Techniki analizy kodu źródłowego w weryfikacji własności rzeczywistych programów w języku Java

Autoreferat rozprawy doktorskiej

Krzysztof Jakubczyk

1 Wstęp

W dzisiejszych czasach komputery można znaleźć praktycznie we wszystkich obszarach działalności człowieka. Nowoczesne społeczeństwo jest niezwykle mocno uzależnione od komputerów — są one wykorzystywane m.in. w edukacji, do zarządzania systemami transportu, w bankowości i handlu, stosują je organizacje rządowe i wojsko. W praktyce komputery wykorzystywane do nadzorowania pracy elektrowni jądrowych i lotów kosmicznych niewiele różnią się od komputerów osobistych czy nawet telefonów komórkowych. Architektura dzisiejszych procesorów może się różnić, ale języki programowania wykorzystane do tworzenia oprogramowania w większości przypadków pozostają takie same.

Wraz ze wzrostem popularności komputerów rośnie zapotrzebowanie na programy komputerowe. Wartość światowego rynku oprogramowania w 2009 roku była szacowana na około 250 miliardów dolarów [9]. Sam rynek aplikacji na telefony komórkowe miał w 2010 wartość około 7 miliardów dolarów i szacuje się jego wzrost do 25 miliardów dolarów w 2015 roku [16]. Ciągle rozwijający się przemysł komputerowy wymaga budowy coraz to bardziej potężnych i skomplikowanych systemów, które potrzebują bardzo zaawansowanego i złożonego oprogramowania. Jego poziom skomplikowania i duży rozmiar niesie za sobą rosnące prawdopodobieństwo wystąpienia błędów. W historii znane są spektakularne wypadki spowodowane przez niewielkie błędy w oprogramowaniu, jak chociażby przypadek samozniszczenia rakiety Ariane 5 tuż po starcie [11].

Rosnąca rola komputerów i powierzanie im coraz to poważniejszych obowiązków wymaga większego nacisku na zapewnienie jakości wytwarzanego oprogramowania. Istnieje wiele metod służących niwelowaniu liczby błędów. Najpopularniejszą jest weryfikacja dynamiczna, w której sprawdza się działającą aplikację, przeprowadzając na niej różnego rodzaju testy, np. testy funkcjonalne, testy jednostkowe, testy wydajności. Popularność tej metody wynika głównie z łatwości jej zrozumienia, co przekłada się także na wykorzystanie testów podczas odbioru oprogramowania przez zamawiającego. Niestety, dynamiczne podejście nie jest w stanie zapewnić, że program nie zawiera błędów, gdyż testowane są tylko wybrane scenariusze. W praktyce bardzo rzadko zdarza się, że można pokryć testami wszystkie przypadki.

Alternatywne podejście do weryfikacji dynamicznej stanowi weryfikacja statyczna. Takie metody weryfikacji nie uruchamiają programu. Wykorzystują one w swojej analizie jedynie kod programu: może to być kod źródłowy, bajtkod czy nawet kod maszynowy. Do statycznych metod zalicza się m.in. sprawdzanie konwencji programistycznych, wyszukiwanie złych fragmentów kodu (tzw. *antywzorce*), liczenie metryk kodu oraz *metody formalne*. W swojej rozprawie doktorskiej koncentruję się na ostatnich — na *metodach formalnych*. Polegają one na wykorzystaniu pewnej matematycznej teorii do wnioskowania na temat wykonania programu bez jego uruchamiania. Można w ten sposób np. udowodnić, że program podczas wykonania nie będzie posiadał pewnego rodzaju błędu jak przepełnienie bufora czy odwołanie się do nieistniejącego obiektu (np. `NullPointerException` w języku Java). Jedną z popularnych metod formalnych jest *abstrakcyjna interpretacja* [6]. Była ona wykorzystywana w praktyce, m.in. do weryfikacji oprogramowania kontroli lotu w samolotach Airbus A340 i A380 [10].

Niestety, metody formalne są rzadko wykorzystywane przez przemysł. Wynika to z przekonania, że rozwiązania teoretyczne są za mało praktyczne, a ich zastosowanie jest zbyt drogie i pracochłonne. Wyjątek stanowią systemy zwiększonego ryzyka jak oprogramowanie sterujące elektrownią jądrową [19] czy wspomniane już oprogramowanie samolotów [10].

Głównym celem mojej rozprawy doktorskiej jest stworzenie metody, która pozwalałaby badać praktyczność dziedzin abstrakcyjnej interpretacji na rzeczywistym kodzie Javy. Całość podzielona jest na trzy części:

- W pierwszej zaprezentowana jest pewna abstrakcyjna dziedzina numeryczna oraz nowy, mocno konfigurowalny operator rozszerzający dla tej dziedziny.
- W drugiej części stworzone jest narzędzie służące do wyszukiwania wzorców programistycznych na dużych projektach w języku Java, które pozwala na generowanie specyfikacji. Użyteczność narzędzia jest pokazana na przykładzie generowania warunków terminacji dla pętli zapisanych w języku JML.
- W ostatniej części narzędzie jest wykorzystane do badania praktycznej siły abstrakcyjnej dziedziny oraz zaproponowanego operatora rozszerzającego.

2 Abstrakcyjna interpretacja

Zazwyczaj nie jest możliwa weryfikacja programu bazująca na konkretnej jego semantyce. Konkretna semantyka może być jednak zbyt precyzyjna, jeżeli koncentrujemy się na szukaniu tylko konkretnych błędów. Na tej właśnie obserwacji bazuje *abstrakcyjna interpretacja*. Pomysł polega na stworzeniu abstrakcji, która zajmuje się jedynie pewnym podzbiorem własności programu. Podstawowym założeniem *abstrakcyjnej interpretacji* jest poprawność analizy: jeżeli abstrakcyjna analiza nie znajdzie błędu, to nie będzie go też w rzeczywistym programie. Ponieważ abstrakcyjna analiza zajmuje się jedynie pewnym wycinkiem własności programu, pewne zależności nie są obserwowane. Takie podejście powoduje przeszacowanie rzeczywistego — konkretnego rezultatu.

Teoria abstrakcyjnej interpretacji opiera się na dziedzinach semantycznych oraz zależnościach pomiędzy nimi. Mamy do czynienia z dwoma dziedzinami:

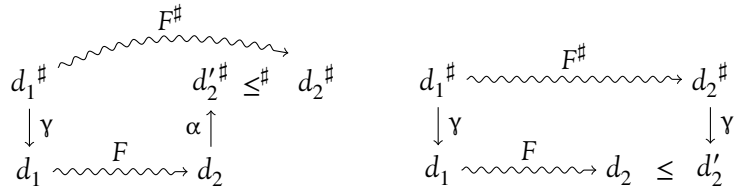
Dziedzina konkretna jest porządkiem $\langle \mathcal{D}, \leq \rangle$. Jest to dziedzina stanów konkretnych lub dokładnych własności.

Dziedzina abstrakcyjna jest porządkiem $\langle \mathcal{D}^\sharp, \leq^\sharp \rangle$. Jest to przybliżenie semantyki konkretnej i zajmuje się przybliżonymi własnościami.

Najbardziej konkretną dziedziną semantyczną programu jest *semantyka standardowa*. Zazwyczaj przyjmuje się, że jest to zbiór stanów programu osiągalnych z któregokolwiek ze stanów początkowych. Najczęściej dziedzina ma strukturę kraty lub kraty zupełnej. Z każdą dziedziną związana jest *funkcja semantyczna*, zwana też *funkcją transferu*, która stanowi interpretację instrukcji programu. Związek pomiędzy dziedzinami wyraża się za pomocą następujących funkcji:

Funkcja konkretyzacji $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$, która dla abstrakcyjnej własności $d^\sharp$ zwraca największy element konkretny, który spełnia własność $d^\sharp$.

Funkcja abstrakcji $\gamma : \mathcal{D} \to \mathcal{D}^\sharp$, która dla konkretnego elementu d zwraca najsilniejszą abstrakcyjną własność, która jest spełniona dla d.



(a) Przy użyciu *Galois connection* (b) Za pomocą konkretyzacji

Rysunek 1: Poprawność abstrakcji funkcji transferu.

Może się zdarzyć, że któraś z funkcji nie istnieje. Jeżeli obie funkcje istnieją, mogą tworzyć tzw. *Galois connection*. Pojęcie poprawności abstrakcji w przypadku istnienia *Galois connection*, a także wyrażone jedynie za pomocą funkcji konkretyzacji, jest przedstawione na rysunku 1.

Operator rozszerzający

Samo przejście do *abstrakcyjnej semantyki* nie gwarantuje terminacji analizy. Aby rozwiązać ten problem, wprowadzono *operator rozszerzający* [5], którego zadaniem jest zapewnienie terminacji analizy. Niech $\langle \mathcal{D}, \leq \rangle$ będzie częściowym porządkiem. Operator rozszerzający $\nabla : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ spełnia następujące własności:

1) przeszacowuje argumenty: dla dowolnych $d, d' \in \mathcal{D}$ zachodzi $d \leq d \nabla d'$ oraz $d' \leq d \nabla d'$,

2) stabilizuje: dla dowolnego ciągu elementów ze zbioru $\mathcal{D}$: $d_0 \leq d_1 \leq \ldots$ ciąg:

$$\begin{aligned} y_0 &= d_0, \\ y_{n+1} &= y_n \nabla d_{n+1} \text{ dla } n \in \mathbb{N} \end{aligned}$$

nie jest ściśle rosnący, tzn. istnieje indeks $i \in \mathbb{N}$, dla którego $y_i = y_{i+1}$.

Istnieje także inny wariant operatora rozszerzającego, gdzie definiuje się operator tylko w przypadku gdy drugi argument jest nie mniejszy niż pierwszy, a w warunku stabilizacji się to zapewnia w następujący sposób: $y_{n+1} = y_n \nabla (y_n \cup d_{n+1})$ [1].

Twierdzenie 1 mówi o terminacji algorytmu abstrakcyjnej interpretacji oraz dokładności wyniku analizy.

Twierdzenie 1 (Abstrakcyjna iteracja z rozszerzaniem)**.** *Niech $\langle \mathcal{D}, \leq \rangle$ będzie kratą zupełną, $\langle \mathcal{D}^\sharp, \leq^\sharp \rangle$ będzie częściowym porządkiem, funkcja transferu $f : \mathcal{D} \to \mathcal{D}$ będzie ciągła, funkcja konkretyzacji $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ będzie monotoniczna. Ponadto niech będzie dana funkcja $f^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, która jest abstrakcją funkcji f, tzn. $f \circ \gamma \leq \gamma \circ f^\sharp$. Niech $x \in \mathcal{D}$ oraz $x^\sharp \in \mathcal{D}^\sharp$ spełniają warunek $x \leq \gamma(x^\sharp)$. Wtedy ciąg zdefiniowany jako:*

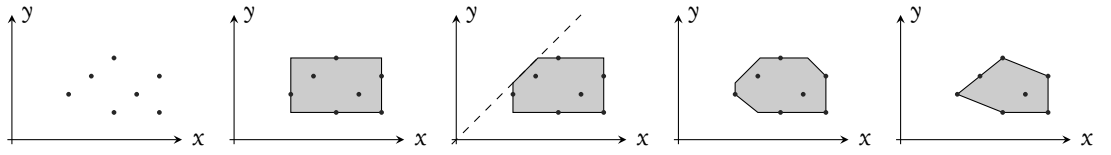
$$y_0{}^\sharp = x^\sharp,$$
$$y_{n+1}{}^\sharp = y_n{}^\sharp \nabla f^\sharp(y_n{}^\sharp) \text{ dla } n \in \mathbb{N}$$

stabilizuje po skończonej liczbie kroków i jego granica przeszacowuje najmniejszy punkt stały funkcji f: $lfp_x f \leq \gamma(\lim(y_n{}^\sharp)_{n\in\mathbb{N}})$.

Przy tworzeniu dziedziny abstrakcyjnej, która ma być wykorzystana w algorytmie abstrakcyjnej interpretacji, należy stworzyć częściowy porządek (często zdarza się, że jest to krata), poprawną funkcję transferu (przeszacowującą konkretną funkcję transferu) oraz operator rozszerzający gwarantujący terminację analizy.

3 Abstrakcyjne dziedziny numeryczne

Jednym z najczęściej poruszanych problemów analizy abstrakcyjnej jest analiza zmiennych numerycznych. Celem *abstrakcyjnych dziedzin numerycznych* jest opisanie w zwarty sposób zbioru możliwych wartości zmiennych numerycznych oraz wydajnych operacji: operacji kratowych, operatora rozszerzającego i funkcji transferu. Na rysunku 2 zaprezentowane są najpopularniejsze abstrakcyjne dziedziny numeryczne.



(a) Punkty konkretne (b) Przedziały (c) Pięciokąty (d) Ośmiokąty (e) Wielokąty wypukłe

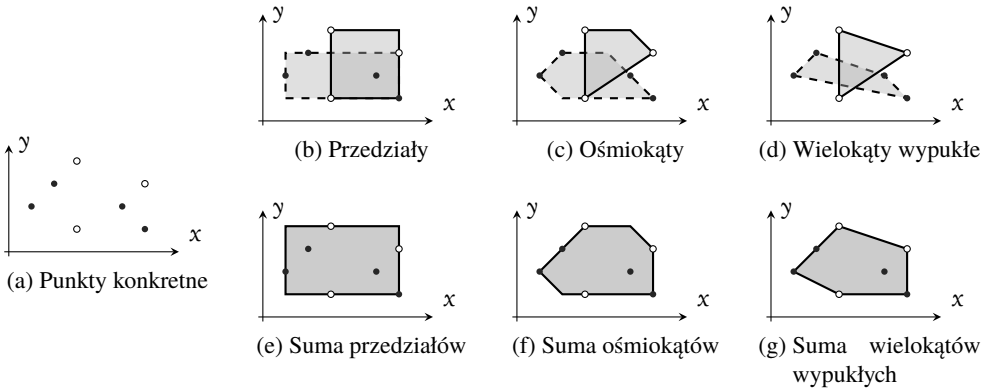
Rysunek 2: Przegląd abstrakcyjnych dziedzin numerycznych.

Jedną z najpopularniejszych dziedzin numerycznych jest *dziedzina przedziałów* [5], w której dla każdej zmiennej pamiętany jest jedynie przedział jej wartości. Jeżeli zmienna ma możliwe wartości 0 oraz 10, abstrakcyjną wartością jest przedział $[0, 10]$. Jest to dziedzina *nierelacyjna*, czyli nie są w niej obsługiwane relacje pomiędzy zmiennymi (np. to, że jedna zmienna ma wartość mniejszą niż druga).

Najprostszą numeryczną dziedziną relacyjną jest *dziedzina pięciokątów* [15], która oprócz przedziału dla zmiennych obsługuje relacje między zmiennymi postaci $x < y$. Kolejną bardzo popularną dziedziną jest *dziedzina ośmiokątów* [17]. Pozwala ona wyrażać nierówności postaci $x \pm y \leq c$, gdzie c jest stałą numeryczną. Jedną z dziedzin mających największą siłę wyrazu jest dziedzina *wielokątów wypukłych* [8], która pozwala na reprezentowanie nierówności liniowych dla wielu zmiennych.

Dziedziny numeryczne, które zostały tutaj omówione, uszeregowane są rosnąco względem skomplikowania, wydajności, a także i wyrażalności. Dziedzina przedziałów posiada liniowy koszt operacji ze względu na liczbę zmiennych, dziedzina ośmiokątów — sześcienny, a dziedzina wielokątów wypukłych — wykładniczy.

Należy zauważyć, że przedstawione dziedziny posiadają jedną wspólną wadę: wszystkie reprezentują zbiory wypukłe i operacja kratowa obliczania kresu górnego dwóch elementów przeszacowuje dokładny wynik. Rysunek 3 ilustruje ten problem.



Rysunek 3: Niedokładne obliczanie sumy w abstrakcyjnych dziedzinach numerycznych.

Istnieją ogólne techniki [7], które pozwalają tworzyć dziedziny abstrakcyjne reprezentujące wiele rozłącznych elementów. Najtrudniejsze do rozstrzygnięcia są w nich dwie kwestie: sprawdzenie, czy dwa elementy są równe (może być wiele sposobów na reprezentowanie tego samego zbioru) oraz stworzenie operatora rozszerzającego [1].

4 Abstrakcyjna dziedzina numeryczna *pudełek*

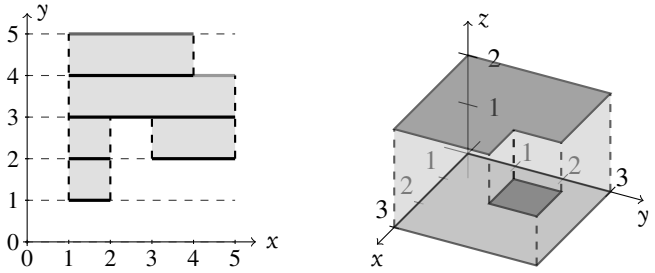
Abstrakcyjna dziedzina *pudełek* (ang. *boxes*) [13] stanowi *dysjunktywne rozszerzenie* (ang. *disjunctive refinement*) dziedziny przedziałowej. Element dziedziny przedziałów nazywany jest *pudełkiem*. Elementami *dziedziny pudełek* są skończone sumy pojedynczych *pudełek*. Niech $\mathbb{B}_n$ będzie zbiorem elementów *dziedziny przedziałowej* dla zbioru zmiennych Var takiego, że $|\mathit{Var}| = n$. Formalnie *dziedzinę pudełek* definiuje się jako uporządkowaną szóstkę $\langle \mathbb{BS}_n, \subseteq, \varnothing, \mathbb{I}^n, \cup, \cap \rangle$, gdzie:

- $\mathbb{I}$ jest zbiorem liczb rzeczywistych $\mathbb{R}$, wymiernych $\mathbb{Q}$ lub całkowitych $\mathbb{Z}$,

- $\mathbb{BS}_n = \{\mathcal{BS} \in \mathbb{I}^n \mid$ istnieją $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_k \in \mathbb{B}_n$ takie, że $\mathcal{BS} = \bigcup_{i=1}^{k} \mathcal{B}_i\}$ jest zbiorem elementów dziedziny,
- $\subseteq$ jest porządkiem zawierania zbiorów,
- $\varnothing$ jest pustym elementem stanowiącym najmniejszy element porządku,
- $\cup$ jest operatorem kresu górnego argumentów,
- $\cap$ jest operatorem kresu dolnego argumentów.

Elementy dziedziny *pudełek* tworzą strukturę kraty (ale nie kraty zupełnej). W odróżnieniu od dziedziny przedziałów, operacja sumy jest operacją dokładną.

Wykorzystanie techniki zamiatania
W rozdziale czwartym rozprawy doktorskiej do reprezentowania elementów dziedziny zaproponowana została technika zamiatania (ang. *sweeping line technique*), wykorzystywana w geometrii obliczeniowej. Przykładem wykorzystania tej techniki jest algorytm Bentleya-Ottmanna do znajdowania wszystkich przecięć w zbiorze odcinków na płaszczyźnie [2]. Pomysł polega na wyobrażeniu sobie prostej (zwanej miotłą), która przesuwa się po płaszczyźnie (zamiata). Z miotłą związana jest pewna struktura danych. W czasie zamiatania prosta zatrzymuje się w niektórych punktach, a struktura danych jest uaktualniana. Po przejściu przez wszystkie punkty wynik obliczany jest na podstawie struktury danych miotły.



(a) Dwa wymiary (b) Trzy wymiary

Rysunek 4: Wykorzystanie zamiatania do reprezentowania elementów dziedziny *Pudełek*.

Na rysunku 4 zaprezentowany jest przykład dwu- oraz trójwymiarowy. Rozpatrzmy najpierw przykład z rysunku 4(a). Załóżmy że mamy już reprezentację jednowymiarową naszej dziedziny. Miotła zamiata wzdłuż osi (zmiennej) y, rozpoczynając od $-\infty$ aż do $+\infty$ (z dołu do góry). Podczas zamiatania obserwujemy, jak zmienia się zbiór wartości dla zmiennej x. Miotła zatrzymuje się w punktach, w których wartość ta ulega zmianie. Jako wartość związaną z miotłą utrzymujemy aktualny zbiór możliwych wartości dla zmiennej x. W przykładzie z rysunku 4(a) wygląda to następująco:

Położenie miotły	Dane związane z miotłą (zbiór wartości x)
$y = -\infty$	$\varnothing$
$y = 1$	$[1, 2]$
$y = 2$	$[1, 2] \cup [3, 5]$
$y = 3$	$[1, 5]$
$y = 4$	$[1, 4]$
$y = 5$	$\varnothing$

Dla przykładu trójwymiarowego z rysunku 4(b) postępowanie jest analogiczne. Jedyna różnica polega na tym, że obserwowana jest dwuwymiarowa wersja dziedziny: zamiatamy wzdłuż osi (zmiennej) z i obserwujemy płaszczyznę zmiennych x oraz y.

Reprezentacja elementów dziedziny *pudełek*

Reprezentacja elementów dziedziny opiera się na przedstawionej powyżej technice zamiatania. Jedyną różnicą jest to, że przy zamiataniu wzdłuż osi (zmiennej) x, struktura związana z miotłą gromadzi dotychczasowe wartości obserwowanej dziedziny wraz z wartościami zmiennej x, dla której obserwowana wartość uległa zmianie. Wynikiem zamiatania jest więc lista par $< p, v >$, gdzie p jest wartością zmiennej x z punktu zatrzymania miotły, a v nową wartością obserwowanej dziedziny, która pojawiła się w punkcie p.

Niech $\mathbb{P}$ oznacza zbiór możliwych wartości zmiennej x. W praktyce zbiór $\mathbb{P}$ jest pewnym rozszerzeniem zbioru $\mathbb{I}$ takim, aby panować zarówno nad silnymi jak i słabymi nierównościami przedziałowymi. Formalnie definiujemy nieskończoną sekwencję zbiorów: $\mathbb{S}_0, \mathbb{S}_1, \mathbb{S}_2, \ldots$ w sposób następujący:

$$\begin{aligned}\mathbb{S}_0 &= \{\epsilon, \mathsf{T}_0\},\\ \mathbb{S}_{n+1} &= \{\epsilon\} \cup \{\big((p_1, v_1), (p_2, v_2), \ldots, (p_m, v_m)\big) \mid v_1, \ldots, v_m \in \mathbb{S}_n, v_1 \neq \epsilon,\\ &\qquad p_1, \ldots, p_m \in \mathbb{P}, \forall_{j \in \{1,\ldots,m-1\}} p_j \prec p_{j+1} \wedge v_j \neq v_{j+1}\},\end{aligned}$$

gdzie ϵ jest pustą sekwencją. W sekwencji $\mathcal{S} \in \mathbb{S}_n$ dla $n > 0$ elementami są pary uporządkowane względem pierwszej zmiennej, gdyż zamiatanie było zgodne z jej wartościami. Dodatkowo drugie elementy sąsiednich par są różne, co wynika z zatrzymywania się w punktach, w których wartość obserwowanej dziedziny się zmieniła.

Niech $\mathcal{S}[i]$ dla $\mathcal{S} \in \mathbb{S}_n$, $i \in \mathbb{I}$ i $n > 0$ oznacza wartość obserwowanej dziedziny w punkcie i w reprezentacji $\mathcal{S}$. Wtedy *relację spełnialności* $sat_n \subseteq \mathbb{I}^n \times \mathbb{S}_n$ definiujemy następująco:

$$\begin{aligned}sat_n(\vec{\epsilon}, \mathcal{S}) &\iff \mathcal{S} \in \mathbb{S}_n \wedge \mathcal{S} = \mathsf{T}_0,\\ sat_{n+1}(\langle i_{n+1}, i_n, \ldots, i_1\rangle, \mathcal{S}) &\iff \mathcal{S}[i_{n+1}] = w \wedge sat_n(\langle i_n, \ldots, i_1\rangle, w)\end{aligned}$$

Za pomocą tak zdefiniowanej relacji definiujemy funkcję konkretyzacji dla naszej dziedziny:

$$\gamma(S) = \{\vec{i} \in \mathbb{I}^n \mid sat_n(\vec{i}, \mathcal{S})\}.$$

Następnym krokiem jest zdefiniowanie operacji dziedzinowych dla stworzonej reprezentacji. W tym celu definiujemy $\blacklozenge$-rozszerzenie, które dla danego operatora bazowego $\blacklozenge : \mathbb{S}_0 \times \mathbb{S}_0 \to \mathbb{S}_0$ jednoznacznie rozszerza go do operatora $\blacklozenge : \mathbb{S}_n \times \mathbb{S}_n \to \mathbb{S}_n$ dla dowolnego $n \geq 0$. Za pomocą $\blacklozenge$-rozszerzenia definiujemy operacje dziedzinowe na elementach $\mathbb{S}_n$ dla dowolnego $n \geq 0$: kres górny $\cup_\blacklozenge$, kres dolny $\cap_\blacklozenge$ oraz porządek $\subseteq_\blacklozenge$.

Twierdzenie 2 (Reprezentacja dziedziny *pudełek*)**.** *Dla dowolnego* $n > 0$ *istnieje funkcja* $F : \mathbb{BS}_n \to \mathbb{S}_n$*, gdzie* $\mathcal{BS} \in \mathbb{BS}_n$ *oraz* $\vec{\boldsymbol{i}} \in \mathbb{I}^n$*, dla której:*

$$\vec{\boldsymbol{i}} \in \mathcal{BS} \iff sat_n(\vec{\boldsymbol{i}}, F(\mathcal{BS})).$$

oraz F *zachowuje operacje dziedzinowe, tzn. dla dowolnych* $\mathcal{BS}, \mathcal{BS}' \in \mathbb{BS}_n$ *zachodzą następujące warunki:* $F(\varnothing) = \epsilon$*,* $\mathcal{BS} \subseteq \mathcal{BS} \iff F(\mathcal{BS}) \subseteq_{\blacklozenge} F(\mathcal{BS}')$*,* $F(\mathcal{BS} \cup \mathcal{BS}') = F(\mathcal{BS}) \cup_{\blacklozenge} F(\mathcal{BS}')$ *oraz* $F(\mathcal{BS} \cap \mathcal{BS}') = F(\mathcal{BS}) \cap_{\blacklozenge} F(\mathcal{BS}')$*.*

Twierdzenie 2 zostało udowodnione w rozprawie. Stanowi ono, że proponowana reprezentacja wraz z operacjami dziedzinowymi jest poprawną reprezentacją dla dziedziny *pudełek*. W rozprawie zaproponowano także konstrukcję funkcji transferu dla tej dziedziny. Jej ogólna idea polega na rozbiciu przedstawionej reprezentacji na odpowiadające jej rozłączne *pudełka*, zastosowaniu funkcji transferu z dziedziny *przedziałów* dla każdego z pudełek i powróceniu do dziedziny *pudełek* w oparciu o dokładny operator kresu górnego. Niektóre prostsze przypadki zostały zaimplementowane w bardziej wydajny sposób.

Operator rozszerzający

Operator rozszerzający dla dziedziny *pudełek* zdefiniowany jest zgodnie z alternatywną definicją, w której drugi argument jest nie mniejszy niż pierwszy. W rozprawie zaproponowane jest wykorzystanie *punktów progowych* [3] w konstrukcji operatora rozszerzającego dla dziedziny *pudełek*. *Punkty progowe* służą do zwiększenia dokładności operatora rozszerzającego — niwelują szybkość z jaką wzrasta wynik. Dla przykładu rozważmy następującą aplikację standardowego operatora rozszerzającego dla dziedziny przedziałowej: $[1, 2]\triangledown[1, 3] = [1, \infty]$. Jeżeli dodamy *punkt progowy* 5, koniec przedziału nie wzrośnie od razu do ∞, ale zatrzyma się w punkcie 5: $[1, 2]\triangledown'[1, 3] = [1, 5]$.

W rozprawie wprowadzona jest definicja *progów kroku rozszerzającego*. Jest to funkcja $spec_{\triangledown} : \mathit{Var} \to 2^{\mathbb{P}}$, która dowolnej zmiennej $v \in \mathit{Var}$ przyporządkowuje skończony zbiór $spec_{\triangledown}(v)$. Funkcja jest wykorzystana w pojedynczej aplikacji operatora rozszerzającego. Następnie definicja zostaje rozszerzona do całej sekwencji aplikacji operatora rozszerzającego: *sekwencja progów operatora rozszerzającego* jest to nieskończony ciąg $spec_{\triangledown,1}, spec_{\triangledown,2}, \ldots$, który od pewnego elementu $k > 0$ nie wprowadza nowych punktów progowych, tzn. dla każdej zmiennej $v \in \mathit{Var}$ zachodzi:

$$spec_{\triangledown,n}(v) \subseteq \bigcup_{i=1}^{k} spec_{\triangledown,i}(v).$$

W oparciu o koncepcję *sekwencji progów operatora rozszerzającego*, w rozdziale czwartym rozprawy zaproponowana jest ogólna definicja operatora rozszerzającego. Przedstawiona konstrukcja jest rekurencyjna, podobnie jak definicja $\blacklozenge$-rozszerzenia. Na zbiorze $\mathbb{S}_0$ operator jest równy kresowi górnemu. Następnie zdefiniowany jest operator dla $(n + 1)$-wymiarowej przestrzeni, dla $n \geq 0$. Niech $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_{n+1}$ będą argumentami operatora rozszerzającego. Pierwszym krokiem jest podział osi zmiennej dla wymiaru $(n + 1)$ na rozłączne segmenty. Podział ten jest tworzony na podstawie argumentów operatora $\mathcal{S}$, $\mathcal{S}'$ oraz *progów kroku rozszerzającego* dla aktualnego kroku oraz zmiennej. Następnie dla każdego segmentu obliczana jest wartość dziedziny n-wymiarowej: jako aplikacja operatora rozszerzającego dla n-wymiarów. Pierwszy argument tej aplikacji jest wartością dziedziny dla tego segmentu w pierwszym argumencie $\mathcal{S}$ operatora.

Drugi argument może być wartością dziedziny dla tego segmentu w drugim argumencie S', ale czasem, aby zapewnić stabilizację, trzeba wziąć wartość lekko *poprawioną*. Zaprezentowany jest dowód, że proponowany operator faktycznie jest operatorem rozszerzającym dla dziedziny *pudełek*.

Proponowana konstrukcja uogólnia dotychczasową konstrukcję operatora rozszerzającego dla dziedziny *pudełek* [13]: warunek na wybór *poprawionej* wartości dla segmentu jest ogólniejszy, a konstrukcja podziału na segmenty jest rozszerzona o *punkty progowe*.

W dalszej części rozdziału czwartego zawarte są rozważania na temat dokładności operatora rozszerzającego. Zaproponowana jest implementacja nowego operatora rozszerzającego: $\nabla_{\leq}$, który jest instancją ogólnego operatora przedstawionego wcześniej. Nowy operator uszczegóławia warunek *poprawiania* wartości dla segmentu, ale nadal jest sparametryzowany *sekwencją progów operatora rozszerzającego*.

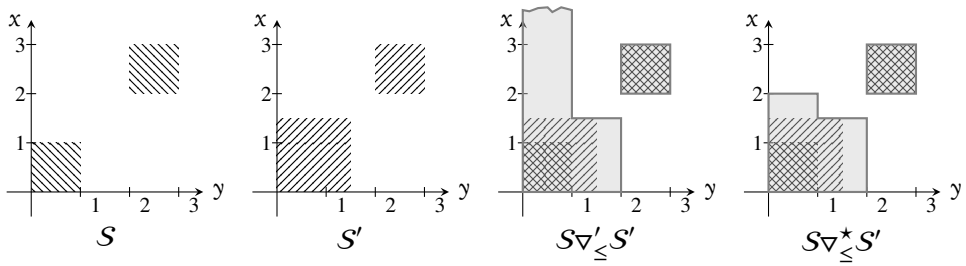
Twierdzenie 3 (Precyzja jednego kroku)**.** *Niech $S, S' \in \mathbb{S}_n$ dla $n \geq 0$. Niech $\nabla'_{\leq}$ oraz $\nabla^{\star}_{\leq}$ będą instancjami operatora rozszerzającego $\nabla_{\leq}$ dla różnych wartości funkcji* progów kroku rozszerzającego *wykorzystanymi do obliczenia wyniku dla argumentów S i S'. Niech:*

$$\forall_{v \in Var} spec_{\nabla'_{\leq}}(v) \subseteq spec_{\nabla^{\star}_{\leq}}(v).$$

Wtedy zachodzi:

$$S \nabla^{\star}_{\leq} S' \subseteq S \nabla'_{\leq} S'.$$

W rozprawie podano dowód twierdzenia 3. Traktuje ono o dokładności jednego kroku proponowanego operatora $\nabla_{\leq}$ w zależności od wyboru funkcji *progów kroku rozszerzającego*. Jeżeli w danym kroku dla tych samych argumentów wybierzemy funkcję, która zawiera jakieś dodatkowe punkty progowe, zawsze dostaniemy wynik nie gorszy. Przykład zwiększenia dokładności



Rysunek 5: Porównanie jednego kroku aplikacji operatora rozszerzającego dla różnych funkcji *progów operatora rozszerzającego*. Operator $\nabla'_{\leq}$ używa funkcji $spec_{\nabla'_{\leq}} = \lambda v.\emptyset$, operator $\nabla^{\star}_{\leq}$ używa funkcji $spec_{\nabla^{\star}_{\leq}} = spec_{\mathrm{G}}(S)$ więc $spec'_{\nabla}(x) = \{0, 1, 2, 3\}$.

operatora $\nabla_{\leq}$ przy dodaniu punktów progowych zaprezentowany jest na rysunku 5. Pokazuje on również, że w niektórych sytuacjach proponowany operator jest dokładniejszy niż dotychczas istniejący [13], który jest instancją operatora $\nabla_{\leq}$ dla pustej funkcji *progów operatora rozszerzającego*. W rozprawie doktorskiej zaproponowanych jest kilka strategii tworzenia *sekwencji progów operatora rozszerzającego*.

5 Tworzenie praktycznych metod formalnych

W rozdziale piątym rozprawy przedstawione jest podejście do tworzenia metod formalnych, które może być przydatne do zastosowań komercyjnych. Głównym czynnikiem, który należy zniwelować przy projektowaniu praktycznej techniki wykorzystującej metody formalne w weryfikacji, jest koszt jej zastosowania. Aktualnie istniejące metody weryfikacji dla języka Java, jak np. ESC/Java2 [4], wymagają często dużej pomocy od użytkownika [18]. Ponadto musi to być osoba o dużych kwalifikacjach, której czas jest kosztowny.

Alternatywą dla marnowania czasu wykwalifikowanych pracowników jest automatyczne generowanie wymaganych specyfikacji na podstawie jakiejś dodatkowej analizy. Zgodnie z *zasadą Pareto* (znanej także jako *reguła "80:20"*) około 80% kodu programu powinno być łatwe, a analiza tej części niezbyt wymagająca. Marnowanie czasu drogich pracowników jest kosztowne. Osoba recenzująca czy sprawdzająca poprawność kodu powinna skupić się na przypadkach trudnych i skomplikowanych, gdyż potencjalnie one właśnie zawierają większość błędów. W rozprawie zaproponowane jest podejście iteracyjne, które docelowo prowadzi do automatycznego pokrycia ustalonego progu (np. około 80%) interesujących przypadków. Proponowane kroki są następujące:

1. Wybieramy interesujące nas konstrukcje w programie, np. pętle `for`.
2. Uruchamiamy stworzoną metodę na wybranym dużym zbiorze projektów.
3. Sprawdzamy, jakie przypadki należy jeszcze obsłużyć.
4. Wybieramy zbiór przypadków, dla których potrafimy stworzyć ogólne rozwiązanie.
5. Zaznaczamy obsłużone przypadki.
6. Generujemy statystyki i obliczamy skuteczność.
7. Jeżeli nie osiągnęliśmy wyznaczonego progu skuteczności, przechodzimy do kroku 2.

Dla przykładu rozpatrzmy klasyfikację pętli `for` w języku Java w kontekście generowania dla nich warunków terminacji. Najpierw rozpoznajemy interesujące nas konstrukcje — w tym przypadku wszystkie pętle `for`. Następnym krokiem jest przejrzenie pętli i próba znalezienia wzorców programistycznych, dla których można łatwo wygenerować warunek terminacji. Przykładem wzorca może być najprostsza pętla `for`, która zwiększa w każdym kroku zmienną kontrolną aż do pewnej stałej, dodatkowo zmienna kontrolna jest uaktualniana tylko w wyrażeniu uaktualniania pętli. Wygenerowanie warunku terminacji w takim przypadku jest łatwe. Następnie generujemy statystyki aby sprawdzić, ile pętli pasuje do naszego wzorca oraz czy udało się osiągnąć wybrany próg. Jeżeli nie udało się go osiągnąć, należy wykonać kolejny krok iteracji, sprawdzając pętle nie objęte dotychczasowymi wzorcami. Warto zauważyć, że w każdej iteracji prezentowanej metody możliwe jest wykorzystanie innej techniki.

CodeStatistics

W rozdziale piątym rozprawy, zaprezentowane jest narzędzie — *CodeStatistics*, które wspomaga proponowaną metodologię. Program ten jest wtyczką do programu Eclipse. Zadaniem *CodeSta-*

tistics jest wyszukiwanie wzorców w projektach stworzonych w języku Java oraz zliczanie statystyk ich wystąpień. Wzorce definiowane są przez przez użytkownika a pozostała część wykonywana jest automatycznie. Dodatkowo *CodeStatistics* potrafi wskazywać fragmenty kodu, które pasują do zdefiniowanych wzorców. Przy zastosowaniu zaproponowanej metodologii możliwe jest wykorzystanie *CodeStatistics* do mierzenia efektywności projektowanych metod formalnych.

Zasada działania *CodeStatistics* jest następująca: najpierw zostaje wczytany kod źródłowy programu i stworzone dla niego *drzewo składniowe* (ang. *syntax tree*). Kolejnym krokiem jest transformacja drzewa do formatu XML, dodatkowo wzbogacona o pewne informacje od kompilatora języka Java, jak np. typy obiektów. *CodeStatistics* zapamiętuje mapowanie pomiędzy węzłami drzewa składniowego a elementami XML. Następnie na wygenerowanym pliku XML wykonywane są zapytania w formie wyrażeń XPath, które są dostarczone przez użytkownika. Zapytania te stanowią opis wzorca, które są przez *CodeStatistics* wyszukiwane w pliku XML. Ostatnim krokiem jest wykorzystanie mapowania do znalezienia elementów drzewa składniowego (czyli fragmentów programu) odpowiadających znalezionym elementom XML oraz wyliczenie statystyk wystąpień.

Eksperyment: generowanie warunków terminacji pętli
Jednym z poważniejszych problemów automatycznej weryfikacji są pętle — często, żeby przeprowadzić dowód wymagane jest dostarczenie specyfikacji niezmiennika pętli czy też warunku jej terminacji. W eksperymencie, który jest opisany dokładnie w rozprawie doktorskiej, zostały skategoryzowane numeryczne pętle *for* rzeczywistych, dużych projektów stworzonych w języku Java. Szczegóły wybranych projektów zostały przedstawione na rysunku 1. Dla tych pętli zostały wygenerowane warunki terminacji w formie specyfikacji `decreases` w języku JML.

Projekt	Wersja	Rozmiar (tys. linii)	Liczba pętli `for`
Oracle Berkeley DB	5.0.34	253	1799
Google App Engine	1.6.4.1	163	967
Apache Hadoop	1.0.1	292	1898
Hibernate	4.1.2	405	855
JEdit	4.5.1	111	894
Tomcat	7.0.27	216	1510
Suma		1440	7923

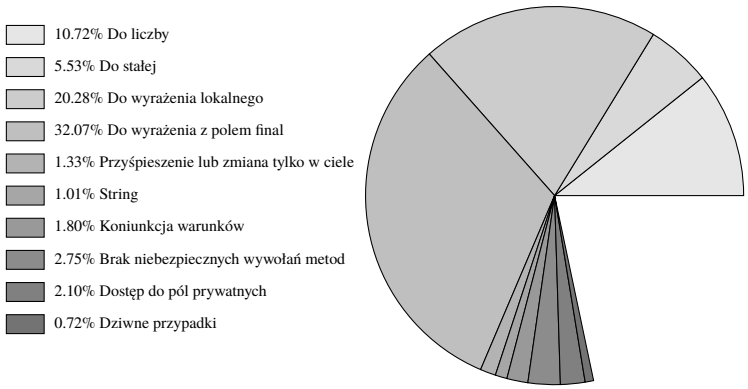
Tablica 1: Szczegóły analizowanych projektów.

Pętle `for` w języku Java mają następującą składnię:

```
for (inicjalizator; warunek ńkoca; krok) {
    S
}
```

gdzie *inicjalizator* jest wyrażeniem wykonanym przed wejściem do pętli, *warunek terminacji* jest sprawdzany przed każdą iteracją, natomiast *krok* jest wyrażeniem wykonywanym po iteracji, tuż przed sprawdzeniem warunku. *Zmienna kontrolna* jest to zmienna, która przyjmuje wartości numeryczne i kontroluje liczbę iteracji pętli. Występuje ona w *warunku terminacji* do sprawdzenia końca pętli. Najczęściej *zmienna kontrolna* jest modyfikowana w *kroku* pętli.

W eksperymencie brane były pod uwagę pętle *for*, które wykorzystują *zmienną kontrolą*. Podstawowy przypadek to pętla, w której *zmienna kontrolna* jest zmienną lokalną, *warunek końca* to porównanie tej zmiennej ze literałem liczbowym, jej wartość jest uaktualniona w *kroku* pętli i zmienna ta nie jest modyfikowana w ciele pętli. Pozostałe kategorie są w większości rozszerzeniami tego warunku, np. porównanie zmiennej liczbowej z wartością wyrażenia, w którym mogą występować zmienne lokalne, które nie są modyfikowane czy pozwolenie na dodatkowe uaktualnienie *zmiennej kontrolnej* w ciele pętli (trzeba uważać, żeby nie odbywało się w wewnętrznej pętli). Istnieje też kategoria, która w wyrażeniu warunku pozwala na wywołanie metody `length` zmiennej lokalnej typu `String`, która nie jest modyfikowana. Wynika to z faktu, że klasa `String` jest klasą finalną a instancje obiektów tej klasy są niemodyfikowalne. Dla wszystkich kategorii za pomocą syntaktycznych reguł zostały wygenerowane warunki terminacji pętli: w formie klauzul *decreases* języka JML. Podsumowanie wyników zostało przedstawione na rysunku 6. W pre-



Rysunek 6: Udział procentowy skategoryzowanych pętli.

zentowanym eksperymencie możliwe było wygenerowanie warunków terminacji dla ponad 78% numerycznych pętli `for`. Na końcu wynik został zweryfikowany na projektach przedstawionych na rysunku 7.

Projekt	**Wersja**	**Rozmiar (tys. linii)**	**Liczba pętli `for`**	**Pokrycie**	
AspectJ	1.6.12	385	5049	4461	88,35 %
Spring toolkit	3.1.1	181	457	349	76,37 %
Vuze	4.7.0.2	496	3689	2395	64,92 %
Total		1062	9195	7205	78,36 %

Rysunek 7: Szczegóły projektów wykorzystanych do weryfikacji wyników.

6 Wykorzystanie wyszukiwania wzorców do oceny praktyczności abstrakcyjnych dziedzin

W rozdziale szóstym rozprawy doktorskiej zaprezentowane jest rozszerzenie narzędzia *CodeStatistics* o semantyczne testy węzłów. Pierwszą część rozdziału stanowi opis statycznego analizatora dla języka Java — *JavaAI*. Analizator jest realizacją techniki *abstrakcyjnej interpretacji*, zawiera implementację abstrakcyjnej dziedziny numerycznej przedziałów oraz abstrakcyjnej dziedziny numerycznej *pudełek* z rozdziału czwartego. Dla drugiej z wymienionych dziedzin stworzonych zostało kilka strategii tworzenia *sekwencji progów operatora rozszerzającego*.

W drugiej części opisywanego rozdziału przedstawiony jest sposób rozszerzania narzędzia *CodeStatistics* o dodatkowe testy. Mechanizm rozszerzania jest realizowany przez system wtyczek, zwanych *wtyczkami analizy*. Przed rozpoczęciem przetwarzania pliku *CodeStatistics* wywołuje odpowiednią metodę *wtyczki analizy*, której zadaniem jest przetworzenie pliku i utworzenie mapy z węzła drzewa AST na wynik przetwarzania dla tego węzła. Dodatkowo każda *wtyczka analizy* udostępnia funkcje rozszerzające język XPath, które mogą zostać wykorzystane w zapytaniach dla *CodeStatistics*. Jako przykład w rozprawie przedstawiona jest wtyczka uruchamiająca analizator *JavaAI*. Funkcje rozszerzające dla języka XPath pozwalają pytać o wartości dziedziny dla węzłów AST, testować trywialność tych wartości (wartość jest trywialna, jeżeli jest największym elementem w kracie) oraz porównywać wartości dziedziny (porównywane są konkretne zbiory odpowiadające elementom dziedziny).

W ostatniej części rozdziału zaprezentowany jest eksperyment, który został przeprowadzony z wykorzystaniem *CodeStatistics* oraz wtyczki *JavaAI*. Celem eksperymentu jest ocenienie siły dziedziny *pudełek* oraz jej operatora rozszerzającego. Testy zostały przeprowadzone na tym samym zestawie projektów jak w eksperymencie terminacji pętli `for`. Testom podlegały trzy abstrakcyjne dziedziny numeryczne:

- *intv*: dziedzina przedziałów,
- *boxes_ldd*: dziedzina *pudełek* z pustą *sekwencją progów operatora rozszerzającego*,
- *boxes_firstbox*: dziedzina *pudełek* z prostą wersją *sekwencji progów operatora rozszerzającego*: wartości progów dla zmiennych pochodzą z pierwszego elementu sekwencji, dla każdej zmiennej jest to najmniejsza i największa możliwa wartość.

W eksperymencie porównane są wartości niezmienników pętli dla wyżej wymienionych dziedzin. Bezpośrednie porównanie dziedziny *intv* oraz *boxes_ldd* wykazało, że druga dziedzina w 23.60% przypadków okazała się dokładniejsza. Niestety, w nielicznych przypadkach niezmienniki były mniej dokładne lub nie dały się bezpośrednio porównać. Porównanie dziedziny *intv* oraz *boxes_firstbox* wykazało, że dodanie prostej funkcji progowej zwiększyło liczbę dokładniejszych niezmienników do 25,14%, a przypadki nieporównywalne czy gorsze zostały zniwelowane praktycznie do 0. Ostatnią częścią eksperymentu było porównanie dwóch wersji dziedziny *pudełek*. Okazało się, że wersja *boxes_firstbox* w 2,36% pozwoliła wygenerować dokładniejszy niezmiennik. Gorszych niezmienników było tylko 0,05%, a nieporównywalnych przypadków 0,13%.

7 Podsumowanie

W rozprawie doktorskiej podjęta została próba stworzenia technik analizy statycznej, które mogą zostać wykorzystane na rzeczywistych, dużych programach w języku Java. Własny wkład teoretyczny jest głównie w rozdziale czwartym. Zaproponowane zostało nowe spojrzenie na dziedzinę *pudełek* oraz zastosowanie techniki *punktów progowych* w operatorze rozszerzającym dla tej dziedziny. Zostały wprowadzone dwie wersje operatora rozszerzającego: pierwsza ogólna, a druga z twierdzeniem o precyzji jednego kroku rozszerzania w uzależnionej od *punktów progowych*. Wstępna wersja rezultatów przedstawionych w tym rozdziale została opublikowana w pracy [14]. Dalsza część rozprawy skupia się na praktyczności metod formalnych. Narzędzie *CodeStatistics* przedstawione w rozdziale piątym oraz opis eksperymentu terminacji pętli `for` zostały opublikowane w pracy [12]. Ostatnią część pracy stanowi rozszerzenie techniki wyszukiwania wzorców o techniki analizy semantycznej. Wykazane zostało, że połączenie takie może zostać praktycznie zastosowane do oceny dziedzin abstrakcyjnej interpretacji na rzeczywistych programach. Ponadto pokazano, że nowy operator rozszerzający z rozdziału czwartego daje wyniki dokładniejsze niż dotychczas znany.

Literatura

[1] Bagnara, R., Hill, P. M., and Zaffanella, E. Widening operators for powerset domains. *International Journal on Software Tools for Technology Transfer* 8, 4-5 (2006), 449–466.

[2] Bentley, J. L. and Ottmann, T. A. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers* 28 (9 Sept. 1979), 643–647.

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In: *The Essence of Computation*. Vol. 2566. Springer, 2002, 85–108.

[4] Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: *FMCO*. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, 342–363.

[5] Cousot, P. and Cousot, R. Static determination of dynamic properties of programs. In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, 106–130.

[6] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *POPL*. ACM, 1977, 238–252.

[7] Cousot, P. and Cousot, R. Systematic Design of Program Analysis Frameworks. In: *POPL*. ACM Press, 1979, 269–282.

[8] Cousot, P. and Halbwachs, N. Automatic Discovery of Linear Restraints Among Variables of a Program. In: *POPL*. ACM Press, 1978, 84–96.

[9] Datamonitor. *Software: Global Industry Guide 2010*. report summary: http://www.research-store.com/fiercewireless/Product/software_global_industry_guide?productid=46A43D30-02A3-4425-BFFB-47FCFD5A3AE5. Jan. 31, 2011.

[10] Delmas, D. and Souyris, J. Astrée: From Research to Industry. In: *SAS*. Vol. 4634. Springer, 2007, 437–451.

[11] Dowson, M. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes* 22 (2 Mar. 1997), 84–.

[12] Fulara, J. and Jakubczyk, K. Practically Applicable Formal Methods. In: *SOFSEM*. Vol. 5901. Lecture Notes in Computer Science. Springer, 2010, 407–418.

[13] Gurfinkel, A. and Chaki, S. Boxes: A Symbolic Abstract Domain of Boxes. In: *SAS*. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, 287–303.

[14] Jakubczyk, K. Sweeping in Abstract Interpretation. *Electronic Notes in Theoretical Computer Science* 288 (2012), 25–36.

[15] Logozzo, F. and Fähndrich, M. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Science of Computer Programming* 75, 9 (2010), 796–807.

[16] Markets and Markets. *World Mobile Applications Market — Advanced Technologies, Global Forecast*. report summary: http://www.marketsandmarkets.com/Market-Reports/mobile-applications-228.html. Aug. 2010.

[17] Miné, A. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100.

[18] Poll, E. and Schubert, A. Verifying an implementation of SSH. In: *In WITS'07*. 2007, 164–177.

[19] Yoo, J., Cha, S. D., and Jee, E. A Verification Framework for FBD Based Software in Nuclear Power Plants. In: *APSEC*. IEEE, 2008, 385–392.