Architectural Specifications and Their Verification

Piotr Hoffman
Autoreferat rozprawy doktorskiej

3 lutego 2005

1 Wprowadzenie

Niniejsza rozprawa jest poświęcona *specyfikacjom architekturalnym*. Przez architekturę systemu (programu) rozumiemy nie tyle pewną własność wymaganą od ukończonego systemu, ile raczej strukturę procesu jego tworzenia. Zgodnie z założeniami koncepcji formalnego rozwijania opisanego za pomocą specyfikacji algebraicznych oprogramowania [ST97], sam proces tworzenia systemu powinien mieć charakter sformalizowany. Uproszczoną, teoretyczną postacią takiej formalizacji jest pojęcie implementacji konstruktorowej [ST88b, SST92]. Specyfikacje architekturalne rozszerzają i uszczegółowiają pojęcie konstruktora. Rozszerzają przez to, że o ile konstruktury dopuszczały jedynie możliwość liniowego rozwijania programu, to specyfikacje architekturalne pozwalają również dzielić zadania programistyczne na mniejsze, niezależne części. W praktyce taka możliwość jest oczywiście niezbędna, można wręcz powiedzieć, że istotą procesu budowy oprogramowania jest właśnie podział zadań większych na mniejsze, wzajemnie niezależne podzadania. Specyfikacje architekturalne uszczegółowiają pojęcie konstruktura przez to, że mają własną składnię i semantykę, nie będąc, jak konstruktory, jedynie abstrakcyjnymi obiektami matematycznymi.

Krótko można specyfikację architekturalną zdefiniować jako formalne wyrażenie, które opisuje, jak rozbić pewne zadanie programistyczne $\mathcal{T}$ na mniejsze, niezależne zadania programistyczne $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Przy tym rozpatruje się zasadniczo dwie klasy zadań — albo zadania implementacji zwykłej specyfikacji SP, tzw. specyfikacji modułu nieparametrycznego, albo zadanie implementacji specyfikacji modułu parametrycznego postaci $SP \to SP'$. Zadanie programistyczne odpowiadające takiej specyfikacji to zadanie zaimplementowania funkcji, która dla każdego programu spełniającego SP zbuduje rozszerzający go program spełniający specyfikację SP'. Można więc powiedzieć, że moduły parametryczne odpowiadają pojęciu funktora w języku Standard ML [Pau96], a specyfikacja modułu parametrycznego odpowiada specyfikacji funktora; przykładem języka, w którym można definiować takie specyfikacje, jest Extended ML [KST97]. Wprowadzenie modułów parametrycznych jest konieczne, ponieważ są one tym mechanizmem, który umożliwia dzielenie skomplikowanych zadań na niezależne części.

Specyfikacje architekturalne zostały po raz pierwszy wprowadzone w języku specyfikacji algebraicznej CASL [ABK$^+$02, BM04, CoF04, BST02]. Język ten można widzieć jako składający się z dwóch do pewnego stopnia niezależnych formalizmów. Jeden formalizm to język specyfikacji algebraicznych; składają się na niego specyfikacje podstawowe, tzn. w zasadzie listy aksjomatów w logice CASL, oraz specyfikacje strukturalne, zbudowane ze specyfikacji podstawowych przy użyciu mechanizmów łączenia specyfikacji w większe całości. Drugi formalizm to właśnie formalizm specyfikacji architekturalnych, który służy do opisywania procesu rozwijania oprogramowania.

Istnieje zasadnicza różnica między pojęciami specyfikacji strukturalnych i architekturalnych. Specyfikacja strukturalna jest zwyczajnym opisem wymaganych własności systemu, a jej "strukturalność" polega na tym, że opis ten nie ma postaci listy aksjomatów, lecz ma własną strukturę. Specyfikacja architekturalna z kolei nie jest opisem wymaganych własności programu, tylko opisem procesu, który pozwoli nam uzyskać program spełniający interesującą nas specyfikację. Obserwację, że powyższe specyfikacje mogą opisywać różne struktury, wyraża hasło "struktura specyfikacji różni się od struktury implementacji" (zob. przykład w [FJ90]).

Podkreślić trzeba, że mechanizm specyfikacji strukturalnych jest zdefiniowany w sposób sparametryzowany przez logikę, w której zapisywane są aksjomaty (zob. [ST88a]). Podobnie mechanizm specyfikacji architekturalnych sparametryzowany jest przez język specyfikacji, w którym specyfikowany jest program, którego stworzenie jest naszym celem. Pewną formalizacją pojęcia logiki, odpowiednią dla ścisłego opisu takiej parametryzacji, jest pojęcie instytucji [GB92, Tar99]. Ze względów technicznych w rozprawie korzystam z opartego na instytucji, jednak nieco się od niej różniącego, abstrakcyjnego pojęcia języka specyfikacji.

Przykładem takiego języka jest język budowania specyfikacji CASL. Ponieważ jest to jeden z najważniejszych obecnie języków specyfikacji, a także dlatego, że ma on wiele interesujących własności, jednym z celów rozprawy jest stworzenie takich metod weryfikacji, które dałoby się zastosować do tego języka. Jest to szczególnie trudne z tego względu, że język ten nie posiada własności amalgamacji [SMT$^+$01b, SMT$^+$05].

Rozprawa podzielona jest na trzy rozdziały. Rozdział wstepny wprowadza podstawowe pojęcia, takie jak język specyfikacji oraz język programowania. Pojęcie języka programowania, całkowicie zresztą oderwane od jakichkolwiek rozważań dotyczących obliczalności, jest nam potrzebne po to, by móc rozważać różnice między podejściem traktującym proces rozwijania oprogramowania jako proces oparty bezpośrednio na programach a podejściem traktującym go jako proces oparty na modelach (algebrach), które reprezentują semantykę programów. W rozdziale wstępnym definiujemy też pojęcia takie jak amalgamacja, własność odbijania izomorfizmów lub własność niesprzeczności sygnatur. Pojęcia te okażą się kluczowe w dalszych rozważaniach.

Rozdział II poświęcony jest zdefiniowaniu i szczegółowej analizie pojęcia specyfikacji architekturalnych.

W rozdziale III prezentujemy metody weryfikacji specyfikacji architekturalnych, tj. metody dowodzenia, że dana specyfikacja architekturalna opisuje poprawny proces budowy programu.

Poniżej przedstawiamy najważniejsze otrzymane wyniki.

2 Analiza specyfikacji architekturalnych

Specyfikacje architekturalne jako takie opisują pojedyncze kroki w procesie rozwijania oprogramowania. W rozprawie wprowadzamy naturalne pojęcie drzewa rozwoju oprogramowania (inną propozycję takiego pojęcia można znaleźć w [MST04]) i definiujemy pojęcia poprawności specyfikacji architekturalnej oraz poprawności drzewa rozwoju oprogramowania. W Twierdzeniu 8 pokazujemy, że drzewo rozwoju oprogramowania można spłaszczyć, jednak tylko przy dodatkowych technicznych założeniach (podobne twierdzenie znajduje się w pracy [BST02]).

Specyfikacje architekturalne definiuje się zwykle na poziomie modeli (algebr) pewnej logiki. Powstaje wobec tego pytanie, czy poprawność specyfikacji architekturalnej na poziomie modeli rzeczywiście gwarantuje nam, że zastosowanie opisanej przez tę specyfikację procedury do programów również da pożądany skutek (por. [ST96]). Odpowiedź na to pytanie ma znaczenie fundamentalne, bowiem naszym ostatecznym celem jest otrzymanie programu, a nie abstrakcyjnego modelu (algebry).

W rozprawie pokazujemy, że by odpowiedź była pozytywna, niezbędna jest zmiana semantyki specyfikacji architekturalnych na tzw. semantykę częściową. W Twierdzeniu 10 pokazujemy, że przy semantyce częściowej spłaszczanie drzew rozwoju oprogramowania możliwe jest zawsze, bez jakichkolwiek dodatkowych założeń. Następnie porównujemy zwyczajną semantykę z semantyką częściową i dowodzimy, że poprawność względem semantyki częściowej m.in. gwarantuje poprawność względem semantyki zwyczajnej (Twierdzenie 14). Pokazujemy też, że pod pewnymi naturalnymi warunkami poprawność względem semantyki zwyczajnej gwarantuje też poprawność względem semantyki częściowej (Twierdzenie 16). Od tego momentu przez poprawność będziemy rozumieć poprawność względem semantyki częściowej.

Ponieważ proste przykłady wskazują, że poprawność na poziomie modeli nie zawsze gwarantuje poprawność na poziomie programów, konieczne jest wprowadzenie nowych założeń; dlatego zakładamy, że interesują nas tylko języki programowania, które mają własność amalgamacji. Dalej definiujemy pojęcie w pełni statycznej poprawności i dowodzimy fundamentalnego Twierdzenia 20. Mówi ono, że przy pewnych prostych założeniach następujące warunki są równoważne dla dowolnej specyfikacji architekturalnej:

- specyfikacja jest w pełni statycznie poprawna,
- poprawność specyfikacji na poziomie modeli gwarantuje jej poprawność na poziomie programów.

Dowód powyższego twierdzenia jest dość skomplikowany i opiera się na dwóch ważnych twierdzeniach pomocniczych, a mianowicie Twierdzeniach 21 i 25. Pierwsze z nich warto przytoczyć: mówi ono, że jeżeli specyfikacja architekturalna jest poprawna na poziomie modeli, a po usunięciu wszystkich aksjomatów jest poprawna na poziomie programów, to jest w całości poprawna na poziomie programów.

Twierdzenie 29, z którego będziemy korzystali w dalszej części rozprawy, zawiera inną charakteryzację w pełni statycznej poprawności. Mówi ono, że język specyfikacji ma własności odbijania izomorfizmów oraz niesprzeczności sygnatur wtedy i tylko wtedy, gdy dla każdej generatywnej specyfikacji architekturalnej następujące warunki są równoważne:

- specyfikacja jest w pełni statycznie poprawna,
- specyfikacja po usunięciu aksjomatów jest poprawna.

Specyfikację architekturalną nazywamy generatywną, jeżeli żaden zadeklarowany w niej moduł parametryczny nie jest wykorzystywany więcej niż jednokrotnie. Jednym z założeń Twierdzeń 20, 21 i 25 również była generatywność rozważanych specyfikacji architekturalnych.

Twierdzenie 30 pokazuje, że generatywność jest rzeczywiście cechą bardzo istotną. Z twierdzenia tego wynika bowiem, że niemożliwe jest stworzenie pojęcia rozszerzającego pojęcie w pełni statycznej poprawności na specyfikacje architekturalne, które nie są generatywne. Z tego względu w dalszych rozważaniach skupiamy się na specyfikacjach generatywnych.

3 Weryfikacja specyfikacji architekturalnych

Metody weryfikacji specyfikacji architekturalnych i drzew rozwoju oprogramowania przedstawione w niniejszej rozprawie są to w zasadzie algorytmy generujące pewną liczbę tzw. zobowiązań dowodowych, to jest twierdzeń, których prawdziwość jest równoważna poprawności specyfikacji architekturalnej lub drzewa rozwoju. Zobowiązania te są postaci

$$[\![SP]\!] \subseteq [\![SP']\!]$$

tj. mówią one, że pewna specyfikacja SP pociąga za sobą specyfikację SP' ($[\![SP]\!]$ oznacza tutaj klasę modeli spełniających specyfikację SP). W wielu logikach prawdziwość takich zobowiązań jest nierozstrzygalna. Stąd proponowane przez nas metody weryfikacji, choć są poprawne i zupełne, w istocie są tylko względnie zupełne, są bowiem zupełne tylko przy założeniu, że umiemy sprawdzić, czy wygenerowane zobowiązania dowodowe są prawdziwe.

Najprostsza proponowana metoda weryfikacji stosuje się do języków specyfikacji, które mają własność amalgamacji. Ma ona zastosowanie do specyfikacji architekturalnych (i analogicznie drzew rozwoju oprogramowania), które są generatywne i w pełni statycznie poprawne. Zgodnie z Twierdzeniem 35 metoda ta jest poprawna i zupełna. Tę właśnie najprostszą metodę należy uznać za wzorcową. Jej jedynym poważnym ograniczeniem jest to, że wymaga od języka specyfikacji własności amalgamacji, a niektóre istotne języki, takie jak Casl, tej własności nie mają.

Następna metoda korzysta z pojęcia rozszerzonej statycznej poprawności; jest to pojęcie uogólniające pojęcie w pełni statycznej poprawności na języki pozbawione własności amalgamacji. Potrzebne jest nam także pojęcie reprezentacji jednego języka specyfikacji w innym; będziemy bowiem reprezentować język specyfikacji bez własności amalgamacji w jakimś języku "lepszym", który tę własność posiada. Wśród reprezentacji wyróżniamy reprezentacje całkowite, tj. takie, które nie korzystają ze zdań języka "lepszego" do reprezentowania klas modeli języka reprezentowanego.

Ponieważ pojęcie rozszerzonej statycznej poprawności jest zdefiniowane w sposób niekonstruktywny, konieczne jest najpierw stworzenie metody jej sprawdzania. Twierdzenie 41 mówi, że jeśli mamy odpowiednią całkowitą reprezentację w języku z amalgamacją i odbijającym izomorfizmy, to można zredukować sprawdzanie rozszerzonej statycznej poprawności w języku reprezentowanym do sprawdzania w pełni statycznej poprawności w języku "lepszym".

Z kolei Twierdzenie 42 mówi, że jeśli przy podobnych założeniach specyfikacja architekturalna (drzewo rozwoju oprogramowania) w języku reprezentowanym jest generatywna i w rozszerzony sposób statycznie poprawna, to redukcja do poprzedniej metody daje nam poprawną i zupełną metodę weryfikacji.

Trzecia metoda weryfikacji nie korzysta w ogóle z pojęć w pełni statycznej lub rozszerzonej statycznej poprawności. Wystarcza jej także dowolna reprezentacja, niekoniecznie reprezentacja całkowita. Natomiast wymaga ona od "lepszego" języka specyfikacji całego szeregu dość szczegółowych własności. Zgodnie z Twierdzeniem 47 przy tych założeniach trzecia metoda weryfikacji jest również poprawna i zupełna. Zaznaczyć należy, że liczba generowanych przez trzecią metodę zobowiązań dowodowych jest znacznie większa niż dla metod poprzednich. Wynika to stąd, że warunki wcześniej sprawdzane statycznie muszą tu być za każdym razem dowodzone. Powoduje to, że wartość praktyczna tej metody jest ograniczona.

W dalszej części rozprawy pokazujemy, że metody druga i trzecia mogą zostać zastosowane do języka specyfikacji CASL. W tym celu konieczne jest zdefiniowanie odpowiedniego "lepszego" języka specyfikacji i odpowiedniej reprezentacji oraz udowodnienie, że mają one wszystkie wymagane w Twierdzeniach 41, 42 i 47 własności.

Najpierw zajmujemy się metodą trzecią. Reprezentujemy język CASL w nieco uboższej wersji jego samego, mianowicie w wersji pozbawionej podrodzajów. To właśnie podrodzaje są przyczyną braku w języku CASL własności amalgamacji. Największym problemem jest teraz udowodnienie, że skomplikowane własności potrzebne dla stosowania metody trzeciej rzeczywiście zachodzą. Jest to wykazane w Twierdzeniu 53, którego dowód opiera się głównie na Twierdzeniu 52.

Następnie przystępujemy do konstruowania całkowitej reprezentacji języka CASL w jego rozszerzeniu, które nazywamy ENRICHEDCASL (idea takiego rozszerzenia została przedstawiona w [SMT+05]). Język ENRICHEDCASL tym różni się od CASL, że pozwala na istnienie wielu różnych zanurzeń podrodzaju w nadrodzaju; dzięki temu ma on własność amalgamacji. Większość wymaganych własności języka ENRICHEDCASL oraz reprezentacji w nim języka CASL można udowodnić bez wielkich trudności. Istnienie tych własności potwierdza Twierdzenie 56.

Inaczej rzecz przedstawia się z własnością odbijania izomorfizmów w języku ENRICHEDCASL. O posiadaniu tej własności, a także własności niesprzeczności sygnatur, mówi Twierdzenie 57. Jednak jego dowód znajduje się dopiero pod sam koniec pracy, gdyż do jego przeprowadzenia konieczne jest skorzystanie z innych wyników.

Przede wszystkim konieczne okazało się wykorzystanie wyników dotyczących teorii amalgamacji dla półgrup [How96] i uogólnienie ich na operady i multikategorie [Lei04]. Dotyczy to w szczególności Twierdzenia Isbella o Zygzaku [Isb66]. Jego uogólnieniem jest Twierdzenia 68, którego dowód jest nieco rozszerzonym nowym i krótkim dowodem Twierdzenia o Zygzaku w wersji dla monoidów (elegancki dowód [Hig90, Hig92] okazał się niepoprawny, podczas gdy dowód [How96] jest nieelementarny). Dzięki Twierdzeniu o Zygzaku możemy pokazać, że funktor generujący algebry nad multikategorią ma słabą wersję własności odbijania izomorfizmów (Twierdzenie 70). Pozwala to udowodnić Twierdzenie 57, a więc własność odbijania izomorfizmów dla języka specyfikacji EnrichedCasl. Znaczy to, że wszystkie warunki konieczne dla zastosowania drugiej metody weryfikacji do Casl są spełnione.

4 Uwagi końcowe

Niniejsza rozprawa zawiera niektóre wyniki lub odwołuje się do wyników opublikowanych w pracach [SMT+01b, SMT+05, KHT+01]. Zawiera też większość wyników opublikowanych w pracach [Hof01], a szczególnie w [Hof03] i w rozdziale mojego autorstwa w [MHAH04]. Znajduje się w niej także wiele wyników jeszcze niepublikowanych.

Oprócz stworzonych metod weryfikacji dla specyfikacji architekturalnych, ważnym efektem pracy są także wnioski dotyczące tego, jakie zmiany należy wprowadzić do semantyki specyfikacji architekturalnych lub jakie wymagania powinno się stawiać wykorzystywanym dla celów specyfikacji architekturalnych językom specyfikacji.

Rozprawa uzupełniona jest przykładem zastosowania zaprezentowanych metod weryfikacji do specyfikacji architekturalnych opisujących budowę oprogramowania systemu kierującego kotłem parowym (tzw. steam-boiler example; zob. [ABL96]). Specyfikacje te są głównym przykładem zastosowania języka Casl przedstawionym w książce [BM04].

Literatura

[ABK+02] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[ABL96] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, LNCS Vol. 1165. Springer, 1996.

[BM04] Michel Bidoit and Peter D. Mosses. Casl *User Manual: Introduction to Using the Common Algebraic Specification Language.*

LNCS Vol. 2900 (IFIP Series). Springer, 2004. With chapters by Till Mossakowski, Donald Sannella, and Andrzej Tarlecki.

[BST02] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*. LNCS Vol. 2960 (IFIP Series). Springer, 2004.

[FJ90] John S. Fitzgerald and Cliff B. Jones. Modularizing the formal description of a database system. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM Europe*, LNCS Vol. 428, pages 189–210. Springer, 1990.

[GB92] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39:95–146, 1992.

[Hig90] Peter M. Higgins. A short proof of Isbell's zigzag theorem. *Pacific Journal of Mathematics*, 144(1):47–50, 1990.

[Hig92] Peter M. Higgins. *Techniques of Semigroup Theory*. Oxford University Press, 1992.

[Hof01] Piotr Hoffman. Verifying architectural specifications. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, 2001, Selected Papers*, LNCS Vol. 2267, pages 152–175. Springer, 2001.

[Hof03] Piotr Hoffman. Verifying generative CASL architectural specifications. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 233–252. Springer, 2003.

[How96] John M. Howie. *Fundamentals of Semigroup Theory*. London Mathematical Society Monographs, New Series, No. 12. Oxford University Press, 1996.

[Isb66] John R. Isbell. Epimorphisms and dominions. In *Conference on Categorical Algebra, La Jolla, California, 1965, Proceedings*, pages 232–246. Springer, 1966.

[KHT+01] Bartek Klin, Piotr Hoffman, Andrzej Tarlecki, Lutz Schröder, and Till Mossakowski. Checking amalgamability conditions for Casl architectural specifications. In J. Sgall, A. Pultr, and P. Kolman, editors, *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001, Marianske Lazne, Czech Republic, Proceedings*, LNCS Vol. 2136, pages 451–463. Springer, 2001.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[Lei04] Tom Leinster. *Higher Operads, Higher Categories*. London Mathematical Society Lecture Notes Series Vol. 298. Cambridge University Press, 2004.

[MHAH04] Till Mossakowski, Piotr Hoffman, Serge Autexier, and Dieter Hutter. Casl logic. In Casl Reference Manual [CoF04], part IV. Edited by T. Mossakowski.

[MST04] Till Mossakowski, Donald Sannella, and Andrzej Tarlecki. A simple refinement language for Casl. In J.-L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 17th International Workshop, WADT 2004, Barcelona, Spain, 2004, Revised Selected Papers*, LNCS Vol. 3423, pages 162–185. Springer, 2004. To appear.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[SMT01a] Lutz Schröder, Till Mossakowski, and Andrzej Tarlecki. Amalgamation in Casl via enriched signatures. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, Proceedings*, LNCS Vol. 2076, pages 993–1004. Springer, 2001.

[SMT+01b] Lutz Schröder, Till Mossakowski, Andrzej Tarlecki, Piotr Hoffman, and Bartek Klin. Semantics of architectural specifications in CASL. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Genova, Italy, Proceedings*, LNCS Vol. 2029, pages 253–268. Springer, 2001.

[SMT+05] Lutz Schröder, Till Mossakowski, Andrzej Tarlecki, Bartek Klin, and Piotr Hoffman. Amalgamation in the semantics of CASL. *Theoretical Computer Science*, 331(1):215–247, 2005. extends [SMT+01b, SMT01a].

[SST92] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992.

[ST88a] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, 1988.

[ST88b] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25(3):233–281, 1988.

[ST96] Donald Sannella and Andrzej Tarlecki. Mind the gap! abstract versus concrete models of specifications. In W. Penczek and A. Szałas, editors, *Mathematical Foundations of Computer Science 1996, 21st International Symposium, MFCS'96, Cracow, Poland, September 2-6, 1996, Proceedings*, LNCS Vol. 1113, pages 114–134. Springer, 1996.

[ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9(3):229–269, 1997.

[Tar99] Andrzej Tarlecki. Institutions: An abstract framework for formal specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, chapter 4. Springer, 1999.