Generowanie obligacji dowodowych dla kodu bajtowego Javy

Tadeusz Sznuk

19 sierpnia 2009

1 Wstęp

Jednym z podstawowych kryteriów oceny programu komputerowego jest jego poprawność. Błędy nie tylko ograniczają użyteczność aplikacji, ale także mogą być zagrożeniem dla bezpieczeństwa całego systemu, w którym owa aplikacja działa. W niektórych przypadkach, na przykład w systemach kontrolujących pracę urządzeń medycznych czy wojskowych, zastosowanie niepoprawnego programu może prowadzić do zagrożenia życia lub poważnych strat materialnych. W większości systemów informatycznych zagrożenie nie jest tak poważne, jednak usterki w oprogramowaniu pozostają istotnym problemem.

Istnieją różne procedury zmniejszające ryzyko wystąpienia błędu w programie komputerowym. Te najpowszechniej stosowane opierają się na testowaniu [Bec03, HT03, CL02]. Polegają one na ocenie wyników działania systemu (lub jego fragmentów) po uruchomieniu go w odpowiednio przygotowanym środowisku (dla ustalonych zestawów danych wejściowych itp.). Testowanie oprogramowania to bardzo rozległa dziedzina. Stworzono liczne narzędzia, biblioteki i metodologie ułatwiające tworzenie zestawów testów. Podejście to ma jednak pewne wady. Przede wszystkim nie może zazwyczaj dać całkowitej gwarancji poprawności. Zwykle nie jest bowiem możliwe zbadanie zachowania aplikacji dla każdej kombinacji danych wejściowych, systemu i innych czynników mogących mieć wpływ na jej wykonanie. Prócz tego należy zauważyć, że stworzenie dobrego zestawu testów dla nietrywialnego programu czy biblioteki jest zadaniem trudnym i pracochłonnym, a w przypadku większych projektów pojawiają się dodatkowo problemy związanie z koniecznością ciągłego uaktualniania testów.

Oczywiście najprostszym sposobem na uniknięcie wszelkich problemów związanych z niepoprawnym oprogramowaniem jest pisanie programów nie zawierających błędów. Nie jest to niestety zadanie proste. Pomocne może okazać się przestrzeganie odpowiednich standardów kodowania. Odpowiednie nazywanie zmiennych i procedur, powstrzymanie się od stosowania pewnych konstrukcji czy nawet czytelne formatowanie kodu prowadzą zwykle do powstania programu z mniejszą ilością błędów. Co więcej, istnieją narzędzia umożliwiające automatyczne badanie, czy wybrany standard kodowania jest przestrzegany [CS, SC]. Ich zastosowanie często pozwala wykryć różnorakie usterki we wczesnej fazie tworzenia oprogramowania.

Wspomniane wcześniej narzędzia nie mogą zagwarantować poprawności programu. Przedstawiają jednak podejście do poszukiwania błędów znacząco różne od testowania. Usterki wykrywane są bez konieczności uruchamiania aplikacji,

na podstawie analizy jej kodu. Przy zastosowaniu tej techniki, zwanej analizą statyczną, można badać różnorakie własności programów, Przykładem takiej analizy jest proces weryfikacji klas przeprowadzany przez maszyny wirtualne Javy [LY99, BH]. Istnieją też narzędzia wyszukujące w kodzie wzorce odpowiadające często popełnianym błędom (takim jak na przykład przedefiniowanie metody `equals`, lecz nie `hashCode` w Javie czy użycie przypisania zamiast porównania w C).

Szczególnym przypadkiem analizy statycznej jest formalne dowodzenie własności programów. Do zastosowania tej techniki niezbędne jest zapisanie semantyki wykorzystywanego języka programowania w wybranym systemie logicznym. Umożliwia to tworzenie dowodów wszelkich[1] własności, które można w owym systemie wyrazić. Zweryfikowanie takiego dowodu za pomocą odpowiednich narzędzi daje gwarancję, że badany program spełnią zadaną specyfikację. Specyfikację taką można stworzyć bezpośrednio jako formułę w odpowiednim systemie lub też wygenerować ją na podstawie opisu w jakimś innym języku. Zaletą drugiego podejścia jest większa przejrzystość owego opisu oraz możliwość wykorzystanie więcej niż jednego systemu logicznego.

Formuły logiczne wyrażające poprawność programu względem specyfikacji nazywane są obligacjami dowodowymi. Generator obligacji dowodowych jest jednym z kluczowych elementów każdego systemu służącego formalnej weryfikacji oprogramowania. Bardzo ważna jest, rzecz jasna, jego poprawność. Duże znaczenie ma też postać generowanych formuł oraz wykorzystywany system dowodzenia. Istotną cechą jest tu skalowalność. Złożoność formalnych dowodów często bowiem sprawia, że w praktyce możliwa (lub opłacalna) jest jedynie weryfikacja bardzo małych systemów.

Dowody poprawności tworzy się zazwyczaj na podstawie programów w postaci źródłowej. Takie podejście ma jednak pewne wady [BRMT05]. Duże znaczenie ma brak błędów w dość złożonej aplikacji, jaką jest kompilator. Ponadto udostępnienie takiego dowodu wiąże się z koniecznością przekazania kodu źródłowego, co nie zawsze jest wskazane. Dlatego czasami lepiej jest korzystać z dowodów odnoszących się bezpośrednio do postaci skompilowanej. W przypadku Javy zastosowanie tej techniki jest znacznie uproszczone, gdyż bajtkod jest językiem stosunkowo wysokiego poziomu.

Drugi rozdział niniejszego dokumentu zawiera krótki opis podstawowych cech wybranych narzędzi do formalnej weryfikacji programów w Javie. Rozdział trzeci skupia się na dokładniejszym przedstawieniu działania generatorów obligacji dowodowych. Zawiera także analizę cech, które taki generator powinien posiadać, aby można go było zastosować przy weryfikacji praktycznych systemów. Przedstawione w tym rozdziale zagadnienia stanowią podstawę moich badań mających na celu stworzenie pracy doktorskiej. Mam więc nadzieję, że opracowanie przynajmniej części z wspomnianych tematów będzie się w owej pracy znajdować.

2 Istniejące narzędzia

Formalna specyfikacja i weryfikacja programów jest dość rozwiniętą dziedziną. Istnieją liczne narzędzia wspierające stosowanie metod formalnych w różnych

[1] Przy nazbyt optymistycznych założeniach odnośnie pełności systemu dowodzenia oraz semantyki

językach programowania (np. SPARK [JC89] dla Ady, Spec♯ [BLR+04] dla C♯, ...). Oczywiście dla Javy również powstają tego typu programy. Poniższy rozdział opisuje najpopularniejsze z nich.

2.1 Dla programów w postaci źródłowej

Większość istniejących narzędzi operuje na programach w postaci źródłowej. W przypadku Javy nie jest ona jednak zbyt odległa od formy skompilowanej. Przedstawione tu programy nie mogą być bezpośrednio zastosowane do kodu bajtowego, ale doświadczenie zdobyte przy ich tworzeniu i wykorzystywaniu może być bardzo użyteczne przy projektowaniu odpowiedników działających na skompilowanych klasach.

2.1.1 JML

Żeby móc mówić o poprawności programu, trzeba najpierw podać jego specyfikację. W przypadku Javy najczęściej stosowanym standardem jej opisu jest JML (ang. Java Modelling Language) [LPC+07, BCC+03]. Język ten pozwala m.in. na uzupełnienie kodu przez dodanie asercji, warunków początkowych i końcowych metod, niezmienników pętli i klas oraz adnotacji rozszerzających system typów. Warunki logiczne definiuje się za pomocą wyrażeń w Javie, wzbogaconej o kilka dodatkowych konstrukcji (pozwalających np. odnieść się do wartości zmiennej z przeszłości).

Istnieje spora liczba programów operujących na kodzie z adnotacjami JML. Niektóre pozwalają na sprawdzanie poprawności asercji w czasie wykonania, ułatwiając tworzenie testów. Inne generują JML na podstawie opisu w innym języku (np. specyfikacji protokołu). Dostępne są również narzędzia automatycznie dodające niektóre adnotacje JML do kodu w Javie.

Z punktu widzenia niniejszego dokumentu najbardziej interesujące są narzędzia umożliwiające formalne dowodzenie poprawności programu względem specyfikacji Najważniejsze z nich przedstawiono poniżej.

2.1.2 ESC/Java2

ESC/Java2 (ang. Extended Static Checker for Java) to narzędzie służące do wyszukiwania błędów w programach z adnotacjami JML. Wykorzystuje systemy do automatycznego dowodzenia twierdzeń (takie jak Simplify czy Z3) w celu zlokalizowania naruszeń specyfikacji. Ciekawą cechą tego narzędzia jest możliwość tłumaczenia kontrprzykładów, dowodzących niepoprawności programu, z języka logiki na ścieżki wykonania powodujące naruszenie specyfikacji [LMS05]. Tego typu informacja zwrotna jest bardzo pożyteczna przy lokalizowaniu i usuwaniu wykrytej usterki.

2.1.3 KeY

KeY [BHS07] to narzędzie służące do dowodzenia poprawności programów w Javie (istnieje również wariant dla C). Jednym z obsługiwanych formatów specyfikacji jest JML. Można także korzystać z OCL [WK99]. Obligacje dowodowe generowane są jako formuły w pewnym wariancie logiki dynamicznej [BK07]. Ich dowody tworzone są przez użytkownika, ale możliwe jest też skorzystanie z

pomocy automatycznych systemów dowodzenia (np. Simplify albo programów obsługujących format SMT-LIB [RT06]).

2.1.4 Krakatoa i Why

Why [Fil03] to system weryfikacji oprogramowania, potrafiący generować obligacje dowodowe z programów i specyfikacji zapisanych w zaprojektowanym do tego celu języku. Krakatoa [MMU04, MPM05] jest narzędziem tłumaczącym kod w Javie i JML do tegoż języka. Why potrafi generować obligacje w formatach wielu różnych narzędzi wspomagających dowodzenie (m.in. Coq, PVS, Isabelle/HOL, Simplify, Z3).

2.2 Dla programów w postaci skompilowanej

Stosunkowo niewiele aplikacji wspiera stosowanie metod formalnych do analizy kodu Javy w postaci skompilowanej. Zestaw narzędzi ułatwiających to zadanie powstały w ramach projektu MOBIUS [BBC+06]. Oprócz tego należy wspomnieć, że niektóre programy do analizy statycznej Javy, takie jak FindBugs [HP04], działają na poziomie bajtkodu (a nie kodu źródłowego).

2.2.1 Mobius

Jednym z założeń projektu MOBIUS było opracowanie zestawu narzędzi umożliwiających stosowanie techniki kodu niosącego dowód (PCC, od ang. Proof-Carrying Code [NL98, CLN+00]) do plików klasowych Javy. Technika ta polega na dodaniu do skompilowanych programów czy bibliotek certyfikatów w postaci dowodów ich poprawności. Oczywiście dowód taki powinien odnosić się do bajtkodu, jako że kod źródłowy nie musi być dostępny podczas weryfikacji. Aby umożliwić korzystanie z wspomnianej techniki stworzono m.in.:

BML BML [CHS+08, CHS09] to oparty na JML język umożliwiający przechowywanie specyfikacji w plikach klasowych.

Umbra Rozszerzenie do Eclipse pozwalające na edycję plików klasowych z adnotacjami BML.

JML2BML Program tłumaczący specyfikację w pliku źródłowym (JML) na specyfikację dla bajtkodu (BML).

bico Program tłumaczący pliki klasowe na struktury w formacie przeznaczonym dla systemu wspomagania dowodzenia Coq [BC04].

bicolano Sformalizowana w Coq-u semantyka bajtkodu Javy. W połączeniu z klasami przetworzonymi przez bico umożliwia formalne wnioskowanie o własnościach skompilowanych programów w Javie.

CCT* Biblioteka umożliwiająca przechowywanie certyfikatów (dowodów poprawności) w plikach `.class`.

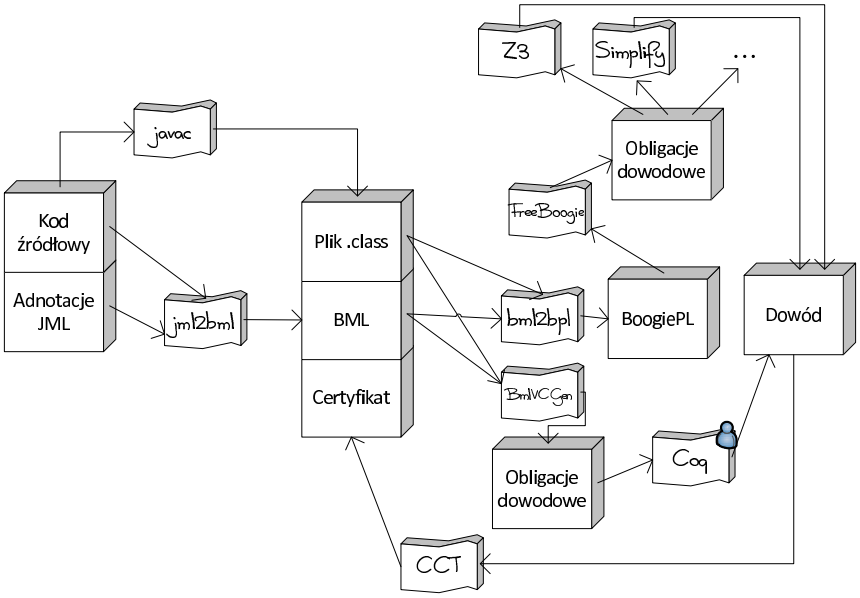
BmlVCGen* Generator obligacji dowodowych, tłumaczący specyfikacje BML na formuły w Coq-u (korzystające z bicolano oraz wyniku działania bico).

FreeBoogie Implementacja Boogie [BCD+06] - generatora obligacji dowodowych, który tłumaczy programy w języku pośrednim, o nazwie BoogiePL [DL05], do postaci formuł logicznych (w formatach stosownych dla różnych systemów dowodzenia).

BML2BPL Program tłumaczący klasę oraz specyfikację BML do BoogiePL.

Narzędzia oznaczona gwiazdką stworzone zostały przez autora niniejszego dokumentu.

Poniższy rysunek pokazuje, jak można wykorzystać opisane wcześniej narzędzia do weryfikacji poprawności plików klasowych.



Rysunek 1: Narzędzia do specyfikacji i weryfikacji programów w kodzie bajtowym Javy.

3 Generowanie obligacji dowodowych

Generator obligacji dowodowych jest jednym z najistotniejszych elementów każdego systemu formalnej weryfikacji oprogramowania. Postać tworzonych formuł, formalizacja semantyki języka programowania oraz własności systemu logicznego, w którym owa semantyka jest wyrażona, mają kluczowe znaczenie dla wydajności i praktyczności procesu weryfikacji. Poniżej przedstawiono kilka własności, które mają wpływ na jakość i użyteczność generatora obligacji dowodowych.

3.1 Poprawność

Jest to dość oczywiste wymaganie, co wcale nie znaczy, że łatwe do spełnienia. Generator obligacji dowodowych dla praktycznych języków programowania i specyfikacji jest z natury narzędziem raczej złożonym. Oczywiście ryzyko błędu można zmniejszyć poprzez formalną weryfikację [NS03, HM94], ale jest to podejście bardzo pracochłonne. Oprócz pomyłki przy tworzeniu samego generatora niełatwo też ustrzec się błędu przy formalizowaniu semantyki, która dla stosowanych w praktyce języków (np. Javy) jest zwykle dość skomplikowana.

Niepoprawność generatora obligacji dowodowych może objawiać się na kilka sposobów. Może powodować błędy wykonania podczas jego działania. Może też być przyczyną powstania obligacji dowodowych, które można udowodnić, mimo, iż program specyfikacji nie spełnia. Niewykluczony jest również przypadek przeciwny – dla poprawnego programu obligacji nie da się udowodnić. Błędy trzeciego rodzaju są szczególnie trudne do zauważenia w początkowej fazie rozwoju systemu.

3.2 Przejrzystość generowanych formuł

Formalne dowodzenie twierdzeń jest zadaniem nietrywialnym. Formuły wyrażające poprawność programów komputerowych są zwykle długie i skomplikowane. Dlatego bardzo ważne jest, aby były one czytelne i zrozumiałe. W przeciwnym wypadku możliwa będzie weryfikacja jedynie bardzo małych programów, z uwagi na pracochłonność tworzenia dowodów. Dobry generator obligacji dowodowych musi zatem zadbać, by produkowane przezeń formuły spełniały wspomniane wcześniej kryteria. Bardzo istotny jest wybór odpowiedniej notacji, która powinna pozwalać na zrozumienie znaczenia poszczególnych (pod-)formuł oraz zawierać informacje o fragmentach kodu, do których odnoszą się owe formuły. Nie bez znaczenia jest też odpowiednie formatowanie wynikowego tekstu.

3.3 Integracja z różnymi formami analizy statycznej

Formalne dowodzenie poprawności pozwala na badanie bardzo wielu własności programów. Zachodzenie niektórych z tych własności można jednak stwierdzić za pomocą narzędzi prostszych lub bardziej efektywnych. Istnieją na przykład programy mogące stwierdzić, że w danej aplikacji nie wystąpi odniesienie do niepoprawnej referencji. Niektóre tego typu narzędzia można rozszerzyć, aby produkowały dowody własności, które stwierdzają. Generator obligacji dowodowych powinien więc umożliwiać łatwą integrację z wspomnianymi programami.

3.4 Modularność

Fundamentalne znaczenie w procesie produkcji oprogramowania ma jego modularność. Możliwość niezależnej pracy nad różnymi elementami systemu jest właściwie niezbędna przy konstruowaniu jakichkolwiek większych aplikacji. Trudno też wyobrazić sobie tworzeniu praktycznych programów bez korzystania z różnorakich bibliotek. Te same czynniki odnoszą się również do procesu weryfikacji. Dowodzenie poprawności jednego modułu nie powinno wymagać odnoszenia się do kodu innych modułów, choć może być niezbędna znajomość ich specyfikacji. Jednak generowanie obligacji dowodowych w sposób zapewniający taką własność jest zadaniem wysoce nietrywialnym [MÖ2].

Większość stosowanych obecnie generatorów obligacji dowodowych działa przy założeniu, że kod całego systemu jest znany. Usunięcie tego założenia powoduje poważne trudności. Nie jest bowiem jasne, w jaki sposób dowodzić takich na przykład własności:

- Wywołanie metody wirtualnej nie powoduje końca świata ani innych nieprzewidzianych efektów?
- Procedura, której treść nie jest znana, modyfikuje tylko te pola, które może modyfikować procedura wołająca?
- Procedura nie narusza niezmienników, których treść nie jest znana podczas weryfikacji tejże procedury? [MPHL06, MHKL05, BDF+03]

Niektóre z wspomnianych problemów można usunąć wprowadzając mechanizm dziedziczenia specyfikacji. Wprowadzenie go w sposób nieodpowiedni może jednak prowadzić do generowania niepoprawnych obligacji dowodowych dla odpowiednio spreparowanego kodu. Oprócz tego wciąż istotny pozostaje problem siły wyrazu języka specyfikacji. Jeśli wymaga on np. żeby po wyjściu z każdej metody zachodziły niezmienniki *wszystkich* żywych obiektów, a zarazem nie wprowadza żadnych ograniczeń odnośnie zbioru zmiennych, od których taki niezmiennik może zależeć, to łatwo może się okazać, że udowodnienie poprawności nawet bardzo prostych metod jest niemożliwe. Ciekawe problemy można też napotkać przy specyfikowaniu zbioru pól, których wartości mogą się zmienić po wykonaniu danej metody. W wielu wypadkach zbioru tego nie da się określić na podstawie samego kodu procedury z uwagi na występujące w nich wywołania wirtualne. Można oczywiście wprowadzić ograniczenie w widocznej specyfikacji wołanej metody, ale czasami jest to zbyt restrykcyjne. Na przykład funkcja, która ma być wywołana po naciśnięciu przez użytkownika przycisku, będzie pewnie zmieniać wartości różnych, trudnych do określenia pól.

Osobną kwestią, związaną z formalnym wnioskowaniem o modularnych programach, jest kwestia modułów pisanych w różnych językach programowania. Często stosowanym podejściem jest stworzenie specyfikacji dla takich modułów i uznanie ich poprawności za aksjomat. Jeśli jednak dla wykorzystywanych przez nie języków programowania istnieje jakiś sposób specyfikacji i dowodzenia własności, można pokusić się o próbę połączenie dowodów. Oczywiście główny problem stanowi fakt, że obie specyfikacje (i dowody) będą w takim wypadku korzystać z różnych systemów logicznych (i innych formalizacji semantyki, modelu pamięci itp.). Poprawne ich połączenie nie musi więc być łatwe. Na szczęście badanie relacji między różnymi systemami logicznymi nie jest dziedziną nową i można skorzystać z pewnych istniejących wyników [GB92, CKTW08].

3.5 Interfejs

Wspomniane wcześniej wymagania odnoszące się do przejrzystości generowanych formuł sugerują, że w procesie weryfikacji bardzo pożyteczny może być graficzny interfejs do edycji dowodów. Dla wielu systemów wspomagających dowodzenie istnieją takie interfejsy (np. ProofGeneral [Asp00], CoqIde czy Mobius Prover Editor [CK08]), jednak nie spełniają one wszystkich wymagań związanych z dowodzeniem poprawności programów. Bardzo pomocny mógłby być interfejs pozwalający z jednej strony zwizualizować zależność formuł od kodu, do którego się odnoszą, z drugiej zaś wydobyć z zestawu udowodnionych już faktów informacje o wskazanym elemencie struktury programu (np. powiedzieć, czy udowodniono już, że dana zmienna ma wartość różną od null albo należącą do jakiegoś zakresu).

3.6 Współbieżność

Wraz z rozwojem systemów wieloprocesorowych i wielordzeniowych coraz większego znaczenia nabiera programowanie współbieżne. Dowodzenie własności programów współbieżnych nie jest jednak zadaniem prostym. Duży problem stanowi tu fakt, że semantyka takich programów jest zwykle bardzo skomplikowana [MPA05]. Istnieją wprawdzie narzędzia pozwalające badać różne własności algorytmów równoległych, jednak dowodzenie poprawności takich programów, napisanych np. w Javie, jest znacznie trudniejsze. Z drugiej strony duża liczba błędów spotykanych w aplikacjach współbieżnych wskazuje, że w tej właśnie dziedzinie techniki weryfikacji formalnej mogą oddać nieocenione usługi.

Literatura

[Asp00] David Aspinall. Proof general: A generic tool for proof development. In *Proceedings of TACAS 2000*, number 1785 in LNCS, page 38, 2000.

[BBC+06] G. Barthe, L. Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. MOBIUS: Mobility, ubiquity, security. objectives and progress report. In *Trustworthy Global Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[BCC+03] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for

object-oriented programs. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, 2006.

[BDF+03] M. Barnett, R. DeLine, M. Fahndrich, K. Rustan, M. Leino, and W. Schulte. Verification of object-oriented programs with invariants, 2003.

[Bec03] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[BH] G. Bracha and G. Hamilton. Jsr 202: Classfile specification update. http://jcp.org/en/jsr/detail?id=202.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[BK07] Bernhard Beckert and Vladimir Klebanov. A dynamic logic for deductive verification of concurrent programs. In Mike Hinchey and Tiziana Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK*. IEEE Press, 2007.

[BLR+04] Mike Barnett, K. Rustan M. Leino, K. Rustan, M. Leino, and Wolfram Schulte. The spec♯ programming system: An overview. pages 49–69. Springer, 2004.

[BRMT05] Gogul Balakrishnan, Thomas W. Reps, David Melski, and Tim Teitelbaum. WYSINWYX: What you see is not what you execute. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2005.

[CHS+08] Jacek Chrząszcz, Marieke Huisman, Aleksy Schubert, Joseph Kiniry, Mariela Pavlova, , and Erik Poll. *BML Reference Manual*, December 2008. In Progress. Available from http://bml.mimuw.edu.pl.

[CHS09] Jacek Chrząszcz, Marieke Huisman, and Aleksy Schubert. BML and related tools. In *Software Technologies Concertation on Formal Methods for Components and Objects (FMCO 2008)*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear.

[CK08] J. Charles and J. Kiniry. A lightweight theorem prover interface for eclipse. In *Proc. 8th International Workshop on User Interfaces for Theorem Provers (UITP 2008)*, 2008.

[CKTW08] María Victoria Cengarle, Alexander Knapp, Andrzej Tarlecki, and Martin Wirsing. A heterogeneous approach to UML semantics. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 383–402. Springer, 2008.

[CL02] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 231–255, Malaga, Spain, June 2002. Springer Verlag.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, New York, NY, USA, 2000. ACM Press.

[CS] Checkstyle. http://checkstyle.sourceforge.net/.

[DL05] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (MSR), March 2005.

[Fil03] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[GB92] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[HM94] P. V. Homeier and D. F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. *Lecture Notes in Computer Science*, 859:269–??, 1994.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM.

[HT03] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. ThePragmaticProgrammers, 2003.

[JC89] T. J. Jennings and B. A. Carré. A subset of ada for formal verification (SPARK). *Ada User*, 9(Supplement):121–126, 1989.

[LMS05] Leino, Millstein, and Saxe. Generating error traces from verification-condition counterexamples. *SCIPROG: Science of Computer Programming*, 55, 2005.

[LPC+07] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML Reference Manual. October 2007.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Second edition, 1999.

[MÖ2] Peter Müller. *Modular specification and verification of object-oriented programs*. Springer-Verlag, New York, NY, USA, 2002.

[MHKL05] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Cooperation-based invariants for OO languages. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS'05)*, 2005.

[MMU04] C. Marche, C. Paulin Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, January 2005.

[MPHL06] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[MPM05] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194, Berlin, August 2005. Springer-Verlag.

[NL98] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*, pages 333–344, New York, NY, USA, 1998. ACM Press.

[NS03] G. Necula and R. Schneck. A Sound Framework for Untrustred Verification-Condition Generators. In *Proc. of LICS03*, 2003.

[RT06] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[SC] Microsoft stylecop. `http://blogs.msdn.com/sourceanalysis/`.

[WK99] Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.