

Formale Verifikationsmethoden für die Entwicklung von High-Integrity-Software für Medizinische Geräte

Jay Abraham
The MathWorks

Paul Jones
FDA / CDRH

Raoul Jetley
FDA / CDRH

Der Umfang und die Komplexität der in eingebetteten medizinischen Systemen eingesetzten Software nimmt stetig zu. Konventionelle Verifikations- und Testmethoden sind für diese Software keine optimale Lösung mehr. Dieser Artikel beschreibt den Einsatz zuverlässiger Verifikationsverfahren für die Entwicklung von High-Integrity-Software für medizinische Geräte. Er konzentriert sich dabei vor allem auf die Anwendung formaler Methoden basierend auf *Abstrakten Interpretationsverfahren*, mit deren Hilfe die Abwesenheit einer definierten Menge von Laufzeitfehlern mathematisch bewiesen werden kann. Im Anschluss daran vergleicht er diese Verifikationslösung mit anderen Analyse- und Testverfahren wie Codereviews, statischen Analysen und dynamischen Tests.

Einführung

Die in medizinischen Geräten enthaltene Embedded Software wird zunehmend anspruchsvoller und komplexer. Moderne Schrittmacher etwa können bis zu 80.000 Zeilen Programcode enthalten, Infusionspumpen sogar über 170.000 Zeilen. Gleichzeitig müssen diese Geräte mit höchsten Anforderungen an Sicherheit und Zuverlässigkeit funktionieren [1].

Rückblickend betrachtet bestanden die für die Verifikation von medizintechnischer Software vorhandenen Möglichkeiten bislang in Codereviews, statischen Analysen und dynamischen Tests. Codereviews stehen und fallen mit der Fachkompetenz des Prüfers und können für große Code-Basen ineffizient sein. Konventionelle statische Analysemethoden fußen hauptsächlich auf einem Mustersuchverfahren, mit dem nach unsicheren Codemustern gesucht wird, können aber nicht die Abwesenheit von Laufzeitfehlern beweisen. Und schließlich ist es angesichts der wachsenden Komplexität der Gerätesoftware praktisch unmöglich, diese unter allen denkbaren Arten von Betriebsbedingungen zu testen.

Medizintechnische Gerätesoftware

Medizinische Geräte werden heute für ein sehr breites Spektrum an Diagnose- und Behandlungszwecken unterschiedlichster Komplexität eingesetzt, das von digitalen Thermometern, Insulinpumpen, Schrittmachern und Herzmonitoren bis hin zu Anästhesiegeräten, großen Ultraschall-Systemen, Geräten für chemische Analysen und Protonenstrahl-

Therapiesystemen reicht. Gemeinsam mit dem wissenschaftlichen und technischen Fortschritt entwickeln sich auch die Fähigkeiten und die Komplexität der jeweils nächsten Gerätegeneration weiter. Alle Geräte dienen dabei dem gemeinsamen Zweck, die medizinische Versorgung direkt oder auch indirekt zu verbessern.

Software wird heute mehr und mehr zu einem allgegenwärtigen Bestandteil medizintechnischer Geräte und erhöht deren Komplexität [5]. Verschiedene weltweit agierende Aufsichts- und Zulassungsbehörden haben daher explizit Vorschriften aufgenommen, die sich direkt an das Thema der Gerätesoftware wendet [2]. Neueste Studien weisen nach, dass die Zahl der Fälle zunimmt, in denen medizinische Geräte durch Software-Programmierfehler versagen. Eine von Bliznakov et al. durchgeführte Studie stellt beispielsweise fest, dass 11,3 % der zwischen 1999 und 2005 von der Food and Drug Administration (FDA) veranlassten Rückrufe auf solche Programmierfehler zurückzuführen waren [6].

Die FDA hat ihre aufsichtstechnische Einbindung in die Prüfung der Entwicklung von Software für medizintechnische Geräte um die Mitte der 1980er Jahre intensiviert, als Programmierfehler in einem für die Strahlentherapie eingesetzten Gerät hauptsächlich für eine Reihe tödlicher Überdosierungen an Patienten waren [2]. Die FDA hat [seitdem] verschiedene Dokumente mit Richtlinien herausgegeben und eine Reihe von Standards anerkannt, die "gute" Verfahren zur

Softwareentwicklung definieren. Erst vor kurzem hat die FDA in ihren Office of Science and Engineering Laboratories (OSEL) des Center for Devices and Radiological Health (CDRH) ein Softwarelabor ins Leben gerufen, das Programmierfehler in Geräten identifizieren soll, die aufgrund eines Rückrufs untersucht werden [1]. Das Softwarelabor überprüft dabei unter anderem den Quellcode mit dem Ziel, die Kernursache eines Geräteversagens zu verstehen und zu identifizieren.

Für die Entwicklung von Embedded Software mit hohen Integritätsanforderungen kann ein Seitenblick auf die in anderen Branchen genutzten Software-Verifikationsprozesse aufschlussreich sein. So werden beispielsweise an die Verifikation von Software in der Luft- und Raumfahrt oder auch der Automobilindustrie strikte Anforderungen gestellt [3]. Diese Industriezweige haben verschiedene Analyse- und Verifikationstools in ihre Entwicklungsprozesse integriert, die die Softwarequalität verbessern. Die US-Luftaufsichtsbehörde FAA (Federal Aviation Administration) fordert als Teil des Zulassungsverfahrens von Flugzeugen als flugtauglich die Zertifizierung von Software nach dem DO-178B-Standard (Software Considerations in Airborne Systems and Equipment Certification). Dieser Standard definiert verschiedene, nach der Schwere ihrer möglichen Folgen in Stufen eingeteilte Kategorien von Softwarefehlern. Je schwerer die Folge eines Versagens, umso rigider sind die Anforderungen an den Entwicklungsprozess, die Verifikation und die Ausfallsicherheit des betreffenden Systems. Die Automobilindustrie richtet sich nach Softwarestandards und orientiert ihre Entwicklung häufig an dem internationalen IEC-61508-Standard (der die "Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme" definiert).

Versagen eingebetteter Geräte

Um in der Vergangenheit aufgetretene Fehler in Zukunft zu vermeiden, lohnt sich eine Untersuchung bekannter Ausfälle und Fehler von eingebetteten Geräten. Dazu seien hier drei der Beispiele herausgegriffen, die von Ganssle in [11] sehr anschaulich vorgetragen wurden.

1. Therac 25 – Ein strahlentherapeutisches Gerät, das Patienten einer Strahlenüberdosis

aussetzte. Das Echtzeitbetriebssystem (RTOS) für das Gerät unterstützte kein Message-Passing-Schema für Threads. Man verwendete darum stattdessen globale Variablen. Weil diese globalen Variablen ungenügend geschützt waren, wurden beim Betrieb des Geräts fehlerhafte Daten verwendet. Ein zweites Problem stellte ein Überlauf in einem 8-Bit-Integerzähler dar. Durch diese Software-Programmierfehler erhielten Patienten Überdosierungen von bis zu dreißigfachen der verschriebenen Dosis.

2. Ariane 5 – Der Erstflug dieser Rakete endete mit der Zerstörung der Trägerrakete und dem Verlust der Nutzlast. Ursache dafür war ein Überlauf in einem Paar redundanter Trägheits-Navigationssysteme, die zur Lage- und Positionsbestimmung der Rakete dienen. Der Überlauf wurde durch die Umwandlung eines 64-Bit-Fließkommawertes in eine 16-Bit-Ganzzahl verursacht. Das Vorhandensein eines redundanten Systems konnte den Fehler nicht beheben, weil das Verhalten beider Systeme exakt identisch war.
3. Space Shuttle-Simulator – Während einer Abbruchübung stürzten sämtliche vier Hauptcomputer des Shuttles ab. Eine Untersuchung des Programmcodes ermittelte ein Problem in der Software der Treibstoffaufbereitung. Nach Durchführung des ersten von mehreren Treibstoff-Ablassvorgängen wurden hier Zähler nicht korrekt reinitialisiert. Der Programmcode sprang zu zufälligen Speicherabschnitten und löste dadurch den Absturz sämtlicher Rechner aus.

Weitere spezifische Beispiele für Probleme mit medizintechnischen Geräten finden sich in der vom CDRH unterhaltenen Manufacturer and User Facility Device Experience Database (MAUDE) [21], die anomales Verhalten und unerwünschte Ereignisse katalogisiert und eine sehr einfache Schlagwortsuche unterstützt. Eine Datenbanksuche nach "Firmware" gibt beispielsweise eine Reihe medizintechnischer Geräte aus, die von der FDA aufgrund gemeldeter unerwünschter Ereignisse weiter verfolgt werden.

Ursachen für Geräteversagen

Codedefekte, durch die Laufzeitfehler entstehen, sind die Hauptursache für die oben beschriebenen Versagensfälle eingebetteter

Systeme. Laufzeitfehler sind eine spezifische Klasse von Softwarefehlern, die als latente Fehler bezeichnet werden. Diese Fehler können im Code vorhanden sein, lassen sich im System aber nur feststellen, wenn sehr spezifische Tests unter ganz besonderen Bedingungen durchgeführt werden. Oberflächlich kann der Programmcode den Anschein erwecken, als funktioniere er völlig normal, es können aber trotzdem unerwartete Fehlfunktionen mit zuweilen fatalen Konsequenzen auftreten. Einige Ursachen für Laufzeitfehler sind unten angeführt (die Liste ist nicht vollständig):

1. Nicht initialisierte Daten – Wenn Variablen nicht initialisiert wurden, können sie auf einen undefinierten Wert gesetzt sein. Bei der Verwendung dieser Variablen kann sich der Code manchmal wie erwartet verhalten, unter bestimmten Umständen werden die Ergebnisse aber unvorhersagbar.
2. Array-Zugriffe außerhalb der definierten Grenzen – Solche Zugriffe finden statt, wenn Daten aus Adressen gelesen oder in sie geschrieben werden, die außerhalb des zugewiesenen Speicherbereichs liegen. Dieser Fehlertyp kann Inkonsistenzen im Code verursachen und während der Ausführung unvorhersagbare Ergebnisse erzeugen.
3. Dereferenzierte Nullzeiger – Ein dereferenzierter Nullzeiger tritt auf, wenn versucht wird, mit einem NULL-Zeiger auf Speicher zu verweisen. Da dies für den Speicher ein "Nicht-Wert" ist, führt jede Dereferenzierung dieses Zeigers zu einem sofortigen Systemabsturz.
4. Berechnungsfehler – Dieser Fehlertyp wird durch einen arithmetischen Fehler verursacht, der auf einen Überlauf, Unterlauf, eine Division durch Null oder das Ziehen der Wurzel einer negativen Zahl zurückgeht. Berechnungsfehler können Programmabstürze oder falsche Ergebnisse erzeugen.
5. Gleichzeitiger Zugriff auf gemeinsam genutzte Daten – Dieser Fehler entsteht, wenn zwei oder mehr Variablen aus verschiedenen Threads auf die gleiche Speicheradresse zuzugreifen versuchen. Dies kann zu einer Race Condition führen und Datenkorruptionen erzeugen, weil mehrere

Threads ungeschützt auf gemeinsam genutzte Daten zugreifen.

6. Illegale Datentypkonvertierungen – Illegale Datentypkonvertierungen können Daten korrumpieren und unbeabsichtigte Folgen haben.
7. Toter Code – Obwohl toter Code (also Code, der niemals ausgeführt wird) nicht unbedingt an sich Laufzeitfehler verursacht, kann es wichtig sein, zu verstehen, warum der Programmierer ihn geschrieben hat. Toter Code kann außerdem ein Hinweis auf schlechte Programmierpraktiken oder verloren gegangene Design-Spezifikationen sein.
8. Endlosschleifen – Diese Fehler werden durch falsche „guard conditions“ für Programmschleifen-Operationen (For, While, usw.) ausgelöst und können zu einem Hängen oder Anhalten des Systems führen.

Verifikation und Testen von Software

Konventionelle Verifikations- und Testverfahren bestehen aus Codereviews und dynamischen Tests. In [15] erörtert Fagan, wie Codeinspektionen und -reviews die Zahl der Programmierfehler senken können. Die Erfahrung hat gezeigt, dass dieser Prozess, obwohl er eine relativ effektive Möglichkeit zur Verifikation von Programmcode darstellt, der Ergänzung um zusätzliche Methoden bedarf. Eine solche Methode ist die statische Analyse. Sie stellt ein noch relativ neues Verfahren dar, das die Software-Verifikation weitgehend automatisiert [16]. Die statische Analyse zielt darauf ab, Fehler im Programmcode zu identifizieren, beweist aber nicht notwendigerweise deren Abwesenheit. Der folgende Abschnitt beschreibt diese Methoden und stellt die Anwendung formaler, auf Abstrakter Interpretation basierenden, Code-Verifikationsmethoden vor.

Code-Reviews

Fagan diskutiert den Prozess von Codereviews und -inspektionen sehr ausführlich in [15]. Die geschilderte Verfahrensweise enthält eine Beschreibung des Teams, welches das Review durchführt (Moderator, Designer, Programmierer und Tester), der Vorbereitung des Review (bei

der beispielsweise eine Checkliste erstellt wird) sowie der Inspektion selbst. Das erklärte Ziel besteht darin, Fehler im Code zu finden. Fagan erwähnt außerdem die am Ende des Review stattfindenden Prozesse, darunter auch die Überarbeitung, bei der die gefundenen Fehler behoben werden, sowie ein Follow-Up, das sicherstellt, dass während der Inspektion gefundene Fehler und Probleme tatsächlich beseitigt wurden. Ein weiterer Aspekt von Codereviews können Prüfungen der Konformität mit bestimmten Programmierstandards sein wie etwa MISRA-C oder JSF++ (für C und C++).

Das Aufspüren subtiler Laufzeitfehler kann sich schwierig gestalten. So können beispielsweise Überläufe oder Unterläufe leicht übersehen werden, die durch komplexe mathematische Berechnungen erzeugt werden, bei denen eine programmatische Steuerung stattfindet.

Dynamische Tests

Dynamische Tests verifizieren den Ablauf der Ausführung von Software, also beispielsweise Entscheidungspfade, Eingaben und Ausgaben. Wagner beschreibt die Methodologie und erklärt die Anwendung dieser Testphilosophie auf der Grundlage der Dimensionen des Typs (funktional und strukturell) sowie der Granularität (Einheit, Integration und System) [16].

Bei dynamischen Tests werden zunächst Testfälle und Testvektoren erzeugt und dann die Software gegen diese Tests ausgeführt. Danach werden die Testergebnisse mit dem erwarteten oder bekannten korrekten Verhalten verglichen. Wagners Artikel enthält außerdem eine Zusammenfassung verschiedener Statistiken zur Effektivität dynamischer Tests. Seine Analyse zeigt, dass die durchschnittliche Effektivität dynamischer Tests bei nur etwa 47% liegt. Mit anderen Worten bleiben bei dynamischen Tests mehr als die Hälfte aller Fehler unentdeckt.

Hailpern [9] und Dijkstra [10] fassen die Kernprobleme beim Testen von Software folgendermaßen zusammen: *"Da wir nicht wirklich zeigen können, dass keine weiteren Fehler mehr im Programm vorhanden sind, wann hören wir auf zu testen?"* [9] und *"Programmtests können dazu dienen, das Vorhandensein von Fehlern aufzudecken, aber niemals, um deren Abwesenheit zu demonstrieren."* [10] Butler und Finelli erklären

weiterhin, das ein sogenanntes "Life Testing", also das Testen eines Gerätes unter sämtlichen möglichen Bedingungen, in der Praxis nicht durchführbar ist. Um etwa eine Fehlerrate von $10^{-8}/h$ quantifizieren zu können, wären über 10 Millionen Teststunden notwendig [8].

Statische Analyse

Die statische Analyse ist eine Methode zur Identifizierung potenzieller und realer Defekte im Quellcode. Der Prozess einer statischen Analyse basiert auf Heuristiken und Statistiken und es müssen weder Testfälle entwickelt noch Code ausgeführt werden. Die damit gefundenen Fehlertypen kann man sich als starke Compiler-Typisierung (etwa die Überprüfung, ob Variablen stets initialisiert oder verwendet werden) im Rahmen einer ausführlichen Datenfluss-Analyse vorstellen.

Wie in [16] und [17] beschrieben, können diese Tools zweifellos Fehler im Code finden, dies aber mit einer hohen Rate von False Positives. Der Begriff False Positive bezieht sich auf die Identifikation eines Fehlers, der nicht real ist. Die für die Verfolgung von False Positives aufgewendete Zeit und Energie kann für die Software-Ingenieure frustrierend sein [17]. Wagner [16] fasst die zu False Positives vorliegenden Ergebnisse in einer Übersicht zusammen. Die mittlere Zahl von False Positives beträgt demnach bei einigen statischen Analysetools 66%.

Neben den False Positives ist es außerdem wichtig, False Negatives zu verstehen. Man spricht von einem False Negative, wenn ein statisches Analysetool einen Fehler nicht entdeckt [18]. In [17] findet sich eine umfassende Erörterung von False Negatives mit der Feststellung, dass jede Verringerung der Wahrscheinlichkeit von False Negatives gleichzeitig die Wahrscheinlichkeit von False Positives erhöht. Der Einsatz statischer Analysen kann also den Verifikationsprozess zu einem gewissen Grad automatisieren, man muss diesen Vorteil aber sorgfältig gegen die Tendenz dieser Tools zur Erzeugung von False Negatives und False Positives abwägen.

Formale Methoden

Der Begriff *Formale Methoden* wurde typischerweise für die auf Beweise gestützte Verifikation eines Systems gegen seine Spezifikation verwendet. Der gleiche Terminus

kann aber auch einen mathematisch stringenten Ansatz zum Beweis der Korrektheit von Programmcode meinen [13]. Dieser Ansatz kann helfen, die Zahl von False Negatives zu verringern, das heißt, er kann helfen, einen Teil der Unfähigkeit zu beseitigen, eindeutig die Abwesenheit bestimmter Laufzeitfehler festzustellen. Der folgende Abschnitt beschreibt den Einsatz der Abstrakten Interpretation als eine auf Formalen Methoden basierende Verifikationslösung, die auf Softwareprogramme angewandt werden kann.

Abstrakte Interpretation

Die Abstrakte Interpretation lässt sich am besten anhand eines einfachen Beispiels erklären. Angenommen, es sollen drei große Ganzzahlen multipliziert werden:

$$-4586 \times 34985 \times 2389 = ?$$

Die obige Beispielrechnung lässt sich per Hand nur schwer sowohl schnell und als auch exakt lösen. Abstrahiert man aber das Rechenergebnis auf sein Vorzeichen (also entweder positiv oder negativ), dann erkennt man unmittelbar, dass dieses am Ende negativ sein muss. Eine solche Vorzeichenermittlung ist eine praktische Anwendung der Abstrakten Interpretation. Mit dieser Methode kann man einige Eigenschaften des Endergebnisses, wie hier das Vorzeichen, von vornherein wissen, ohne die Ganzzahlen komplett miteinander multiplizieren zu müssen. Ebenso weiß man, dass das Vorzeichen für diese Rechnung niemals positiv werden wird. Die Abstrakte Interpretation beweist also strikt, dass das Vorzeichen der gezeigten Operation immer negativ und niemals positiv sein wird.

Betrachten wir nun eine vereinfachte Anwendung der formalen Mathematik der Abstrakten Interpretation auf Software-Programme. Die Semantik einer Programmiersprache ist in der konkreten Domäne S dargestellt. Es sei nun A die Abstraktion dieser Semantik. Die Abstraktionsfunktion α bildet die konkrete Domäne auf die abstrakte Domäne ab. Die Konkretisierungsfunktion γ bildet von der abstrakten Domäne A auf die konkrete Domäne S ab. α und γ bilden eine Galois-Verbindung und sind monoton [22]. Bestimmte Beweiseigenschaften der Software können in der abstrakten Domäne A durchgeführt werden. Es ist ein einfacheres Problem, den Beweis in der

abstrakten Domäne A durchzuführen als in der konkreten Domäne S .

Das Konzept der logischen Korrektheit (englisch treffender Soundness) ist im Kontext einer Diskussion über die Abstrakte Interpretation von großer Bedeutung. Korrektheit bedeutet, dass Aussagen über eine Eigenschaft nur dann getätigt werden, wenn diese Aussagen als korrekt bewiesen wurden. Die Ergebnisse einer Abstrakten Interpretation gelten als korrekt, weil sich durch strukturelle Induktion mathematisch beweisen lässt, dass die Abstraktion das richtige Ergebnis vorhersagt. Wendet man die Abstrakte Interpretation auf Software-Programme an, dann kann sie dazu eingesetzt werden, bestimmte Eigenschaften der Software zu beweisen, darunter beispielsweise, dass diese Software frei von bestimmten Typen von Laufzeitfehlern ist [20].

Cousot und Cousot [12, 13] beschreiben die Anwendung der Abstrakten Interpretation auf die statische Programmanalyse. Deutsch stellt in [19] die Anwendung dieser Methode auf eine kommerziell erhältliche Lösung vor. In der Praxis wird für die Abstrakte Interpretation mit Hilfe der Abstraktionsfunktion α eine angenäherte Semantik des Software-Codes berechnet, so dass diese in der abstrakten Domäne verifiziert werden kann. Dabei werden Gleichungen oder Bedingungen erzeugt, deren Lösung eine Computerdarstellung der abstrakten Semantik des Programms darstellt.

Werte von Variablen werden mit Hilfe von Graphen (engl. lattices) dargestellt. Für das zuvor beschriebene Vorzeichenbeispiel kann man mit dem in Abb. 1 gezeigten Graphen die Fortpflanzung abstrakter Werte in einem Programm verfolgen (von unten nach oben). Das Erreichen jedes einzelnen Knotens bedeutet den Beweis einer bestimmten Eigenschaft. Erreicht man die Spitze des Graphen, dann bedeutet das, dass eine bestimmte Eigenschaft unbewiesen ist; was darauf deutet, dass diese Eigenschaft unter bestimmten Bedingungen als richtig, unter anderen dagegen als falsch bewiesen wird.

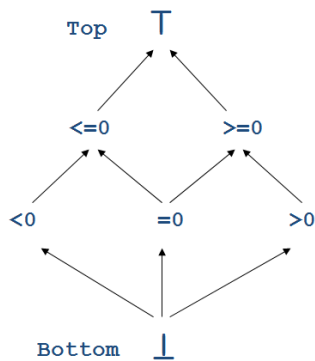


Abb. 1: Graphendarstellung von Variablen

Sämtliche möglichen Ausführungspfade in einem Programm werden einer Über-Approximation unterzogen. Methoden wie die Erweiterung und Verengung [22] sowie die Iteration mit einem Solver werden zur Lösung von Gleichungen und Nebenbedingungen eingesetzt, um so die Abwesenheit von Laufzeitfehlern im Quellcode zu beweisen.

Nutzung der Abstrakten Interpretation für die Code-Verifikation

Die Identifikation sowie der Nachweis der Abwesenheit dynamischer Fehler wie etwa Laufzeitfehler, die während der Ausführung von Programmcode auftreten können, lässt sich durch Definition einer Stärksten Globalen Invariante $SGI(k)$ erzielen. $SGI(k)$ ist hierbei die Menge aller möglichen Zustände, die am Punkt k in einem Programm P erreicht werden können. Ein Laufzeitfehler wird auftritt, wenn $SGI(k)$ eine verbotene Zone schneidet. $SGI(k)$ ist das Ergebnis formaler Beweismethoden und kann als kleinste Fixpunkte eines monotonen Operators am Graphen einer Menge von Zuständen ausgedrückt werden [19].

Um die Anwendung der Abstrakten Interpretation auf die Codeverifikation zu verdeutlichen, sei angenommen, dass der Code die folgende Operation enthält:

$$X = X/(X-Y)$$

Eine ganze Reihe von Problemen kann hier einen potenziellen Laufzeitfehler erzeugen. Im Einzelnen sind dies:

1. Nicht initialisierte Variable.

2. Ein Über- oder Unterlauf bei der Subtraktions-Operation ($X-Y$).
3. Ein Über- oder Unterlauf bei der Divisions-Operation.
4. Eine Division durch Null, falls $X = Y$ wird.
5. Ein Über- oder Unterlauf durch die Zuweisung zu X .

Bedingung Nummer 4 (Division durch Null) soll hier einmal näher betrachtet werden. In der Auftragung von X gegen Y (s. Abbildung 2) erkennt man, dass die 45°-Linie und damit der Fall $X = Y$ einen Laufzeitfehler erzeugt. Im Streuplot sind sämtliche Werte dargestellt, die X und Y annehmen können, wenn das Programm diese Codezeile ausführt (als + gekennzeichnet).

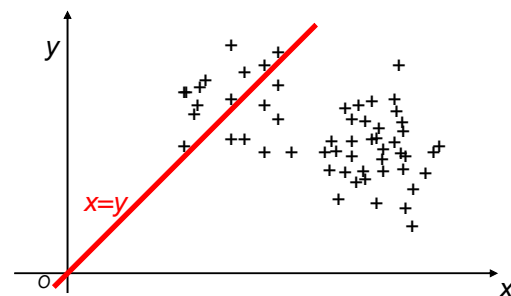


Abb. 2: Auftragung der Datenwerte von X gegen die von Y

Dynamische Tests würden nun X und Y über viele verschiedene Kombinationen ausführen, um zu ermitteln, ob hierbei ein Fehler auftritt. Angesichts der großen Zahl der dafür notwendigen Tests kann diese Art zu testen den Laufzeitfehler "Division durch Null" möglicherweise nicht entdecken noch seine Abwesenheit beweisen.

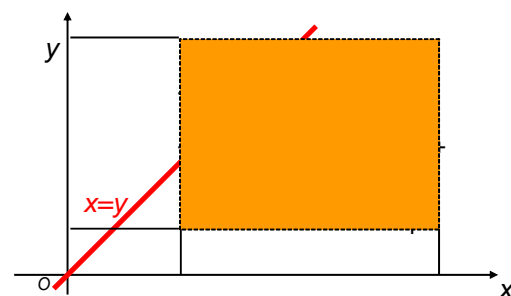


Abb. 3: Typenanalyse

Eine andere mögliche Methode bestünde darin, den Wertebereich von X und Y im Rahmen der Laufzeitfehlerbedingung (also $X = Y$) mit Hilfe einer Typenanalyse zu untersuchen. In

Abbildung 3 ist der durch die Typenanalyse eingegrenzte Wertebereich als Kasten eingezeichnet. Schneidet die Gerade $X = Y$ diesen Kasten, dann ist hier ein potenzieller Fehler vorhanden. Einige statische Analysetools nutzen diese Technik. Die Typenanalyse ist in diesem Fall jedoch zu pessimistisch, weil sie unrealistische Werte für X und Y berücksichtigt.

Die Abstrakte Interpretation erzeugt eine exaktere Darstellung der Wertebereiche für X und Y . Da eine ganze Reihe von Programmier-Konstrukten Einfluss auf die Werte von X und Y nehmen können (etwa Zeiger-Berechnungen, Schleifen, If-Then-Else, Multitasking usw.), definiert man einen abstrakten Graphen [19]. Eine vereinfachte Darstellung dieses Konzepts wäre, sich die Gruppierung von Daten durch Polyederzüge vorzustellen, wie dies in Abbildung 4 gezeigt ist. Weil das Polyeder die Gerade $X = Y$ nicht schneidet, kann eindeutig geschlossen werden, dass keine Division durch Null auftritt.

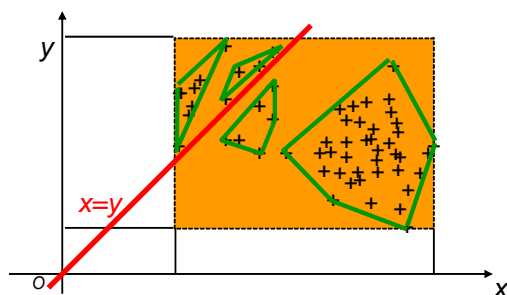


Abb. 4: Abstrakte Interpretation

Wie weiter oben beschrieben, ist die Abstrakte Interpretation eine auf dem Beweis der logischen Korrektheit basierende Verifikationsmethode. Bei der Abstrakten Interpretation werden Aussagen bezüglich spezifizierter Laufzeitaspekte der Software als richtig bewiesen.

Ein auf Abstrakter Interpretation basierendes Code-Verifikationstool

Das Konzept der Abstrakten Interpretation kann als das einer Sammlung von Tools verallgemeinert werden, mit deren Hilfe sich eine breite Palette von Laufzeitfehlern in Software ermitteln lässt. Dieser Artikel nimmt PolySpace® als Beispiel und erklärt daran, wie ein solches Tool helfen kann, Laufzeitfehler wie Überläufe, Divisionen durch Null, Array-

Zugriffe außerhalb gültiger Bereiche usw. aufzuspüren oder deren Abwesenheit zu beweisen.

PolySpace ist ein Code-Verifikationsprodukt von The MathWorks®, das sich der Abstrakten Interpretation bedient. Den Input für PolySpace bildet C-, C++- oder Ada-Quellcode. Der Output besteht aus vierfarbig markiertem Quellcode. PolySpace informiert den Anwender mit Hilfe der in Tabelle 1 aufgeführten und in Abb. 5 gezeigten Farbcodierung über die Qualität des Codes.

Grün	<i>Code ist korrekt und zuverlässig</i>
Rot	<i>fehlerhafter Code, der Laufzeitfehler auslöst</i>
Grau	<i>Toter oder unerreichbarer Code</i>
Orange	<i>Unbewiesener, potentiell unsicherer Codeabschnitt</i>

Abb. 1: Erläuterung des in PolySpace verwendeten Farbschemas

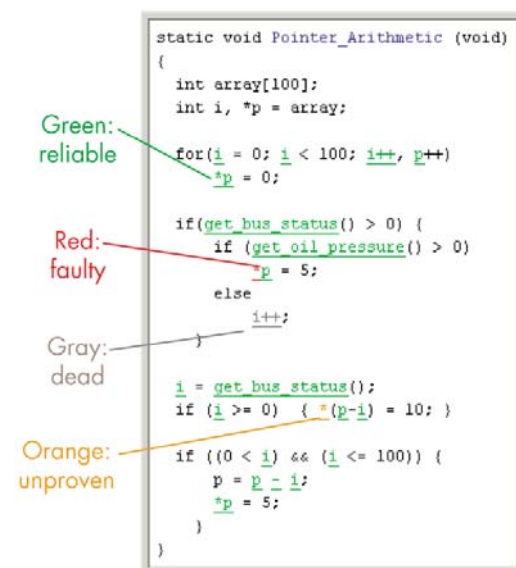


Abb. 5: Farbcodierung in PolySpace

Die folgende Übung demonstriert die Effektivität der Anwendung der Abstrakten Interpretation mit PolySpace. Dazu sei folgende Funktion angenommen:

```

1      int where_are_errors(int input)
2      {
3          int x, y, k;
4
5          k = input / 100;
6          x = 2;

```

```

7     y = k + 5;
8     while (x < 10)
9     {
10        x++;
11        y = y + 3;
12    }
13
14    if ((3*k + 100) > 43)
15    {
16        y++;
17        x = x / (x - y);
18    }
19
20    return x;
21 }

```

Das Ziel besteht darin, Laufzeitfehler in der Funktion `where_are_errors()` zu finden. Die Funktion führt verschiedene mathematische Berechnungen aus und enthält eine `while`-Schleife sowie eine `if`-Anweisung. Man beachte, dass alle Variablen initialisiert sind und verwendet werden. In Zeile 17 könnte eine Division durch Null auftreten, falls $x = y$ wird. Erlauben aber die eingesetzten Steuerstrukturen und die an x und y durchgeführten mathematischen Operationen, dass $x = y$ werden kann?

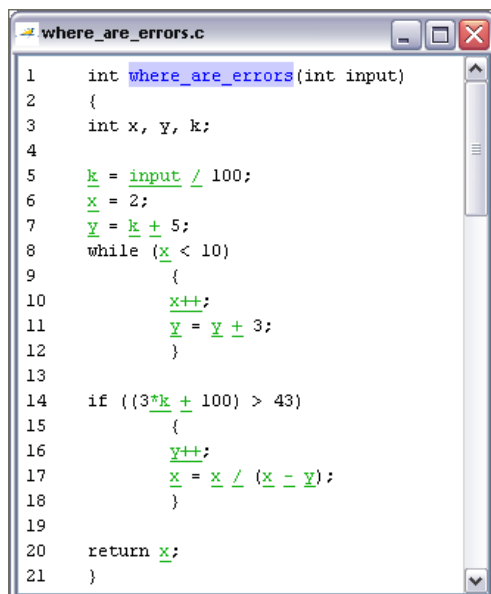


Abb. 6: PolySpace-Ergebnisse für korrekten Code

Wie in Abbildung 6 dargestellt, hat PolySpace bewiesen, dass es in diesem Code keine Laufzeitfehler gibt. Der Grund dafür ist, dass Zeile 17 nur ausgeführt wird, wenn die Bedingung $(3*k + 100 > 43)$ als true ausgewertet wird. Weil aber außerdem der Wert von y von

dem von k abhängt, ermittelt PolySpace, dass y in Zeile 17 stets größer als 10 ist, wenn x gleich 10 ist. Darum kann in dieser Zeile keine Division durch Null auftreten.

Dieses Ergebnis wird effizient und ohne Ausführung, Instrumentierung und Debugging des Codes oder die Erzeugung von Testfällen ermittelt. PolySpace identifiziert außerdem sämtliche Operationen des Codes, die potenziell einen Laufzeitfehler erzeugen könnten und unterstreicht sie (s. Abb. 6). Im obigen Beispiel sind sämtliche unterstrichenen Passagen grün markiert, weil keine Laufzeitfehler vorhanden sind. In Zeile 17 beweist beispielsweise die grüne Markierung des `/`-Operators, dass dieser sowohl sicher gegen Überläufe als auch gegen Divisionen durch Null ist.

Codeverifikation und medizinische Gerätesoftware

Die Nutzung einer auf der Abstrakten Interpretation basierenden Code-Verifikation ist nicht neu. So dokumentieren Brett und Klemm den Einsatz von PolySpace für missionskritischen Code in der Flight-Software des Mars Exploration Rover [14]. Best Practices aus Industriezweigen wie der Luft- und Raumfahrttechnik können auf einfache Weise auf Software für medizinische Geräte übertragen werden [1]. Es überrascht darum nicht, dass das Software-Labor der FDA Code-Verifikationstools einsetzt, um den Quellcode in medizinischen Geräten zu untersuchen, deren Embedded Software im Verdacht steht, unerwünschte Effekte verursacht zu haben [1]. Der Artikel stellt außerdem fest, dass diese Tools sowohl geholfen haben, Probleme im Code zu finden, als auch dazu beitragen, im Falle von Produktrückrufen Softwarefehler zu beseitigen.

Die Anwendung dieses Verifikationsverfahrens auf medizintechnische Gerätesoftware hat eine Reihe von Vorteilen. Mit auf dieser Methode basierenden Tools können sowohl Softwareentwicklungs- als auch Qualitätssicherungsteams auf effiziente Weise nachweisen, dass ihr Code (im Rahmen der Schlüssigkeit der eingesetzten Tools) frei von Laufzeitfehlern ist. Der folgende Abschnitt beschreibt eine Reihe weiterer Bedingungen für Laufzeitfehler, die mit Hilfe der Abstrakten Interpretation ermittelt werden können. Zum Zweck der Illustration wurde die Verifikation

wieder mit PolySpace durchgeführt; die Ergebnisse werden daher so dargestellt, wie dies in PolySpace der Fall wäre.

Beispiel: Dereferenzierung eines Zeigers außerhalb des gültigen Bereichs

```

1 void out_of_bounds_pointer(void)
2 {
3
4     int ar[100];
5     int *p = ar;
6     int i;
7
8     for (i = 0; i < 100; i++)
9     {
10         *p = 0;
11         p++;
12     }
13
14     *p = 5;
15
16 }
```

Im gezeigten Beispielcode wurde das Array *ar* für eine Größe von 100 Elementen angelegt. Durch Pointer Aliasing zeigt der Zeiger *p* auf das erste Element des Arrays *ar*. Die *for*-Schleife mit dem Zähler *i* zählt von 0 bis 99 und dies tut auch der Zeiger *p*. Am Ende der *for*-Schleife zeigt der Index auf Element 100 des Arrays *ar*. Der Versuch, an dieser Speicherstelle Daten zu speichern, erzeugt einen Laufzeitfehler.

Im Rahmen der Abstrakten Interpretation wird ein Graph erzeugt, der die Variable für den Zeiger *p* enthält. Durch nachfolgende Iteration und Lösung lässt sich nun beweisen, ob der für *p* reservierte Wertebereich überschritten werden kann und dadurch ein Array-Zugriff außerhalb des gültigen Bereichs erzeugt wird. Abbildung 7 zeigt die Ergebnisse der Codeverifikation mit PolySpace. Man beachte die grün markierten Anweisungen (Zeilen 8, 10, 11 sowie zum Teil 14). Diese stellen PolySpace-Checks dar, die anzeigen, dass in diesen Abschnitten kein Laufzeitfehler auftreten wird. Darüber hinaus hat PolySpace aber in Zeile 14 einen Fehler nachgewiesen. Die mit dem Zeiger *p* in den Speicher geschriebenen Daten stellen einen (rot markierten) Programmierfehler dar, der einen Laufzeitfehler auslöst.

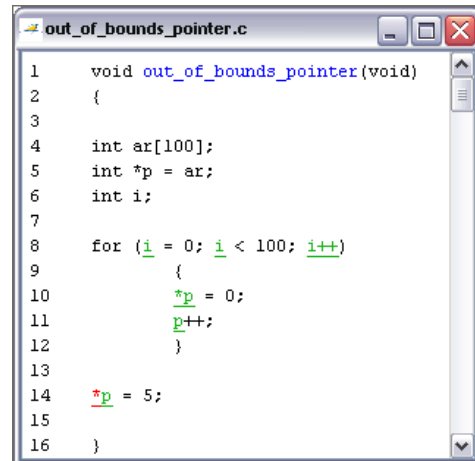


Abb. 7: PolySpace-Ergebnis für einen Zeiger außerhalb des gültigen Bereichs

Beispiel: Interprozedurale Aufrufe

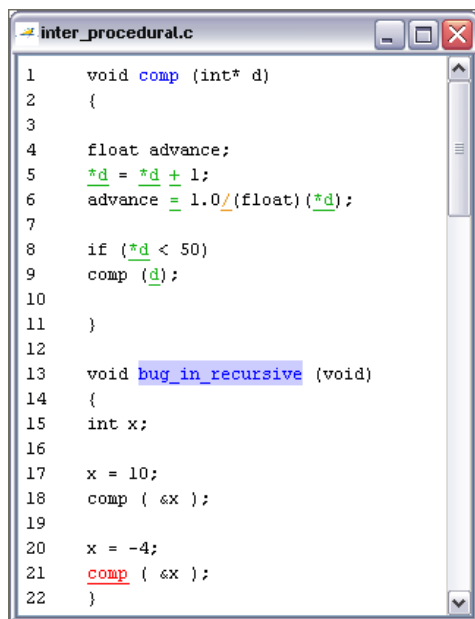
```

1 void comp (int* d)
2 {
3
4     float advance;
5     *d = *d + 1;
6     advance = 1.0/(float)(*d);
7
8     if (*d < 50)
9     comp (d);
10
11 }
12
13 void bug_in_recursive (void)
14 {
15     int x;
16
17     x = 10;
18     comp ( &x );
19
20     x = -4;
21     comp ( &x );
22 }
```

Im nächsten Beispielcode ist *d* eine Ganzzahl in der Funktion *comp()*, die jeweils um 1 erhöht wird. Sie dient danach als Nenner, mit dessen Hilfe der Wert der Variablen *advance* ermittelt wird, und wird schließlich wieder rekursiv an die gleiche Funktion übergeben. Zur Prüfung, ob die Divisions-Operation in Zeile 6 eine Division durch Null erzeugt, ist eine interprozedurale Verifikation erforderlich, die ermittelt, welche Werte im Laufe der Ausführung an die Funktion *comp()* übergeben werden.

Im Beispielcode werden zwei Werte an die Funktion *comp()* übergeben. Wird diese Funktion mit einem Startwert 10 aufgerufen, dann wird **d* eine monoton diskrete Variable,

die von 11 bis 49 wächst. In Zeile 6 entsteht hierbei keine Division durch Null. Wird `comp()` dagegen mit einem Startwert von -4 aufgerufen, dann wächst `*d` von -3 bis 49 und geht dabei insbesondere durch `*d = 0`. Zeile 6 erzeugt dann eine Division durch Null.



```

1  void comp (int* d)
2  {
3
4  float advance;
5  *d = *d + 1;
6  advance = 1.0/(float) (*d);
7
8  if (*d < 50)
9  comp (&d);
10
11 }
12
13 void bug_in_recursive (void)
14 {
15 int x;
16
17 x = 10;
18 comp (&x);
19
20 x = -4;
21 comp (&x);
22 }

```

Abb. 8: PolySpace-Ergebnisse für interprozeduralen Code

Eine einfache Syntaxprüfung entdeckt diesen Laufzeitfehler nicht. Eine Abstrakte Interpretation mit PolySpace, wie sie in Abbildung 8 gezeigt ist, beweist, dass aller Programmcode mit Ausnahme der Zeilen 6 und 21 lauffehlerfrei ist. Dazu ist zu beachten, dass der Funktionsaufruf von `comp()` in Zeile 18 erfolgreich ist, der in Zeile 21 jedoch nicht. Die Division durch Null ist daher orange hervorgehoben, denn beim Aufruf von `comp()` in Zeile 18 entsteht keine Division durch Null, beim Aufruf in Zeile 21 jedoch sehr wohl. Dieses Beispiel illustriert die einzigartige Fähigkeit der Abstrakten Interpretation, interprozedurale Analysen mit Pointer Aliasing durchzuführen und dadurch problematische Funktionsaufrufe von einwandfreien zu unterscheiden.

Verbesserung der Zuverlässigkeit medizinischer Geräte

Die auf der Abstrakten Interpretation basierende Codeverifikation stellt eine nützliche Methode

dar, die zur Verbesserung der Qualität und Zuverlässigkeit von Embedded Software in medizintechnischen Geräten eingesetzt werden kann. Durch die Anwendung eines umfassenden, auf mathematischen Beweisen basierenden Prozesses erkennt die Abstrakte Interpretation nicht nur Laufzeitfehler, sondern beweist auch die Abwesenheit dieser Laufzeitfehler im Quellcode. Konventionelle Verifikationstools sind darauf abgestimmt, Programmierfehler zu finden, verifizieren aber nicht die Zuverlässigkeit des restlichen Programmcodes. Eine auf Abstrakter Interpretation basierende Codeverifikation ermöglicht die eindeutige Identifikation von Softwarecode, der entweder einen Softwarefehler verursacht oder dies eben nicht tut.

Durch den Einsatz von Code-Verifikationstools in einem frühen Entwicklungsstadium können Softwareteams erhebliche Zeit- und Kostenersparnisse erzielen, indem sie Laufzeitfehler finden und eliminieren, während sie noch am einfachsten und kostengünstigen behoben werden können. Da die Abstrakte Interpretation auf der Ebene des Quellcodes arbeitet, muss die untersuchte Software zur Ermittlung definierter Programmierfehler nicht ausgeführt werden. Und schließlich optimiert die Abstrakte Interpretation den Debugging-Prozess, indem sie im Quellcode direkt die Ursache für Laufzeitfehler identifiziert, anstatt nur deren Symptome zu diagnostizieren.

Durch diese Lösung wird keine Zeit darauf verschwendet, Systemabstürze und durch korrupte Daten entstehende Fehler bis zum Quellcode zurückzuverfolgen. Auch Aufwände, sporadisch auftretende Fehler zu reproduzieren, werden damit überflüssig. Insgesamt lässt sich festhalten, dass eine Verifikationslösung, die neben anderen Methoden auch eine Abstrakte Interpretation einschließt, maßgeblichen Anteil an der Gewährleistung sowohl der Sicherheit von Software als auch eines qualitativ akzeptablen Entwicklungsprozesses haben kann. Dieser Verifikationsprozess ist logisch korrekt und mit ihm lässt sich eine hohe Integrität in medizintechnischen Geräten erzielen. Aufsichts- und Zulassungsbehörden wie die FDA und weitere Industriezweige erkennen den Wert von formalen Verifikationsprinzipien an und arbeiten daher mit Tools, die auf diesen Prinzipien basieren.

Dank

Die Autoren danken Brett Murphy, Gael Mulat, Jeff Chapple, Marc Lalo, Parasar Kodati, Patrick Munier, Paul Barnard und Tony Lennon von The MathWorks für ihre Hilfe beim Entwurf und der Durchsicht dieses Artikels.

Literatur

1. Taft, "CDRH Software Forensics Lab: Applying Rocket Science To Device Analysis", The Gray Sheet, 2007
2. Fries, "Reliable Design of Medical Devices", 2006
3. Knight, "Safety Critical Systems, Challenges and Directions", International Conference on Software Engineering, 2002
4. Ganssle, "When Disaster Strikes", Embedded Systems Design, 2004
5. Lee, et al., "High-Confidence Medical Device Software and Systems," IEEE Computer, 2006
6. Bliznakov, Mitalas, Pallikarakis, "Analysis and Classification of Medical Device Recalls", World Congress on Medical Physics and Biomedical Engineering, 2006
7. Wallace, Kuhn, "Failure Modes in Medical Device Software", International Journal of Reliability, Quality and Safety Engineering, 2001
8. Butler, Finelli "The Infeasibility of Quantifying the Reliability of Life-Critical Real Time Software", IEEE Transactions on Software Engineering, 1993
9. Hailpern, Santhanam, "Software Debugging, Testing, and Verification", IBM Systems Journal, 2002
10. Dijkstra, "Notes On Structured Programming", 1972
11. Ganssle, "Learning From Disaster", Embedded Systems Conference Boston, 2008
12. Cousot, "Abstract Interpretation: Theory and Practice", International SPIN Workshop on Model Checking of Software, 2002
13. Cousot, "Abstract Interpretation Based Formal Methods and Future Challenges", Informatics. 10 Years Back. 10 Years Ahead, 2001
14. Brat, Klemm, "Static Analysis of the Mars Exploration Rover Flight Software", First International Space Mission Challenges for Information Technology, 2003
15. Fagan, "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, 1976
16. Wagner, "A Literature Survey of the Software Quality Economics of Defect Detection Techniques", ACM/IEEE International Symposium on Empirical Software Engineering, 2006
17. Engler et al, "Weird Things That Surprise Academics Trying to Commercialize a Static Checking Tool", Static Analysis Summit, 2006
18. Chapman, "Language Design for Verification", Static Analysis Summit, 2006
19. Deutsch, "Static Verification of Dynamic Properties, SIGDA, 2003
20. Cousot, "Abstract Interpretation", ACM Computing Surveys, 1996
21. FDA, www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfMAUDE/TextSearch.cfm, 2009
22. Cousot and Cousot, "Comparing the Galois Connection and Widening / Narrowing Approaches to Abstract Interpretation", Symposium on Programming Language Implementation and Logic Programming, 1992