożliwia projektantowi (graficzne) tworzenie i edycję modeli CSM komponentów systemu. Modele są zapamiętywane w formacie CXL lub eksportowane do formatu emf.
- *Product Engine* tworzy reprezentację komponentów w postaci BDD i wykonuje operację mnożenia (\otimes) komponentów tworzących jeden projekt. Wynik ma postać BDD, może być także przetworzony na postać tekstową CXL,
- *TempoRG* przyjmuje od projektanta wymagania wyrażone w postaci wyrażeń logiki temporalnej QsCTL [23] [26] i przeprowadza ich ewaluację w grafie stanów osiągalnych (produkcie) dostarczonym przez *Product Engine*,
- *Cntrexample Editor* produkuje w postaci graficznej lub tekstowej kontrprzykłady, dostarczane przez *TempoRG* w przypadku negatywnego rezultatu ewaluacji,
- *Reduction Engine* przeprowadza redukcję wskazanego produktu, dostarczonego przez *Product Engine* lub pobranego z repozytorium.

Narzędzie COSMA wspiera również rozszerzony model CSM (Extended CSM, ECSM, [24]). Nie jest on modelem skończenie-stanowym i nie podlega model-checkingowi, jest natomiast wykonywany w drodze symulacji i może służyć do ewaluacji wydajności systemu (moduł *ECSM Simulator*, ([22])). Rozszerzenia w stosunku do modelu CSM są następujące:

- istnieje możliwość definiowania dowolnych zmiennych lokalnych dla komponentu lub globalnych dla projektu,
- zarówno stanom, jak przejściom komponentów CSM można przyporządkowywać akcje, będące po prostu sekwencyjnymi programami w C,
- formułom boolowskim na krawędziach grafów CSM można dodawać czynniki odpowiadające wyrażeniom boolowskim na zmiennych,
- symbole generowane w stanach mogą być uzupełnione o atrybuty (modelowanie zawartości komunikatów),

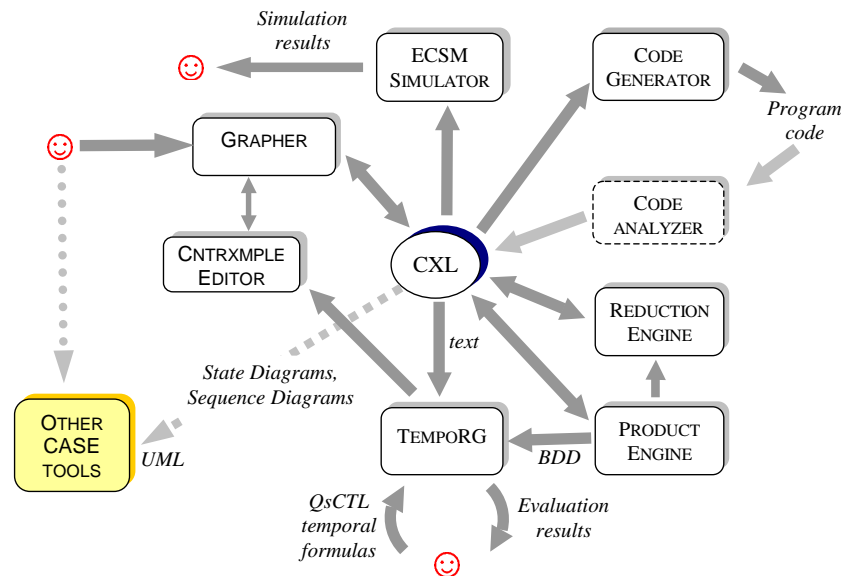


Fig. 5. Ideowy schemat środowiska programowego COSMA

- dla celów badań wydajnościowych, stanom można przypisywać czasy (deterministyczne lub losowe czasy przebywania w stanie), zaś rozejściom niedeterministycznym - prawdopodobieństwa.

Do modułów środowiska COSMA należy również *Code Generator* — generator kodu programu z modułu ECSM, istniejący w nieopublikowanej wersji eksperymentalnej. Moduł *Code Analyzer* był z kolei planowany jako narzędzie produkujące modele CSM komponentów zapisanych w postaci programów w jęz. Java. Wejściem dla modułu jest nie sam program, lecz jego odpowiednik w postaci pośredniej (t. zw. BIR) produkowanej przez środowisko Bandera ([13], [14]). Eksperymentalna wersja tego modułu [21], choć działa, nie zachęca jednak do kontynuacji: wydaje się, że skończenie stanowe modele można skutecznie produkować dla bardzo ograniczonego podzbioru Javy.

3 Podsumowanie

Próba przerzucenia pomostu między praktyką projektowania a formalną weryfikacją musi wiązać się przede wszystkim z określeniem miejsca formalnej weryfikacji w procesie projektowania. Dotychczasowe doświadczenia skłaniają do poglądu, że zalety model checkingu ujawniają się najlepiej na wczesnym etapie projektowania. Analiza koncepcyjnego szkieletu systemu, uwzględniającego koordynację i komunikację między głównymi współbieżnymi komponentami może ujawnić istnienie błędów komunikacji i synchronizacji, które (po usunięciu) nie będą propagowały się do dalszych etapów projektowania. Tak zweryfikowane automatowe modele komponentów mogą następnie posłużyć jako wzorce do szczegółowej implementacji w postaci programów czy układów sprzętowych.

Ze względu na pożądaną 'przyjazność' metodologii, w pracach nad narzędziami do model checkingu należałoby przywiązywać większą wagę do ich współpracy z powszechnie znanymi narzędziami CASE i środowiskami programowania. W szczególności, warto wykorzystać podobieństwa między modelami skończenie stanowymi a diagramami stanów, sekwencji, aktywności i kooperacji (UML) lub statecharts Harela [11], [12]. Eksport i import tego typu specyfikacji między narzędziem CASE a model checkerem takim, jak COSMA, przybliżyłyby możliwość formalnej weryfikacji projektu na wczesnym etapie projektowania.

Automaty współbieżne CSM i środowisko COSMA wydają się obiecujące jako koncepcyjna i narzędziowa podstawa tego typu prac. Przemawiają za tym:

- koncepcyjna prostota modelu komponentów systemu,
- graficzny tryb definiowania zachowań komponentów (zgodnie ze współczesnymi tendencjami, por. choćby UML),
- proste zasady komunikacji wewnątrz systemu (asynchroniczność, jednoczesność, brak domniemanego 'demonia' przepłatającego lub szeregującego zdarzenia i akcje),

- istnienie podstawowych narzędzi do otrzymywania i analizy grafu stanów osiągalnych systemu, a także symulacji modelu ESCM.

Oczywiście, konieczny jest dalszy rozwój metodologii, zwłaszcza zaś:

- opracowanie metod hierarchicznej specyfikacji automatów CSM,
- wsparcie dla algorytmów eksportu/importu automatów CSM z/do UMLowych diagramów stanów, sekwencji, aktywności i kooperacji,
- wprowadzenie ograniczeń czasowych do modelu CSM (Timed CSM),
- modelowanie operacji na danych,
- opracowanie specyficznych dla CSM algorytmów redukcji, pozwalających złagodzić efekt wykładniczej eksplozji modelu,

Tak ukierunkowane badania są w toku. Dotychczasowe rezultaty są zachęcające [28, 29, 27]. Niezależnie jednak od ich wartości poznawczej, do szerszego stosowania opisanej metodologii w praktyce najbardziej zachęciłyby przykłady, *case studies*, które mają największą siłę przekonywania.

References

1. E. M. Clarke, O. Grumberg, D. A. Peled: *Model Checking*, MIT Press, 2000.
2. B. Berard (ed.) et al.: *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer Verlag, 2001,
3. <http://archive.museophile.sbu.ac.uk/formal-methods.html>
4. T. Kropf: *Introduction to Formal Hardware Verification*, Springer Verlag, 1999.
5. D. A. Peled: *Software Reliability Methods*, Springer Verlag, 2001,
6. G. J. Holzmann: *The Model Checker SPIN*, IEEE Trans. on SE, Vol. **23**, No. 5 (May 1997), p. 279–295.
7. *SPIN*: <http://spinroot.com/spin/whatispin.html>
8. K. L. McMillan: *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
9. *SMV*: <http://www-2.cs.cmu.edu/modelcheck/smv.html>
10. *FormalCheck*: www.cadence.com/datasheets/formalcheck.html
11. Harel D., *StateCharts: A visual formalism for complex systems*, Science of Computer Programming, **8**, p. 231–274, 1987.
12. Harel D. et al.: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*, IEEE Transactions on Software Engineering, **16** (4), p. 403–414, April 1990.
13. J. Hatcliff, M. Dwyer: *Using the Bandera tool set to model-check properties of concurrent Java software*, Proc. CONCUR 2001, 2001, pp. 39–58.
14. Bandera: www.cis.ksu.edu/santos/bandera
15. *Unified Modeling Language*, <http://www.rational.com/uml>
16. *COSMA*, www.ii.pw.edu.pl/cosma/
17. S. Budkowski et al.: *The Estelle Development Toolset*, Institut National des Télécommunications, Evry, France, 1998, <http://www-lor.int-evry.fr/edt>
18. *Information Processing Systems. Estelle: A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC97/SC21, 1997.
19. W. B. Daszczuk: *Evaluation of temporal formulas based on checking by spheres*, Proc. Euromicro Symposium on Digital Systems Design—Architectures, Methods and Tools, IEEE Computer Society, September 4–6, 2001, Warsaw, pp. 158–164.
20. W. B. Daszczuk, W. Grabski, J. Mieścicki, J. Wytrębowski J.: *System modeling in the COSMA environment*, Proc. Euromicro Symposium on Digital Systems Design - Architectures, Methods and Tools, IEEE Computer Society, September 4–6, 2001, Warsaw, pp. 152–157.
21. P. Fusik: *Weryfikacja modelowa współbieżnych programów w języku Java przy użyciu środowiska Bandera i COSMA*, Praca dyplomowa magisterska, Instytut Informatyki PW, 2004.
22. A. Krystosik: *Projektowanie i formalna weryfikacja oprogramowania dla reaktywnych systemów ochrony informacji przy pomocy automatów ESCM*, Materiały z VI Krajowej Konferencji Zastosowań Kryptografii Enigma 2002, Warszawa, 15–17 maja 2002, str. 185–193.
23. W. B. Daszczuk: *Verification of temporal properties in concurrent systems*, Rozprawa doktorska, Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych, Warszawa, styczeń 2003,
24. A. Krystosik: *ESCM—Extended Concurrent State Machines*, ICS Research Report 2/2003.
25. J. Mieścicki: *Concurrent State Machines, the formal framework for model-checkable systems*, ICS Research Report, 5/2003,
26. W. B. Daszczuk: *Temporal model checking in the COSMA environment (the operation of TempoRG program)*, ICS Research Report, 7/2003, Warszawa, 2003.
27. W. B. Daszczuk: *Timed Concurrent State Machines*, ICS Research Report, 27/2003, Warszawa, 2003.
28. J. Mieścicki: *Multi-phase model checking in the COSMA environment*, ICS Research Report, 14/2003, Warszawa, 2003.
29. J. Mieścicki, B. Czejdo, W. B. Daszczuk: *Multi-phase model checking in the COSMA environment as a support for the design of pipelined processing*, ICS Research Report, 16/2003, Warszawa, 2003.