Überprüfung von Softwareskizzen und -entwürfen

Ehrich, Hans-Dieter

Veröffentlicht in:
Jahrbuch 2000 der Braunschweigischen Wissenschaftlichen Gesellschaft, S.69-72

J. Cramer Verlag, Braunschweig

HANS-DIETER EHRICH, Braunschweig

Überprüfung von Softwareskizzen und -entwürfen

Braunschweig, 13.10.2000*

1. Einleitung

Software ist ein besonderer Stoff: Produkt auf einem expansiven Markt und ein universelles Medium zur Darstellung von Information.

Als Produkt wird Software geplant, entworfen, implementiert, geprüft, verpackt, versandt, installiert, lizensiert, gewartet – und bezahlt. Planung, Entwurf und Implementierung erfordern einen großen Aufwand an hochqualifiziertem Personal und sind entsprechend teuer, der eigentliche Herstellungsprozess aber ist einfach und billig; man braucht dazu weder besonders qualifiziertes Personal noch fertigungstechnischen Aufwand und kaum Materie oder Energie (was den Diebstahl, d.h. die Herstellung von Raubkopien, begünstigt).

Software ist aber auch ein Medium zur Darstellung von Information, hier gibt es Parallelen zu biologischen Medien wie der DNS und dem Nervensystem sowie den kulturellen Medien wie dem Buch, dem Film und den Tonträgern. Im Unterschied zu diesen kulturellen Medien ist die dargestellte Information aber dynamisch änderbar, und sie kann durch die darunterliegende Hardware operativ wirksam werden: über angeschlossene Sensoren kann die Umwelt wahrgenommen und durch Aktoren manipuliert werden.

Welches also ist die richtige Modellvorstellung für die Herstellung von Software? Zuweilen wird hierfür das Bild der Fabrik bemüht, aber es passt nicht: der Schwerpunkt liegt beim kreativen Entwerfen und Herstellen eines Prototypen, nicht bei dessen Vervielfältigung. Dies trifft auch für das Schreiben von Büchern zu, aber auch dies taugt nicht recht als Analogon: Autoren von Programmen fühlen sich eher als Techniker; man spricht zu Recht von Software *Engineering.* Nicht schlecht passen Bilder aus dem Bauwesen und der Architektur: Software wird mit erlernbaren Ingenieursmethoden geplant, entwickelt, auf individuelle Bedürfnisse zugeschnitten und realisiert, wobei durchaus Raum ist für die individuelle Kreativität des Ingenieurs. Das Besondere an Software ist, daß sich dies alles in der immateriellen Welt der Zeichen, Symbole und Regeln abspielt; an die Stelle der materiellen naturwissenschaftlichen Grundlagen der konventionellen Ingenieursdisziplinen tritt die mathematische Logik.

2. Modellierung und Implementierung

Wie Bauwerke werden Softwareprodukte geplant, bevor sie implementiert werden, und wie bei Bauwerken sind Modelle unerläßliche Mittel der Planung und Analyse. Modelle

---

* Kurzfassung eines Vortrags gehalten in der Klasse für Mathematik und Naturwissenschaften der Braunschweigischen Wissenschaftlichen Gesellschaft.

ermöglichen es, zwischen Auftraggeber und Entwerfer akzeptable Lösungen auszuhandeln und schriftlich festzuhalten, die dann verläßliche (und ggf. gerichtsfeste) Vorgaben für die Implementierung bilden.

Ein gutes Modell sollte es insbesondere erlauben, zu beurteilen, ob das Anwendungsproblem adäquat gelöst ist, und später, ob die Lösung korrekt implementiert wurde. Natürlich sind auch andere Dinge wichtig, z.B. abzuschätzen, wie lange die Entwicklung dauern wird, wieviel sie kosten wird, wie angenehm das Produkt zu benutzen sein wird, wie sicher es sein wird, wie gut es wartbar sein wird und Einiges mehr. Angesichts der hohen volkswirtschaftlichen Kosten fehlerhafter Software ist korrektes Funktionieren aber ein vordringliches Anliegen; es ist wie mit der Gesundheit: Korrektheit ist nicht Alles, aber ohne Korrektheit ist alles Nichts.

Und Korrektheit hat diese beiden Hauptaspekte: Adäquatheit des Modells und Korrektheit der Implementierung. Modellierungsfehler, die man erst nach der Implementierung erkennt, sind besonders teuer, sie können große Teile der Implementierung und damit viel teure Arbeit obsolet machen. Wie man Modellierungsfehler vor der Implementierung finden kann ist also ein kritisches Problem.

3. Überprüfung

Je nach Phase der Softwareentwicklung und zu prüfender Systemeinheit bieten sich unterschiedliche Prüfverfahren an. Generell gibt es dynamische Methoden, bei denen das Prüfstück im Betrieb erprobt wird, und statische Methoden, bei denen das Prüfstück außer Betrieb analysiert wird. Stand der Technik ist i.w. dynamisches Prüfen im finalen Stadium der Entwicklung, nämlich Testen der Implementierung. Hier gibt es einige Fortschritte bei der Computerunterstützung, bis hin zum automatischen Testen von (relativ kleinen) Systemkomponenten.

Die Überprüfung von Modellen vor der Implementierung hat bisher nur zögernd Eingang in die Praxis gefunden, jedoch läßt die zunehmende Unzufriedenheit mit Qualität und Kosten von Software ein steigendes Interesse erwarten. Modelle werden in der Praxis vorwiegend statisch überprüft, nämlich durch Inspektion und Diskussion.

In Teilbereichen der Softwareentwicklung, z.B. bei Kommunikationsprotokollen und anderen wenig datenintensiven Anwendungen, finden jedoch zunehmend logikbasierte Verifikationsmethoden Anwendung, wie sie für Hardware erfolgreich eingesetzt wurden. Es gibt im Wesentlichen zwei Ansätze: Deduktion und Model Checking.

Um Deduktionsmethoden, d.h. Methoden des automatischen Beweisens, anwenden zu können, muß man das Modell in einem geeigneten Logik-Kalkül durch Axiome beschreiben. Mittels eines Deduktionssystems für diese Logik kann dann eine gegebene Prüfbedingung φ durch den Nachweis ihrer Herleitbarkeit aus den Axiomen A überprüft werden: $A \vdash \varphi$.

Model Checking erfordert, daß ein Zustands-Übergangsmodell des Systems konstruiert wird. Die Prüfbedingung φ wird in einer temporalen Logik formuliert. Der Model Checker, ein komplexes Programm ([CGPOO]), überprüft dann automatisch, ob die Bedingung φ im Modell M gilt: $M \models \varphi$.

Beide Methoden sind schon bei Software überschaubarer Komplexität außerordentlich aufwendig und teuer. Trotzdem finden sie Eingang in Teile der Praxis, vor allem in sicherheitskritischen Anwendungen. Aber auch wirtschaftliche Interessen können den Einsatz teurer Methoden rechtfertigen: nachdem es gelang, einen 500 Millionen US-Dollar teuren Fehler in der Gleitpunktarithmetik des Pentium III-Prozessors im Nachhinein mittels der gerade neu entwickelten Methode des Model Checking zu "finden", hat diese Methode eine stürmische Entwicklung erfahren.

Hier wie auch bei den Deduktionsmethoden zielt die aktuelle Forschung darauf ab, den Aufwand zu reduzieren und die Methoden für größere Probleme praktikabel zu machen.

4. Logik für Systeme

Die Grundlage von Software ist Logik – aber welche? Bereits in der Pionierphase der Computerentwicklung fand die Aussagenlogik Verwendung, nämlich bei den so genannten *logischen Schaltungen.* Es geht um die Wahrheitswerte von Aussagen, die mittels logischer Verknüpfungen wie *und, oder, wenn ... dann, nicht* u.s.w. zusammengesetzt sind. Die in der Mathematik üblicherweise verwendete Prädikatenlogik erlaubt es, Aussagen über Gegenstände eines gegebenen Universums zu machen, über deren Beziehungen zueinander und über den auf ihnen definierten Funktionen. Dabei sind auch Quantifizierungen wie *für alle ...* und *es gibt ...* möglich.

Für Aussagen über dynamische Softwaresyteme eignen sich besonders *modale* Logiken, in denen Aussagen über Aussagen wie *φ ist notwendig wahr* und *φ ist möglicherweise wahr* gemacht werden können. Neben Programmlogiken, in denen Aussagen wie *φ ist nach jeder/ irgendeiner Ausführung des Programms wahr* haben insbesondere temporale Logiken große Beachtung gefunden. Hier erscheinen die Modalitäten als *φ ist immer wahr* **(G** *φ)* bzw. *φ ist irgendwann einmal wahr* **(F** *φ*). Die heute eingesetzten Model-Checking-Methoden basieren entweder auf der linearen temporalen Logik LTL oder auf der *Computation Tree Logic* CTL, in der noch über die möglichen Abläufe eines Systems quantifiziert werden kann: *φ gilt bei allen möglichen Systemdurchläufen* **(A** *φ*) bzw. *φ gilt bei mindestens einem möglichen Systemdurchlauf* **(E** *φ).*

Hier sind zur Illustration einige typische Aussagen über Systeme, formuliert in CTL (s. [HROO]):

E F (started ∧ ¬ ready)
ein Zustand kann eintreten, in dem started gilt, aber nicht ready;

A G (requested ⇒ **A F** acknowledged)
eine Anforderung wird in jedem Fall irgendwann bestätigt;

A G (**A F** enabled) (ein gegebener Prozess) ist immer mal wieder dienstbereit;

A F (**A G** deadlock) (...) gerät unvermeidlich in eine Dauerblockade;

A G (**E F** restart) (...) kann jederzeit einen Wiederanlaufpunkt erreichen.

5. Kommunikation

Mittels Model Checking können zustandsbasierte Systeme überprüft werden, die zu jedem Zeitpunkt in einem Zustand sind oder sich im Stadium des Übergangs in einen anderen Zustand befinden. Diese Modellvorstellung trägt weit, hat aber ihre Grenzen bei datenintensiven verteilten Systemen, da die Anzahl der Zustände in's Astronomische wächst: man bekommt es mit dem *state explosion problem* zu tun. Bei verteilten Systemen kommt hinzu, daß die sequentielle Sicht als Menge von Gesamtzuständen mit Übergängen auch deswegen nicht adäquat ist, weil solche fiktiven Gesamtzustände sich der Beobachtbarkeit entziehen. Natürlicher und vorteilhafter erscheint es, das System als Menge sequentieller Komponenten aufzufassen, zwischen denen *Kommunikation* stattfindet. Das heißt, daß nicht davon ausgegangen wird, daß alle Komponenten sich bei jedem Schritt synchronisieren, sondern daß Synchronisation nur noch zur nötigen Abstimmung zwischen den beteiligten Komponenten stattfindet.

Die Überprüfung von verteilten Systemen nach dieser Modellvorstellung ist Gegenstand aktueller wissenschaftlicher Arbeiten des Autors (siehe z.B. [EC00]). Grundlage ist ein auf [LRT92] zurückgehender Ansatz zur Erweiterung der Temporalen Logik: jede sequentielle Komponente bekommt ihre eigene lokale Logik, die um Aussagen über Kommunikation mit anderen sequentiellen Komponenten erweitert wird.

Dies kann auf verschiedene Weisen geschehen. Untersucht werden z.Z. zwei Logiken, D_0 und D_1. Beide gehen von der separaten lokalen Verwendung von CTL (bzw. LTL) aus. Sie unterscheiden sich jedoch in der Art der Beschreibung von Kommunikation: D_0 lässt nur ganz einfache Aussagen über die Synchronisation elementarer Aktionen zu, während in D_1 beliebige Aussagen über andere Objekte in deren Logik möglich sind. In [EC00] wird gezeigt, dass sich D_1 in D_0 vollständig und korrekt übersetzen lässt. Daraus folgt, dass D_1 nicht expressiver ist als D_0.

Mit dieser Übersetzung gelingt es in einigen Fällen, globale Prüfbedingungen in lokal zu überprüfende Einzelbedingungen einerseits und Kommunikationsanforderungen andererseits aufzuteilen. Diese können insgesamt mit erheblich geringerem Aufwand überprüft werden. Die Idee bedarf der weiteren Untersuchung und Erprobung.

Literatur

[CGP00] E.M. Clarke, 0. Grumberg and D.A. Peled. Model Checking. MIT Press, Cambridge (Mass.), 2000.

[EC00] H.-D. Ehlich and F. Caleiro. Specifying Communication in Distributed Information Systems. Acta Informatica Vol 36 Fasc. 8, pages 591-616, 2000.

[HR00] M.R.A. Huth and M.D. Ryan. Logic in Computer Science, modelling and reasoning about systems. Cambridge University Press, Cambridge 2000.

[LRT92] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal logics for communicating sequential agents: I. International Journal of Foundations of Computer Science, 3:117-159, 1992.

Prof. Dr. H.-D. Ehrich
Gaußstraße 12
D-38092 Braunschweig