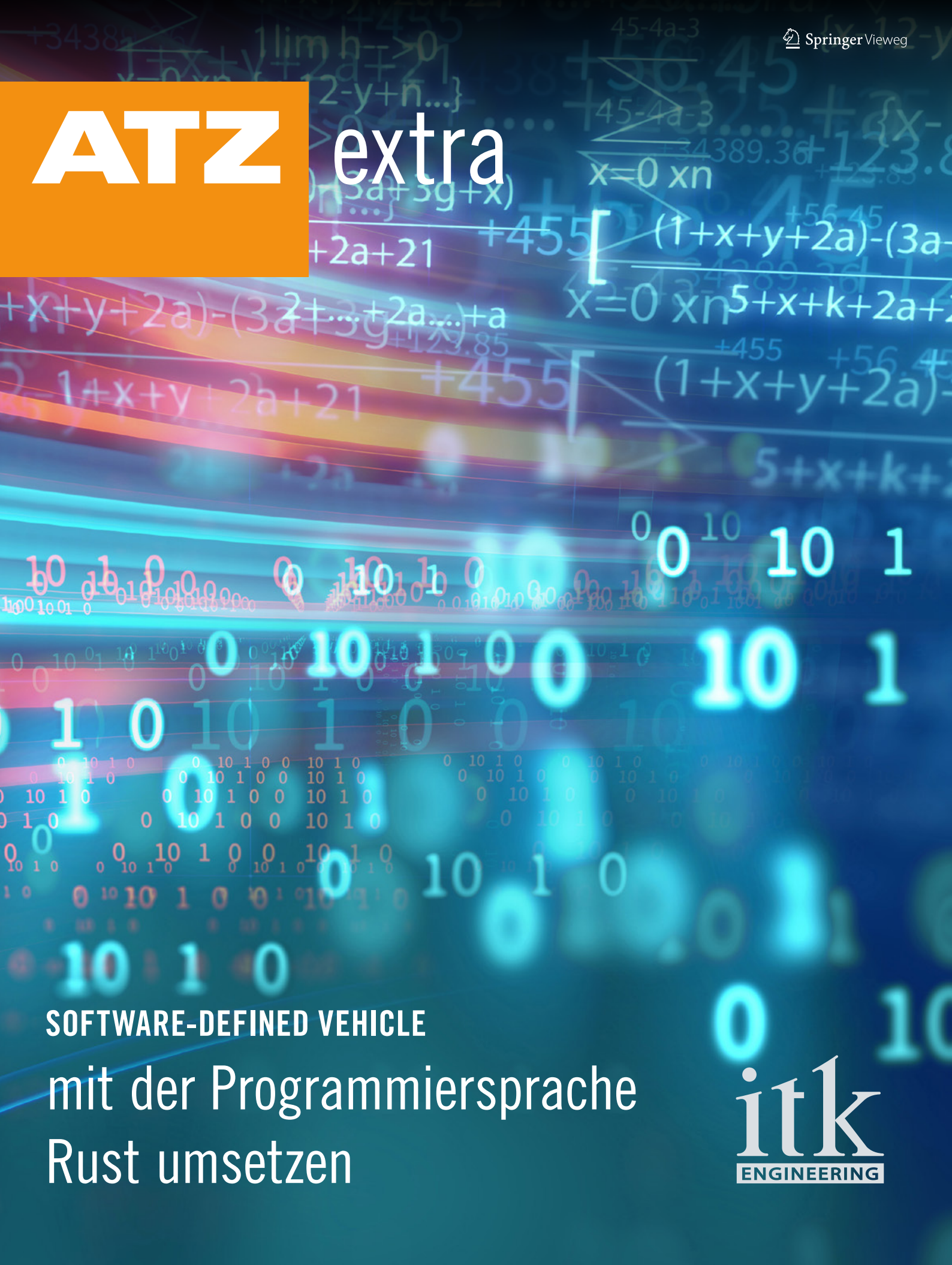
Springer Vieweg

ATZ extra



SOFTWARE-DEFINED VEHICLE

mit der Programmiersprache Rust umsetzen

itk ENGINEERING

Rust-Integration auf Basis von Interoperabilität in bestehender Software

VERFASST VON

Christopher Schwager
ist Fachreferent Realtime Architectures bei der ITK Engineering GmbH in Rülzheim.

Gängige Programmiersprachen stoßen aufgrund steigender Komplexität, schneller Entwicklungszyklen und wachsender Qualitätsansprüche an ihre Grenzen. Rust hat das Potenzial, die führende Sprache der modernen Softwareentwicklung zu werden. ITK Engineering beschreibt, wie eingebaute Funktionen für Softwarequalität sicheres Coden ermöglichen und den Aufwand für Debugging und Tests minimieren.

Über alle Branchen hinweg spielen digitale Funktionen eine zentrale Rolle, und auch bei Endverbrauchern sind Apps und digitale Lösungen fest etabliert. Künftig werden vermehrt Vernetzungs-, Automatisierungs- und Personalisierungsfunktionen durch Software(SW)-Lösungen erwartet und realisiert. Dies wird das Kundenerlebnis und die Spezifikationen der zugrunde liegenden Hardware maßgeblich beeinflussen. Die steigende Integration von software-

definierten Elementen erfordert eine kontinuierliche Verbesserung der SW-Entwicklung. Künftig ist ein ständiger Datenaustausch mit der Cloud notwendig, um SW innerhalb der Hardwaregrenzen laufend zu optimieren. Hersteller müssen schnellere Entwicklungszyklen einhalten, um sich durch die Aktualisierung von Funktionen und Upgrades über den gesamten Lebenszyklus abzuheben.

Dies führt zwangsläufig zu einer steigenden SW-Komplexität, was mehr Zeit für die Validierung erfordert, insbesondere durch umfassendes Testen und Debugging, um Fehler zu vermeiden oder zu korrigieren – insbesondere, wenn Aspekte wie funktionale Sichereit zunehmend an Relevanz gewinnen. Ein Grund dafür liegt in den klassischen Programmiersprachen C und C++ der Embedded-SW-Entwicklung, denen grundlegende Sicherheitsgarantien wie Speicher- oder Threadsicherheit fehlen. Diese Defizite zählen zu den Hauptursachen für Programmabstürze. Zahlreiche Werkzeuge und Ansätze wurden entwickelt, um die Qualität in einer umfassenden Validierungsphase zu sichern. In einer softwaredefinierten Welt sind die damit verbundenen steigenden Kosten nicht mehr akzeptabel [1]. Es bedarf neuer Lösungen, um Effizienz und Qualität bereits während der Programmierung zu gewährleisten. Eine Lösung könnte eine Programmiersprache wie Rust sein.

RUST ALS MODERNE PROGRAMMIERSPRACHE

Rust [2] erlebt in den letzten Jahren einen großen Aufschwung und wird von Technologieunternehmen wie Google [3] oder Amazon [4] als neue Sprache übernommen. Die Multiparadigmen-Programmiersprache wurde entwickelt, um Sicherheit zu gewährleisten, Parallelprogrammierung zu unterstützten, einfach erlernbar zu sein und sich gleichzeitig für leistungskritische Anwendungen zu eignen. Gerade im Bereich der Sicherheit setzt sie mit ihrer neuartigen Speicherverwaltung Maßstäbe. Das Konzept der Eigentümerschaft von Variablen gewährleistet zur Kompilierzeit die Speichersicherheit. Alternativprinzipien wie Garbage Collection müssen daher nicht zur Laufzeit angewendet werden, was zu einer signifikanten Leistungssteigerung der SW führt. Jede Variable und jeder Wert haben einen Eigentümer. Variablen müssen ausgeliehen werden, wenn andere Funktionen den Wert manipulieren wollen. Dies stellt sicher, dass der Wert nicht gleichzeitig von jemand anderem verändert wird. Darüber hinaus wird die Variable aufgeräumt, sobald der Eigentümer den Gültigkeitsbereich verlässt. Zusätzlich gibt es aufgrund des strengen Typsystems keine impliziten Konvertierungen zwischen Variablen. Ein weiterer Aspekt sind Codierrichtlinien wie MISRA. Ein Abgleich hat ergeben, dass mit Rust 80 % der MISRA-Richtlinien irrelevant werden, was zu höherer Effizienz und reduzierten Kosten führt. Darüber hinaus kann Performance, Effizienz und Kontrolle von Rust mit etablierten Sprachen mithalten oder übertrifft diese sogar [5]. Insgesamt eignet sich Rust aufgrund seiner guten Leistungs- und Sicherheitsaspekte für alle Arten von Anwendungen, von eingebetteten Echtzeit- bis hin zu webbasierten Anwendungen, und hat daher das Potenzial, etablierte Programmiersprachen zu ersetzen.

HERAUSFORDERUNGEN BEI DER INTEGRATION

Ein vollständiger Wechsel der Programmiersprache ist jedoch in den seltensten Fällen möglich, da die vorhandene SW oftmals über Jahre mit erheblichem Aufwand aufgesetzt und entwickelt wurde. Es ist daher in der Regel finanziell und zeitlich nicht sinnvoll, diese Entwicklung in einer anderen Programmiersprache zu wiederholen. Darüber hinaus wurde die SW mit entsprechenden Tools qualifiziert und validiert. Zudem haben etablierte SW-Plattformen oder Technologiestacks, zu denen Betriebssysteme wie Linux und QNX, aber auch Frameworks wie das Robot Operating System (ROS) oder Autosar gehören, eines gemeinsam: Sie wurden in C beziehungsweise C++ entwickelt und bieten Programmierschnittstellen (Application Programming Interfaces, API) in diesen Programmiersprachen an. Um von Rust profitieren zu können, sind Migrationspfade oder Möglichkeiten erforderlich, um Interoperabilität zu ermöglichen. Dabei spielt nicht nur die grundsätzliche Kompatibilität eine Bedeutung, sondern auch, wie die gegenseitige Integration von Softwarebestandteilen funktioniert und Schnittstellen realisiert werden.

FOREIGN FUNCTION INTERFACE ALS GRUNDLAGE DER INTEROPERABILITÄT

Das Foreign Function Interface (FFI) ist ein Mechanismus, der es einer SW ermöglicht, in einer anderen Programmiersprache verfasste Funktionen oder Dienste zu nutzen. Es dient als Verbindung zwischen Aufrufkonventionen und Semantik beider Programmiersprachen. Auf Maschinencodeebene spielt das Application Binary Interface (ABI) eine Rolle, das Konventionen wie das Speicherlayout sowie die Kodierung von Bits und Bytes auf Maschinencodeebene definiert. Letztendlich müssen beide Ebenen zusammenpassen, **BILD 1**. Rust stellt ein FFI bereit, das Funktionsaufrufe von und zu C erlaubt, **BILD 2**. Über diese Schnittstelle kann auch mit weiteren Programmiersprachen wie C++ interagiert werden. Grundsätzliche Anwendungsfälle sind die Integration von C in Rust oder vice versa, wobei in Rust das FFI definiert werden muss [6].

Ausgangspunkt für die Integration von C in Rust ist die C-API des zu integrierenden Codes. Die Schnittstelle, sowohl Datentypen als auch Funktionssignaturen, sind in einem externen Block in Rust abzubilden. Aufgrund der Annahme von Rust, dass fremde Funktionen unsicher sind, sind Aufrufe in einem unsafe Block zu packen. Deshalb empfiehlt sich zusätzlich ein sicheres Interface, um das rohe C-Interface zu implementieren, das die verlorenen Garantien wiederherstellt. Der Compiler kann nicht prüfen, ob die Deklarationen korrekt sind. Bei der Integration von Rust in C können einzelne öffentliche Funktionen mit dem Schlüsselwort extern „C“ und dem Attribut *#[no_mangle]* für die Verwendung in C vorbereitet werden. Entsprechend kompatible C-Header müssen zusätzlich erstellt werden. Hier sind keine Besonderheiten zu berücksichtigen, solange die Datentypen kompatibel sind.

INTEROPERABILITÄT VON DATENTYPEN

Skalare Datentypen wie Ganzzahlen, beispielsweise Unsigned 32-bit Integer u32, und Gleitkommazahlen (Float/Double) sind zwischen Rust und C binärkompatibel. Dies rührt bei den Gleitkommazahlen daher, dass beide Sprachen

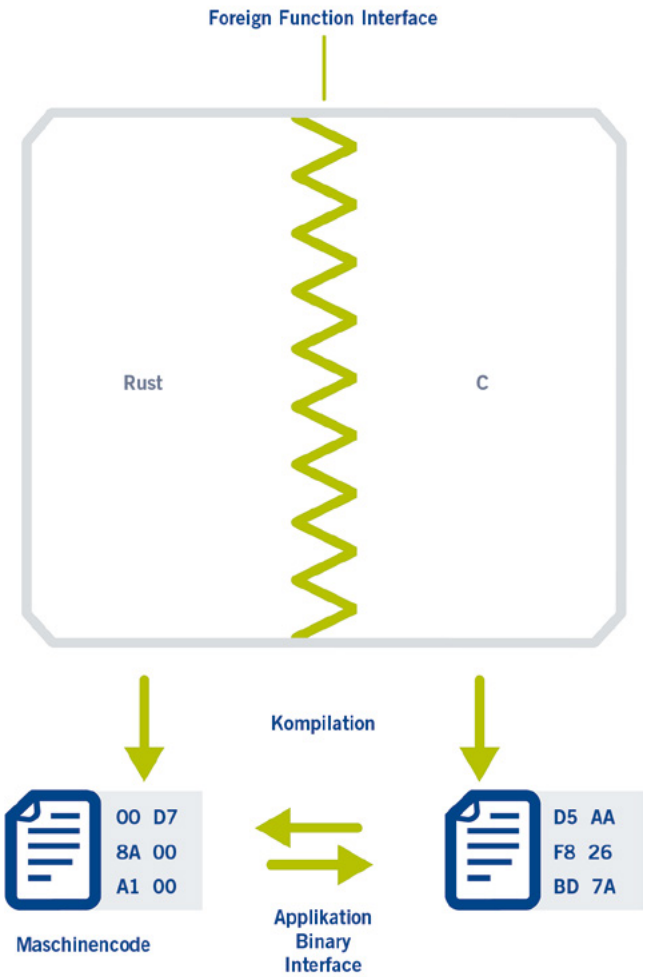


BILD 1 Interoperabilität auf FFI- und ABI-Ebene (© ITK Engineering)

eine Repräsentation konform zum IEEE-754 Standard verwenden. Beim booleschen Datentyp oder jenem für Zeichen hingegen gibt es Unterschiede zu beachten. In C wird der Wert false von einer Null repräsentiert, wohin gegen jede Zahl ungleich Null true entspricht.

Erst mit der Einführung des C99-Standards gibt es einen expliziten Datentyp (bool/_Bool) und entsprechende Werte für true = 1 und false = 0. Dies entspricht auch der Definition in Rust [7]. Viele C-Bibliotheken nutzen aber eine eigene Typdefinition auf Grundlage einer Ganzzahl und entsprechende Makros. Da die Datentyplänge und die Definitionen der Wahrheitswerte abweichen können, ist undefiniertes Verhalten am Interface möglich.

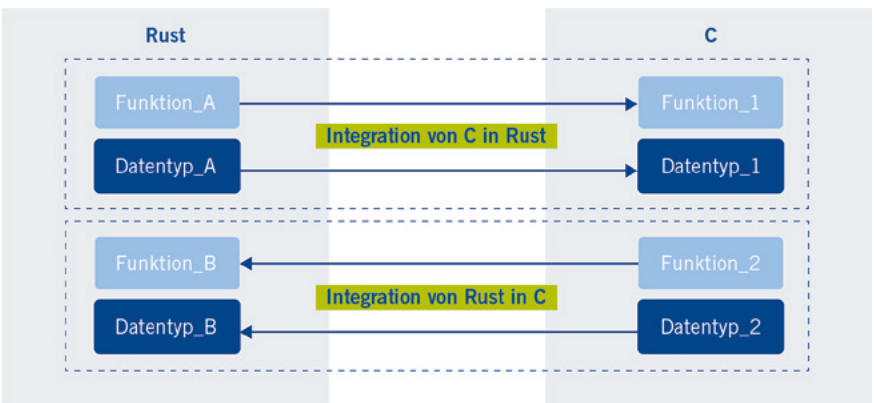


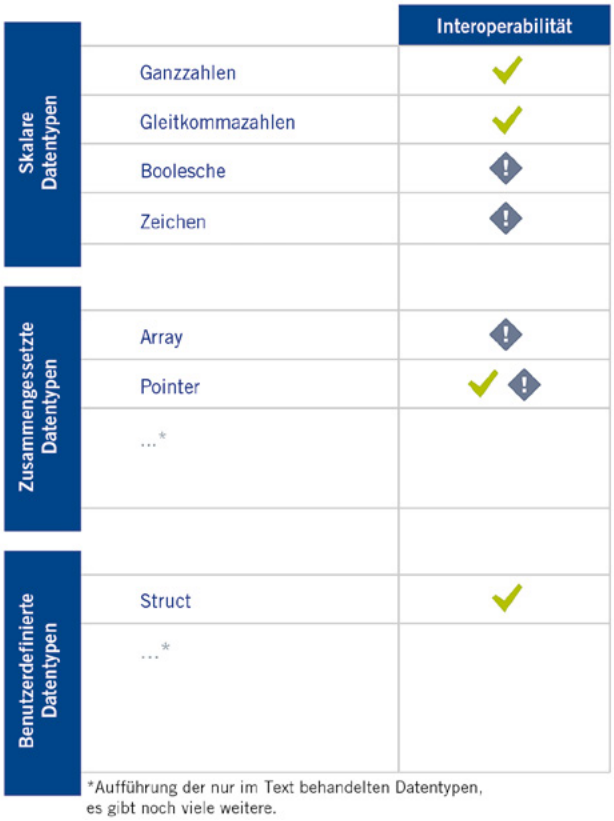
BILD 2 Verschiedene Anwendungsfälle der Interoperabilität mit C (© ITK Engineering)

Bei der Repräsentation von Zeichen ist der Unterschied noch gravierender. So werden Zeichen (char) in Rust als ein Unicode Scalar Value mit der Größe von 4 Byte repräsentiert [8]. Stattdessen wird in C in einem Byte ein Zeichen aus dem Basic Character Set (typischerweise ein ASCII-Zeichen) in Form einer Ganzzahl dargestellt. Daher kann hier keine vollständige Interoperabilität auf Basis des C char hergestellt werden, da lediglich die gemeinsame Menge, das Basic Character Set, ausgetauscht werden kann. Seitens Rust ist jeweils eine Typkonvertierung durchzuführen und aus Sicht von C müssen ungültige Zeichen gefiltert und entsprechend behandelt werden.

Auch bei zusammengesetzten Datentypen gibt es Hindernisse. Beispielsweise spielt bei Arrays neben den eigentlichen Datenelementen in Rust auch die Länge derselben als Metadaten eine besondere Rolle, da so beim Zugriff festgestellt werden kann, ob dieser innerhalb der Grenzen des Arrays liegt. In C sind Arrays Zeiger auf das erste Element. Eine äquivalente Überprüfung ist hier nur in seltenen Ausnahmefällen möglich. Ist die Länge dynamisch, muss diese als weiteres Argument neben der Startadresse mitgegeben werden. Dies führt bei der Rekonstruktion in Rust aus einem C-Zeiger und dessen Länge dazu, dass lediglich ein Slice erzeugt werden kann. Dessen maßgeblicher Nachteil gegenüber einem Array ist, dass die Länge nicht zur Compilezeit feststeht. Dadurch kann die Überprüfung auf Zugriffe außerhalb der Grenzen erst zur Laufzeit geschehen. Des Weiteren ist für die Erzeugung des Slice ein unsafe Block notwendig [9], was zu undefiniertem Verhalten führen kann.

Rohe Zeiger sind grundsätzlich ABI-kompatibel, wobei der Rust Compiler hier keine Garantien, etwa hinsichtlich Speichersicherheit, geben kann. Ihre Verwendung bedingt daher die Nutzung eines unsafe Blocks. Zur Vermeidung kann zumindest auf Options und/oder Referenzen zurückgegriffen werden, was aber bedingt, dass der Aufrufende für die Qualität der Daten verantwortlich wird. Benutzerdefinierte Datentypen wie Strukturen weisen weniger

BILD 3 Verschiedene Datentypen sind interoperabel zwischen den Programmiersprachen C und Rust (© ITK Engineering)



potenzielle Fehlerquellen auf. Mit dem Attribut *#[repr(C)]* können diese in eine C-Repräsentation gebracht werden. Themen wie Bit-/Byte-Padding, Data-Alignment und Packing sind dadurch gelöst. **BILD 3** zeigt eine Zusammenfassung der Interoperabilität von Datentypen.

FFI-DEFINITION

Ein kompatibles Interface zwischen Rust und C von Hand zu schreiben, erscheint mühsam. Da die technische Umsetzung allerdings geradlinig ist, gibt es hierfür FFI-Generatoren, wie etwa bindgen [10] oder cbindgen, welche die notwendigen Rust- und C-Files automatisch erzeugen können. Sofern kein komplexes Interface mit vielen Abhängigkeiten existiert, ist dies erfahrungsgemäß ein guter und gängiger Weg. Beim Aufruf von C-Code aus Rust kann anschließend manuell ein sicherer Wrapper ergänzt werden. Dieser muss einerseits sicherstellen, dass der Aufruf trotz des unsicheren Aufrufs des C-Interfaces für alle Ein- und Ausgaben sicher ist, wozu gegebenenfalls auch die Typkonvertierung gehört, sowie andererseits Lifetimes und Ownership berücksichtigen. Dazu gehört etwa, dass entsprechend Speicher wieder freigegeben wird, sofern dieser vollständig von C an Rust übergeben wurde, auch im Fall einer Panik.

Im umgekehrten Fall (Aufruf von Rust aus C) ist das externe Rust-Interface zunächst manuell festzulegen. Ziel ist es, hier auf unsafe Blöcke zu verzichten und trotzdem ein fehlertolerantes Interface zu definieren: die Verwendung von *Option<&T>* oder *Option<Box<T>>* anstatt rohen Zeigern, um null-Zeiger tolerant zu werden, sowie die Berücksichtigung von Lifetimes und Ownership.

ZUSAMMENFASSUNG

Die softwaredefinierte Welt verlangt nach Ansätzen, die mit steigenden Kosten und Komplexität umgehen können und die Effizienz und Qualität bereits während der Programmierung gewährleisten. Rust als sichere Programmiersprache hat dieses Potenzial. Ein vollständiger Wechsel der Programmiersprache ist jedoch nur selten möglich und sinnvoll. Deshalb braucht es Optionen, um Rust mit etablierten Programmiersprachen wie C zu kombinieren. Essenziell ist die kompatible ABI, jedoch gibt es bei der FFI-Definition einige Herausforderungen. Werden diese gelöst, ist die Interoperabilität beider Sprachen gewährleistet.

LITERATURHINWEISE

[1] Roland Berger (Hg.): Computer on wheels Part 4, (July 2022). Online: https://content.roland-berger.com/hubfs/07_presse/Roland_Berger_Article_Computer_on_Wheels_4_2022.pdf, aufgerufen: 21. März 2024
[2] Rust Team (Hg.): Rust. A language empowering everyone to build reliable and efficient software. Online: https://www.rust-lang.org/, aufgerufen: 26. Januar 2024
[3] Miller, S.; Lerche, C.: Sustainability with Rust. In: AWS (Hg.): AWS Open Source Blog. Online: https://aws.amazon.com/de/blogs/opensource/sustainability-with-rust/, aufgerufen 26. Januar 2024
[4] Vander Stoep, J.; Hines, S.: Rust in the Android platform. Online: https://security.googleblog.com/2021/04/rust-in-android-platform.html, aufgerufen: 26. Januar 2024
[5] Ng, V.: Rust vs C++, a Battle of Speed and Efficiency. In: Journal of Mathematical Techniques and Computational Mathematics 2 (2023), Nr. 6, S. 216-220
[6] Rust Community (Hg.): The Rustonomicon. Foreign Function Interface. Online: https://doc.rust-lang.org/nomicon/ffi.html, aufgerufen: 26. Januar 2024
[7] Beingessner, A: Notes on Type Layouts and ABIs in Rust. Online: https://faultlore.com/blah/rust-layouts-and-abis/#the-layoutsabis-of-builtins, aufgerufen: 26. Januar 2024
[8] Rust Community (Hg.): The Rust Reference. Online: https://doc.rust-lang.org/reference/types/textual.html, aufgerufen: 26. Januar 2024
[9] Rust Community (Hg.): The Rust Standard Library. Online: https://doc.rust-lang.org/std/slice/fn.from_raw_parts.html, aufgerufen: 26. Januar 2024
[10] Rust Community (Hg.): Rust-Bindgen. Online: https://github.com/rust-lang/rust-bindgen, aufgerufen: 26. Januar 2024

IMPRESSUM
Sonderausgabe 2023 in Kooperation mit ITK Engineering GmbH, Bergfeldstraße 2, 83607 Holzkirchen; Springer Fachmedien Wiesbaden GmbH, Postfach 1546, 65173 Wiesbaden, Amtsgericht Wiesbaden, HRB 9754, USt-IdNr. DE81148419

GESCHÄFTSFÜHRER:
Stefanie Burgmaier I Andreas Funk I Joachim Krieger
PROJEKTMANAGEMENT: Anja Trabusch

TITELBILD: © Shutterstock I whiteMocca

ITK Engineering

Seit der Firmengründung 1994 stehen wir für Stabilität, Sicherheit und Methodenexpertise. Damals wie heute bildet branchenübergreifendes Spezialwissen insbesondere im Bereich der Regelungstechnik und der modellbasierten Entwicklung die Basis, um unsere Kunden von der Idee bis zur Serienproduktion durchgängig und partnerschaftlich zu begleiten.

Unsere Kompetenzen umfassen u.a.:

- Softwareentwicklung
- Hardwareentwicklung
- Elektrik/Elektronik
- Systemintegration
- Software als Produkt
- Komplettlösungen
- Auftragsentwicklung
- Technische Beratung
- Schulungen
- Qualitätssicherung

Die Zufriedenheit all unserer Partner und ein respektvolles Miteinander prägen unsere Unternehmensphilosophie, in der vier Werte fest verankert sind: Lesen Sie gerne mehr darüber im Web.

V.1.0.0_d_2021

ITK Engineering GmbH
Hauptsitz Rülzheim
Im Speyerer Tal 6
76761 Rülzheim
Tel.: +49 (0)7272 7703-0
Fax: +49 (0)7272 7703-100
info@itk-engineering.de

www.itk-engineering.de
www.itk-karriere.de

Gegründet 1994 –
heute hat ITK deutschlandweit Niederlassungen und ist international vertreten.

Folgen Sie uns auch auf: