Programmierung ist ein Spiel

Programming is a Game

Roderick Bloem, Krishnendu Chatterjee

Roderick Bloem ist Professor für Informatik am Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie der Technischen Universität Graz und Sprecher des Nationalen Forschungsnetzwerks für Rigorous Systems Engineering. Er promovierte 2001 an der University of Colorado in Boulder. Seine Forschungsschwerpunkte sind sichere und fehlerfreie Systeme.

Roderick Bloem is a professor of computer science at the Institute for Applied Information Processing and Communications, Graz University of Technology, and is the speaker of the FWF National Research Network on Rigorous Systems Engineering. He received his PhD from the University of Colorado at Boulder in 2001. His research interests are in secure and correct systems.

Computer sind überall: Computer kennen wir meist als Laptops und PCs, doch auf jeden klassischen Computer kommen bereits 50 andere als Bestandteil von Artikeln des täglichen Gebrauchs, von Bankomatkarten bis hin zu Flugzeugen. Computer haben unser Leben zwar einfacher gemacht, doch sind wir stark davon abhängig, dass sie auch fehlerfrei funktionieren. Man denke nur an die Bremse im Auto oder an die kritische Infrastruktur eines Landes: 2003 legte ein Computerfehler die Stromversorgung für 55 Millionen Amerikaner und Amerikanerinnen lahm.

Gleichzeitig tauchen immer mehr Computer in Netzwerken auf. Wenn die Programmierung eines einzigen Computers bereits schwierig ist, um wie viel schwieriger ist die Programmierung eines gesamten Netzwerks. Das menschliche Gehirn ist nun einmal nicht dafür geschaffen, sich über Maschinen Gedanken zu machen, die sequenziell und völlig starr reagieren. Dazu eine Anekdote über ein US-amerikanisches Gesetz, das die Fahrsicherheit von Zügen bei einer Eisenbahnkreuzung gewährleisten soll: „Wenn sich zwei Züge einem Kreuzungspunkt nähern, sollen beide anhalten und so lange nicht weiterfahren, bis der andere die Kreuzung verlassen hat“. Das Gesetz ist eindeutig Unsinn, doch stört dies nicht weiter, solange Menschen daran beteiligt sind – sie werden schon einen gemeinsamen Nenner finden. Diese Flexibilität fehlt jedoch bei Rechnern völlig. Im **Nationalen Forschungsnetzwerk Rigorous Systems Engineering (RiSE)** beschäftigen wir uns mit Methoden, die es auf lange Sicht ermöglichen, fehlerfrei zu programmieren. Im RiSE suchen neun Vollzeitwissenschafter und -wissenschafterinnen und mehr als 20 Nachwuchswissenschafter und -wissenschafterinnen nach Wegen, die Qualität von Softwareprogrammen zu verbessern. In diesem Artikel werden wir uns mit

Computers are most visible as laptops and PCs, but for each “classical” computer, there are 50 computers embedded in everyday items from ATM cards to airplanes. Computers have made are our lives much easier, but we also depend greatly on their functioning correctly. Computer programs that are responsible for the brakes in a car or the critical infrastructure of a country must be programmed correctly. This is not always the case: for instance, a computer error contributed to the 2003 power blackout in the North America, which affected 55 million people.

At the same time, computers increasingly appear in networks. Where programming one computer is hard, programming a network of computers is extremely hard. The human mind is simply not equipped to reason about machines that act concurrently without any flexibility. An apocryphal story tells of a US state law to control the safe behaviors of trains at an intersection: “When two trains approach an intersection, both shall come to a full stop, and neither shall move until the other is gone.” The law is clearly nonsensical, but that is not a problem when humans are involved – they will work something out. Computers, however, lack that flexibility and when confronted with the computer equivalent of such a rule, will grind to an interminable halt.
In the **NFN’s Rigorous Systems Engineering (RiSE)**, we study methods to systematically design error-free programs. In RiSE, nine primary investigators and over 20 junior researchers study ways to improve the quality of software. In this article, we will look at the theory of correct programming and describe one approach, synthesis, which builds on game theory.

Abb. 1: Die Überprüfung der Fehlerfreiheit der Software kann mittels der Spieltheorie erfolgen.

Fig. 1: Checking correctness of software can be done using game theory.

der Theorie des richtigen Programmierens beschäftigen und einen spieltheoretischen Ansatz beschreiben, die Synthese.

Programmieren heißt, ein Spiel korrekt aufzulösen

Grundsätzlich geht es beim Programmieren darum, Spiele zu *lösen:* eine Spielanleitung zu schreiben, sodass man immer gewinnt. Manche Spiele sind leicht zu lösen. Die meisten Leute spielen perfekt „Tic Tac Toe“ („Drei gewinnt“) – sie verlieren nie, egal, gegen welchen Gegner. Auch das Spiel „Vier gewinnt“ wurde bereits gelöst. Es gibt eine Taktik, mit welcher die Person, die beginnt, immer gewinnt. (Und demgemäß keine, mit welcher der zweite Spieler immer gewinnt.) Spiele wie Schach sind jedoch außer Reichweite. Zwar sind Schachprogramme mittlerweile extrem ausgereift und können fast jeden schlagen, Gewinngarantie gibt es jedoch keine, und genau darum geht es uns hier.

Der Zusammenhang zwischen dem Lösen von Spielen und dem Schreiben von Programmen ist eng. Während der Ausführung erhält das Programm Inputs, z. B. von einer Benutzerin, die Tasten drückt. Es produziert Outputs: Pixel auf einem Schirm, Geräusche oder Netzwerktraffic. Entscheidend ist dabei, dass das Programm immer fehlerfrei abläuft und nie ein falsches Ergebnis generiert. Daher sehen wir Umwelt und Programm als zwei gegnerische Spieler: Das Programm versucht, alles richtig zu machen, während die Um-

Programming is solving a game

At the foundations, programming is like *solving* a game: writing a recipe that tells you how to play a game so that you will never lose. Some games can be solved easily. Most people are perfect players at tic-tac-toe – they will never lose, no matter how good the opponent. The game “connect-four” has also been solved. There is a recipe for the beginning player to always win. (Consequently, there is no recipe that allows the second player to always win.) Games like chess, however, are beyond reach. Computer programs have become extremely strong at chess, and can beat almost any player, but there is no guarantee that they win, and that is what we are after.

The connection between solving games and writing computer programs is close. During its execution, a computer program receives inputs from the environment, for instance from a user who presses buttons. It produces outputs: pixels on a screen, sounds, or network traffic. It is crucial that the program is correct under any circumstance and never produces a wrong output. Thus, we consider the environment and the program as opposing players: the program tries to do everything right, whereas the environment tries to provide inputs that are impossible to react to. Typically, inputs and outputs alternate like moves in chess. The rules of the game are given by a specification that states which outputs are allowed when and the system wins if it fulfills the specifications.

Krishnendu Chatterjee ist Assistent Professor am Institute of Science and Technology (IST) Austria. Er promovierte 2008 an der University of California in Berkeley. 2008 erhielt er auch den Ackerman Award für die weltweit beste Dissertation in Computerlogik. Sein Forschungsschwerpunkt sind die theoretischen Grundlagen der formellen Verifikation und Spieltheorie.

Krishnendu Chatterjee is an assistant professor at the Institute of Science and Technology (IST) Austria. He received his PhD from the University of California, Berkeley in 2008. He is the receiver of the 2008 Ackerman Award for the best thesis worldwide in computer science logic. His main research interest is in the theoretical foundations of formal verification and game theory.

Abb. 2: Das Team von RiSE (v.l.n.r.) Thomas A. Henzinger, Christoph Kirsch, Ulrich Schmid, Helmut Veith, Armin Biere, Roderick Bloem, Uwe Egly, Laura Kovács, Krishnendu Chatterjee.

Fig. 2: The team of RiSE (from left to right: Thomas A. Henzinger, Christoph Kirsch, Ulrich Schmid, Helmut Veith, Armin Biere, Roderick Bloem, Uwe Egly, Laura Kovács, Krishnendu Chatterjee.

welt versucht, Inputs zu liefern, auf die es nicht reagieren kann. Typischerweise wechseln sich Inputs und Outputs ab, wie Spielzüge im Schach. Die Spielregeln definieren eine Vorgabe darüber, welche Outputs wann erlaubt sind, und das System gewinnt, wenn es alle Vorgaben erfüllt.
Nehmen wir zum Beispiel eine Druckersteuerung. Die Vorgabe könnte lauten, dass

1. Druckaufträge eine Sekunde in Anspruch nehmen dürfen,
2. jeder Auftrag in zwei Sekunden abzuarbeiten ist und
3. keine zwei Druckaufträge gleichzeitig abgearbeitet werden dürfen (um Kollisionen zu vermeiden).

Wenn das System drei Nutzerinnen hat, kann diese Vorgabe unmöglich erfüllt werden – wenn jede Nutzerin zur gleichen Zeit einen Druckauftrag startet, *muss* das System eine der drei Anforderungen verletzen. Ändern wir Vorgabe zwei jedoch dahingehend, dass die Aufträge in bis zu drei Sekunden abzuwickeln sind, so kann die Vorgabe erfüllt werden. Die Druckersteuerung stellt sicher, dass alle Regeln eingehalten werden, unabhängig davon, wie sich die Umwelt verhält.

Fehlerfreiheit gewährleisten

Es gibt mehrere Möglichkeiten, um zu gewährleisten, dass Softwareprogramme fehlerfrei ablaufen. Der übliche Weg ist, sie zu testen. Beim **Testen** probieren wir verschiedene Inputs aus und überprüfen, ob auch die Outputs stimmen. Aus der Analogie mit dem Spiel wird jedoch deutlich, dass diese Vorgangsweise nicht ideal ist. Die Tatsache, dass Sie 1, 2 oder eine Million Spiele verlieren, bedeutet nicht, dass Ihr Gegner unfehlbar ist und gegen jedermann gewinnen würde.
Eine systematischere Möglichkeit, fehlerfreies Funktionieren zu gewährleisten, ist die **Verifikation.** Dabei wird ein Beweis dafür aufgebaut, dass ein bestimmtes Programm fehlerfrei abläuft,

An example would be an arbiter for a printer. The specification may say that

1. Print jobs take one second,
2. Each job must be handled within two seconds, and
3. Two print jobs may not be served at the same time (lest garbage ensues).

If there are three users of the system, this specification cannot be fulfilled by any system – if each user issues a print job at the same time, the system *must* violate one of the three requirements. If we change requirement 2 to state that jobs can take up to three seconds to be completed, the specification can be satisfied: there is an arbiter that fulfills all the rules regardless of the behavior of the environment.

Ensuring correctness

There are multiple ways to make sure that software works correctly. The usual way is **testing**. When testing a program, we try many different inputs and check that the outputs are correct. It is clear from the game analogy that this approach is not ideal. The fact that you lose 1, or 2, or a million games does not mean that your opponent is infallible and would win against anyone. The main reason to perform tests is that a first indication of a program's correctness can be obtained very quickly.
Verification is a systematic way to ensure correctness. We construct a proof that a given program is correct regardless of the inputs. This approach has great appeal because we can gain complete confidence in the program. It is like making sure that a player is perfect. The drawback is also clear: it is not easy to see that a strategy is perfect, and constructing a proof of correctness may be hard. Still, the community has made tremendous progress in this field over the last 2 decades. So-called model checkers can automatically construct proofs of correctness for programs

unabhängig von den Inputs. Wir können damit große Sicherheit in der Richtigkeit des Programms erreichen, so, als würden wir uns versichern, dass ein Spieler oder eine Spielerin perfekt ist. Der Nachteil ist, dass es nicht leicht ist zu beweisen, dass eine Strategie bzw. ein Programm perfekt ist. Die Softwarecommunity hat im Laufe der letzten 2 Jahrzehnte aber beträchtliche Erfolge erzielt. Sogenannte Model Checker sind heute in der Lage, automatisch den Nachweis der Fehlerfreiheit für so kleine, aber wichtige Programme wie Gerätetreiber zu liefern.

Ein Nachteil der Verifikation besteht darin, dass Benützer/Benützerinnen sowohl ein fehlerfreies Programm als auch eine perfekte Vorgabe erstellen müssen. Idealerweise sollte es genügen, eine gute Vorgabe zu schreiben, woraus ein fehlerfreies Programm automatisch erstellt wird. Dieser Ansatz, **Synthese** genannt, ist die Königsklasse in Sachen Fehlerfreiheit. Doch nun zurück zur Analogie mit dem Spiel: In der Synthese wird versucht, automatisch einen perfekten Spieler zu konstruieren, der jeden Gegner bezwingt. Genau darum geht es in der Spieltheorie.

Bis vor fünf oder zehn Jahren galt die Synthese als von rein theoretischem Interesse; man war skeptisch, Real-Life-Systeme aufgrund von Vorgaben automatisch generieren zu können. Die letzten fünf Jahre haben uns diesem Ziel jedoch erheblich näher gebracht. Innovative Konzepte der Spieltheorie ermöglichen es uns, die Fähigkeiten von Synthesetools auf kleine, aber realistische industrielle Systeme auszudehnen. Das ist zweifellos ein großer Erfolg, der umso spannender ist, als er viele neue und unerwartete theoretische Fragen aufwirft – über die richtige Sprache der Vorgabe, das Erfordernis der Robustheit usw. Diese Fragen an der Schnittstelle zwischen Theorie und Praxis sind Kernthemen des RiSE-Netzwerks und, wie wir meinen, ein wichtiger Meilenstein auf dem Weg zur Programmierung fehlerfreier Systeme.

like device drivers (which are small but may create large problems).

Another drawback of verification is that the user must construct both a working program and a perfect specification. Ideally, it should suffice to write a good specification; a correct program should then be constructed automatically. This approach, called **synthesis**, is the major league of correctness. Returning to the game analogy, synthesis attempts to construct a perfect player automatically, a player that wins against any opponent. This problem is the topic of game theory, a discipline that originates from economics. Up to 5 or 10 years ago, synthesis was considered to be only of theoretical interest; it was considered completely unrealistic to construct real-life systems automatically from specifications. Over the last five years, however, we have made tremendous progress towards this goal. Using novel game-theoretic concepts, we have been able to extend the capability of synthesis tools to small, but realistic industrial systems. This is a great success, and it is even more exciting as it raises many new and unexpected theoretical questions concerning the right specification languages, the need for robustness, etc. These new questions on the intersection between theory and practice will be one of the topics of the RiSE network, and we expect them to be an important cornerstone in future approaches to programming correct systems.

RiSE

Rigorous Systems Engineering ist ein vom FWF finanziertes nationales Forschungsnetzwerk. Das Projekt bündelt die Kompetenzen von Weltklasseforschern und -forscherinnen für Verifikation in Österreich: Armin Biere (JKU Linz), Roderick Bloem (TU Graz, Sprecher), Krishnendu Chatterjee (IST Austria), Uwe Egly (TU Wien), Tom Henzinger (IST Austria), Christoph Kirsch (PLU Salzburg), Laura Kovács (TU Wien), Ulrich Schmid (TU Wien) und Helmut Veith (TU Wien).

RiSE

Rigorous Systems Engineering is an FWF-funded National Research Network. The project bundles the strengths of world-class researchers in verification in Austria: Armin Biere (JKU Linz), Roderick Bloem (Graz University of Technology, speaker), Krishnendu Chatterjee (IST Austria), Uwe Egly (TU Wien), Tom Henzinger (IST Austria), Christoph Kirsch (PLU Salzburg), Laura Kovács (TU Wien), Ulrich Schmid (TU Wien), and Helmut Veith (TU Wien).