Robust Projectional Editing

Marcus Frenkel and Friedrich Steimann

FernUniversität in Hagen, Germany

Zusammenfassung Transaktionen stellen eine Möglichkeit dar, um in Code-Editoren eine dauerhafte semantische Korrektheit des Programmcodes zu gewährleisten, insbesondere, wenn sie durch Methoden wie die constraintbasierte Programmreparatur unterstützt werden. In ihrer bisherigen Erforschung sind sie allerdings lediglich für den lokalen Einsatz durch einen einzelnen Programmierer geeignet. Insbesondere beim kollaborativen Arbeiten auf einer gemeinsamen Codebasis entsteht das Problem, dass zwangsläufig Konflikte auftreten, wenn 2 oder mehr Programmierer die gleiche Stelle im Programmcode modifizieren wollen. Eine Lösung zum Beseitigen des Problems stellt die Verwendung des 2-Phase-Lockings dar, wobei die Herausforderung darin besteht, dieses vom (üblichen) Datenbank-Kontext auf die kollaborative Programmierung und das Setzen von Sperren auf Bestandteile des Programmcodes zu übertragen. In dem vorliegenden Kurzbericht diskutieren wir hierfür eine Lösungsmöglichkeit, welche insbesondere auf der Verwendung eines projizierenden Code-Editors aufbaut.

1 Ausgangslage

Projizierende Editoren (wie etwa MPS[1]) zeichnen sich dadurch aus, dass sie das Modell eines Programms zwar dem Programmierer in textueller Form präsentieren, intern jedoch alle Prozesse -- insbesondere auch das Hinzufügen, Verändern und Löschen von Programmelementen -- auf Basis des Programmmodells und entsprechend der Regeln des Metamodells der zugrundeliegenden Programmiersprache durchführen. Durch dieses Vorgehen ist zwar immer syntaktische Korrektheit gewährleistet, die Einhaltung semantischer Regeln kann hierdurch jedoch nicht erzwungen werden.

Eine Möglichkeit zum Gewährleisten semantischer Korrektheit in einem projizierenden Editor ist es, nur die Eingabe semantisch korrekter Werte beim Editieren einer Programmstelle zuzulassen. Eine Möglichkeit, dies sicherzustellen, ist über die Formulierung der semantischen Regeln mit Hilfe von formal notierten Invarianten (wie etwa in unseren Arbeiten zur constraintbasierten Programmreparatur[2] bereits erforscht wurde). Das Problem dieses Ansatzes ist, dass viele

[1] https://www.jetbrains.com/mps/

[2] Friedrich Steimann, Jörg Hagemann, and Bastian Ulke. 2016. Computing repair alternatives for malformed programs using constraint attribute grammars. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. 711–730.

gewünschte Modifikationen an Programmcode nicht nur auf eine einzelne Codestelle beschränkt sind, sondern Änderungen an mehreren Stellen erfordern. Beispiele hierfür sind das Ändern des Typs einer Variablendeklaration mit anschließender Anpassung von Feldzugriffen und Methodenaufrufen auf dieser Variablen, das Verschieben eines Feldes in einen anderen Typ mit entsprechenden Anpassungen der Zugriffe auf dieses Feld oder das Hinzufügen einer neuen Methode in einen Typ im Kontext eines neuen Methodenaufrufes. Einige dieser Änderungen lassen sich in atomare Einzeloperationen aufsplitten, etwa das Hinzufügen einer neuen Methode (mit leerem oder Standard-Rumpf) und das anschließende Einfügen eines Aufrufes dieser Methoden; viele Änderungen erfordern jedoch einen temporär inkonsistenten Zustandes des Programms.

Eine Lösung dieses Problems ist das Zusammenfassen mehrerer Einzeländerungen in einer Transaktion, die während ihrer Durchführung inkonsistente Programmzustände erlaubt und die erst abgeschlossen werden kann, wenn nach Änderungen ein konsistenter Programmzustand hergestellt wurde. In einer vorherigen Veröffentlichung (siehe Fußnote 2) haben wir zu diesem Zweck den Solution Space Explorer (*SSE*) entwickelt, welcher dem Programmierer ausgehend von einem zu ändernden Attribut eines Programmelementes (etwa dem Typ einer Variablendeklaration) weitere Attribute von Programmelementen präsentiert, die mit Blick auf die ursprünglich gewünschte Änderung und mit Ziel der Herstellung von Konsistenz angepasst werden müssen.

Jede der durch den SSE durchgeführten Transaktionen erfüllt in einem solchen Fall das ACID-Prinzip. Die *atomicity* wird durch den Transaktionscharakter selber und die *consistency* über die semantische Prüfung mittels Invarianten sichergestellt. Werden Änderungen nur lokal durch einen einzelnen Programmierer durchgeführt, ist die *isolation* der Transaktionen ebenfalls erfüllt; die *durability* ergibt sich implizit aus dem Speichern des geänderten Programmcodes.

2 Problemstellung und Lösung

Der Solution Space Explorer funktioniert, solange lediglich ein einzelner Programmierer den Code lokal modifizieren möchte. Anders sieht es aus, wenn kollaborativ programmiert werden soll, also mehrere Programmierer gleichzeitig auf der gleichen Codebasis arbeiten. Prinzipiell erlaubt der Solution Space Explorer diesen Ansatz, und zwar indem er nach jeder abgeschlossenen Transaktion die Codeänderungen per Broadcast an alle Clients verteilt.

Ein Problem ergibt sich jedoch in den Fällen, in denen mehrere Programmierer gleichzeitig die gleiche Codestelle bearbeiten möchten. In einem solchen Fall kann das Prinzip der *isolation* verletzt werden, wenn miteinander in Konflikt stehende Änderungen durchgeführt werden sollen.

Ein verbreiteter Mechanismus zum Gewährleisten von *isolation* gleichzeitig durchgeführter Transaktionen ist das 2-Phase-Locking. Bei diesem Ansatz werden in einer Transaktion auf zu lesenden und zu ändernden Daten Read- beziehungsweise Write-Locks gesetzt, um konfligierende Änderungen zu verhindern. Um Reihenfolgeabhängigkeiten zu vermeiden, erfolgt die Lock-Behandlung in 2

Phasen: In der ersten werden durch eine Transaktion Locks ausschließlich gesetzt und keine freigegeben; nach Durchführung aller benötigten Änderungen werden in der zweiten Phase alle Locks wieder freigegeben, ohne neue zu setzen.

Übertragen auf den Solution Space Explorer würde dieser Read-Locks auf alle Attribute von Programmelementen setzen, die dem Programmierer zur Auswahl präsentiert werden. Entscheidet er sich für einen bestimmten Wert eines Attributs, wird vor Ausführen der Änderung ein Write-Lock auf das entsprechende Attribut gesetzt. Beim Erweitern der Auswahl der präsentierten Attribute für eine umfangreichere Lösung müssen dementsprechend weitere Read-Locks gesetzt werden. Wurde das Programm derart in einen (neuen) konsistenten Zustand überführt, können die Transaktion bestätigt und im Zuge dessen alle gesammelten Locks freigegeben werden. Eine Propagierung der neuen Werte per Broadcast an alle Clients ist erst nach Abschluss der Transaktion notwendig, da durch die gesetzten Write-Locks kein anderer Client auf die entsprechenden Attribute zugreifen könnte. Projizierende Editoren wie MPS stellen eine ideale Grundlage für die Implementierung eines derartigen Verhaltens dar, da sie im Gegensatz zu Texteditoren intern auf einem Syntaxbaum arbeiten, dessen Knoten sich durch IDs eindeutig identifizieren und damit sperren lassen; textbasierte Editoren müssten dagegen etwa zeilenweise sperren, was weit ungenauer -- und bei neu eingefügten Zeilen deutlich schwieriger -- umzusetzen wäre.

Der beschriebene Ansatz weist zwei Probleme auf, die für das 2-Phase-Locking üblich sind und die behandelt werden müssen: Deadlocks und eine hohe Anzahl von Locks, die das gleichzeitige Durchführen mehrerer Transaktionen verhindern. Im Falle des SSE entstehen beide Probleme dadurch, dass durch die Auswahl von weiteren Attributen von Programmelementen, welche das ursprünglich zu ändernde Attribut einschränken, sehr schnell sehr viele Read-Locks gesetzt werden müssen, die dementsprechend eine Parallelisierung von Transaktionen unterschiedlicher Clients verhindern oder zu Deadlocks führen. Wichtig ist daher bei der Umsetzung des Locking-Ansatzes im SSE eine *gezielte* und *sparsame* Auswahl von präsentierten Elementen, um die Anzahl der Locks niedrig zu halten.