Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Entkopplung der Z3 Komponente in ProB mit ZeroMQ

Bachelorarbeit

vorgelegt von

Silas Alexander Kraume

22. Januar 2025

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Dr. C. Bolz-Tereick

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 22. Januar 2025

Silas Alexander Kraume

Zusammenfassung

Fassen Sie hier die Fragestellung, Motivation und Ergebnisse Ihrer Arbeit in wenigen Worten zusammen.

Die Zusammenfassung sollte den Umfang einer Seite nicht überschreiten.

Macht man am Ende ...

Danksagung

Me, Myself and I!

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einführung

Die digitale Transformation hat unsere Welt grundlegend verändert und macht Software-Systeme zu einem unverzichtbaren Bestandteil des täglichen Lebens. Von sicherheitskritischen Anwendungen wie der Steuerung autonomer Fahrzeuge bis hin zu Finanzsystemen und medizinischen Geräten sind wir zunehmend auf Software angewiesen, die zuverlässig und fehlerfrei funktioniert. Die Gewährleistung von Korrektheit und Stabilität ist jedoch eine anspruchsvolle Aufgabe insbesondere angesichts der Komplexität moderner Systeme. Ein zentraler Baustein zur Bewältigung dieser Herausforderung ist der Einsatz von Modellierungs- und Verifikationswerkzeugen. Diese ermöglichen es, komplexe Systeme systematisch zu analysieren und sicherzustellen, dass sie den gewünschten Spezifikationen entsprechen. Besonders hervorzuheben ist der Einsatz von SMT[1]-Solvern, die sich als leistungsfähige Werkzeuge etabliert haben, um schwierige logische Probleme effizient zu lösen. SMT-Solver wie Z3 [MB08] bieten durch ihre Fähigkeit zur präzisen und schnellen Verarbeitung logischer Ausdrücke eine wertvolle Unterstützung bei der Verifikation und Validierung. Ein prominentes Beispiel für die Integration eines solchen Solvers ist die Software ProB [LB03]. Der Animator, Constraint-Solver und Model-Checker ProB nutzt den SMT-Solver Z3, um formale Modelle effizient zu analysieren und zu überprüfen. Dies macht die Software zu einer wichtigen Instanz in der Welt der formalen Methoden, insbesondere im Kontext von Modellierungs- und Verifikationsaufgaben.

1.1 Motivation

Innerhalb von ProB birgt der Einsatz des Z3-Solvers jedoch auch Herausforderungen, die die Effizienz und Zuverlässigkeit der Anwendung beeinträchtigen können. Ein bekanntes Problem besteht in dem sporadischen Auftreten von Speicherlecks und Segmentation Faults, die sowohl die Stabilität als auch die Nutzbarkeit von ProB's Z3-Interface negativ beeinflussen. Diese technischen Mängel erschweren nicht nur die Durchführung formaler Verifikationen, sondern können auch zu einer zeitraubenden Verwendung der Z3-Solver Komponente sowie Unterbrechung von Arbeitsprozessen führen. Die Problematik wird in einem Paper, geschrieben von Körner und Leuschel [KL23], genauer erörtert.

Ein weiterer Mangel liegt in der aktuellen sequenziellen Lösung mehrerer Prädikate. Dieser Ansatz, bei dem die Prädikate nacheinander gelöst werden, ist in seiner Natur ressourcenintensiv und zeitaufwendig. Angesichts der steigenden Komplexität formaler Modelle und der wachsenden Nachfrage nach schnellerer Verifikation wird die Limitierung durch die sequenzielle Verarbeitung immer offensichtlicher. Eine Parallelisierung der Lösung von Prädikaten könnte hier erhebliche Leistungsverbesserungen bringen, indem moderne Mehrkernarchitekturen effizienter ausgenutzt werden, um den Anforderungen der Nutzer und der immer komplexer werdenden Modelle gerecht zu werden.

[1] Satisfiability Modulo Theories

Die Kombination dieser Herausforderungen (sporadische technische Instabilitäten und begrenzte Effizienz durch sequenzielle Verarbeitung) macht es notwendig, alternative Ansätze oder Verbesserungen für die Integration des Z3-Solvers in ProB zu erforschen und bildet die Grundlage und Motivation für die vorliegende Arbeit.

1.2 Ziele

Das Hauptziel dieser Arbeit ist es, die Integration des Z3-Solvers in ProB zu verbessern, indem die bestehende Vorgehensweise, die Prädikate direkt im Z3-Interface von ProB zu lösen, verworfen wird. Stattdessen wird eine neue Architektur vorgeschlagen und implementiert, welche eine vollständige Entkopplung der Z3-Solver-Komponente von ProB vorsieht. Hierzu wird also der Z3-Solver in einen eigenständigen, separaten Prozess ausgelagert, wodurch ein System eingeführt wird, bei dem ProB und der Z3-Solver als zwei unabhängige Prozesse agieren, die über eine Kommunikationsschnittstelle miteinander verbunden sind. Prädikate werden hierdurch innerhalb des Z3-Interfaces an den Z3-Solver gesendet, wo diese gelöst und zurückgeschickt werden. Es entsteht somit ein simples Client-Server Modell, welches eine chronologische Folge von Anfragen und Antworten implementiert. ProB stellt hierbei den Client und der Z3-Solver den Server dar. Diese geplante Architekturänderung ist in der folgenden Abbildung 1 visualisiert.

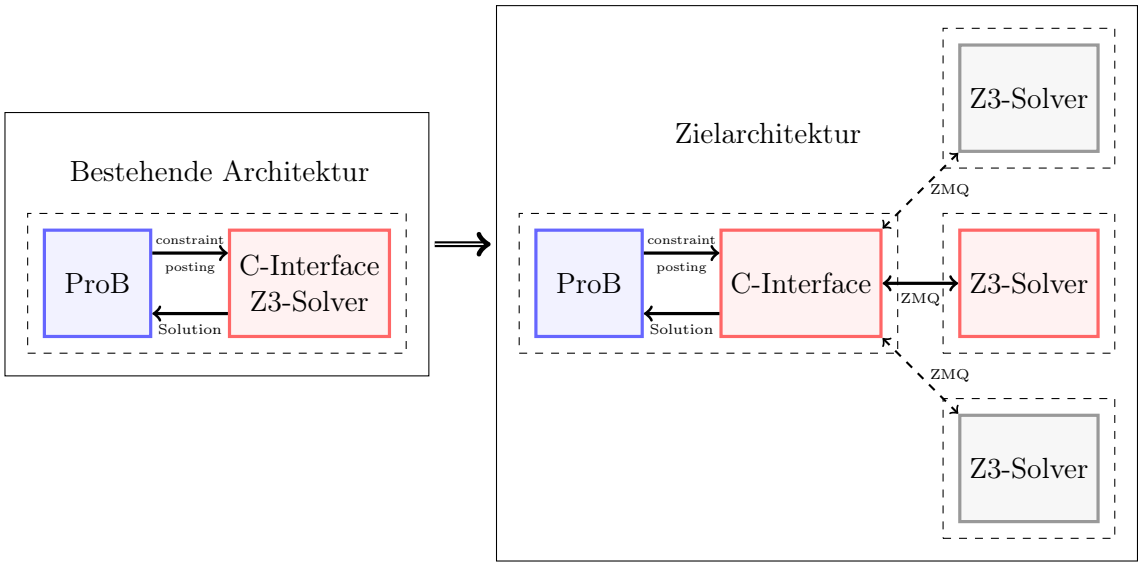

Abbildung 1: Die umgesetzte Architekturänderung (Die Komponenten-Entkopplung ist in den roten Boxen dargestellt. Die gestrichelten Boxen zeigen die verschiedenen Prozesse an.)

Diese Arbeit wird einerseits mit dem Interesse der Erweiterbarkeit verrichtet, sodass zukünftig die Option besteht, gegebenenfalls mehrere Instanzen des Z3-Prozesses zu starten und das Lösen der Prädikate zu parallelisieren. Andererseits dient die Entkopplung selbst

bereits zur Verbesserung der Stabilität und Zuverlässigkeit von ProB, da bei eventuellen Fehlern im Z3-Solver-Prozess dieser unabhängig von ProB neu gestartet werden kann, was zu einem robusteren Gesamtsystem führt.

Die genauen Technologien und Konzepte, die hierfür zum Einsatz kommen und Relevanz zeigen, sowie ihre Funktionsweise und Vorteile werden im folgenden Kapitel detailliert erläutert. Daraufhin wird die Planung und Implementierung der neuen Architektur beschrieben und die Leistungsfähigkeit der vorgenommenen Entkopplung anhand von Benchmarks und Tests hinsichtlich des Vergleichs zur vorherigen Systemstruktur evaluiert. Zuletzt wird auf zukünftige Erweiterungen und Verbesserungen eingegangen und eine abschließende Konklusion genannt.

2 Grundlagen

Zur Förderung eines einheitlichen Verständnisses werden in diesem Abschnitt zunächst die erforderlichen Hintergrundinformationen illustriert. Im Folgenden werden die drei zentralen Konzepte behandelt, die für das Verständnis dieser Arbeit von Bedeutung sind: ProB, Z3 und ZeroMQ.

2.1 ProB

Die B-Methode, entwickelt von J.-R. Abrial [Abr96], ist eine formale Methode zur Entwicklung von Softwaresystemen, die auf der Idee der abstrakten Maschinen basiert. Mit abstrakten Maschinen lassen sich Zustände und deren Veränderungen mithilfe mathematischer Konzepte wie Mengen, Relationen und Funktionen modellieren [LB03]. Durch sogenannte Verfeinerungen wird schrittweise von einer abstrakten Beschreibung zu einer konkreten Implementierung übergegangen. Dabei stellt die Methode sicher, dass Invarianten stets eingehalten werden, um die Korrektheit des Systems zu garantieren.

Die an der HHU[2] am Lehrstuhl der Softwaretechnik und Programmiersprachen entwickelte Software ProB [LB03] ist ein Validierungstoolset für Modelle der B-Methode. Als solcher unterstützt ProB mitunter die Modellierung, Animation und Verifikation von B-Modellen, indem Funktionalitäten wie Consistency Checking und Constraint Solving bereitgestellt werden. ProB findet bereits in vielen Systemen Verwendung zur Datenvalidierung und Validierung komplexer Eigenschaften für sicherheitskritische Systeme. Es wird bereits von mehreren Unternehmen eingesetzt und ist November 2022 mit dem „AlainColmerauer Prize“ ausgezeichnet worden.

ProB ermöglicht es, formale Spezifikationen zu visualisieren und zu animieren. Nutzer können durch die Simulation in Echtzeit einen Einblick in die Zustandsübergänge einer

[2] Heinrich-Heine-Universität

Maschine erhalten und schrittweise die Veränderungen nachvollziehen. Der aktuelle Zustand der Maschine wird dabei in einer grafischen Benutzeroberfläche dargestellt.

Ein weiterer Kernbestandteil von ProB ist das Consistency Checking, welches in zwei Ansätzen realisiert wird: Model Checking und Constraint-Based Checking.

Beim Model Checking wird versucht, eine Sequenz von Operationen zu finden, die ausgehend von einem Anfangszustand zu einer Verletzung der Invariante oder einem anderen Fehler führt. Im Gegensatz dazu fokussiert sich das Constraint-based Checking auf die Suche nach einem Zustand des Systems, der die Invariante noch erfüllt. Von dort aus wird geprüft, ob es eine einzelne Operation gibt, welche die Invariante verletzt oder anderweitige Fehler erzeugt.

Während das Model Checking eine umfassende Exploration aller Zustände ermöglicht, ist das Constraint-based Checking spezifischer, da es nur auf Fehler bei einzelnen Operationen fokussiert ist. Zusammen lassen sich so vollständige Fehler und problematische Operationen identifizieren.

Beide Ansätze bieten wertvolle Instrumente für die Konsistenzprüfung von B-Modellen, und sind in der Lage, die Verletzung von Invarianten und daraus folgenden Bedingungen sowie die Abwesenheit von Deadlocks und das Erreichen von spezifizierten Zielprädikaten zu überprüfen [LB08].

Zuletzt bietet ProB auch eine Constraint-Solving-Funktionalität, die es ermöglicht, unter Berücksichtigung von gegebenen Constraints (Einschränkungen) Lösungen für spezifische Prädikate zu finden, was die Grundlage des Animators und Model-Checkers bildet. Derartige Einschränkungen oder Bedingungen können in Form von logischen Ausdrücken oder Gleichungen gegeben sein, die es zu erfüllen gilt. Ein Constraint-Solver ist ein Algorithmus oder System, welches darauf abzielt, unter Berücksichtigung eben jener Bedingungen eine Belegung aller Variablen zu finden, die die gegebenen Prädikate erfüllt und somit ein Problem auf dessen Erfüllbarkeit zu prüfen. ProB implementiert hierfür verschiedene Constraint-Solving-Strategien, die auf unterschiedlichen Algorithmen basieren und es ermöglichen, Prädikate effizient zu lösen. Einerseits wird CLP(FD)[3] [CD96] verwendet, um auf endlichen Domänen beispielsweise Gleichheits- und Ungleichheitsbedingungen sowie arithmetische Relationen zu lösen. Ein weiterer Ansatz ist die Integration des SAT-basierten Kodkod [TJ07], einem effizienten Constraint-Solver für die Prädikatenlogik erster Ordnung mit Relationen, transitiven Hüllen, Bit-Vektor-Arithmetik und partiellen Modellen. Zuletzt wird auch der SMT-Solver Z3 in ProB integriert, um komplexere Prädikate zu lösen, die über simple boolsche und arithmetische Ausdrücke hinausgehen.

ProB ist im Kern in SICStus Prolog [CWA+88] implementiert, bietet jedoch verschiedene Programmerweiterungen, welche zumeist in C oder C++ geschrieben sind. Einer dieser Erweiterungen ist das Z3-Interface, welches die Integration des Z3-Solvers in ProB ermöglicht.

[3] Constraint Logic Programming over Finite Domains

2.2 Z3 Solver

Das Satisfiability Modulo Theories (SMT)-Problem lässt sich als eine natürliche Erweiterung des klassischen Boolean Satisfiability (SAT)-Problems verstehen. Während SAT sich ausschließlich mit der Erfüllbarkeit von booleschen Formeln beschäftigt, umfasst SMT zusätzliche Hintergrundtheorien wie Arithmetik, Bit-Vektoren, Arrays und uninterpretierten Funktionen. Diese Erweiterung ermöglicht eine umfassendere Form der logischen Schlussfolgerung.

Der Z3-Solver [MB08] wurde von Microsoft Research entwickelt. Seit seiner ersten Veröffentlichung hat sich Z3 zu einem der leistungsfähigsten und am weitesten verbreiteten SMT-Solver behauptet. Microsoft hat Z3 als Open-Source-Software frei zugänglich unter der MIT-Lizenz veröffentlicht, was seine Verbreitung und Nutzung in verschiedenen Bereichen weiter gefördert hat.

Der Solver wurde in C++ implementiert, bietet jedoch eine Vielzahl von externen API-Anbindungen, die es ermöglichen, den Solver in verschiedenen Programmiersprachen zu verwenden, wie beispielsweise OCaml, Python, Ruby und Rust. Er ist ein leistungsfähiger SMT-Solver, der sich speziell auf die Lösung von Problemen im Bereich der Softwareverifikation und -analyse spezialisiert. Zum Beispiel wird der Z3-Solver in der Softwareverifikation eingesetzt, um die Korrektheit von Programmen zu überprüfen, indem er formale Spezifikationen testet und beweist. Andere Einsatzbereiche sind die automatische Generierung von Testfällen basierend auf formalen Modellen sowie dem Modellieren von Entscheidungsproblemen und dem Abstrahieren von Prädikaten.

Der Aufbau von Z3 ist modular und basiert auf dem DPLL[4](T)-Framework, das die grundlegenden Prinzipien des DPLL-Algorithmus zur Lösung des CNF-SAT-Problems um weitere Theorien (wie SMT) ergänzt. Der DPLL ist ein Algorithmus zur Entscheidung der Erfüllbarkeit einer booleschen Aussagenlogikformel in konjunktiver Normalform (CNF). In Z3 dient der DPLL-Algorithmus als Kern des SAT-Solvers. Weitere Theorien wie lineare und nicht lineare Arithmetik, Bit-Vektoren und Arrays werden durch weitere Theorie-Solver behandelt, welche als spezialisierte Module eng mit dem SAT-Solver integriert sind.

Ein zentrales Merkmal von Z3 ist seine hohe Effizienz und seine Fähigkeit, große und komplexe Probleme in akzeptabler Zeit zu lösen. Diese Leistungsfähigkeit wird durch verschiedene Optimierungen und Heuristiken erreicht, die den Suchraum effizient einschränken und die Lösung von Constraints beschleunigen. Zudem ist ein Simplifer in die System Architektur integriert, der gegebene Constraints vereinfacht. Z3 hat nur wenige Abhängigkeiten an externe Bibliotheken und ist daher leicht zu integrieren und zu verwenden [MR15]. Es wird die C++ Bibliothek für das Threaden der Anwendung verwendet, um die parallele Verarbeitung zu ermöglichen und damit die Leistungsfähigkeit weiter zu steigern.

Innerhalb von ProB wird Z3 vor allem dann eingesetzt, wenn ProBs interner Constraint-Solver bei der Lösung bestimmter Probleme an seine Grenzen stößt [KL16]. Dies betrifft

[4] Davis-Putnam-Logemann-Loveland

Quellcode 1: Ein Beispiel zur Verwendung des Z3-Solvers innerhalb der ProB REPL.

```
1:  >>> :z3 X<Y & Y<X & X:INTEGER
2:  PREDICATE is FALSE
```

insbesondere große oder komplexe Constraints mit nicht-linearen Bedingungen. Beispielsweise lässt sich die Formel, welche in Quellcode 1 gezeigt wird, in Z3 aber nicht in ProB lösen.

Insgesamt macht die Leistungsfähigkeit und Flexibilität des Z3-Solvers ihn zu einem wichtigen Bestandteil von ProB, um Verifikationsaufgaben zu bewältigen, die über die Fähigkeiten des internen Constraint-Solvers hinausgehen.

2.3 ZeroMQ

In der geplanten Architektur wird als Technologie zur Kommunikation zwischen den zwei separaten Prozessen die ZeroMQ-Bibliothek [Hin13] verwendet. ZeroMQ[5] ist eine hochleistungsfähige, asynchrone Nachrichtenaustauschbibliothek, die speziell für verteilte Systeme entwickelt wurde. Da ZeroMQ in der Sprache C implementiert und ursprünglich für die Börse geschrieben wurde, lag extreme Performanceoptimierung lange Zeit im Fokus [S+12], was es zu einer der schnellsten und effizientesten Bibliotheken für den Nachrichtenaustausch macht.

ZeroMQ bietet eine Vielzahl von Kommunikationsmustern, die es ermöglichen, verschiedene Arten von verteilten Systemen zu realisieren. Dazu gehören unter anderem:

- Request-Reply (REQ/REP): Ein einfacher Nachrichtenaustausch, der für synchrone Kommunikation geeignet ist.
- Publish-Subscribe (PUB/SUB): Ermöglicht die Verteilung von Nachrichten an mehrere Empfänger, die sich für bestimmte Themen registrieren.
- Push-Pull (PUSH/PULL): Ein Muster für Lastverteilung, bei dem Nachrichten an Worker-Threads oder Prozesse verteilt werden.
- Dealer-Router (DEALER/ROUTER): Ein erweiterbares und flexibles Muster für komplexere (und asynchrone) Kommunikationsstrukturen.

Im Kern stellt ZeroMQ ein API über traditionelle Sockets bereit, welche die Komplexität des Netzwerkprotokollmanagements abstrahiert. Statt sich mit niedrigstufigen Details wie Verbindungen, Paketverwaltung und Fehlerbehebung auseinanderzusetzen, kann so

[5] auch ØMQ, 0MQ oder ZMQ

Quellcode 2: Ein minimales Client Modell mit ZeroMQ.

```
1:  s = socket(REQ);
2:  s.connect("ipc:///tmp/zmq");
3:  s.send("Hello");
4:  reply = s.recv();
```

Quellcode 3: Ein minimales Server Modell mit ZeroMQ.

```
1:  s = socket(REP);
2:  s.connect("ipc:///tmp/zmq");
3:  request = s.recv();
4:  s.send("Hello back");
```

die Implementierung von Anwendungslogik drastisch vereinfacht werden. Derartige API Anbindungen sind für viele Programmiersprachen verfügbar, darunter C++, Python, Java, Go und viele mehr.

ZeroMQ ist flexibel und einfach skalierbar, da es ohne einen dedizierten Message Broker auskommt und direkt zwischen den Prozessen kommuniziert. Es werden verschiedene Transportprotokolle unterstützt, darunter TCP (Transmission Control Protocol), IPC (Inter-Process Communication), (E)PGM ((Encapsuled) Pragmatic General Multicast) und viele mehr, die es ermöglichen, ZeroMQ in verschiedenen Umgebungen zu verwenden.

Insgesamt bietet ZeroMQ also eine hohe Geschwindigkeit, Flexibilität und Skalierbarkeit sowie ein simples Programmierinterface, was es zu einer idealen Wahl für die Kommunikation zwischen ProB und dem Z3-Solver macht. Zu Illustrationszwecken ist in Quellcode 2 und Quellcode 3 beispielhaft ein minimaler Client und Server in C++ mit ZeroMQ dargestellt.

Für die geplante Architekturänderung wird ZeroMQ verwendet, um eine direkte IPC-Protokollverbindung zwischen ProB und dem Z3-Solver herzustellen. Da der entkoppelte Solver auf demselben Rechner ausgeführt wird, bietet sich IPC als Kommunikationsprotokoll an, um eine effiziente und schnelle Kommunikation zu gewährleisten. Es wurde sich für das Request-Reply Muster entschieden, da es im Kontext die am besten passendste Lösung darstellt, ohne dem System unnötige Komplexität hinzuzufügen. Bei dem gewählten Muster ist zu beachten, dass die Kommunikation synchron erfolgt, was bedeutet, dass eine strikte Reihenfolge der Nachrichten eingehalten werden muss. Angefangen bei der ersten Request Nachricht ausgehend von ProB müssen sich die Nachrichten abwechseln, um eine korrekte Kommunikation zu gewährleisten. Dies ist im Kontrollfluss der Anwendungen zu berücksichtigen, um Deadlocks oder anderweitige Probleme zu vermeiden.

3 Architekturänderung

Im Folgenden wird die Planung und Implementierung der Architekturänderung zwecks Entkopplung der Z3-Komponente aus ProB beschrieben.

3.1 Planung

maybe every $$ zu textbf

Um eine korrekte Systemrefaktorisierung durchzuführen, ist es notwendig, zuvor grundlegende Kernaspekte des neuen Systems vollständig zu planen. Dies beinhaltet insbesondere die Struktur des neuen Server-Prozesses und den Aufbau der Nachrichten, die zwischen den Prozessen ausgetauscht werden.

3.1.1 Prolog Datentypen

zitate zu deutsch oder so

Innerhalb des bestehenden Z3-Interfaces werden verschiedene Prolog-Datentypen verwendet. Da durch die Entkopplung der Prozesse Prolog selbst und die Z3-Komponente nicht länger direkt miteinander kommunizieren können, müssen die Prolog-Datentypen in eine Form umgewandelt werden, die über das Netzwerk übertragen werden kann und von Z3 verstanden wird. Primär werden die folgenden Prolog-Datentypen verwendet: SP_atom, $SP\_integer$ und SP_term_ref. Mithilfe der C++ Bibliothek SICStus Prolog können diese Datentypen konvertiert werden. So stehen bei der Umwandlung von SP_atom in einen C++ String die Funktionen $SP\_string\_to\_atom$ und $SP\_atom\_to\_string$ zur Verfügung. Die Dokumentation von SICStus Prolog [al.23] schlägt unter dem Kapitel „Conversions between Prolog Arguments and C Types“ zudem zur Umwandlung von $SP\_integer$ den Datentyp C long vor. Der Typ SP_term_ref wird in der Dokumentation wie folgt beschrieben: „The argument could be any term.“. Er lässt sich nur schwer gezielt in einen C++ Datentyp umwandeln und muss daher mit besonderer Vorsicht behandelt werden.

3.1.2 Struktur der Nachrichten

Innerhalb von ZeroMQ lassen sich verschiedene Datenwerte gemeinsam in einen Nachrichtenblock packen. Ein solcher Nachrichtenblock wird über das abstrakte *zmsg* Objekt repräsentiert, welches es ermöglicht, komplexere Daten zusammen in sogenannten Frames zu übermitteln. Da sich der konkrete Inhalt der Nachrichten je nach Interfacefunktion unterscheidet, bietet es sich an, dieses Konzept zu nutzen und den Inhalt spezifisch für jede Funktion zu definieren und zu interpretieren.

Zusätzlich zu den eigentlichen Daten, die übermittelt werden, ist es jedoch notwendig, Metadaten zu übermitteln, die die korrekte Interpretation der Nachrichten auf der Empfängerseite ermöglichen. Wenn eine Anfragenachricht von ProB an den Z3-Server gesendet wird,

Quellcode 4: Ein Ausschnitt des Funktionsidentifikations-Enums.

```
1:  enum SP_Function {
2:    INIT = 0,
3:    PRETTY_PRINT_SMT = 1,
4:    PRETTY_PRINT_SMT_FOR_ID = 2,
5:    MK_VAR = 2,
6:    // ...
7:  }
```

Quellcode 5: Das Statusidentifikations-Enum.

```
1:  enum Z3Status {
2:    NOK,
3:    OK,
4:    UNFINISHED
5:  }
```

muss der Server wissen, welche Funktion aufgerufen werden soll. Dafür wird ein Funktionsidentifikator benötigt, der die Funktion eindeutig identifiziert. Diese Identifikatoren werden als ein Enum-Typ definiert, der die verschiedenen Funktionen als Konstanten repräsentiert und in sowohl der ProB- als auch der Z3-Komponente eingebunden wird. Ein Ausschnitt des Enum-Typs ist in Quellcode 4 gezeigt.

Wenn eine Antwortnachricht vom Z3-Server an ProB gesendet wird, muss die Nachricht ebenfalls Metadaten enthalten, die die korrekte Interpretation der Nachricht auf der ProB-Seite ermöglicht. Dafür wird ein Statusidentifikator benötigt, der den Status der Anfrage repräsentiert. Der hierfür verwendete Enum-Typ ist in Quellcode 5 gezeigt.

Die Statuswerte OK und NOK repräsentieren den Erfolg oder Misserfolg einer Anfrage. Diese Werte sind fundamental, da bei dem Aufkommen von potenziellen Fehlern oder Exceptions (siehe Abschnitt 3.2.6) in Z3 der ProB Prozess über den Misserfolg informiert werden muss, um eine entsprechende Fehlerbehandlung durchzuführen. Der Statuswert $UNFINISHED$ wird verwendet, wenn eine Anfrage noch nicht abgeschlossen ist und der Server auf weitere Informationen wartet (siehe Abschnitt 3.2.2).

Insgesamt besteht eine Nachricht also immer aus einem Identifikator (von Typ Integer), der entweder eine Funktion oder einen Status repräsentiert, gefolgt von den eigentlichen Daten, die übermittelt werden sollen, in Form eines *zmsg* Objektes.

3.1.3 Server Struktur

Da sich der Z3-Server mit nur einem einzigen Client beschäftigen muss, ist die Kernlogik des Servermodells simpel gehalten. Nach dem Instanziieren des Sockets wird eine

Quellcode 6: Die Kernstruktur des Z3-Servers.

```
1:  int f_id;
2:  zmsg_t *request = nullptr;
3:  zmsg_t *response;
4:
5:  while (true) {
6:    if (!receive_z3_request(&f_id, &request)) {
7:      break;
8:    } // extrahiere Funktionsidentifikator und Nachricht
9:
10:   response = zmsg_new();
11:   switch (f_id) {
12:     case SP_FUNCTION::INIT:
13:       init_contexts();
14:       break;
15:     case SP_FUNCTION::MK_VAR:
16:       // extrahiere Daten aus Nachricht
17:       t_type = receive_sp_string(request);
18:       s_value1 = receive_sp_string(request);
19:       // rufe Funktion auf und fuege Ergebnis zur Antwort hinzu
20:       zmsg_addstrf(response, "%ld", mk_var(t_type, s_value1));
21:       break
22:     // ...
23:   }
24:   if (!z3exc_occured) {
25:     send_z3_response(Z3Status::OK, &response);
26:   }
27: }
```

Endlosschleife gestartet, die auf eingehende Nachrichten wartet. Sobald eine Nachricht empfangen wird, wird zunächst der Funktionsidentifikator extrahiert und die entsprechende Funktion aufgerufen. Dies geschieht mittels einem Switch-Case-Block, der alle möglichen Funktionsidentifikatoren abdeckt. Nachdem die Funktion, mit den aus dem *zmsg* Objekt extrahierten Daten als Parameter ausgeführt wurde, wird eine Antwortnachricht an den ProB-Prozess gesendet, die den Status der Anfrage und die Ergebnisse enthält. Das Ergebnis ist in den meisten Fällen der Rückgabewert eben jener Funktion, die aufgerufen wurde. Die beschriebene Struktur des Servers ist in Quellcode 6 gezeigt.

Zusätzlich besitzen manche Hilfsfunktionen ihre eigene Logik zum Empfangen und Senden von Nachrichten, da sie in der Lage sein müssen, auf weitere Informationen zu warten, bevor sie ihre Berechnung abschließen können. Hierauf wird in Abschnitt 3.2.2 genauer eingegangen. Weitere Bestandteile, die zur Struktur des Servers gehören, sind die Initialisierung des Servers, die Verbindungsherstellung zum ProB-Prozess, das implementierte Logging und die Behandlung von Exceptions. Diese Aspekte werden in den entsprechenden Abschnitten (Abschnitt 3.2.4, Abschnitt 3.2.5, Abschnitt 3.2.6) genauer erläutert.

3.2 Implementierung

In den folgenden Abschnitten wird die Implementierung detailliert beschrieben. Dabei werden die angewandten Refaktorisierungsschritte sowie deren Umsetzung im Code erläutert.

3.2.1 Interfacefunktionen

Es existieren 56 Funktionen im Z3-Interface, welche direkt über die Prolog Schnittstelle aufgerufen werden können. Diese Interfacefunktionen müssen in der neuen Architektur so implementiert werden, dass sie die Nachrichten an den Z3-Server senden und die Antwort empfangen. Der Ablauf der Portierung einer Interfacefunktion ist dabei immer gleich:

1. Konvertieren der Funktionsparameter in einen C++ Datentyp
2. Erstellen einer Anfragenachricht mit Funktionsidentifikator und Daten
3. Senden der Anfragenachricht an den Z3-Server
4. Gegebenenfalls das Abarbeiten aller nötigen Hilfsfunktionen
5. Empfangen der Antwortnachricht
6. Extrahieren des Status und der Daten aus der Antwortnachricht
7. Konvertieren der Daten in einen Prolog-Datentyp
8. Rückgabe der Daten

Die Kernlogik der ursprünglichen Interfacefunktionen bleibt dabei erhalten, da sie effektiv nur auf die Seite des Z3-Prozesses ausgelagert wird, wobei SICStus Prolog Datentypen in C++ Datentypen umgewandelt werden.

Als Beispiel einer derartigen Refaktorisierung ist in Quellcode 7 die ursprüngliche Implementierung der Interfacefunktion mk_var gezeigt. In den Quellcodes 8 und 9 ist die neue Implementierung der Funktion in dem ProB- und Z3-Prozess dargestellt. Es ist zu erkennen, dass der ursprüngliche Funktionskörper der Funktion nahezu identisch zu der Implementierung innerhalb des Z3-Prozesses ist. Die ProB-Seite der Funktion hingegen wurde hinsichtlich der beschriebenen Vorgehensweise umstrukturiert. So werden in den Zeilen 5 bis 7 (Quellcode 8) die Funktionsparameter in einen $zmsg$ Block eingefügt und dabei in entsprechende Datentypen umgewandelt. In der Zeile 8 wird die Anfragenachricht mit dem Funktionsidentifikator an den Z3-Server gesendet. Die Zeilen 11 bis 13 zeigen die nachfolgende Abarbeitung aller notwendigen Hilfsfunktionen (in diesem Fall mk_type). Zuletzt wird ab Zeile 15 die Antwortnachricht empfangen, deren Informationen extrahiert, zurück in einen Prolog-Datentyp (in diesem Fall $SP\_Integer$) umgewandelt und schließlich zurückgegeben.

Quellcode 7: Die Interfacefunktion *mk_var* in der alten Architektur.

```
SP_integer mk_var(const SP_atom translation_type_atom,
  const SP_term_ref type, const char *varname) {
  std::shared_ptr<ContextData> ctx_data =
   get_translation_representant_ctx_data(translation_type_atom);
  z3::sort t = mk_type(translation_type_atom, type);
  std::shared_ptr<z3::context> ctx = ctx_data->get_context();
  try {
    return insert_id(ctx_data, ctx->constant(varname, t));
  } catch(z3::exception& e) {
    raise_Z3_exception(e, "mk_var");
    return -1;
  }
}
```

Quellcode 8: Die Interfacefunktion *mk_var* in der neuen Architektur (ProB Seite).

```
SP_integer mk_var(zsock_t *zocket,
 const SP_atom translation_type_atom, const SP_term_ref type,
 const char *varname) {
  zmsg_t *request = zmsg_new();
  zmsg_addstr(request,
   SP_string_from_atom(translation_type_atom));
  zmsg_addstr(request, varname);
  if (!send_z3_request(zocket, SP_FUNCTION::MK_VAR, &request)) {
    return -1;
  }
  if (!mk_type(zocket, type)) {
    return -1;
  }

  zmsg_t *response = nullptr;
  if (!receive_z3_response(zocket, &response)) {
    return -1;
  }
  if (response) {
    char *response_str = zmsg_popstr(response);
    SP_integer result = std::stol(response_str);
    free(response_str);
    return result;
  }
  zmsg_destroy(&response);
  return -1;
}
```

Quellcode 9: Die Interfacefunktion *mk_var* in der neuen Architektur (Z3 Seite).

```
1:  long mk_var(const std::string translation_type_atom,
2:   const std::string varname) {
3:    try {
4:      std::shared_ptr<ContextData> ctx_data =
5:       get_translation_representant_ctx_data(translation_type_atom);
6:      z3::sort t = mk_type(translation_type_atom);
7:      std::shared_ptr<z3::context> ctx = ctx_data->get_context();
8:      return insert_id(ctx_data, ctx->constant(varname.c_str(), t));
9:    } catch(z3::exception& e) {
10:     raise_Z3_exception(e, "mk_var");
11:     return -1;
12:   }
13: }
```

Im Gegensatz zu einer globalen Definition des Sockets im Z3-Prozess wird der Socket in der ProB-Seite als Parameter an jede Interfacefunktion gegeben. Dies wird erreicht, indem der Socket innerhalb der Prolog Schnittstelle als erstes Argument injiziert wird. Dadurch wird die Erweiterbarkeit des Codes gewährleistet, da es die Möglichkeit bietet, zukünftig mehrere Sockets zu verwalten und somit mehrere Z3-Server zu bedienen.

3.2.2 Hilfsfunktionen

Hilfsfunktionen werden definiert als diejenigen Funktionen, die nicht direkt über das Z3-Interface von ProB aufgerufen werden können, sondern als Unterstützung für die Implementierung der Interfacefunktionen dienen. Viele dieser Hilfsfunktionen weisen keine Kopplung zwischen Prolog und Z3 auf und können daher problemlos auf die Z3-Seite portiert werden. Andererseits gibt es auch Hilfsfunktionen, die eine minimale bis stark enge Kopplung besitzen. In diesen Fällen ist es notwendig, die Funktionen so zu refaktorisieren, dass sie auf der Z3-Seite ohne Prolog-Abhängigkeiten funktionieren.

Hilfsfunktionen, die nur eine minimale Kopplung aufweisen, benötigen nur wenige Kommunikationsnachrichten zwischen den Prozessen. Sie lassen sich in den meisten Fällen ähnlich wie die Interfacefunktionen refaktorisieren. Wichtig zu beachten ist, dass die Hilfsfunktionen inmitten der Funktionskörper der Interfacefunktionen aufgerufen werden. Da eine strikte Reihenfolge der Nachrichtenübermittlung eingehalten werden muss, ist es notwendig, dass die Hilfsfunktionen eine invertierte Kommunikationsstruktur verwenden. Das bedeutet, jede Hilfsfunktion, welche auf einen Nachrichtenaustausch angewiesen ist, ist auf eine Art und Weise zu implementieren, die immer zuerst eine Nachricht von dem Z3-Server an den ProB-Client sendet. Gleichermaßen endet jede Hilfsfunktion mit einer Nachricht von ProB an Z3. Somit ist eine Modularität gewährleistet, durch die jede Hilfsfunktion innerhalb einer Interfacefunktion verwendet werden kann.

Einige Hilfsfunktionen weisen hingegen eine extrem starke Kopplung auf. Insbesondere ist die Funktion *mk_type* hervorzuheben, welche die Typen der in einem Prädikat enthaltenen Elemente konstruiert. Bei Bedarf dekonstruiert sie den Datentyp von iterierbaren Elementen, wie etwa Listen oder Sets, um an die Typen der wiederum darin enthaltenen Elemente zu gelangen. Des Weiteren ist die Funktion indirekt rekursiv durch den Aufruf anderer Hilfsfunktionen, welche erneut *mk_type* selbst aufrufen. Die Kopplung zwischen SICStus Prolog und Z3 ist hierbei durch die enthaltenen Bedingungen im Kontrollfluss zusätzlich komplex. Sowohl die Verwendung von Funktionalitäten der Prolog Bibliothek sowie der Z3-Komponente verändern und bedingen den Verlauf der Methode. Zur Entkopplung wurde die Logik einer „State Machine“ implementiert, die in verschiedenen Szenarien zusätzliche Informationen an dem entsprechend anderen Prozess anfordern oder diese an ihn zu senden. Die Kommunikation macht sich dabei den Statusidentifikator $UNFINISHED$ zu Nutze, um den Prozess über den aktuellen Zustand zu informieren. Die Funktion wird in den meisten Interfacefunktionen aufgerufen und ist daher ein zentraler Bestandteil des Z3-Interfaces.

In Abbildung 2 ist das Sequenzdiagramm der Hilfsfunktion *mk_type* dargestellt, welches die Komplexität der Funktion verdeutlicht und veranschaulicht, wie die Entkopplung der Funktion durchgeführt wurde. Auf beiden Seiten des Diagramms sind die ProB- und Z3-Prozesse abgebildet, die schrittweise ihren Zustand ändern, welchen sie durch die Nachrichtenübermittlung synchron halten. Zunächst wird überprüft, ob der Typ ein Atom ist, wie *integer*, *real* oder *boolean*. Wenn dies nicht der Fall ist, wird überprüft, ob der Typ bereits zuvor konstruiert wurde. Sollte er nicht existieren, wird der Typ dekonstruiert. Die Dekonstruktion unterscheidet zwischen den Typen *set*, *global*, *couple* und *record*.

3.2.3 Optimierungen

Nach der erfolgreichen Portierung der Schnittstelle und der Implementierung aller notwendigen Funktionen lassen sich zusätzlich kleine Optimierungen und Verbesserungen vornehmen, um Effizienz, Lesbarkeit und Wartbarkeit des Codes zu erhöhen.

Eine der elementarsten Optimierungsmöglichkeiten ist die Vermeidung von Nachrichtenaustausch beider Prozesse innerhalb von Schleifen. Dieses Verhalten tritt insbesondere dann auf, wenn Prolog Datenstrukturen übermittelt werden, die iterierbar sind, wie Listen, Vektoren und Records. In diesen Fällen wird für jedes Element der Struktur eine Nachricht an den Server gesendet, um das aktuelle Element zu übermitteln. Dieses Verhalten kann in einzelnen Fällen durch die Übermittlung der gesamten Struktur in einer einzigen Nachricht vermieden werden. Beispielsweise lässt sich eine Prolog Liste zunächst in einem String Vektor akkumulieren, um anschließend in einer einelnen Nachricht an den Server gesendet zu werden.

Eine weitere Optimierungsmöglichkeit ist die Vermeidung von unnötigen Nachrichten an den Server. Beispielsweise illustriert Quellcode 10 vermeidbare Komplexität durch die Verwendung unnötiger Datenobjekte. Anhand der Variablen *translation_type_atom* wird das Objekt *ctx_data* ermittelt, welches an die Funktion *prolog_type_list_to_sort_vector*

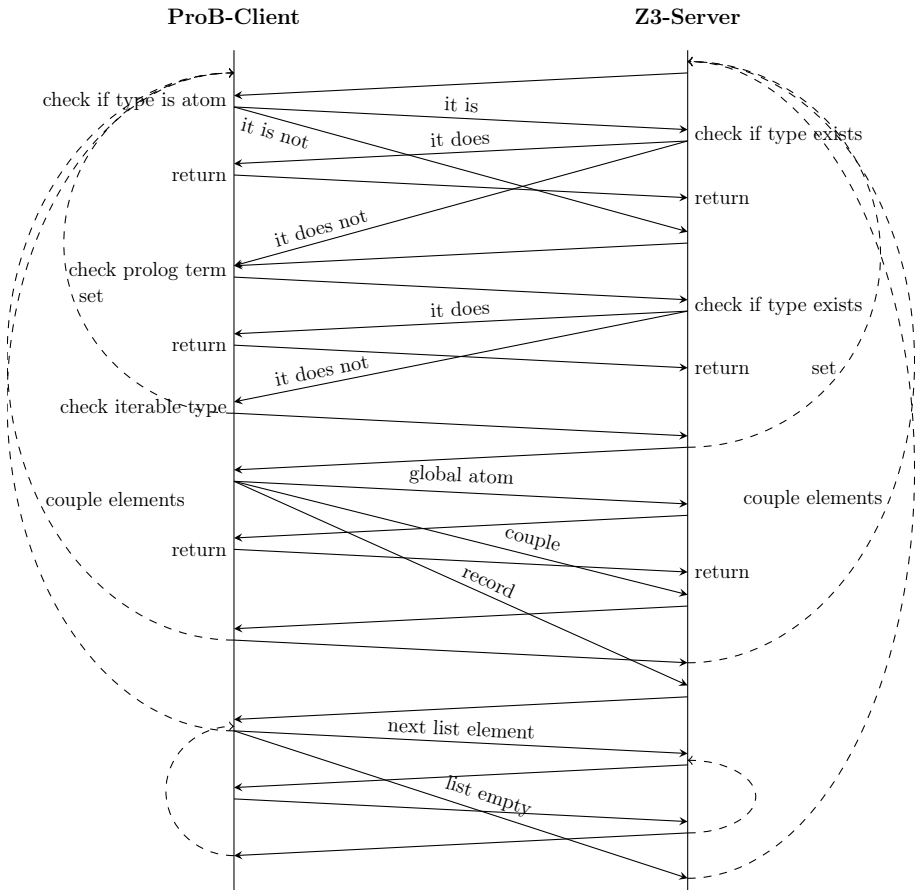

Abbildung 2: Das Sequenzdiagramm der Hilfsfunktion *mk_type*. Der Übersicht halber sind nicht alle Nachrichten beschriftet. Derartige Nachrichten übermitteln zumeist ausschließlich Statusindikatoren.

Quellcode 10: Illustrationsbeispiel zur Optimierung von vermeidbarer Objektkomplexität.

```
1:  // function: mk_op_comprehension_set_multi
2:  ContextData ctx_dta =
3:    get_translation_representant_ctx_data(translation_type_atom);
4:  prolog_type_list_to_sort_vector(ctx_data, couple_types);
5:  // ...
6:
7:  // function: prolog_type_list_to_sort_vector
8:  mk_sort(ctx_data->get_translation_type_atom());
```

übergeben wird. Diese verwendet das Objekt ausschließlich zur Ermittlung der Variablen *translation_type_atom*. Das Problem hierbei ist, dass *ctx_data* in der neuen Architektur auf der Seite des Server-Prozesses liegt und somit einen Nachrichtenaustausch erfordert. Die Problematik lässt sich umgehen, indem die Funktion *prolog_type_list_to_sort_vector* dahingehen refaktorisiert wird, direkt mit *translation_type_atom* aufgerufen zu werden oder sogar gänzlich darauf zu verzichten.

Zuletzt wurde das DRY[6]-Prinzip angewendet, um die Wartbarkeit des Codes zu erhöhen, indem gewisse Hilfsfunktionen ausschließlich auf entweder der Prolog- oder der Serverseite implementiert wurden. Der ursprüngliche Kontrollfluss verlangte zum Beispiel die Implementierung einzelner Hilfsfunktionen in beiden Prozessen. Durch die Anwendung des Programmierprinzips und minimaler Anpassung im Quellcode wurden diese Funktionen ausschließlich innerhalb eines Prozesses implementiert.

3.2.4 Serveranbindung

Der Z3-Prozess wird nicht manuell gestartet, sondern, um eine angenehme Nutzerfreundlichkeit zu gewährleisten, automatisch im Z3-Interface als Subprozess von ProB erzeugt. Insbesondere bedeutet das, dass der Z3-Server ausschließlich auf Anfrage von ProB gestartet wird, wenn das entsprechende Interface verwendet wird. Als Subprozess wird der Z3-Solver mit der Beendigung von ProB ebenfalls terminiert. Dieses Verhalten ist innerhalb der Prolog-Seite des Z3-Interfaces implementiert und wird durch das Prädikat *init_z3interface* realisiert, welches dahingehend erweitert wurde. Die Implementierung des Prädikats ist in Quellcode 11 gezeigt.

In den entsprechenden Zeilen 8 bis 10 wird der Endpoint zur ZeroMQ Kommunikation generiert und als Funktionsargument an die *init* Funktion übergeben, welche die Verbindung zum Z3-Server herstellt. Die Funktion *process_create* wird eingesetzt, um den Z3-Server als Subprozess zu starten, welcher durch die Verwendung der Systemargumente ebenfalls über den generierten Endpoint informiert wird. So sind beide Prozesse in Kenntnis desselben Endpoints und können miteinander kommunizieren. Der Endpoint selbst ist hierbei effektiv ein

[6] Don't Repeat Yourself

Quellcode 11: Das Prädikat zur Initialisierung des Z3-Interfaces.

```
1:  init_z3interface :- is_initialised(_), !.
2:  init_z3interface :-
3:      catch(load_foreign_resource(library(z3interface)),E,
4:            (format(user_error,'*** LOADING Z3 library failed~n',[]),
5:             assert(z3_init_exception(E)),
6:             fail)
7:            ),
8:      get_path_to_fresh_z3_endpoint(Endpoint),
9:      process_create('./lib/z3rver', [Endpoint], [stdout(null)]),
10:     init(Endpoint, Zocket),
11:     assertz(is_initialised(Zocket)).
```

Pfad zu einer temporären Datei, die als Kommunikationskanal zwischen den Prozessen dient. Er wird als einzigartiger Dateiname innerhalb von *get_path_to_fresh_z3_endpoint* generiert.

3.2.5 Logging

Der Z3-Server wurde mit einem Logging-System ausgestattet, um die potentielle Fehlersuche und -behebung zu erleichtern. Insbesondere hilft das Logging-System dabei, eine klare Übersicht über den Verlauf des Servers zu erhalten und damit die Nachvollziehbarkeit zu erhöhen. Durch das Protokollieren von Ereignissen, wie etwa dem Empfangen und Senden von Nachrichten, der aktuellen Funktion, die ausgeführt wird, sowie dem Auftreten von Fehlern, wird eine umfassende Dokumentation des Serververhaltens gewährleistet. Derartige Informationen erleichtern auch die zukünftige Wartung und Erweiterung des Systems. Innerhalb des Prolog Prädikates *process_create* lässt sich die Ausgabe des Z3-Servers allerdings nur schwer kontrollieren und verlangt die Verwendung der standard shell (sh). Aus diesem Grund wurde das Logging-System auf der Seite des Z3-Servers selbst implementiert, um gegebenenfalls die Ausgabe in einer Datei zu speichern. Die Implementierung funktioniert durch das Umleiten des Standardausgabestreams in einen eigenen Dateideskriptor. Dieser Dateideskriptor zeigt bei Verwendung des Logging-Systems in eine valide Datei im aktuellen Dateiverzeichnis. Sollte das Logging-System nicht verwendet werden, wird der Standardausgabestream auf „/dev/null“ umgeleitet, um die Ausgabe zu unterdrücken. Somit lässt sich über die Systemargumente beim Start des Servers entscheiden, ob das Logging-System aktiviert werden soll oder nicht. Zur Aktivierung muss lediglich die Zeile 9 in Quellcode 11 so angepasst werden, dass neben dem Endpoint zusätzlich das Argument „--log“ an den Server übergeben wird.

Da die Erweiterbarkeit des Systems gewährleistet werden soll, muss beachtet werden, dass der Z3-Server in eine Datei schreibt, die einzigartig zu seinem Prozess ist. Sollten in Zukunft mehrere Z3-Prozesse existieren, würden diese andernfalls in dieselbe Datei

schreiben und die Logausgabe dadurch unübersichtlich machen. Womöglich könnte dies auch zu Konflikten führen, wenn mehrere Prozesse gleichzeitig in dieselbe Datei schreiben, sodass Informationen vollständig verloren gehen oder nur ein einziger Z3-Solver die Datei beschreiben kann. Um dies zu verhindern, wird der Dateiname der Logdatei durch den Endpoint des Z3-Servers ergänzt. Beispielsweise schreibt der Z3-Solver bei einem Endpoint $ipc:///tmp/z3rver96485092.probz$ in die Datei $z3rver96485092.probz.log$.

3.2.6 Exceptions

Die korrekte Behandlung von Exceptions ist ein wichtiger Bestandteil der Implementierung, um die Stabilität des Systems zu gewährleisten. Sie ist jedoch grundlegend fehleranfällig und kann zu unerwarteten Abstürzen führen, wenn sie nicht ordnungsgemäß behandelt wird. Die Struktur der neuen Architektur fügt dem Behandlungsprozess von Exceptions eine zusätzliche Komplexität hinzu, da die ProB- und Z3-Prozesse unabhängig voneinander auf Fehler stoßen können. Beim Auftreten einer Exception in einem der Prozesse muss der andere Prozess über den Fehler informiert werden, um eine entsprechende Fehlerbehandlung durchzuführen. Hierbei ist zu beachten, dass das Kommunikationsmuster der beiden Prozesse eingehalten werden muss, wenn eine Exception auftritt. Dies ist nicht trivial, da Exceptions den natürlichen Kontrollfluss fundamental verändern.

Wenn eine Exception geworfen wird, verlässt der Kontrollfluss einer Anwendung grundsätzlich den aktuellen Codeblock und propagiert aufwärts im Funktionsaufruf-Stack (engl. „Call Stack“), bis ein passender Exception-Handler gefunden wird, der den Fehler behandelt oder das Programm terminiert. Zusätzliche Komplexität entsteht durch das Z3-Interface von SICStus Prolog auf der Seite des ProB-Prozesses, da die Schnittstelle einen durch $SP\_raise\_exception$ geworfenen Fehler als ausstehend (engl. „pending“) behandelt und den Kontrollfluss nicht wie gewöhnlich unterbricht. Stattdessen wird der Fehler also zwischengespeichert und erst bei der Rückkehr aus der Interfacefunktion nach Prolog behandelt.

In der originalen Architektur des Z3-Interfaces existiert keine klare Struktur zur Behandlung von Exceptions. Beispielsweise haben manche Hilfsfunktionen ihre eigenen Exception-Handler, die im Falle eines Fehlers einen nicht-validen Wert (wie etwa *nullptr*) zurückgeben. Dieser Wert wird in gewissen Interfacefunktionen nicht überprüft, was zu unerwarteten Abstürzen führen kann, wenn die entsprechende Exception auftritt. Manche Funktionen implementieren einen Exception-Handler für den gesamten Funktionskörper, andere hingegen nur für einzelne Codeblöcke, die teilweise nicht alle Fehlerfälle abdecken. Ebenfalls existieren Interfacefunktionen (wie etwa mk_sort oder mk_record_field), die keine Fehlerbehandlung implementieren und somit bei einem Fehler unerwartet (mit einem Segmentation Fault) terminieren.

Aus diesen Gründen wurde das Exception-Handling in der neuen Architektur des Z3-Interfaces grundlegend überarbeitet. Die Exception-Handler wurden konsistent in allen Interfacefunktionen implementiert, sodass der gesamte Funktionskörper abgedeckt ist. Auch wenn große Codeblöcke innerhalb eines Try-Catch-Blocks einen unschönen Codestyle

Quellcode 12: Der Exception-Handler des Z3-Prozesses.

```
1:  void raise_Z3_exception(const z3::exception& e, std::string name) {
2:    fprintf(logFile, "[ERROR] z3_exception in %s : %s\n",
3:     name.c_str(), e.msg());
4:    if (z3exc_occured) {
5:        return;
6:    }
7:    z3exc_occured = true;
8:    zmsg_t *response = zmsg_new();
9:    zmsg_addstr(response, e.msg());
10:   zmsg_addstr(response, name.c_str());
11:   send_z3_response(Z3Status::NOK, &response);
12: }
```

darstellen, ist so gewährleistet, dass nicht nur alle Fehler, sondern auch daraus resultierende Folgefehler abgefangen werden. Entsprechend wurden die Hilfsfunktionen so refaktorisiert, dass sie selbst keine Exceptions mehr abfangen, sondern diese an die aufrufende Funktion weiterreichen. Die aufrufende Funktion ist nach einer endlichen Propagierung zwangsläufig eine Interfacefunktion, welche den Fehler somit abfängt und entsprechend behandelt.

Um die Einhaltung des Request-Reply-Modells zu garantieren, wurde ein spezieller Exception-Handler in dem Z3-Prozess implementiert. Dieser wird immer aufgerufen, wenn eine Exception in einer der Interfacefunktionen auftritt und leitet den Fehler gegebenenfalls an den ProB-Prozess weiter. Die Implementierung des Exception-Handlers ist in Quellcode 12 gezeigt. Insbesondere ist auf die Variable *z3exc_occured* zu achten, die verhindert, dass mehrere Exceptions an den ProB-Prozess gesendet werden, bevor der erste Fehler behandelt wurde. Entsprechend wird auch in der Zeile 25 von Quellcode 6 die schlussendliche Antwort einer Anfrage nur dann gesendet, wenn keine Exception aufgetreten ist. Es ist nämlich davon auszugehen, dass der Fehler innerhalb des ProB-Prozesses in diesem Fall bereits behandelt wurde und die Interfacefunktion dort verlassen worden ist. Somit würde eine weitere Nachricht gegen das Kommunikationsmuster verstoßen und zu unerwarteten Fehlern führen.

Als grundlegende Konvention sollte der Z3-Server nach dem Senden einer Antwort und vor dem Erhalten der nächsten Anfrage keinen Code ausführen, der Fehler verursachen könnte. Andernfalls könnte der ProB-Prozess nicht über die Exception in Kenntnis gesetzt werden, da der Z3-Server in diesem Fall nicht mehr in der Lage wäre, eine Nachricht zu senden.

3.3 Kompilierung

Nach der vollständigen Umsetzung der neuen Systemarchitektur muss sowohl das Z3-Interface als auch der Z3-Solver kompiliert werden. Hierzu werden Änderungen im Makefile der Programmerweiterung von ProB vorgenommen.

Quellcode 13: Ausschnitt des Makefiles zur Kompilierung des Z3-Solvers.

```
include ../shared_libs.mk

$(target_dir)/z3rver: z3rver
  cp -f z3rver $@

z3rver: $(LIBCZMQ_PATH) $(target_dir)/$(LIBZ3)
        $(z3solver_files) $(z3solver_header)
  $(CXX) $(Z3CXXFLAGS) $(Z3LDFLAGS) $(z3solver_files)
   $(Z3LDLIBS) -o z3rver

install: $(target_dir)/z3rver
```

Quellcode 14: Der Kompilierungsbefehl.

```
g++ -std=gnu++11  -Wall -O2 -g -I../zmq_lib/prefix/include
 -I./shared -I./prefix/include -DZ3VERS="4.13.2"
 -L../zmq_lib/prefix/lib -L./prefix/bin -Wl,-rpath,'$ORIGIN'
 z3server/context_data.cpp z3server/z3rver.cpp -lczmq -pthread -lz3
 -o z3rver
cp -f z3rver ../../lib/z3rver
```

3.3.1 Makefile

Das eigentliche Z3-Interface wird als Bibliothek in Form einer dynamischen Shared Library kompiliert. Dieses Verhalten ändert sich nicht, es müssen lediglich minimale Anpassungen vorgenommen werden, wie das Entfernen der Z3- und das Hinzufügen der ZMQ-Bibliothek. Durch das Importieren der *shared_libs.mk* Datei in das Makefile wird die Kompilierung der Interface-Bibliothek mithilfe der *install* Regel automatisiert.

Es wird eine weitere Regel definiert, welche die Kompilierung des Z3-Solvers durchführt und die resultierende Binärdatei in das Zielverzeichnis kopiert. Diese Regel wird als Abhängigkeit von *install* hinzugefügt. Die Implementierung der Regel ist in Quellcode 13 gezeigt. Es ist zu beachten, dass die Abhängigkeiten $\$(LIBCZMQ\_PATH)$ und $\$(target\_dir)/\$(LIBZ3)$ sicherstellen, dass sowohl die ZeroMQ- als auch die Z3-Bibliothek vor der Kompilierung des Z3-Solvers vorhanden sind und entsprechend korrekt eingebunden werden können. Der resultierende Befehl, welcher durch das Makefile ausgeführt wird, ist in Quellcode 14 gezeigt.

Nach Abschluss der Implementierung und Kompilierung konnten alle vorhandenen Tests zur Überprüfung der Funktionalität des Z3-Interfaces erfolgreich durchgeführt werden. Dies bestätigt, dass die Architekturänderung erfolgreich umgesetzt wurde.

Quellcode 15: Problematische Endlosschleife.

```
1: // send_solver_interrupts("all"); // workaround
2: while (true) {
3:  { // wait for threads to finish
4:   std::lock_guard<std::mutex> threads_guard(running_threads_mutex);
5:   if (running_threads.size() == 0)
6:    break;
7:  }
8:  std::this_thread::sleep_for(std::chrono::milliseconds(5));
9: }
```

4 Zusätzliche Komplikationen

Im Verlauf der Arbeit traten einige Probleme auf, die nicht direkt mit der Entkopplung der Z3-Komponente zusammenhängen. Diese werden im Folgenden kurz erläutert.

4.1 Softlock

Bei der exzessiven Ausführung der relevanten Testkategorie $cat(smt\_solver\_integration)$. fiel während der Entwicklung auf, dass die Tests sporadisch nicht terminieren. Mithilfe eines simplen Skripts wurde die Testkategorie endlos ausgeführt, um das Problem zu reproduzieren und einen Einblick in dessen Häufigkeit zu erhalten. Am meisten war die Testnummer 2395 betroffen, welche sich auf der Entwicklungsumgebung etwa alle 18 Durchläufe nicht beendete. Nach einer Analyse des Problems stellte sich heraus, dass die Endlosschleife in Quellcode 15 die Ursache war, welche in der Methode *reset* ausgeführt wird. Die Schleife wartet auf das Beenden aller Threads, die zur Lösung eines Prädikates gestartet wurden.

Mithilfe des GNU Debuggers (GDB) [SPS+88] konnte das Problem dahingehend eingegrenzt werden, dass die laufenden Threads sich während der problematischen Endlosschleife innerhalb der Z3-Bibliothek befanden und dort mutmaßlich feststeckten. Genauer befanden sich die Threads in einer der beiden Methoden $sat :: local\_search :: flip\_walksat(unsignedint)()$ oder $sat :: local\_search :: pick\_flip\_walksat()()$. Die Ursache für das Problem konnte nicht abschließend geklärt werden, jedoch wurde ein (unschöner) Workaround implementiert, der das Problem behebt. Durch den in Zeile 1 als Kommentar dargestellten Aufruf zur Methode *send_solver_interrupts* werden alle laufenden Threads unterbrochen und somit die Endlosschleife verlassen. Da zum Zeitpunkt der Ausführung der *reset* Funktion mindestens ein Thread bereits ein Ergebnis berechnet hat, stellt das Unterbrechen der anderen Threads kein Problem dar.

4.2 Versionsinkompatibilität

Ein weiteres Problem, das während der Entwicklung auftrat, war die Inkompatibilität der Z3-Bibliothek (*z3lib.so*) mit dem Betriebssystem. Diese Problematik trat auf der automatischen Testumgebung auf, welche die Darwin plattform xyz innerhalb von Docker nutzt. Anhand der Fehlermeldungen konnte festegestellt werden, dass sich die Inkompatibilität auf die glibc-Version zurückführen lässt. Die Darwin-Plattform verwendete eine ältere Version von glibc, sodass die Z3-Bibiliothek auf inkorrekte oder nicht vorhandene Funktionen zugriff. Das Problem wurde gelöst, indem die Testumgebung auf die neuere Systemversion Darwin 12 aktualisiert wurde, welche mit der Z3-Bibliothek kompatibel ist.

5 Leistungsbewertung

Nach Abschluss der durchgeführten Architekturänderung ist es elementar, die Auswirkungen auf die Laufzeitperformance zu bewerten. Da sich die grundlegende Funktionsweise des Z3-Solvers in der neuen Architektur nicht geändert hat, ist es zu erwarten, dass die Laufzeitperformance der ProB-Systemerweiterung durch die Einführung der ZeroMQ-Kommunikation negativ beeinflusst wird. Zusätzlich zu den Aufrufen des Z3-Interfaces müssen zur Lösung eines einzelnen Prädikates nun mehrere Anfragen und Antworten über den Socket serialisiert werden. Diese zusätzliche Kommunikation führt zu einem Performance-Overhead, welcher quantifiziert und evaluiert werden muss. Ebenfalls besteht ein Interesse zum Vergleich verschiedener ZeroMQ-Protokolle und deren Auswirkungen auf die Performance. Hierbei ist zu erwarten, dass das Inter-Process-Communication (IPC) Protokoll schneller ist als das Transmission-Control-Protocol (TCP) Protokoll, da es auf dem gleichen Rechner arbeitet und keine Netzwerkkommunikation benötigt, sondern das Dateisystem verwendet. Im nachfolgenden Abschnitt werden die Methodik der Leistungsbewertung, die erzielten Ergebnisse und ihre Interpretation detailliert beschrieben.

5.1 Performance-Overhead

Um eine Bewertung des Performance-Overheads zu ermöglichen, müssen zunächst empirische Daten erhoben werden. Hierzu werden die Tests zur Verifikation der Funktionsweise des Z3-Interfaces umfunktioniert, um die Laufzeit der einzelnen Anfragen zu messen. Im Code des Z3-Interfaces wird ein Zeitstempel bei Beginn und Ende des Lösungsvorgangs eines Prädikates gesetzt, dessen Differenz berechnet und gespeichert. Insgesamt stehen 53 Tests zur Verfügung, die in der Testumgebung des Z3-Solvers ausgeführt werden können. Diese Tests umfassen insgesamt 679 Prädikate, welche eine ausreichende Grundgesamtheit zur Bewertung der Performance bieten. Ebenfalls wird die Anzahl der Anfragen und Antworten, die über das Netzwerk gesendet werden, gemessen. Ein kleiner Auszug dieser Messdaten sind in Tabelle 1 dargestellt.

Tabelle 1: Auszug der Daten der Performance-Messung.

TestID	QueryID	Old(ns)	New(IPC, ns)	New(TCP, ns)	Req. Count
⋮	⋮	⋮	⋮	⋮	⋮
1510	1	114 815 014	283 392 117	113 503 997	191
1510	2	52 048 678	59 273 375	44 489 012	166
1510	3	30 853 103	24 983 820	25 212 332	286
1511	1	69 240 686	199 325 313	61 823 314	21
1511	2	61 404 523	62 220 124	48 522 297	20
1513	1	80 829 324	202 694 845	75 877 790	25
⋮	⋮	⋮	⋮	⋮	⋮

Die Zeitmessungen sind in Nanosekunden (ns) angegeben und zeigen die Laufzeit der Anfragen in den verschiedenen Konfigurationen. Alle Messwerte liegen in einer Größenordnung von mindestens Millisekunden. Damit sind die zu erwartenden Messfehler in Bereich von Nanosekunden gering genug, um die Daten adäquat analysieren zu können. Innerhalb der eigentlichen Daten wurden die Messungen mehrfach unabhängig voneinander wiederholt, um eine statistische Aussagekraft zu gewährleisten. Da dennoch von Messunsicherheiten und Schwankungen auszugehen ist, werden die Daten in einem statistischen Kontext betrachtet. Es wird angenommen, dass sowohl die tatsächlichen Laufzeiten innerhalb eines Tests als auch deren Messunsicherheiten normalverteilt sind.

Um sich zunächst einen Überblick über die Rohdaten zu verschaffen, werden in Abbildung 3 die durchschnittlichen Laufzeiten in den verschiedenen Konfigurationen dargestellt. Der obere Subgraph zeigt die durchschnittlichen Laufzeiten aller Anfragen in sowohl der alten als auch der neuen Architektur, wobei die neue Architektur in das IPC-Protokoll und TCP-Protokoll unterteilt ist. Wider Erwarten zeigt sich, dass die durchschnittliche Laufzeit in der neuen Architektur bei vielen Prädikaten geringer ausfällt als in der alten Architektur. Insgesamt sind von den 679 Prädikaten nur 172 Prädikate in der alten Architektur schneller gelöst worden. 265 Prädikate wurden in der IPC-Konfiguration und 242 Prädikate in der TCP-Konfiguration am schnellsten gelöst.

Die unteren beiden Subgraphen stellen interessante Bereiche der Rohdaten erneut dar und zeigen zusätzlich die Standardfehler der Mittelwerte in Form der Fehlerbalken. Diese geben an, wie sehr die Mittelwerte der Laufzeiten von den tatsächlichen Mittelwerten der Grundgesamtheit abweichen. Kalkuliert wird dieser Standardfehler mittels der folgenden Formel:

$$\sigma_{\overline{x}} = \frac{\sigma}{\sqrt{n}} \tag{1}$$

Hierbei ist $\sigma_{\overline{x}}$ der Standardfehler des Mittelwertes, σ die Standardabweichung der Grundgesamtheit und n die Anzahl der Messungen. Somit werden ausschließlich statistische Fehler betrachtet und systematische Fehler nicht berücksichtigt.

So zeigt der untere linke Subgraph die Rohdaten im Bereich der Prädikate 60 bis 115

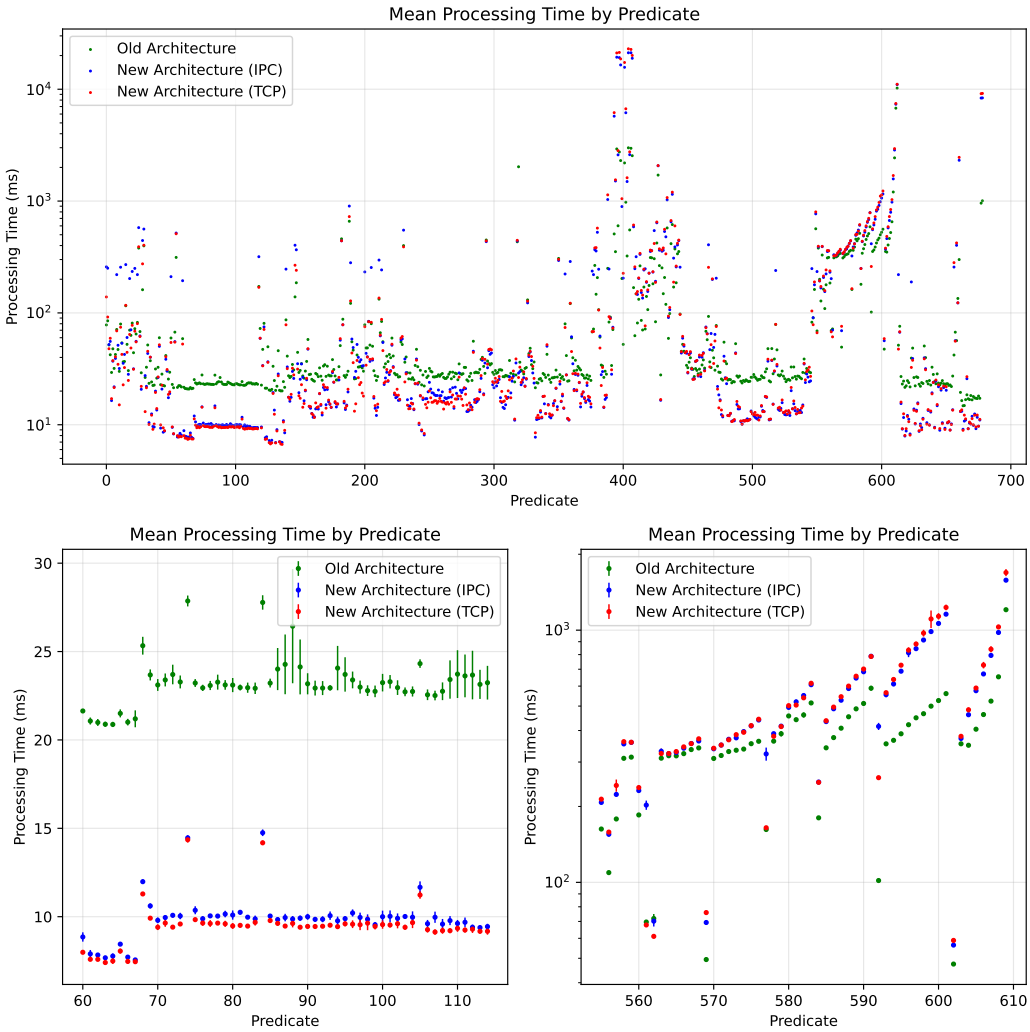

Abbildung 3: Durchschnittliche Laufzeiten der Anfragen in den verschiedenen Architekturen.

erneut, wobei klar zu erkennen ist, dass hier die neue Server-Architektur konstant schneller ist. Dieser Trend ist über die gesamte Messung hin unterhalb einer gewissen Grenze zu beobachten. Ebenfalls ersichtlich ist der nur minimal ausfallende Unterschied zwischen IPC und TCP.

Auf der anderen Seite zeigt der untere rechte Subgraph die Rohdaten im Bereich der Prädikate 555 bis 610. Hier haben die Laufzeit Messungen einen vergleichsweise hohen Wert und es ist zu erkennen, dass die alte Architektur in diesem Bereich schneller ist.

Entsprechend wurde durch die Protokollkommunikation ein Performance-Overhead eingeführt, der sich in den Rohdaten dahingehend widerspiegelt, dass Prädikate, welche bereits eine hohe Laufzeit aufgewiesen haben, nun in der neuen Architektur noch langsamer gelöst werden. Neben diesem Overhead wurde jedoch eine signifikante Laufzeitverbesserung erworben, die bis zu einer gewissen Gesamtlaufzeitsgrenze den Overhead nicht nur annulliert, sondern überkompensiert. Insgesamt scheint die neu eingeführte Server-Architektur hinsichtlich der Laufzeitperformance um einen gewissen Faktor effizienter geworden zu sein.

Ebenfalls ist zu erkennen, dass der Unterschied zwischen IPC und TCP nur minimal ist und keine signifikanten Unterschiede aufweist. Innerhalb der 679 Testprädikaten ist IPC in 365 (53.76%) Fällen schneller als TCP und entsprechend TCP in 314 (46.24%) Fällen schneller als IPC. Da zur TCP-Kommunikation die Adresse $tcp://127.0.0.1$ verwendet wurde, ist eine mögliche Erklärung hierfür, dass die Kommunikation über das Loopback-Interface einen Großteil des Netzwerk-Stacks umgeht und vom Betriebssystem speziell optimiert wird. In der weiteren Analyse wird daher nur das IPC-Protokoll betrachtet.

Um den Overhead der neuen Architektur zu quantifizieren, wird die Differenz der durchschnittlichen Laufzeiten der neuen ($\overline{t_{\text{IPC}}}$) und alten ($\overline{t_{\text{Old}}}$) Architektur berechnet und gegen die Anzahl der ZeroMQ-Anfragen analysiert. Der induzierte Overhead t_{induced} wird also wie folgt definiert:

$$t_{\text{induced}} = \overline{t_{\text{IPC}}} - \overline{t_{\text{Old}}} \tag{2}$$

Der oberste Subgraph in Abbildung 4 zeigt den induzierten Overhead der neuen Architektur in Abhängigkeit der Anzahl der ZeroMQ-Anfragen auf einer logarithmischen Skala. Im groben Verlauf ist zu erkennen, dass der Overhead mit steigender Anzahl der Anfragen annähernd linear zunimmt. Zusätzlich scheint jedoch eine ausgeprägte Systematik vorzuliegen, welche sich inhaltlich durch die horizontale Verzerrung der Daten ausprägt. Einige wenige Datenpunkte weichen stark von jeglicher Systematik ab und sind als Ausreißer zu klassifizieren.

Im zweiten Subgraphen wird die lineare Abhängigkeit des Overheads von der Anzahl der Anfragen untersucht, indem der Zusammenhang linear gefittet[7] wird. Dies geschieht unter Berücksichtigung des kombinierten Standardfehlers der Mittelwerte der alten und neuen IPC-Architektur. Die Unsicherheitsberechnung wird berechnet mit der gaußschen Fehlerfortpflanzung

$$\sigma_{\overline{comb}} = \sqrt{\sigma^2_{\overline{IPC}} + \sigma^2_{\overline{Old}}} \quad , \tag{3}$$

welche ebenfalls in den Fehlerbalken dargestellt wird. Das Ergebnis des linearen Fits zeigt eine Steigung von $0.03ms$ pro Anfrage, was bedeutet, dass der Overhead pro Anfrage um $0.03ms$ steigt. Zusätzlich gibt es einen y-Achsenabschnitt von $-13.13ms$, was bedeutet, dass der Overhead bei 0 Anfragen $-13.13ms$ beträgt. Dieser Wert beschreibt die zuvor erfasste Laufzeitverbesserung der neuen Architektur. Insgesamt ergibt sich die folgende lineare Funktion zur Beschreibung des Overheads:

$$Overhead = 0.03ms \cdot Anfragenanzahl - 13.13ms \tag{4}$$

Durch das Gleichsetzten der Overhead-Funktion mit 0 ergibt sich eine Anzahl an Anfragen von etwa 438, bis der Overhead nicht länger von der Laufzeitverbesserung kompensiert werden kann.

Eine separate Messreihe wurde durchgeführt, um die durchschnittliche Laufzeit einer einzelnen Anfrage an den Z3-Server zu ermitteln. Hierbei wurde die durchschnittliche Laufzeit von $0.0054ms$ bestimmt. Da für jede Anfrage zwei Nachrichten (Anfrage und Antwort) über den Socket gesendet werden, ergibt sich eine Laufzeit von $0.0216ms$ pro Anfrage unter der

[7] Der lineare Fit wurde mit dem von scipi.optimize bereitgestellten curve_fit berechnet.

Annahme, dass das Erhalten einer ZeroMQ-Nachricht die gleiche Laufzeit aufweist wie das Senden einer Nachricht. Die Differenz von $0.0084ms$ zur ermittelten Steigung des linearen Fits könnte beispielsweise auf die Größe der Nachrichten zurückzuführen, welche in der kontrollierten Messreihe einer einzelnen Anfrage kleiner ausfällt als in den 52 Testszenarien. Zudem sind Messwerte dieser Größenordnung zunehmend ungenau, weshalb sie annehmbar plausibel bezüglich der Steigung des linearen Fits sind.

Der lineare Fit zeigt augenscheinlich eine gute Übereinstimmung mit den Daten, welche in dem dritten Subgraphen aus Abbildung 4 verifiziert wird. Hierbei wird die Differenz des induzierten Overheads und des linearen Fits gegen die Anzahl der Anfragen dargestellt. Es ergibt sich ein Residuenplot. In diesem Plot ist die Systematik des obersten Subgraphen erneut zu erkennen. Sie bildet ein kleines Cluster oberhalb der 0-Linie, bei sehr kleinen Anfragenzahlen. Es stellt sich heraus, dass innerhalb dieses Datenclusters alle Datenpunkte die QueryID 1 aufweisen. Somit lässt sich die Abweichung durch das Initialisieren des Sockets, der Verbindung des Z3-Servers und dessen initialen Overheads zum Aufsetzten der Z3-Konfigurationen erklären. Innerhalb einer TestID wird der Z3-Server nur einmalig initialisiert und folgende Prädikate desselben Tests nutzen denselben Z3-Server, indem sie die Konfiguration zurücksetzten und nicht grundlegend neu initialisieren.

Zur Überprüfung stellt der letzte Subgraph erneut den Residuenplot dar, jedoch ohne diejenigen Datenpunkte mit QueryID 1. Hierbei ist zu erkennen, dass die Systematik des vorherigen Graphen verschwindet und der lineare Fit eine gute Übereinstimmung mit den Daten aufweist. Somit ist gezeigt, dass der Overhead der neuen Architektur durch das Serialisieren auf den Socket annähernd linear mit der Anzahl der Anfragen zunimmt und keine weiteren relevanten Systematiken besitzt. Eine Voraussage des Overheads ist mithilfe des linearen Fits möglich. Es ist zu beachten, dass die absoluten Werte des Overheads nur in der gegebenen Testumgebung gültig sind und nicht auf andere Umgebungen übertragen werden können.

Ein einzelner Datenpunkt scheint eine unwahrscheinlich hohe Laufzeitverbesserung von durchschnittlich 2 Sekunden aufzuweisen, weist jedoch hohe Messunsicherheiten auf. In Tabelle 2 ist dieser Datenpunkt dargestellt.

Tabelle 2: Ausschnitt der gesammelten Messwerte eines Ausreißers.

TestID-QueryID	$Old_1(ms)$	$Old_2(ms)$	$Old_3(ms)$	$Old_4(ms)$	**Req. Count**
2122-90	36	4033	3993	43	119

Die verschiedenen Messwerte der alten Architektur in Millisekunden zeigen zwei starke Ausreißer bei der zweiten und dritten Messung. Diese Varianz ist womöglich auf die in Abschnitt 4.1 beschriebene Problematik zurückzuführen. In jedem Fall ist dieser Datenpunkt als individueller Ausreißer zu betrachten und weist keine besondere statistische Relevanz auf.

Die ermittelte Performance-Verbesserung wird an dieser Stelle nicht weiter analysiert,

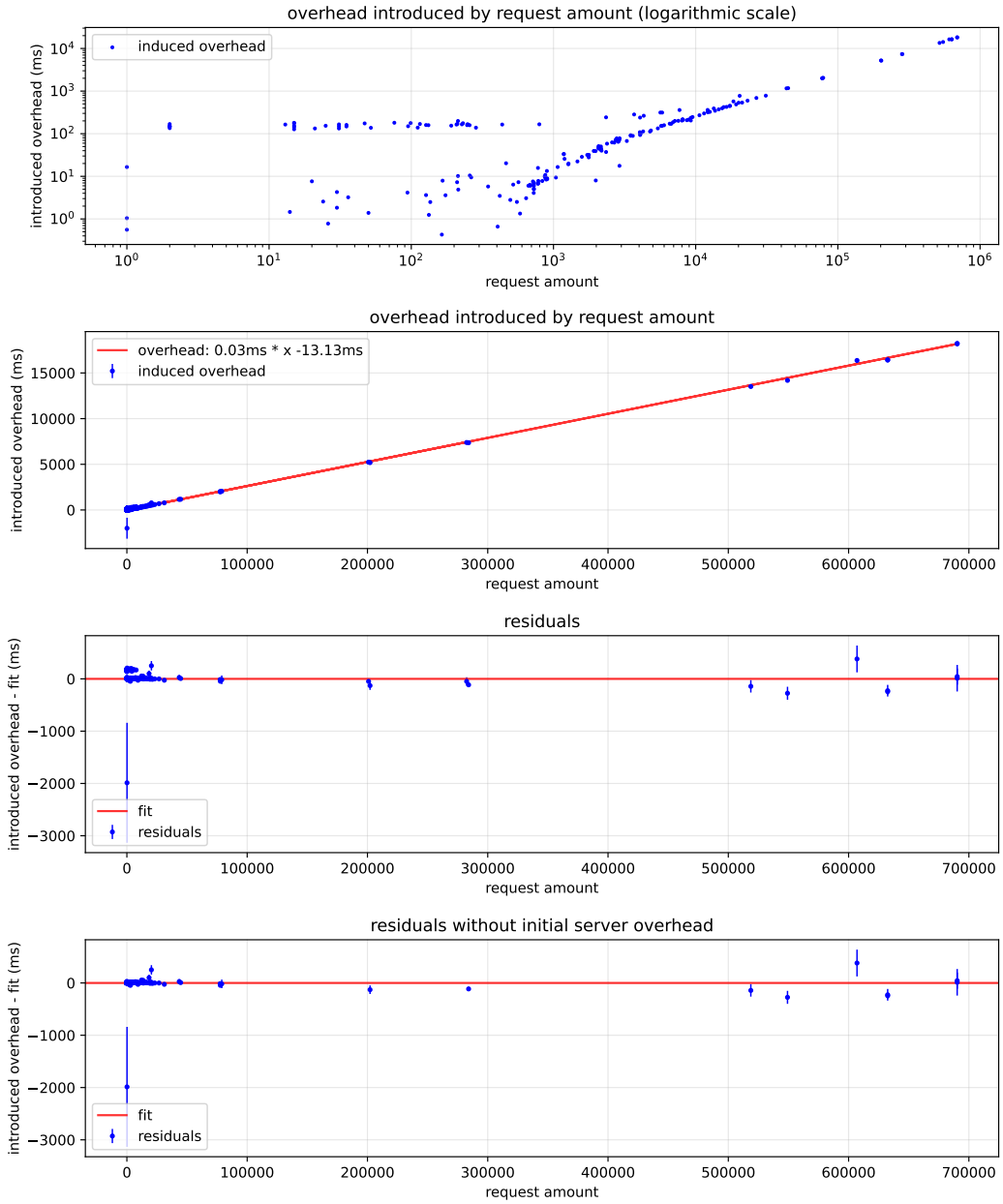

Abbildung 4: Induzierter Overhead der neuen Server-Architektur durch das Serialisieren auf den Socket.

da sie nicht Gegenstand dieser Arbeit ist. Ein möglicher Grund für die Verbesserung könnte die Kompilierung sein, die durch die neue Architektur ermöglicht wird, da der Z3-Solver als eigenständiger Prozess womöglich besser vom Compiler optimiert werden kann. Als eigenständiger Prozess ist es ebenfalls möglich, dass der runtime linker die Z3-Bibliothek optimierter laden und ausführen kann. Zusätzlich könnten die in Abschnitt 3.2.3 beschriebenen Optimierungen eine Rolle spielen.

6 Zukünftige Arbeiten

Es existieren verschiedene Möglichkeiten, um auf die in dieser Arbeit vorgestellte Implementierung aufzubauen und sie weiter zu verbessern. Folgende Abschnitte stellen einige dieser Möglichkeiten vor.

6.1 Weitere Analyse und Optimierungen

Die in Abschnitt 5 vorgestellte Performance-Evaluation ist eine erste Analyse der Implementierung und der Auswirkungen durch die Entkopplung von ProB und Z3. Sie stellt überraschenderweise eine Verbesserung der Performance fest, obwohl die Systemänderung an sich einen Overhead darstellt. Um ein tieferes Verständnis über die Laufzeit zu erlangen, ist es sinnvoll, die Analyse zu erweitern und zu vertiefen. Eine ausreichend tiefgründige Analyse könnte insofern von Vorteil sein, um ein besseres Verständnis über die Performanceverbesserung zu erlangen und somit gegebenenfalls gezielt weitere Optimierungen vorzunehmen.

6.2 Deinit Hook

Nach der erfolgreich abgeschlossenen Systemarchitekturänderung, die in dieser Arbeit vorgestellt wurde, bleibt ein letzter Schritt offen, um die Implementierung vollständig auszubauen. Derzeitig wird der Z3-Solver-Prozess nicht sauber beendet, sondern existiert als sogenannter *child process* von ProB und ist somit an die Existenz des ProB-Prozesses gekoppelt. Dies stellt insofern grundsätzlich kein großes Problem dar, da der Z3-Solver-Prozess nach Beendigung des ProB-Prozesses automatisch beendet wird. Dennoch besteht in der Theorie die Möglichkeit, dass der Z3-Server als sogenannter *dangling process* zurückbleibt. Grundsätzlich ist es zudem eine sauberere Lösung, den Z3-Prozess manuell zu beenden.

Des Weiteren bleibt bei Beendigung des Solvers der von ZeroMQ instanziierte Socket geöffnet und als *dangling socket* im Arbeitsspeicher zurück. Dieser Umstand kann zu Speicherlecks führen und sollte daher vermieden werden.

Um diese Probleme zu beheben, ist es sinnvoll, die Implementierung um einen Deinit-Hook zu erweitern. Dieser Hook wird aufgerufen, wenn der Z3-Solver beendet beziehungsweise

deinitialisiert wird, und dient dazu, den Z3-Solver-Prozess und den Z3-Socket sauber zu beenden und zu schließen.

6.3 Parallelisierung

Die Entkopplung von ProB und Z3 dient elementar als Grundlage für die Parallelisierung des Z3-Solvers beziehungsweise die Parallelisierung der Lösung mehrerer Prädikate. Wie in Abschnitt 1.2 beschrieben, ist die Entkopplung mit eben jenem Interesse der Erweiterbarkeit verrichtet worden.

Es bestehen verschiedene Ansätze, um die Parallelisierung zu realisieren. Eine Möglichkeit wäre, mehrere Instanzen des Z3-Solvers zu starten und die Prädikate auf diese Instanzen zu verteilen. In diesem Fall müsste die Kommunikationsschnittstelle auf der Seite von ProB so erweitert werden, dass sie mehrere Z3-Solver-Prozesse unterstützt. Der Z3-Server selbst müsste in diesem Szenario nicht verändert werden.

Eine andere Möglichkeit besteht darin, auch den Z3-Server zu erweitern, sodass dieser mehrere Prädikate gleichzeitig lösen kann. Hierbei müsste die Kommunikationsschnittstelle auf der Seite des Z3-Servers so erweitert werden, dass sie mehrere Prädikate gleichzeitig empfangen und lösen kann. Es würde sich anbieten, in diesem Fall das Kommunikationsmustern zu wechseln und auf ein asynchrones Modell umzusteigen, wie beispielsweise das Dealer-Router Modell, welches in Abschnitt 2.3 bereits erwähnt wurde.

7 Konklusion

hier wird konkludiert. alles supi mit rundem ende oder so...

stuff

Literatur

[Abr96] Abrial, Jean-Raymond: *The B-Book: Assigning Programs to Meanings*. New York, NY, USA : Cambridge University Press, 1996

[al.23] al., Mats C.: *SICStus Prolog User's Manual*. RISE Research Institutes of Sweden AB, December 2023

[CD96] Codognet, Philippe ; Diaz, Daniel: Compiling constraints in clp (FD). In: *The Journal of Logic Programming* 27 (1996), Nr. 3, S. 185–226

[CWA+88] Carlsson, Mats ; Widen, Johan ; Andersson, Johan ; Andersson, Stefan ; Boortz, Kent ; Nilsson, Hans ; Sjöland, Thomas: *SICStus Prolog User's Manual*. Kista, Sweden : Swedish Institute of Computer Science, 1988

[Hin13] Hintjens, P: *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013

[KL16] Krings, Sebastian ; Leuschel, Michael: SMT Solvers for Validation of B and Event-B Models. In: Ábrahám, Erika (Hrsg.) ; Huisman, Marieke (Hrsg.): *Integrated Formal Methods*. Cham : Springer International Publishing, 2016. – ISBN 978–3–319–33693–0, S. 361–375

[KL23] Körner, Philipp ; Leuschel, Michael: Towards Practical Partial Order Reduction for High-Level Formalisms. In: Lal, Akash (Hrsg.) ; Tonetta, Stefano (Hrsg.): *Verified Software. Theories, Tools and Experiments.* Cham : Springer International Publishing, 2023. – ISBN 978–3–031–25803–9, S. 72–91

[LB03] Leuschel, Michael ; Butler, Michael: ProB: A model checker for B. In: *FME 2003: Formal Methods: International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings* Springer, 2003, S. 855–874

[LB08] Leuschel, Michael ; Butler, Michael: ProB: an automated analysis toolset for the B method. In: *International Journal on Software Tools for Technology Transfer* 10 (2008), S. 185–203

[MB08] Moura, Leonardo de ; Bjørner, Nikolaj: Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R. (Hrsg.) ; Rehof, Jakob (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. – ISBN 978–3–540–78800–3, S. 337–340

[MR15] Microsoft-Research: *GitHub - Z3Prover/z3: The Z3 Theorem Prover*. 2015

[S+12] Sústrik, Martin u. a. ; Brown, Amy (Hrsg.) ; Wilson, Greg (Hrsg.): *The Architecture of Open Source Applications (Volume 2) ZeroMQ*. 2012. – 359–372 S.

[SPS+88] Stallman, Richard ; Pesch, Roland ; Shebs, Stan u. a.: Debugging with GDB. In: *Free Software Foundation* 675 (1988)

[TJ07] Torlak, Emina ; Jackson, Daniel: Kodkod: A relational model finder. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* Springer, 2007, S. 632–647