## Unser Lösungsansatz

**Nebenläufigkeit** Wir haben die Nebenläufigkeit immer nach demselben Prinzip implementiert: Für jede Funktionalität, die nebenläufig realisiert werden soll, gibt es eine eigene Klasse A. In dieser Klasse steht dann eine Unterklasse B, die das Interface Runnable implementiert. Zunächst muss ein Objekt der Klasse A erzeugt werden, auf welchem dann eine Methode ausgeführt wird. Durch jene Methode werden dann Instanzen von B (im Folgenden: Agenten) erstellt und an den ThreadPoolExecuter übergeben. Dieser hält eine feste Anzahl von Threads bereit, ordnet neue Agenten in eine Warteschlange ein und startet den vordersten Agenten, sobald ein Thread frei ist. Dann wird gewartet bis alle Agenten vollständig ausgeführt wurden und der ThreadPoolExecuter terminiert. So wird sichergestellt, dass der nächste Berechnungsschritt im Hauptprogramm erst ausgeführt wird, wenn der vorherige fertig ist. Die Kommunikation der Agenten läuft dann über von Java vorimplementierte threadsichere Klassen wie ConcurrentHashMap, die selbstimplementierte threadsichere Klasse Lts, oder es wird explizit gelockt.

**Algorithmus allgemein** Es wird zunächst ein LTS berechnet, welches dieselben Zustände und Aktionen hat wie das eingegebene LTS E,in dem alle schwachen Transitionen von E als starke Transitionen existieren. Dies wird auch schwaches LTS genannt. Auf das schwache LTS wird dann im zweiten Schritt die gröbste starke Bisimulation bestimmt. Dann werden bisimulative Zustände zusammengefasst und redundante Transitionen gelöscht. Zuletzt wird noch bei Bedarf eine reflexive $\tau$-Transition an den Startzustand hinzugefügt um ein LTS zu erhalten, das beobachtungskongruent zur Eingabe ist.

**Berechnung der schwachen Transitionen** Die Berechnung der schwachen Transitionen wird in der Klasse WeakLtsCalculator nebenläufig realisiert. Zunächst wird eine Instanz dieser Klasse erzeugt, wobei weakLts als Lts initialisiert wird, welches nur den initialen Knoten enthält. Darauf wird dann calculate ausgeführt. Zuerst werden die unsichtbaren (calculateInvisible) und dann die sichtbaren (calculateVisible) schwachen Transitionen berechnet, wobei beide Schritte jeweils nebenläufig implementiert sind. In calculateInvisible wird zunächst ein InternalReachabilityChecker konstruiert und berechnet. Dabei werden nebenläufig für jeden Zustand über Tiefensuche alle Zustände bestimmt, die ausschließlich über $\tau$-Transitionen erreichbar sind. Dies ist so implementiert, dass man nach Konstruktion für jeden Zustand in konstanter Zeit sowohl jene Zustandsmenge abrufen kann, als auch ebenfalls in konstanter Zeit für zwei Zustände ihre $\tau$-Erreichbarkeit überprüfen kann. Dann werden für jeden Zustand nebenläufig mithilfe des InternalReachabilityCheckers alle ausgehenden $\tau$-Transitionen in weakLts geschrieben. Bei der Bestimmung der sichtbaren Transitionen werden für jeden Zustand s nebenläufig alle schwachen sichtbaren Transitionen zu weakLts hinzugefügt, die sich durch eine starke sichtbare Transition ausgehend von s begründen lassen. Folgendes wird für jede Aktion a durchgeführt: Es werden mithilfe der bereits vorhandenen $\tau$-Transitionen alle Nachfolger Post(Post(s,a), $\tau$) in weakLts und alle Vorgänger Pre(s, $\tau$) bestimmt, miteinander und mit a zu einer Transition verknüpft, die zu weakLts hinzugefügt wird. Entfernung redundanter Transitionen

**Die Partitionierung** Um die Zustände in Äquivalenzklassen bezüglich starker Bisimilarität einzuteilen, richten wir uns im Wesentlichen nach dem vorgeschlagenen Algorithmus von Trainingsblatt 13-14. Für die nebenläufige Umsetzung, verwenden wir Pipelining, wobei eine Partitionierung in Form einzelner Äquivalenzklassen, die wir auch Blöcke nennen, die Pipeline durchläuft und in jedem Schritt vom jeweiligen Agenten weiter verfeinert wird, bis am Ende eine Partitionierung erreicht wird, die eine Bisimulation beschreibt.

Während der Ausführung wird die Pipeline dynamisch erweitert, indem sich neu gestartete Agenten hinten anhängen. Dafür wird der aktuelle Ausgabekanal der Pipeline immer über eine geteilte Variable referenziert, die durch ein Lock geschützt ist.

Die Agenten in der Pipeline werden jeweils mit einer Menge von Zuständen $C$ und einer Aktion $\alpha$ erstellt. Sobald ein Agent vom Executor gestartet wird, reiht er sich am Ende der Pipeline ein, indem er sich das Lock für die Referenz auf den Ausgabekanal der Pipeline nimmt, diesen Kanal als seinen eigenen Eingabekanal festlegt und die Referenz auf einen neu erstellten Kanal setzt, den er für seine Ausgabe verwenden wird. Dann wird das Lock wieder freigegeben. Um später schnelle Abfragen zu ermöglichen, wird $\mathrm{Pre}(C, \alpha)$ vorberechnet. Anschließend werden solange neue Blöcke aus dem Eingabekanal entnommen und verarbeitet, bis ein spezielles Marker-Objekt übergeben wird, das anzeigt, wann der Agent seine Arbeit erledigt hat und terminieren kann. Dieses Marker-Objekt wird einfach durch den Ausgabekanal weitergereicht. Die Verarbeitung eines regulären Blocks $B$ besteht darin, die enthaltenen Zustände in zwei disjunkte Teilblöcke $B \cap \mathrm{Pre}(C, \alpha)$ und $B \setminus \mathrm{Pre}(C, \alpha)$ einzuordnen. Falls keiner der Teilblöcke leer ist, wird für jedes Paar aus einem der zwei Teilblöcke und einer Aktion ein neuer Agent mit diesem Paar als Parameter erstellt und an den Executor übergeben. Um überflüssige Arbeit zu vermeiden, verwenden die Agenten eine gemeinsame ConcurrentHashMap, in der vermerkt wird, welche Blöcke schon weiter aufgespalten wurden. Falls ein Agent zu Beginn seine Ausführung feststellt, dass der eigene Block eingetragen ist, beendet er sich sofort, weil die Betrachtung der entstandenen Teilblöcke seine Aufgabe schon beinhaltet.

Um die oben beschriebene Architektur für die Berechnung der gröbsten Bisimulation zu benutzen, wird nur ein Block in den anfänglichen Ausgabekanal der Pipeline gelegt, der alle Zustände des LTS enthält. Zusätzlich wird noch das Marker-Objekt nachgereicht. Dann wird für jede Aktion ein Agent erstellt, der die Menge der Zustände und diese Aktion als Parameter erhält, und dem Executor übergeben. Nebenbei wird mitgezählt, wie viele Agenten aktiv sind oder auf ihre Ausführung warten. Sobald die Anzahl auf 0 fällt, können die Mengen aus dem Ausgabekanal der Pipeline zurückgegeben werden. Sie stellen die Äquivalenzklassen der Partitionierung dar.

**Entfernung redundanter Transitionen** Zuletzt müssen noch redundante Transitionen entfernt werden. Es wird zunächst das alte LTS bis auf die Transitionen geklont. Dann werden nebenläufig alle Transitionen, die nicht redundant sind hinzugefügt, wobei Redundanz nach den folgenden Kriterien überprüft wird:

$$\mathrm{Transition}(s, \tau, t) \text{ redundant} \Leftrightarrow \exists s \neq r \neq t.\ s \xrightarrow{\tau} r \xrightarrow{\tau} t$$

$$\mathrm{Transition}(s, a, t) \text{ redundant} \Leftrightarrow (\exists s' \neq s.\ s \xrightarrow{\tau} s' \xrightarrow{a} t) \vee (\exists t' \neq t.\ s \xrightarrow{a} t' \xrightarrow{\tau} t)$$

Diese Kriterien ergeben sich aus der Definition von schwachen Transitionen und der Tatsache dass das LTS nach verschmelzen von Zuständen immer noch alle schwachen Transitionen auch als starke Transitionen enthält.

und der Tatsache, dass das übergebene LTS jede schwache Transition auch als starke Transition enthält, her.