

Erste Begegnung mit Alloy 6

Burkhardt Renz

4. April 2022

Inhaltsverzeichnis

1 Der Echo-Algorithmus	3
2 Die statische Struktur: Graphen	6
3 Zustandsübergänge: die Schritte des Echo-Algorithmus	15
4 Verifikation mit Alloy 6	30
5 Visualisierung der Transitionsprädikate	38
6 Variante der Spezifikation	41
7 Szenarien	44
8 Laufzeitmessungen	48
Literatur	51

Meine erste Begegnung mit *Alloy*, Sprache und Werkzeug zur formalen Spezifikation von Softwareartefakten liegt eigentlich schon länger zurück¹. Strukturelle Gegebenheiten, wie sie in Softwareartefakten vorkommen, können zum Beispiel Datenstrukturen sein, wie Graphen, Bäume, Listen u.ä., oder Objektstrukturen, wie sie etwa in der Kollaboration von Objekten in Entwurfsmustern vorkommen, oder Beziehungen von Entitäten, wie sie in der Entity/Relationship-Modellierung von Datenbankschemata vorkommen. Auch kombinatorische Beziehungen von Objekten in Spielen oder Rätseln wie Sudoku oder dem Damenproblem lassen sich mit Alloy spezifizieren. Das Herausragende an Alloy ist dabei die interaktive Art und Weise des Spezifizierens im *Alloy Analyzer*. Der Alloy Analyzer gibt unmittelbar Rückmeldung durch Diagramme, an denen man erkennen kann, ob man wirklich das Gewünschte beschrieben hat. Dabei wird einem oft klar, dass man etwas schlampig Spezialfälle vergessen hat oder bestimmte Eigenschaften einfach unterstellt hat, die jedoch explizit spezifiziert werden müssen.

Software beschreibt natürlich nicht nur Strukturen, sondern auch ihre Veränderung über die Zeit. Ganz abstrakt kann man sich ein laufendes Softwaresystem vorstellen als ein System, das zu jedem Zeitpunkt einen bestimmten Zustand hat und dann ausgelöst durch ein Ereignis im nächsten Zeitpunkt in einen anderen Zustand wechselt. Diese *Dynamik* kann man mit Alloy auch ausdrücken², jedoch sieht Alloy keine „eingebauten“ Konstrukte vor, wie dies zu tun ist. Deshalb hat die Spezifikation der Dynamik eines Systems in Alloy bisher spezielle Idiome und Disziplinen beim Spezifizieren erfordert.

Mit Alloy 6³ ist das anders. Deshalb ist die Begegnung mit Alloy 6 etwas ganz Neues: In Alloy 6 kann man schon in der Sprache unterscheiden zwischen Objekten, die *statisch* sind, sich also im Laufe der Zeit nicht ändern und solchen, die *dynamisch* sind, deren Struktur sich also von Zeitpunkt zu Zeitpunkt unterscheiden kann. Der Alloy Analyzer zeigt dann nicht nur möglichen statischen Strukturen an, die der Spezifikation entsprechen, sondern auch mögliche zeitliche Verläufe, die sich aus ihr ergeben. Alloy 6 setzt dabei ein Konzept um, das Leslie Lamport in seiner Spezifikationssprache TLA⁺ eingeführt hat: Die Ausführung eines Softwaresystems besteht in einer Folge von Zuständen. Dabei ist jeder Zustand definiert durch die Zuweisung von Werten zu den Variablen des

¹ siehe Burkhardt Renz und Nils Asmussen: *Kurze Einführung in Alloy* [RA09], basierend auf dem Buch *Software Abstractions* von Daniel Jackson [Jac12].

² siehe Nils Asmussen: *Ansätze zur Modellierung von Dynamik in Alloy* [Asm10].

³ Alloy 6 wurde als Erweiterung von Alloy zunächst unter dem Namen *Electrum* entwickelt durch Julien Brunel, David Chemouil, Alcino Cunha und Nuno Macedo [Bru+18], siehe auch <http://haslab.github.io/Electrum/>. Electrum führt in die Sprache von Alloy Operatoren der linearen temporalen Logik LTL ein. Außerdem kann man in Electrum auch die Model Checker NuSMV oder nuXmv für die Verifikation verwenden. Im Frühjahr 2021 wurde Electrum in die Codebasis von Alloy übernommen und bildet so die aktuelle Version von Alloy: Alloy 6, veröffentlicht am 14.11.2021.

Systems. Die Spezifikation eines Systems ist also nichts anderes als die Beschreibung aller möglicher erwünschter Ausführungsabläufe.⁴ Dieses Konzept aufgreifend ist Alloy 6 ein Werkzeug zum *interaktiven Spezifizieren und Verifizieren von Softwareartefakten*. Sein herausragendes Merkmal bleibt die Eleganz der Sprache Alloy und das interaktive (für Hype empfängliche Menschen würden sagen: agile) Arbeiten mit dem Alloy Analyzer.

In dieser ersten Begegnung wollen wir Alloy 6 am Beispiel eines verteilten Algorithmus kennenlernen. Der *Echo-Algorithmus*, den wir spezifizieren und analysieren wollen, konstruiert in einem ungerichteten zusammenhängenden Graphen einen aufspannenden Baum. Zunächst werden wir den Echo-Algorithmus selbst kennenlernen und an einem Beispiel durchspielen. Dann setzen wir Alloy 6 ein, zunächst um die statische Struktur zu beschreiben, eben ungerichtete zusammenhängende Graphen. Danach kümmern wir uns um die Dynamik, wie sich jeder Knoten des Graphen beim Echo-Algorithmus von Zustand zu Zustand verhalten muss, also welche Bedingungen (*constraints*) ein Folgezustand zu einem gegebenen Zustand erfüllen muss. Und schließlich verifizieren wir in Alloy 6, dass unser Algorithmus in der Tat einen Spannbaum des Graphen ergibt. Let's get started.

1 Der Echo-Algorithmus

Der Echo-Algorithmus ist ein Algorithmus, dessen Ziel es ist durch die Kooperation der Knoten in einem ungerichteten zusammenhängenden Graphen einen aufspannenden Baum zu konstruieren.

Für den Algorithmus wird ein Knoten des Graphen als *Initiator* ausgezeichnet, die anderen Knoten sind die *Teilnehmer*. Die Beteiligten verhalten sich folgendermaßen, siehe etwa [Fok18, Chap. 4.3]:

- Der Initiator sendet eine Nachricht mit seiner eigenen Identifizierung *Id* an alle seine Nachbarn.
- Jeder Teilnehmer nimmt eine Nachricht aus seinem Posteingang und wenn er noch kein Elternteil markiert hat, wird die *Id* in dieser Nachricht sein Elternteil. Danach sendet er seine eigene *Id* an alle anderen seiner Nachbarn.

⁴“...we represent the execution of a system as a sequence of states. Formally, we define a *behavior* to be a sequence of states, where a state is an assignment of values to variables. We specify a system by specifying a set of possible behaviors—the ones representing a correct execution of the system.” Leslie Lamport in *Specifying Systems*, [Lam02, S. 15].

- Ein Teilnehmer notiert weitere Nachrichten, die er in seinem Posteingang erhält. Dies macht er solange, bis er von allen seinen Nachbarn eine Nachricht verarbeitet hat. Ist dies der Fall, sendet er eine Nachricht mit seiner Id an sein Elternteil.
- Wenn schließlich der Initiator ein Echo von allen seinen Nachbarn erhalten hat, bildet die im Verlauf konstruierte Beziehung via Elternteil einen aufspannenden Baum des Graphen mit dem Initiator als Wurzel.

Wir spielen den Algorithmus an dem Beispiel des Graphen in Abb. 1 durch.

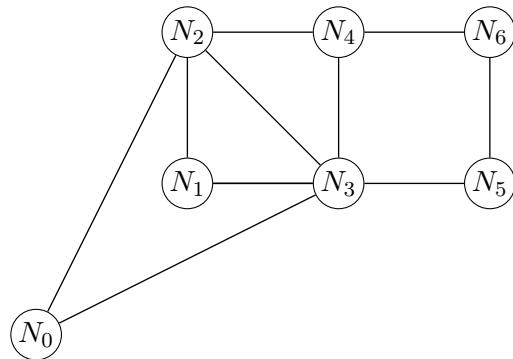


Abbildung 1: Beispielgraph für den Echo-Agorithmus

Wir gehen davon aus, dass der Knoten N_2 der Initiator ist. Dann gibt es viele Möglichkeiten, in welcher Reihenfolge die Nachrichten versandt werden. Nach dem ersten Schritt durch den Initiator, könnte sich etwa folgendes Szenario entwickeln:

1. Der Initiator N_2 sendet an seine Nachbarn N_0, N_1, N_3, N_4 seine Id.
2. Einer der Empfänger, sage N_0 markiert N_2 als sein Elternteil und sendet seine Id an seinen anderen Nachbarn N_3 .
3. N_4 markiert ebenfalls N_2 als sein Elternteil und sendet seine Id an N_3 und N_6 .
4. N_3 nimmt N_4 aus seinem Posteingang und markiert ihn als sein Elternteil. Dann sendet N_3 seinen anderen Nachbarn N_0, N_1, N_2, N_5 seine Id.
5. N_1 nimmt N_3 aus seinem Posteingang und markiert N_3 als sein Elternteil und sendet seinem anderen Nachbarn N_2 seine Id.
6. N_6 markiert N_4 als sein Elternteil und sendet seine Id an N_5 .
7. N_5 wählt N_3 aus seinem Posteingang als Elternteil und sendet seine Id an N_6 .

8. N_5 hat nun Nachricht von allen seinen Nachbarn erhalten und sendet seine Id an sein Elternteil N_3 .
9. Bei N_6 ist dies auch so und deshalb bekommt N_4 die Id von N_6 zugesandt.
10. N_1 sendet das Echo an N_3 .
11. N_3 sendet das Echo an N_4 .
12. N_4 hat Nachricht von allen seinen Nachbarn erhalten und sendet sein Echo an N_2 .
13. N_0 sendet das Echo an N_2 . Nun hat N_2 Nachricht von allen seinen Nachbarn erhalten. Der Spannbaum ist konstruiert.

Abb. 2 zeigt fett hervorgehoben den im geschilderten Szenario gebildeten Spannbaum des Graphen.

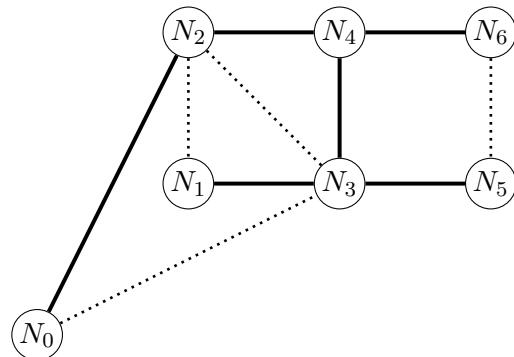


Abbildung 2: Spannbaum im Beispielgraph für den Echo-Agorithmus

*

Im folgenden Abschnitt wollen wir zunächst die statische Struktur für den Echo-Algorithmus in Alloy 6 spezifizieren, also ungerichtete zusammenhängende Graphen.

Ich empfehle die folgenden Abschnitte direkt in Alloy 6 nachzuvollziehen. Die Github-Seite von Alloy ist <https://github.com/AlloyTools/org.alloytools.alloy>, dort findet man auch die aktuelle Version von Alloy 6.

Bei der Installation von Alloy 6 werden gleichzeitig SAT-Solver bereitgestellt, die man für die Überprüfung von Spezifikationen verwenden kann. Sie können allerdings nur Ausführungspfade bis zu einer vorgegebenen endlichen Zahl von Schritten generieren.

Möchte man die Model Checker NuSVM und nuXMV für beliebig viele Schritte, also die unbeschränkte Verifikation von Spezifikationen verwenden, muss man diese Programme selbst installieren und den Pfad zur Installation setzen. Auf einem Mac tut man das zum Beispiel so:

1. Installation von NuSMV mit `brew install nu-smv`.
2. Installation von nuXmv von der Webseite <https://nuxmv.fbk.eu>.
3. Damit in der App der Pfad auch tatsächlich verwendet wird, muss man folgendes Kommando ausführen:

```
sudo launchctl config user
path /usr/local/Cellar/nu-smv/2.6.0/bin:<path to bin of nuXmv>
```
4. Danach muss man den Mac neu starten und auch Alloy 6 neu aufrufen. Dann erscheinen im Menü unter `Options>Solver` die zusätzlichen Auswahlmöglichkeiten `Electrod/NuSMV` und `Electro/nuXmv`.

Die Quellen für die Spezifikationen in dieser Ausarbeitung sind auf meiner Webseite unter <https://esb-dev.github.io/mat/echo.zip> zu finden.

2 Die statische Struktur: Graphen

Graphen haben Knoten, die durch Kanten verbunden sind. Das ist in Alloy 6 leicht auszudrücken:

```
sig Node{
    neighbors: set Node
}
```

Mit `sig`, einer *Signatur*, definiert man die Typen in unserer Spezifikation⁵. Die Knoten haben eine binäre Relation `neighbors` zu anderen Knoten, die als *Feld* der Signatur definiert werden. Damit ist `neighbors` eine Teilmenge des kartesischen Produkts der Menge `Node` mit sich selbst, welches man als `Node -> Node` in Alloy 6 schreibt. Bei der Definition des Feldes kann man eine *Multiplizität* angeben, in unserem Fall `set`, wodurch festgelegt wird, dass ein Knoten unseres Graphen mehrere benachbarte Knoten haben kann. Gibt man keine Multiplizität explizit an, ist sie `one`.

⁵ Die Bezeichnung *Signatur* kommt aus der Prädikatenlogik, siehe Burkhardt Renz: *Logik und formale Methoden* Vorlesungsskript [Ren21], Kap. 12 *Die formale Sprache der Prädikatenlogik*.

Mit dieser ersten Definition können wir direkt mit dem interaktiven Inspizieren der dadurch festgelegten möglichen „Welten“ beginnen, indem wir im Alloy Analyzer das Kommando `run` ausführen:

```
run {}
```



Abbildung 3: Resultat des Kommandos `run` mit dem ersten Ansatz zur Spezifikation von Graphen

Der Alloy Analyzer erzeugt nun mehrere Beispiele von „Welten“, die der Spezifikation entsprechen. Diese Beispiele sind in der Sprache der formalen Logik *Modelle*⁶, man könnte auch von *Instanzen* sprechen, wie dies Brunel et al. in ihrem Online-Buch [Bru+21] tun⁷. Das heißt, dass wir Alloy 6 als *model finder* einsetzen, der zu einer gegebenen Spezifikation Modelle erstellt, die ihr genügen.

⁶ Diese Bezeichnung stammt von der Sprechweise in der formalen Logik, siehe [Ren21, Kap. 1.2 *Mathematische Logik*].

⁷ “When presenting the semantics of a logic, *instances* of formulas are usually known as *models*: a model of a formula is a valuation of its free variables that makes the formula true. In Alloy the term *model* is frequently used to denote the specification of a system, and the term *instance* is used to denote a satisfying valuation. This use of the term *model* in Alloy parlance is consistent with other techniques and languages for software design, namely the Unified Modeling Language, where a software model is

Nach dem Ausführen (Schaltfläche `Execute`) wird im rechten Teil des Fensters im Alloy Analyzers das Ergebnis der Ausführung des Kommandos `run` angezeigt, wie in Abb. 3 dargestellt: Ein oder mehrere Modelle (`Instance`) wurden gefunden.

Durch einen Klick auf `Instance` öffnet sich das Fenster für die Visualisierung der gefundenen Modelle und wir sehen Abb. 4.

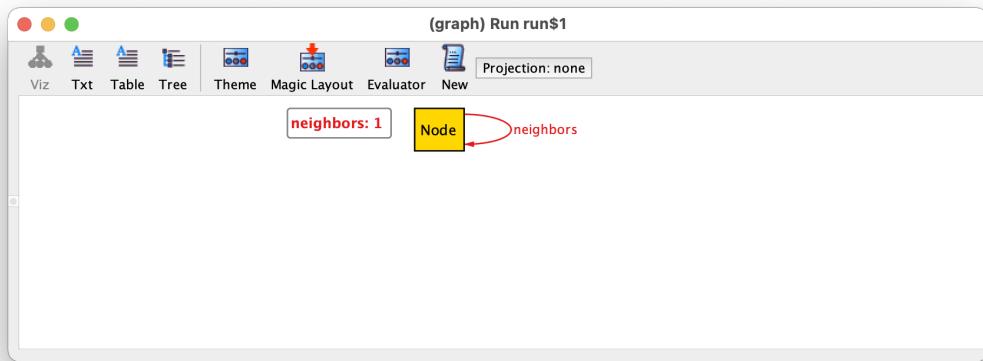


Abbildung 4: Erstes Modell unserer ersten Spezifikation von Graphen

In Abb. 4 wurde ein Modell der Spezifikation erzeugt mit einem Knoten und einer Kante, die auf den Knoten selbst zeigt. Es wird auch angezeigt, dass die Relation `neighbors` ein Tupel enthält.

Man kann nun auf die Schaltfläche `New` drücken und erhält jeweils ein weiteres Modell, das der Spezifikation genügt. In Abb. 5 werden einige der gefundenen Modelle angezeigt. Alloy 6 erzeugt Modelle mit einer endlichen Anzahl von Elementen je spezifizierter Signatur, per Default ist diese Anzahl auf 3 begrenzt.

Dies ist eine grundlegende Eigenschaft von Alloy. Das Erfüllbarkeitsproblem der Prädikatenlogik, also die Frage ob es zu einer Formel der Prädikatenlogik ein Modell gibt (und wenn ja welches) ist im Allgemeinen nicht lösbar; aber schon, wenn man sich auf endliche Universen von Objekten beschränkt. Und genau dies tut Alloy. Gleichwohl kann Alloy mit großem Nutzen in der Analyse von Softwareartefakten eingesetzt werden. Dies liegt daran, dass nicht gewünschte Konstellationen oder Fehler oft schon bei

precisely the kind of artefact we have been denoting by a specification.”

Ich werde der Tradition der formalen Logik entsprechend von *Spezifikation* und *Modell* sprechen.

einer kleinen Menge von Objekten auftreten. Daniel Jackson nennt dies die *small scope hypothesis*: “Most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or small instance makes no difference. So if the analysis considers all small instances, most flaws will be revealed. This observation, which I call the *small scope hypothesis*, is the fundamental premise that underlies Alloy’s analysis.” [Jac12, S. 15]

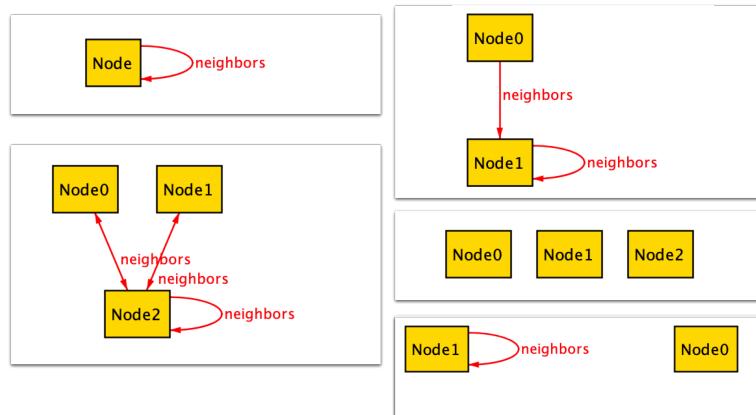


Abbildung 5: Einige Modelle unserer ersten Spezifikation von Graphen

An den Modellen, die der Alloy Analyzer erzeugt hat, kann man erkennen, dass Graphen vorkommen, die nicht zusammenhängend sind, und solche, die gerichtet sind, d.h. die Relation `neighbors` nur in einer Richtung gilt. Außerdem kann es vorkommen, dass eine Knoten eine Kante auf sich selbst hat, was der Definition eines ungerichteten Graphen widersprechen würde. Unsere Spezifikation ist also unterspezifiziert: wir müssen Integritätsbedingungen angeben, die diese unerwünschten Möglichkeiten ausschließen. Gesagt, getan: wir ergänzen unsere Spezifikation um folgende Zeilen:

```
fact irreflexive {
    all n: Node | n not in n.neighbors
}
```

Mit einem `fact` legen wir eine Eigenschaft fest, die für alle erzeugten Modelle gelten soll. Die Eigenschaft `irreflexive` sagt aus, dass es keine Kante eines Knotens auf sich selbst geben darf.

Bei der Formulierung dieser Eigenschaft wird der Operator `.`, der *dot join operator* verwendet. Bei diesem Operator handelt es sich um die *Komposition*, d.h. zwei Tupel

werden kombiniert, wenn die letzte Komponente des ersten mit der ersten Komponente des zweiten übereinstimmt und dann wird diese Komponente weggelassen. In unserem Beispiel `n.neighbors` hat das erste Tupel `n` nur eine Komponente, während das zweite Tupel `neighbors` zwei Komponenten hat, nämlich einen Knoten und alle seinen Nachbarn. Bildet man nun `n.neighbors`, dann erhält man gerade alle Nachbarn des Knotens `n`. Wenn nun der Knoten selbst nicht in der Menge seiner Nachbarn enthalten ist, gibt es keine Kante auf ihn selbst.

The screenshot shows the Alloy Analyzer interface with the following details:

- Title Bar:** /Users/br/ssb/fnd/lsv-In/examples/echo1.als
- Toolbar:** New, Open, Reload, Save, Execute, Show
- Model Specification (Left Panel):**

```

sig Node {
    neighbors: set Node
}

fact irreflexive {
    all n: Node | n not in n.neighbors
}

fact undirected {
    neighbors = ~neighbors
}

fact connected {
    all n1, n2: Node | n2 in n1.*neighbors
}
run {} for exactly 4 Node

```
- Execution Log (Right Panel):**
 - Warning:** Alloy4 defaults to SAT4J since i
For faster performance, go to Options menu
If these native solvers fail on your comp
 - Executing "Run run\$1"**
Solver=sat4j Bitwidth=4 MaxSeq=4 Skole
87 vars. 12 primary vars. 126 clauses.
Instance found. **Predicate** is consistent
 - Executing "Run run\$1 for exactly 4 Node"**
Solver=sat4j Bitwidth=4 MaxSeq=4 Skole
268 vars. 16 primary vars. 409 clauses.
Instance found. **Predicate** is consistent
- Status Bar:** Line 16, Column 21 [modified]

Abbildung 6: Spezifikation von ungerichteten zusammenhängenden Graphen

Wir benötigen noch zwei weitere Eigenschaften:

```

fact undirected {
    neighbors = ~neighbors
}
fact connected {
    all n1, n2: Node | n2 in n1.*neighbors
}

```

Der Operator `~` bezeichnet zu einer binären Relation die *transponierte Relation*, ein

Tupel $b \rightarrow a$ ist genau dann in der Relation \sim_{rel} , wenn $a \rightarrow b$ in der Relation rel enthalten ist. Das Fakt `undirected` legt also fest, dass wenn zwischen zwei Knoten eine Kante da ist, die Kante in der umgekehrten Richtung auch vorhanden ist.

Der Operator $*$ ist der *reflexive transitive Abschluss* einer binären Relation. Der Ausdruck `n1.*neighbors` ergibt also alle Knoten, die auf einem Pfad beginnend beim Knoten `n1` erreicht werden können sowie den Knoten `n1` selbst. Wenn nun von jedem Knoten aus alle anderen erreicht werden können, dann ist der Graph zusammenhängend.

Wir können nun überprüfen, welche Graphen wir mit dieser Spezifikation erhalten. In Abb. 6 wird die zweite Spezifikation gezeigt, diesmal für ungerichtete zusammenhängende Graphen. Das Kommando `run for exactly 4 Node`, sorgt dafür, dass wir Modelle mit exakt vier Knoten erhalten.

Es werden dann insgesamt 6 verschiedene Modelle angezeigt, was genau die 6 möglichen ungerichteten zusammenhängenden Graphen mit 4 Knoten modulo Isomorphie sind, wie sie in Abb. 7 dargestellt sind.

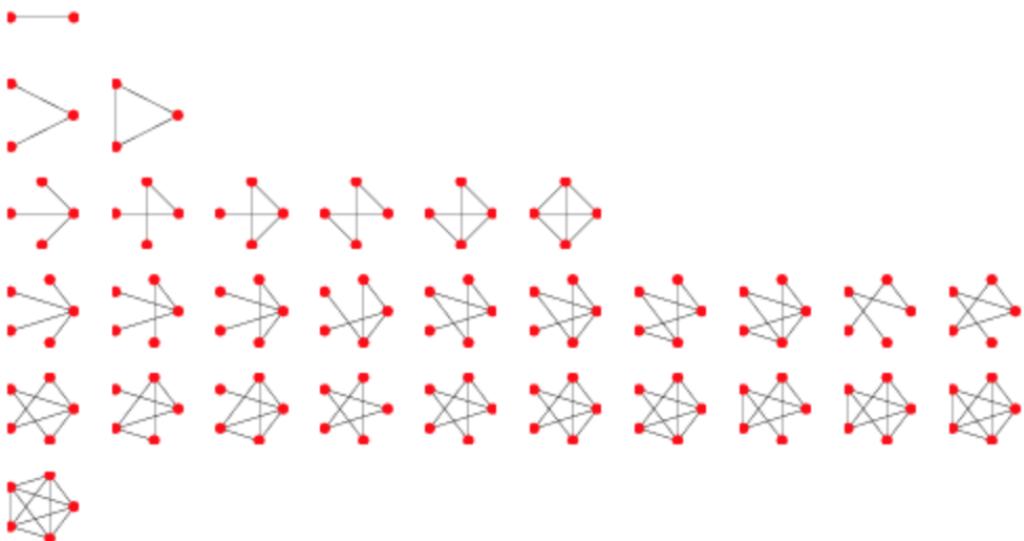


Abbildung 7: Zusammenhängende Graphen mit bis zu 5 Knoten (Quelle: <https://mathworld.wolfram.com/ConnectedGraph.html>)

Die Anzeige weiterer möglicher Modelle endet mit der Meldung „There are no more satisfying instances.“, siehe Abb. 8.

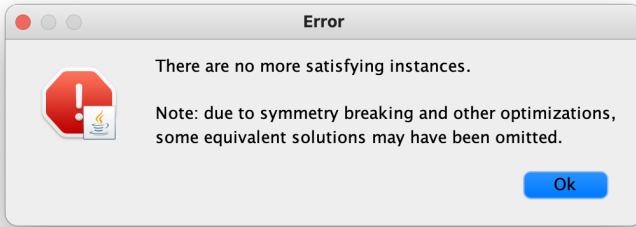


Abbildung 8: Meldung, dass es keine weiteren Modelle mehr gibt

Die Meldung hat einen Hinweis auf das *symmetry breaking*, was darin besteht, dass weitere Modelle, die sich nur durch die Benennung der einzelnen Knoten unterschieden hätten nicht angezeigt werden. Die Atome eines Modells haben keine spezielle Bedeutung, die Bezeichnungen sind also nur zur Unterscheidung untereinander da, ansonsten aber bedeutungslos. Das bedeutet, dass zwei Modelle, die sich nur durch die Namen der Atome unterscheiden tatsächlich identisch sind⁸. Der Alloy Analyzer verwendet intern einen Algorithmus, der dies erkennt: dieser Mechanismus wird *symmetry breaking* genannt.

Bisher haben wir ungerichtete zusammenhängende Graphen spezifiziert. Beim Echo-Algorithmus unterscheidet man jedoch zwischen einem speziellen Knoten, dem Initiator, und den restlichen Knoten, den Teilnehmern. Dies wollen wir nun in unserer Spezifikation auch tun. Dazu erweitern wir die Signature `Node`:

```
abstract sig Node{
    neighbors: set Node
}

one sig INode extends Node{}
sig PNode extends Node{}
```

Mit dem Schlüsselwort `extends` definieren wir Teilmengen einer Signatur, und zwar so dass sie paarweise disjunkt sind, wenn mehrere Teilmengen spezifiziert werden. Unsere

⁸ In unserem Fall der zusammenhängenden Graphen erkennt der Alloy Analyzer tatsächlich die Isomorphie der Graphen bis zu 4 Knoten. Bei 5 Knoten ist das nicht mehr der Fall. Abb. 7 zeigt, dass es 21 nicht-isomorphe zusammenhängende Graphen mit 5 Knoten gibt. Der Alloy Analyzer erzeugt aber 30 Modelle mit 5 Knoten.

beiden Teilmengen sind `INode`, wobei das „I“ für *initiator* steht, und `PNode`, wobei das „P“ für *participant* steht. Das Schlüsselwort `one` legt fest, dass es genau ein Element in der Teilmenge der `INode` geben soll, eben unseren Initiator. Nun könnte es sein, dass es außerdem noch Knoten in `Node` gibt, die weder in der Teilmenge `INode` noch in der Teilmenge `PNode` enthalten sind. Um dies auszuschließen legen wir fest, dass die Signatur `Node` `abstract` ist. Auf diese Weise haben wir eine Partition von `Node` in die disjunkten Teilmengen `INode` und `PNode` spezifiziert⁹.

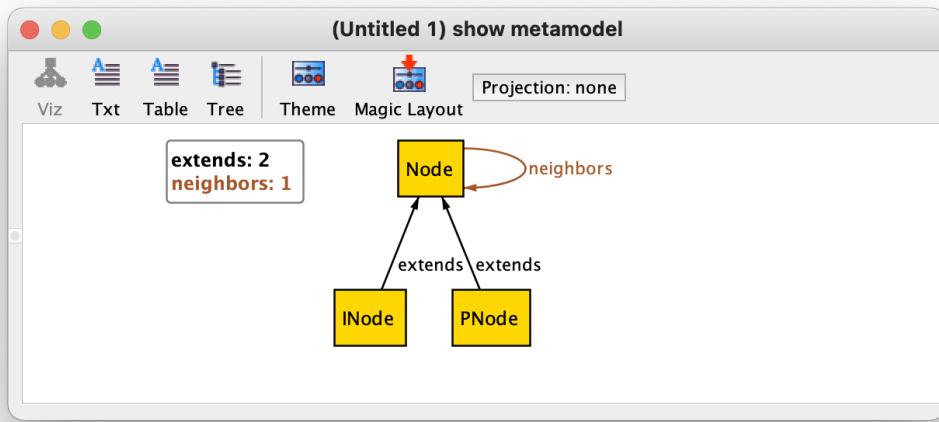


Abbildung 9: Das Metamodell der Spezifikation der Partitionierung von `Node`

Der Menüpunkt `Execute>Show Metamodel` zeigt unsere Spezifikation als Diagramm, siehe Abb. 9.

Wenn man nun auf Basis dieser Spezifikation Modelle mit 3 Knoten erzeugt, dann erhält man nicht nur die beiden nicht-isomorphen Graphen mit 3 Knoten (siehe Abb. 8), sondern weitere Modelle, denn jetzt spielt die Position des Initiators im Graphen eine Rolle. Man erhält die in Abb. 10 abgebildeten Modelle.

Für die Analyse des Echo-Algorithmus spielt die Position des Initiators tatsächlich eine Rolle, denn wir wollen uns ja sicherlich davon überzeugen, dass der Echo-Algorithmus

⁹ In der Sprechweise der *Unified Modeling Language* (UML) hat eine Spezialisierung dieser Art die Eigenschaft `{complete, disjoint}`.

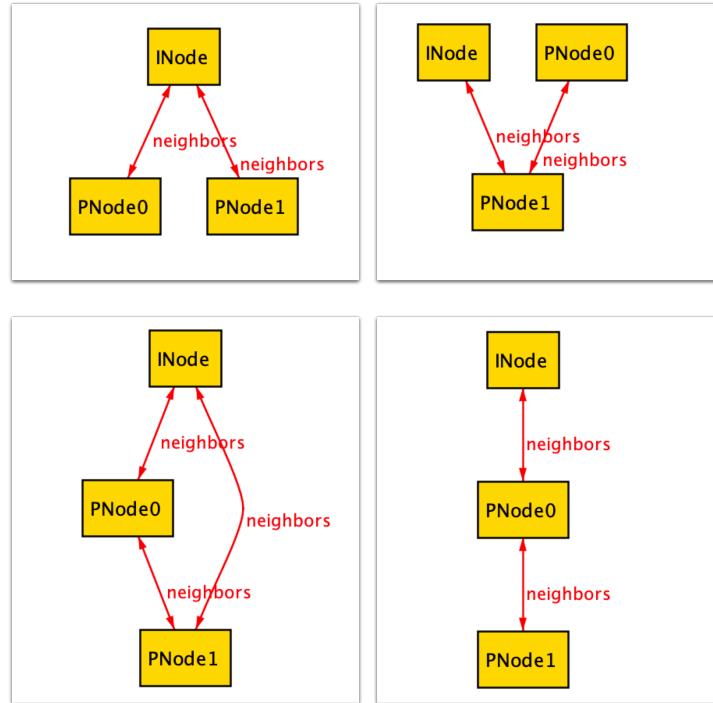


Abbildung 10: Die Modelle mit 3 Knoten

korrekt ist, und zwar unabhängig davon, welcher Knoten in einem zusammenhängenden Graphen die Rolle des Initiators spielt.

Wir können noch eine weitere Veränderung an unserer Spezifikation vornehmen. Alloy kennt das Konzept von *Modulen*, die es erlauben, Spezifikationen von bestimmten Eigenschaften wiederzuverwenden. So gibt es ein Modul `graph`, das wir in unsere Spezifikation einbinden können.

Im Menü des Alloy Analyzers kann man über den Menüpunkt `Open Sample Models...` im Verzeichnis `models/util` die in Alloy 6 bei der Installation vorhandenen Module laden und ihren Code ansehen. Im Modul `util/graph` wird man sehen, dass es das Modul `util/relation` verwendet und dort die Definitionen für den ungerichteten und irreflexiven Graphen genau so stehen, wie wir sie oben selbst definiert haben. Was den Zusammenhang des Graphen angeht, können wir aus `util/graph` das Prädikat `rootedAt` verwenden, weil wir ja jetzt einen ausgezeichneten Knoten, nämlich

`INode` haben. Und der Graph ist natürlich zusammenhängend, wenn jeder Knoten im reflexiven transitiven Abschluss von `INode` enthalten ist.

Das Modul `graph` ist parametrisiert, es erwartet, dass wir die Signatur der Knoten des Graphen angeben. Das Modul enthält unter Anderem verschiedene Prädikate, deren Parameter die Relation der Knoten-Signatur sein muss, die die Kanten des Graphen definiert.

Wenn wir dieses Modul verwenden, dann ergibt sich folgende Spezifikation:

```
open util/graph[Node]

abstract sig Node{
    neighbors: set Node
}

one sig INode extends Node{}

sig PNode extends Node{}

fact {
    noSelfLoops[neighbors]
    undirected[neighbors]
    rootedAt[neighbors, INode]
}
```

Damit haben wir die Struktur der Graphen festgelegt, auf denen wir den Echo-Algorithmus anwenden wollen. Die nächste Aufgabe besteht nun darin zu spezifizieren, wie die Zustandsübergänge bei der Ausführung der einzelnen Schritte des Algorithmus aussehen.

3 Zustandsübergänge: die Schritte des Echo-Algorithmus

Bisher haben wir die statische Datenstruktur spezifiziert: der Graph bleibt unverändert während der Echo-Algorithmus abläuft. In diesem Abschnitt werden wir nun die Datenstruktur erweitern durch Komponenten, die während der Ausführung des Algorithmus sich von Zustand zu Zustand ändern können. Dies war in Alloy schon immer möglich, hat aber spezielle Muster der Spezifikation erfordert. Die entscheidene Erweiterung in Alloy 6 besteht darin, dass Veränderungen der Struktur zwischen Zuständen besonders

einfach und elegant festgelegt werden können. Dazu gibt es in Alloy 6 das Schlüsselwort `var`, das angibt, dass eine Signatur oder ein Feld einer Signatur pro Zustand unterschiedlich ausgeprägt sein kann.

Die Bezeichnung `var` kommt natürlich von *variable*. Man darf sich aber nicht eine Variable vorstellen, wie man sie aus imperativen Programmiersprachen kennt, wo einer Variable Werte zugewiesen werden. Tatsächlich besteht das Konzept darin, dass wir beispielweise zu einem Feld `var inbox: set Node` gewissermaßen zwei Felder definiert haben, nämlich `inbox` und `inbox'`, wobei das eine in einem Zustand definiert wird und das mit *prime* gekennzeichnete die Gegebenheiten im Folgezustand angibt.

Zunächst müssen wir überlegen, welche „variablen“ Strukturen wir für den Echo-Algorithmus benötigen. Dies kann man sicherlich auf unterschiedliche Weise tun, eine Möglichkeit ist die Erweiterung der Signatur `Node` auf folgende Weise:

```
enum Color{ Red, Green }

abstract sig Node{
    neighbors: set Node,
    var parent: lone Node,
    var inbox: set Node,
    var color: Color
}
```

Die Idee für die Spezifikation des Algorithmus ist dabei:

1. Die Relation `parent` soll die sukzessive aufgebaute Baumstruktur repräsentieren. Das bedeutet, dass wenn ein Knoten die allererste Nachricht bearbeitet, trägt er den entsprechenden Knoten in dieses Feld ein. Die Multiplizität `lone` bedeutet, dass ein Knoten kein oder ein Elternteil haben kann.
2. Die `inbox` dient dazu, die erhaltenen Nachrichten aufzunehmen. Wenn also der Knoten `PNode1` zum Beispiel seine Id an `PNode2` sendet, ist zu spezifizieren:

`PNode2.inbox' = PNode2.inbox + PNode1,`

was bedeutet, dass die `inbox'` des Folgezustands die Vereinigung ihres bisherigen Inhalts mit `{PNode1}` ist.

3. Die `color` verwenden wir, um zu notieren, dass ein Knoten „fertig“ ist und sein Echo erzeugt hat. Am Anfang sind die Knoten als `Red` markiert, haben sie ihr Echo gegeben, werden sie `Green`. Für die Definition der Farben wurde das Konzept

der Enumeration von Alloy eingesetzt: ein `enum` erzeugt die angegebenen *singles* sowie eine Ordnung derselben durch die implizite Verwendung des Moduls `util/ordering`.

Nach dieser Erweiterung können wir uns Modelle der Spezifikation erzeugen lassen und stoßen auf eine neue Ansicht in der Visualisierung wie in Abb. 1 beispielhaft dargestellt.

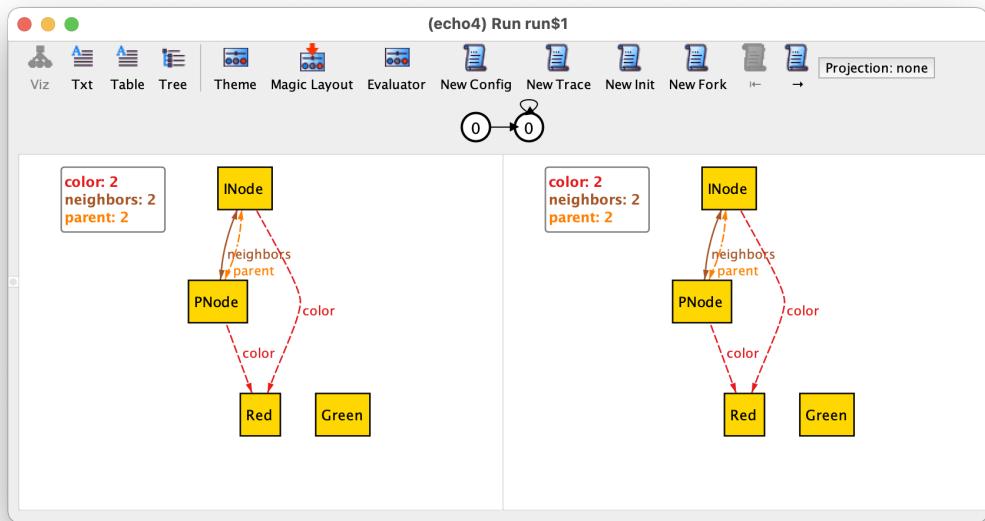


Abbildung 11: Visualisierung ohne Integritätsbedingungen für `var s`

Zunächst fällt auf, dass wir zwei Modelle gezeigt bekommen. Da wir `var s` spezifiziert haben, werden zwei Zustände angezeigt, sowie über den beiden Zuständen der Ausführungs pfad. In unserem Beispiel ist er nicht sehr spektakulär: Wir sehen zweimal denselben Zustand mit der Nummer `0`, sowie die Schleife. Der Analyzer hat also einen Ausführungs pfad erzeugt, bei dem sich immerzu der erste Zustand wiederholt.

Außerdem ist die Darstellung recht unübersichtlich geworden, weil wir jetzt eine ganze Menge von Relationen haben, die dargestellt werden. Um die Darstellung übersichtlicher zu machen, kann man in Alloy Einstellungen treffen, das Thema (*Theme*) festlegen. Dazu wählt man die Schaltfläche `Theme`. Jetzt kann man die Art der Darstellung verändern.

In Abb. 12 kann man sehen, dass die Farbe eines Knotens nicht als Kante dargestellt

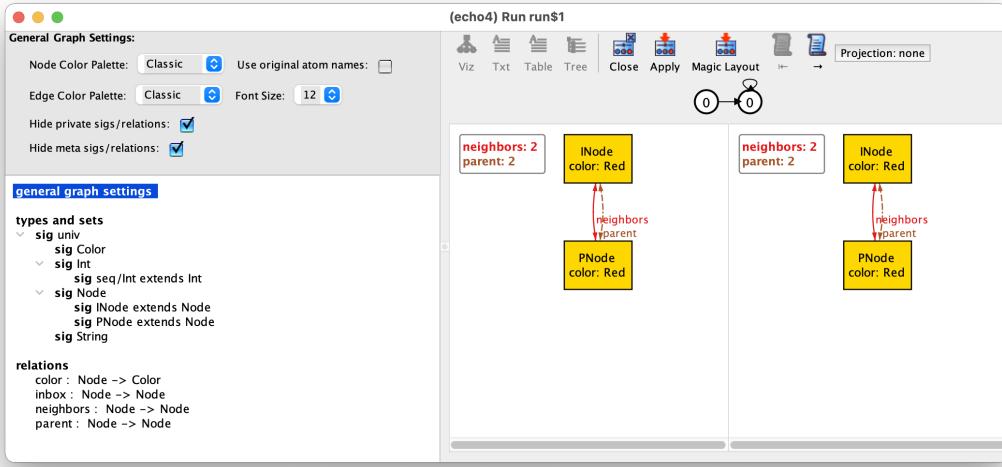


Abbildung 12: *Themes*: Einstellungen für die Visualisierung

werden soll, sondern als Attribut. Es ist für unser Beispiel sinnvoll, auch die Relation `inbox` als Attribut anzuzeigen. Außerdem kann man unter `types and sets` die Anzeige der Signatur `Color` ausschalten. Dadurch werden die Atome für die Farben selbst nicht mehr angezeigt, die Farben tauchen nur noch als Attribut der Knoten auf.

Drückt man nun auf `Apply` sieht man den Effekt der Änderung der Darstellung unmittelbar im rechten Teil des Fensters, wie in Abb. 12. Nach dem Schließen des Fenster zur Einstellung des Themas durch `Close` erhalten wir die übersichtlichere Darstellung wie in Abb. 13. Man kann übrigens das eingestellte Thema speichern, so dass man es später auch wieder laden kann.

Gegenüber der Darstellung der statischen Modelle im vorherigen Abschnitt gibt es nun eine Reihe neuer Schaltflächen, die wir jetzt ausprobieren wollen¹⁰.

`New Config` ergibt ein andere Konfiguration, d.h. ein anderes Modell für die statischen Strukturen einer Spezifikation. In unserem Fall erhalten wir also einen anderen zusammenhängenden ungerichteten Graphen. Da der Default mit dem Kommando `run {}` bis zu drei Objekte je Signatur erzeugt werden, erhalten wir die sechs nicht-isomorphen

¹⁰ Im Text schildere ich die Beispiele so, wie sie bei mir bei der Ausarbeitung des Texts aufgetreten sind. Das muss nicht unbedingt exakt gleich sein, wenn jemand die Beispiele nachvollzieht. Die Ausführung von Alloy ist nicht deterministisch und kann sich eventuell sogar von Lauf zu Lauf ändern.

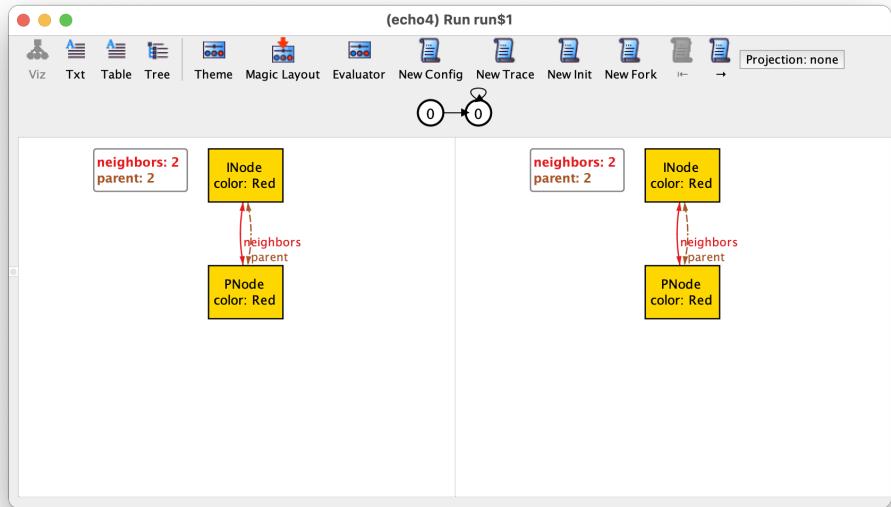


Abbildung 13: Übersichtlichere Visualisierung durch geeignete Einstellungen

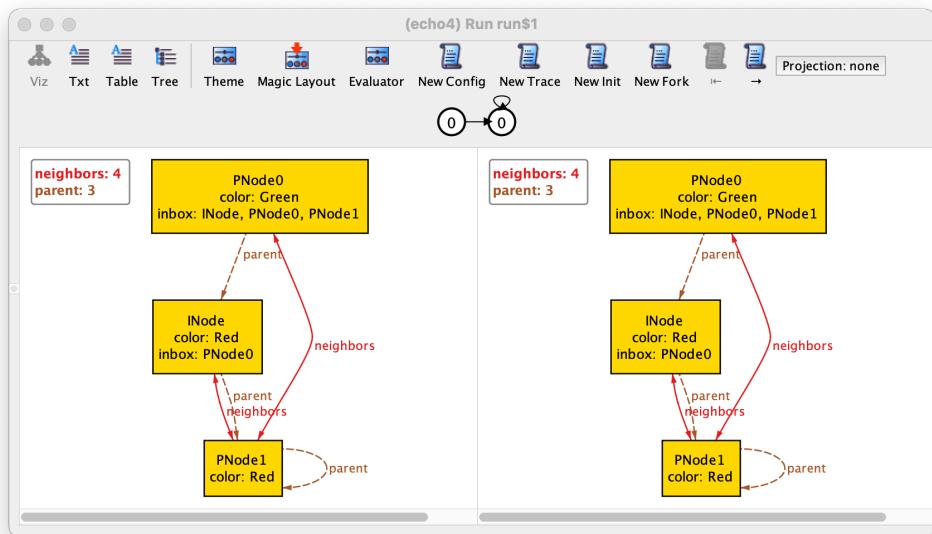


Abbildung 14: Eine Konfiguration nach New Config

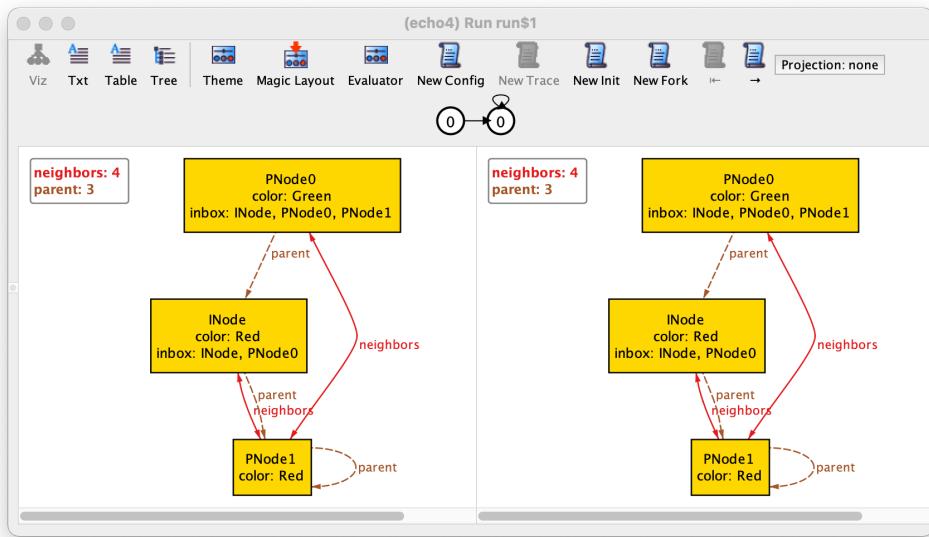


Abbildung 15: Ein neuer Anfangszustand für den Ausführungspfad

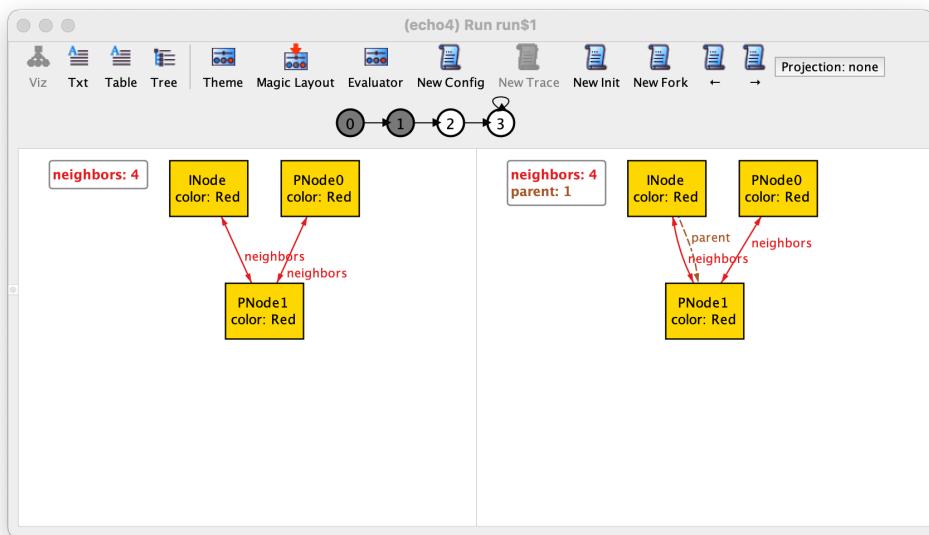


Abbildung 16: Der Ausführungspfad nach zwei Verzweigungen

Modelle: drei Möglichkeiten mit 3 Knoten, eine Möglichkeit mit 2 Knoten und eine Möglichkeiten mit 1 Knoten, dem Initiator allein. In Abb. 14 wird ein Graph mit 3 Knoten angezeigt, den man durch mehrfaches Drücken von `New Config` erhält.

`New Trace` ergibt eine neue Ausführungssequenz. Da wir bisher keine Einschränkungen für die Zustände definiert haben, führt das dazu, dass wir einfach bei einmaligem Drücken einen neuen Initialzustand bekommen und dann eine Sequenz, die immer in diesem Zustand verbleibt.

`New Init` ergibt einen neuen Anfangszustand zu einem Pfad. In Abb. 15 kann man sehen, dass die `inbox` von `INode` einen anderen Wert hat.

Wir können uns auf einem Ausführungspfad mit den Pfeilen vorwärts bzw. rückwärts bewegen. Es wird dann immer angezeigt, welches gerade die beiden Zustände auf dem Pfad sind, die wir in den beiden Hälften des Fenster sehen. Da wir nichts vorgeschrieben haben, führt das zunächst nur dazu, dass wir im selben Zustand verbleiben. Aber wir können auch auf dem aktuellen Pfad am aktuellen Zustand verzweigen mit `New Fork`, wenn das möglich ist. Wenn wir also zum Beispiel folgende Reihenfolge wählen

`→, New Fork, →, New Fork`

dann, erhalten wir die in Abb. 16 abgebildete Situation.

Auf diese Weise können wir im Alloy Analyzer die verschiedenen Möglichkeiten an Ausführungspfaden erkunden, die sich aus unserer Spezifikation ergeben. Bisher ist das natürlich nicht sehr sinnvoll, weil wir ja noch gar nichts spezifiziert haben, was die Zustandsübergänge angeht.

Dazu jetzt, was ist die Grundidee? Zuerst spezifizieren wir den gewünschten Anfangszustand durch das Prädikat `init` und dann Prädikate, die immer einen Zustand und seinen gewünschten Folgezustand beschreiben. Nennen wir diese Prädikate, die die Transitionen präzisieren, zur Veranschaulichung der Grundidee `transpred1` usw., dann ergibt sich im Ergebnis eine Spezifikation des Verhaltens des zu spezifizierenden Systems nach folgendem Schema:

```
fact trans {  
    init  
    always (transpred1 or transpred2 or ... or transpredn)  
}
```

Das Schlüsselwort `always` steht dabei für den Operator \square der temporalen Logik, wobei $\square\phi$ bedeutet, dass die Eigenschaft ϕ in jedem Zustand gelten muss. Das Fakt `trans` legt

also fest, dass ausgehend vom Initialzustand `init` in jedem Folgezustand eines der Transitionsprädikate zutreffen muss.

Der Stil, so die Dynamik eines System zu spezifizieren, wird als *events-as-predicates idiom* bezeichnet. Ich würde die Bezeichnung *Transitionen als Prädikate* vorziehen. Es ist dabei wichtig, die Übergänge vollständig zu spezifizieren. Das bedeutet man muss

1. spezifizieren welche Bedingungen der aktuelle Zustand erfüllen muss, damit der Übergang erfolgen soll, auch *guard* genannt,
2. angeben, welche Veränderungen sich ergeben, in dem man das Verhältnis zwischen einem Element `x` und seinem Wert in nächsten Zustand `x'` festlegt, sowie
3. explizit angeben, was alles unverändert bleibt. Dieses Letztere nennt man auch die *frame condition* und man neigt dazu, sie zu vergessen.

Diese Grundidee wollen wir nun anwenden für die Spezifikation des Echo-Algoritmus.

Im Initialzustand wurden noch keine Nachrichten verschickt und alle Knoten sind rot. Wir nehmen als Elternteil des Initiators ihn selbst, haben also eine Schleife. Diese Schleife kann natürlich nicht Teil des gesuchten aufspannenden Baums des Graphen sein. Aber sie vereinfacht später die Spezifikation der Zustandsübergänge. Wir müssen nur dann später berücksichtigen, dass diese Schleife eben nicht Teil des Spannbaums ist.

```
pred init {  
    parent = INode->INode  
    no inbox  
    color = Node->Red  
}
```

In der ersten Zeile dieses Prädikats legen wir fest, dass die Relation `parent` aus dem Tupel `INode->INode` besteht. In der zweiten Zeile, die übrigens automatisch mit einem `and` zum Rest verbunden wird, wird die Relation `inbox` als leer definiert und dann in der dritten Zeile die Relation `color` als das kartesische Produkt `Node -> Red` definiert.

Letzteres hätte man auch durch `all n: Node | n.color = Red` erreicht. In Alloy kann man Ausdrücke im *relationalen Stil* formulieren oder auch im Stil der *Prädikatenlogik*. Manchmal ist dies Geschmacksache, manchmal eignet sich der eine oder der andere Stil besser für das, was man ausdrücken möchte.

Wenn wir nun verlangen, dass dieses Prädikat immer erfüllt sein muss,

```
fact trans {
    init
}
run {}
```

dann erhalten wir sechs verschiedene Konfiguration mit bis zu 3 Knoten, die alle im Initialzustand die gewünschten Eigenschaften haben.

Die Schaltfläche `New Init` führt zu einer Meldung, dass kein weiteres Modell mehr angezeigt werden kann. Das ist klar, denn wir haben ja alle „variablen“ Elemente durch das Prädikat `init` festgelegt.

Mit `New Trace` hingegen erhalten wir neue Modelle. Denn in unserem `fact trans` wurde ja nur der Initialzustand spezifiziert, was danach sein kann, haben wir nicht festgelegt. Drückt man etwa lange genug auf `New Path`, wird ein Pfad generiert, bei dem der Initiator im zweiten Zustand die Eigenschaft `INode.color = Green` hat.

Wir müssen also festlegen, was in den Folgezuständen erlaubt sein soll. Um die generierten Modelle besser erfassen zu können, passen wir das Thema der Visualisierung noch weiter an. Man kann Form und Farbe der Signaturen wählen. Damit wir besser sehen können, welcher Knoten der Initiator ist, stellen wir als Form `House` ein.

Als erstes Prädikat definieren wir einen Schritt, bei dem sich gar nichts ändert. Alle Ausführungspfade sind in der linearen temporalen Logik unendlich, d.h. die diskrete Zeit schreitet immer weiter fort. Da wir einen Algorithmus spezifizieren, der nach endlich vielen Schritten enden soll, sehen wir vor, dass dann das System immer in diesem Zustand verbleibt, es sich also nichts mehr ändert. Außerdem hat dieser „Stotterschritt“ den Vorteil, dass wir unsere Spezifikation der Prädikate, die Zustandübergänge festlegen, schrittweise aufbauen können.

Da alle von Zustand zu Zustand veränderbaren Strukturen in unserer Spezifikation Relationen sind, müssen wir ausdrücken, dass diese unverändert bleiben, also zum Beispiel `parent' = parent`. Damit diese Eigenschaft in unserer Spezifikation expliziter ausgedrückt wird, verwenden wir den Makro-Mechanismus von Alloy (siehe <http://alloytools.org/quickguide/macro.html>) und definieren vor der Spezifikation der Signaturen folgendes Makro:

```
let unchanged[r] { (r)' = (r) }
```

Nun können wir den „Stotterschritt“ folgendermaßen ausdrücken:

```
pred stutter {
  unchanged[parent]
  unchanged[inbox]
  unchanged[color]
}
```

Wir verwenden `stutter` und lassen uns Modelle generieren:

```
fact trans {
  init
  always { stutter }
}

run {}
```

Abb. 17 zeigt die Situation mit der Konfiguration mit 2 Knoten nach einigen Schritten mit `→`.

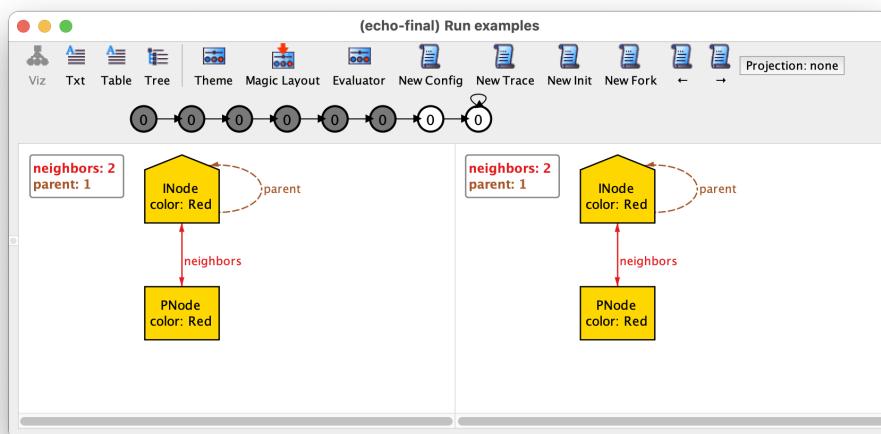


Abbildung 17: Der Ausführungspfad nach `init` mit `stutter`

Es gibt in dieser Situation dann keinen `New Trace`, `New Init` und auch keine Verzweigung `New Fork`. Wir haben ein System spezifiziert, das in jedem Zeitschritt wieder in den aktuellen Zustand, den Initialzustand, wechselt.

Auf dieser Basis beginnen wir nun Prädikate zu definieren, die den Echo-Algorithmus ausmachen. Wir definieren, wie der Initiator den Algorithmus initiiert: Er sendet an alle seine Nachbarn seine eigene Id.

Da später die anderen Knoten auch ihren Nachbarn (mit Ausnahme ihres Elternteils) eine Nachricht senden, machen wir dies als das Prädikat `broadcast` wiederverwendbar:

```
pred broadcast[n, fp: Node] {
    all q: n.neighbors - fp | q.inbox' = q.inbox + n
    all u: Node - n.neighbors + fp | u.inbox' = u.inbox
}
```

Die Parameter des Prädikats `broadcast` sind der aktuelle Knoten `n` sowie die Nachricht, durch die der Broadcast verursacht wurde, d.h. dem Knoten, der Elternteil sein wird. Die Bezeichnung `fp` des zweiten Parameters steht für *future parent*. Die Adressaten der Nachricht müssen im Folgezustand die Id des aktuellen Knotens `n` in ihrem Posteingang haben. Und schließlich brauchen wir noch die Bedingung, dass der Posteingang der restlichen Knoten unverändert bleibt. Das Prädikat `broadcast` ist so konstruiert, dass wir es auch für den initialen Broadcast des Initiators verwenden können.

Wir belassen die Nachricht `fp` übrigens im Posteingang, weil wir uns ja merken müssen, von welchen unserer Nachbarn wir bereits eine Nachricht erhalten haben.

Das Prädikat `initiate` für den Start des Algorithmus ist nach dem obigen Schema aufgebaut:

1. Spezifiziere die Situation, in der Übergang stattfinden soll, in unserem Fall `init`, also den Initialzustand,
2. spezifiziere, wie sich der Folgezustand vom aktuellen Zustand unterscheiden soll, also *lax* gesprochen was sich ändern soll, in unserem Fall der Broadcast, und
3. vergiss nicht die *frame condition*, also zu spezifizieren, was alles gleich bleibt, in unserem Fall die Relationen `parent` und `color`.

```
pred initiate {
    init
    broadcast[INode, INode]
    unchanged[parent]
    unchanged[color]
}
```

```

fact trans {
    init
    always { initiate or stutter }
}

run {}

```

Wenn wir nun Modelle ansehen, dann erhalten wir zunächst auch wieder Beispiele, in denen gar nichts passiert. Dies sind Ausführungspfade, bei denen immer wieder der Schritt `stutter` passiert. Aber jetzt können wir `New Fork` verwenden oder auch `New Path` um einen Ausführungspfad zu erhalten, in dem die initialen Nachrichten verschickt wurden. Abb. 18 zeigt diesen Ausführungspfad am Beispiel des vollständigen Graphen mit 3 Knoten: Der Initiator hat seine Id an die beiden anderen Knoten verschickt.

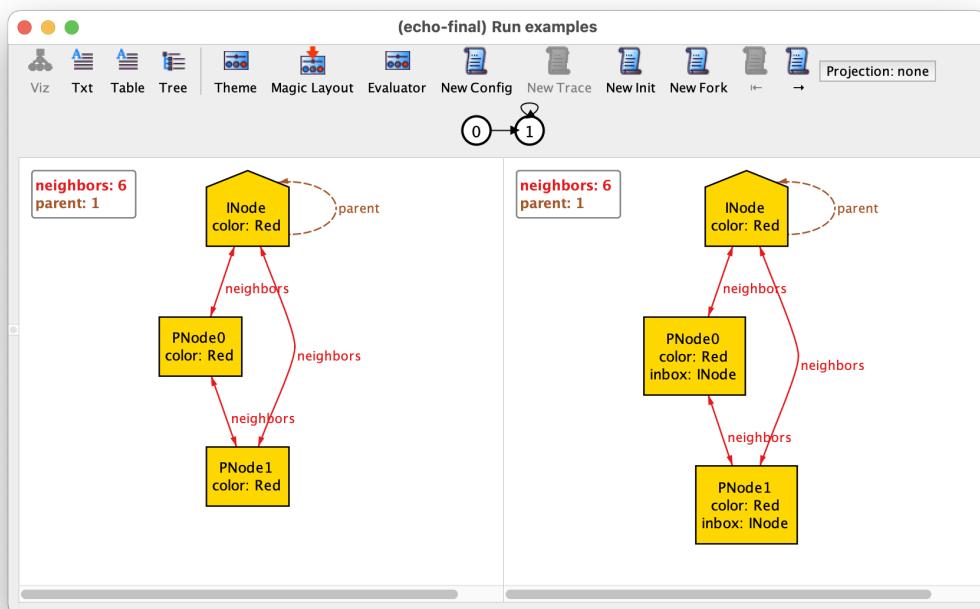


Abbildung 18: Der Ausführungspfad nach `initiate` mit `stutter`

Wir brauchen noch zwei weitere Prädikate für Zustandsübergänge:

`forward` spezifiziert den Übergang, wenn ein Knoten eine Nachricht in seinem Posteingang vorfindet, jedoch bisher noch keine andere Nachricht verarbeitet hat: Der Knoten, der die Nachricht gesendet hat, wird Elternteil, die eigene Id wird an alle anderen Nachbarn geschickt.

```
pred forward [n: Node, msg: Node] {
    no n.parent and msg in n.inbox
    parent' = parent + n->msg
    broadcast[n, msg]
    unchanged[color]
}
```

Die erste Zeile gibt die Bedingung für diesen Zustandübergang an: es muss sich um einen Knoten handeln, der noch kein Elternteil eingetragen hat und die Nachricht befindet sich in seinem Posteingang. In der folgenden Zeile wird dann das Elternteil eingetragen. Dann verwenden wir wieder `broadcast`.

Nun müssen wir noch spezifizieren wann das Echo zurückgeschickt wird. In `echo` hat ein Knoten von allen seinen Nachbarn eine Nachricht erhalten, er kann also das Echo an sein Elternteil zurücksenden.

```
pred echo [n: Node] {
    one n.parent and n.inbox = n.neighbors and n.color = Red
    unchanged[parent]
    n = INode implies
        inbox' = inbox
    else
        inbox' = inbox ++ n.parent->(n.parent.inbox + n)
        color' = color ++ (n->Green)
}
```

Wenn ein Knoten ein Elternteil hat und von allen Nachbarn eine Nachricht erhalten hat, aber noch die Farbe `Red` hat, d.h. das Echo noch nicht zurückgeschickt hat, dann tut er dies und wechselt seine Farbe. Mit `... implies ... else ...` können wir spezifizieren, was in einem bestimmten Fall und seinem Gegenteil gelten soll. In unserem Beispiel behandeln wir den Initiator speziell, er soll nicht sich selbst als Sender in seinen Posteingang eintragen¹¹.

¹¹ Das müssten wir nicht unbedingt tun. Der Algorithmus würde auch ohne diese Spezialbehandlung des Initiators funktionieren. Aber die Visualisierung ist konsistenter und außerdem haben wir das Schlüsselwort `else` kennengelernt.

In der Zeile

```
inbox' = inbox ++ n.parent->(n.parent.inbox + n)
```

wird der Operator `++` eingesetzt. Er definiert ist die Vereinigung zweier Relationen mit „Überschreiben“ der Tupel, die in der linken Relation mit derselben ersten Komponente vorkommen wie ein Tupel in der zweiten Relation. In unserem Fall bedeutet das, dass die Relation `inbox'` die Relation `inbox` unverändert enthält für alle Knoten bis aus das Tupel für den Elternteil des aktuellen Knotens, der den angegebenen neuen Wert erhält: nämlich seinen bisherigen Posteingang vereinigt mit der Echo-Nachricht. Der Operator `++` ist ein Beispiel dafür, wie der relationale Stil zu eleganten Spezifikationen führt.

Nun können wir die komplette Spezifikation des Echo-Algorithmus zusammenstellen:

```
fact trans {
  init
  always {
    initiate or
    some n, msg: Node | forward[n, msg] or
    some n: Node | echo[n] or
    stutter }
}
```

Wenn wir nun mit dem Kommando `run {}` Modelle erzeugen, dann erhalten wir immer auch solche, in denen die Stotterschritte vorkommen, in der Regel sogar diese zuerst. Um unseren Algorithmus „auszuprobieren“ müssen wir also immer den Button `New Fork` verwenden, um auch Zustandübergänge zu erhalten, bei denen was „passiert“. Oder aber wir erzwingen solche Beispiele, indem wir fordern, dass

```
eventually INode.color = Green
```

gilt, wobei wir den temporalen Operator \diamond verwenden. Er hat folgende Semantik: $\diamond \phi$ ist wahr, wenn die Formel ϕ nach endlich vielen Zeitschritten schließlich gilt. In unserem Beispiel bedeutet das also, dass schließlich das Echo zum Initiator zurückgekommen ist.

```
run {eventually INode.color = Green}
```

Abb. 19 zeigt die Sequenz von Zuständen für den vollständigen Graphen mit 3 Knoten.

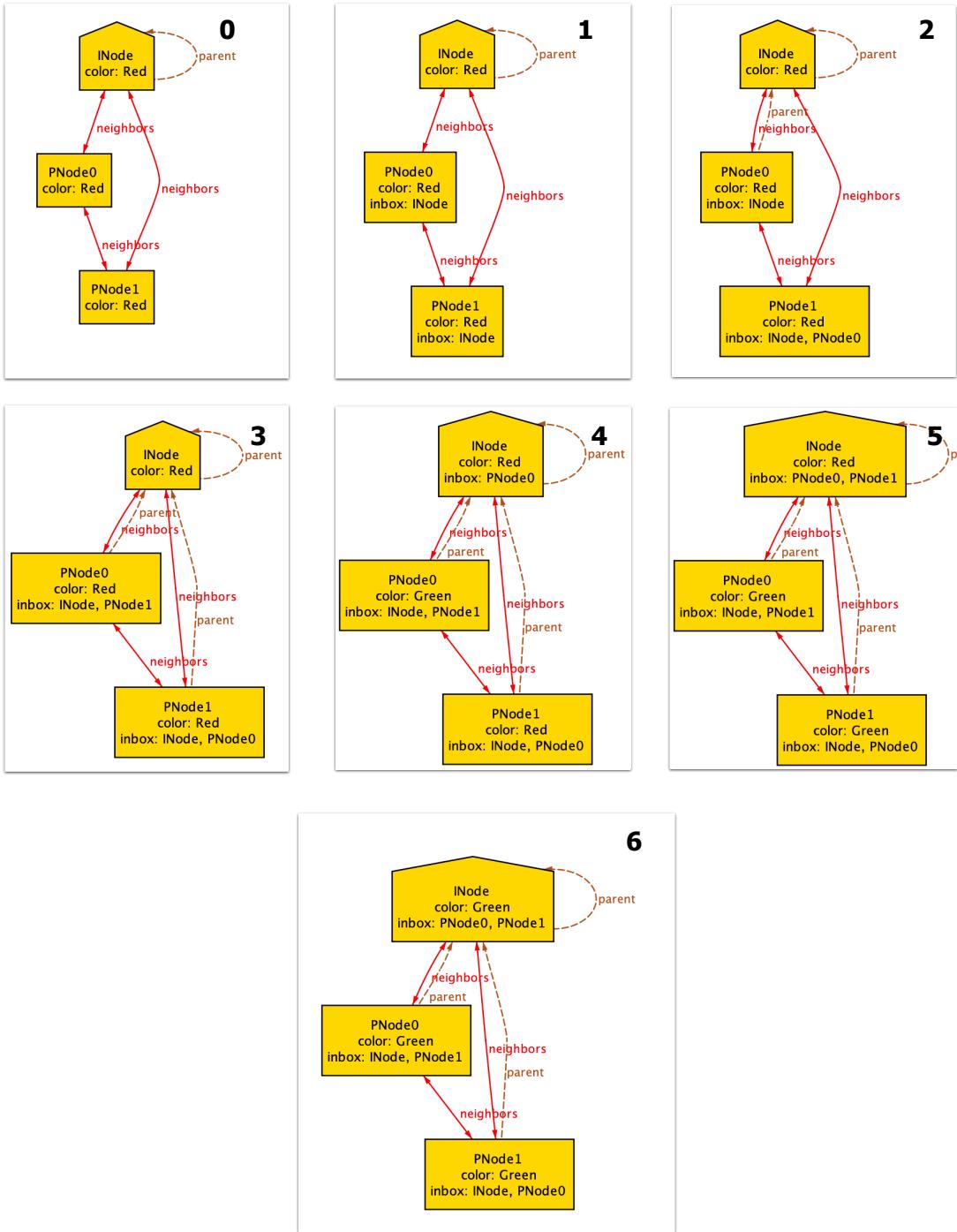


Abbildung 19: Echo-Algorithmus auf den vollständigen Graphen mit 3 Knoten

Allerdings gilt es bei dieser Vorgehensweise zu beachten, dass wir mit dem Kommando

```
run eventually INode.color = Green
```

nur Modelle sehen, bei denen das Echo beim Initiator tatsächlich angelangt ist. Es könnte ja aber sein, dass unsere Spezifikation auch Ausführungspfade erlaubt, bei denen das Echo den Initiator nie erreicht. Dies können wir mit dieser Vorgehensweise niemals erkennen.

Also müssen wir bei der Verifikation unserer Spezifikation anders vorgehen. Davon handelt der folgende Abschnitt.

4 Verifikation mit Alloy 6

Nachdem wir den Echo-Algorithmus spezifiziert haben, interessieren uns natürlich zwei Fragen:

1. Kommt das Echo auch wirklich schließlich zum Initiator zurück?
2. Wenn dies der Fall ist, haben wir dann einen aufspannenden Baum des Graphen gefunden?

Die Vorgehensweise bei der Verifikation von Annahmen in Alloy besteht darin, dass man die zu überprüfende Eigenschaft als `assert` formuliert und dann mit dem Kommando `check` die Überprüfung der Annahme startet. Im Ergebnis wird der Alloy Analyzer ausgeben, ob er ein Gegenbeispiel gefunden hat.

Da die Analyse immer nur mit Modellen mit endlich vielen Objekten durchgeführt wird, per Default begrenzt auf 3, könnte es natürlich sein, dass kein Gegenbeispiel gefunden wird, weil es erst bei einer größeren Anzahl von Objekten auftritt. Es ist also ratsam, bei der Verifikation die Größe des untersuchten Universums sukzessive zu erhöhen. Andererseits kann man davon ausgehen, dass in der Regel für den Typ von Verifikation, die wir machen, Fehler oft schon in Konstellationen mit einer kleinen Anzahl von Objekten gefunden werden können.

In Alloy 6 ist nicht nur die Zahl der Objekte des statischen Teils der Spezifikation beschränkt, sondern auch die Zahl der Schritte, der Zustandsübergänge, die analysiert werden. Hier ist die Schranke per Default 10, d.h. es werden alle der Spezifikation entsprechenden Ausführungspfade mit bis zu 10 Schritten von Zustand zu Folgezustand erzeugt. Auch hier kann es sein, dass uns interessante Fälle (oder auch Fehler in unserer

Spezifikation) entgehen, weil mehr Schritte erforderlich sind, bis sie auftauchen. Auch hier haben wir die Möglichkeit, die Zahl der Schritte zu erhöhen. Aber nicht nur das: Alloy 6 kann auch *Model Checker* aufrufen, die Verifikationen für beliebig viele Schritte durchführen. So kann etwa NuSMV¹² oder nuXmv als Model Checker eingesetzt werden. Dies werden wir später auch tun. Doch beginnen wir nun direkt in Alloy 6 die beiden am Anfang aufgeworfenen Fragen zu untersuchen.

Wenn es darum geht zu prüfen, ob unsere Spezifikation garantiert, dass der Echo-Algorithmus tatsächlich zum Ende kommt, d.h. das Echo zum Initiator gelangt, könnte man versucht sein, Folgendes zu formulieren:

```
assert EchoComesBack {
    eventually INode.color = Green
}

check EchoComesBack
```

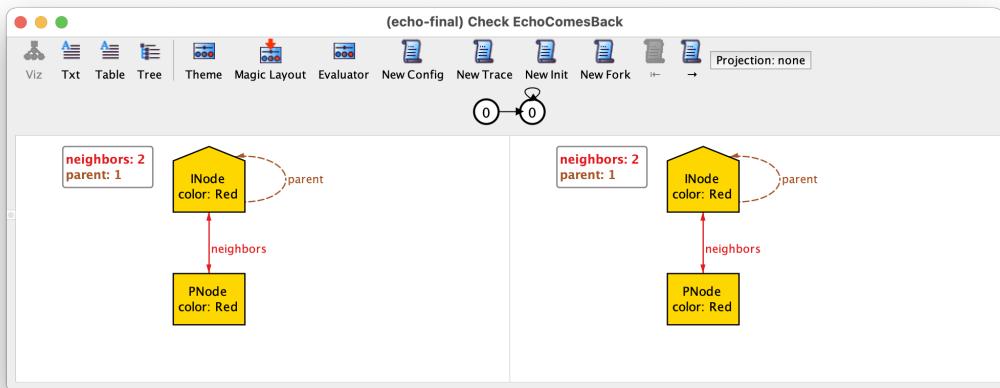


Abbildung 20: Gegenbeispiel für unseren ersten Versuch der Verifikation

Die Analyse liefert jedoch sofort ein Gegenbeispiel, siehe Abb. 20. Nicht sehr verwunderlich: unsere Spezifikation hat als möglichen Zustandsübergang das Prädikat `stutter`, bei dem sich gar nichts ändert. Also ist es klar, dass es Ausführungspfade gibt, bei denen niemals das Echo zum Initiator kommt, weil einfach immerzu nichts passiert.

¹² Die Webseite von NuSMV ist <https://nusmv.fbk.eu/index.html>, die Architektur des Model Checkers wird in [Cim+02] beschrieben, Dokumentation und Tutorial findet man auf der Webseite von NuSMV. Dort ist auch der Link auf die Webseite von nuXmv zu finden.

Wir müssen also bei der Verifikation berücksichtigen, dass wir Ausführungspfade betrachten, bei denen der Algorithmus tatsächlich immer wieder mal einen Fortschritt macht. Dies kann man erreichen, in dem man sogenannte Fairness-Eigenschaften formuliert¹³, etwa *schwache Fairness*, die formal lautet:

$$\Diamond \Box \psi \rightarrow \Box \Diamond \phi$$

Also: wenn schließlich immerzu ψ gilt, dann wird auch immer wieder ϕ erfüllt sein. Informell: „Wenn schließlich etwas immerzu versucht wird, dann wird es unendlich oft gelingen“.

Schwache Fairness können wir nun so einsetzen, dass wir an die Stelle von ψ gerade die Bedingung setzen, die erfüllt sein muss, damit ein Zustandsübergang eintreten darf und an die Stelle von ϕ die Spezifikation des Übergangs selbst. Tun wir das für alle auftretenden Möglichkeiten und probieren das Prädikat `fairness` aus:

```
pred fairness {
    all n, msg: Node {
        (eventually always init)
            implies (always eventually initiate)
        (eventually always (no n.parent and msg in n.inbox))
            implies (always eventually forward[n, msg])
        (eventually always (one n.parent and n.inbox = n.neighbors and
                            n.color = Red))
            implies (always eventually echo[n])
    }
}

run withProgress {fairness}
```

Wir erhalten nun Ausführungspfade für alle (nicht-isomorphen) Graphen mit bis zu drei Knoten und einem dezidierten Anfangsknoten und können sehen, dass in allen diesen Modellen das Echo zum Initiator zurückkommt. Wir können auch alle Fälle mit genau vier Knoten inspizieren:

```
run withProgress {fairness} for exactly 4 Node
```

Wir sehen dann, dass 9 Schritte erforderlich sind und wir erhalten 11 Konfigurationen. In allen von ihnen ist die Annahme erfüllt, dass der Initiator schließlich das Echo erhält, es braucht jeweils 9 Schritte dafür.

¹³ Über typische Aussagen in der linearen temporalen Logik LTL wie Sicherheit und Fairness handelt [Ren21, Kap. 19.3].

Nun erhöhen wir die Zahl der Knoten in den Konfigurationen noch auf fünf. Die Analyse dauert dann schon etwas länger, auf meinem MacBook Pro knapp mehr als 5 Minuten und – es wurde kein Modell gefunden. Warum? Die Zahl der Schritte für die Analyse ist auf 10 per Default begrenzt. Es werden jedoch mindestens 11 Schritte benötigt bei Graphen mit 5 Knoten.

```
run withProgress {fairness} for exactly 5 Node, 11 steps
```

Es dauert jetzt noch länger, nämlich fast 8 Minuten. Es werden dann Modelle gefunden.

Nebenbei: wie kann man die minimale Zahl von Schritten herausfinden? Wir können einen vollständigen Graphen spezifizieren. Hier sind alle Knoten miteinander verbunden, also sind sicherlich maximal viele Nachrichten zwischen den Knoten nötig. Wir probieren nun für den vollständigen Graphen mit 5 Knoten den Algorithmus mit einer hohen Zahl von Schritten aus und sehen dann, wieviele Schritte tatsächlich benötigt wurden. Wie Abb. 21 zeigt, sind es in der Tat 11 Schritte.

```
pred complete {
    neighbors = (Node->Node) - iden
}

run completeGraph {
    complete
    eventually INode.color = Green
} for exactly 5 Node, 20 steps
```

Tatsächlich hätten wir dies auch leicht berechnen können. Unsere Spezifikation sieht vor, dass jeder Knoten zweimal Nachrichten sendet, einmal an alle seine Nachbarn außer dem Elternknoten und einmal das Echo zurück an den Elternknoten. Und ein weiterer Schritt kommt hinzu: die Schleife ganz am Ende. Also hat ein minimaler Ausführungsgraph bei einem Graphen mit $n(n > 1)$ Knoten $2n + 1$ Schritte.

Nun wollen wir die erste Frage überprüfen: Kommt das Echo auch wirklich schließlich zum Initiator zurück?

```
assert EchoComesBack {
    fairness implies eventually INode.color = Green
}

check EchoComesBack
```

Die Analyse ergibt: “No counterexample found. Assertions may be valid.”

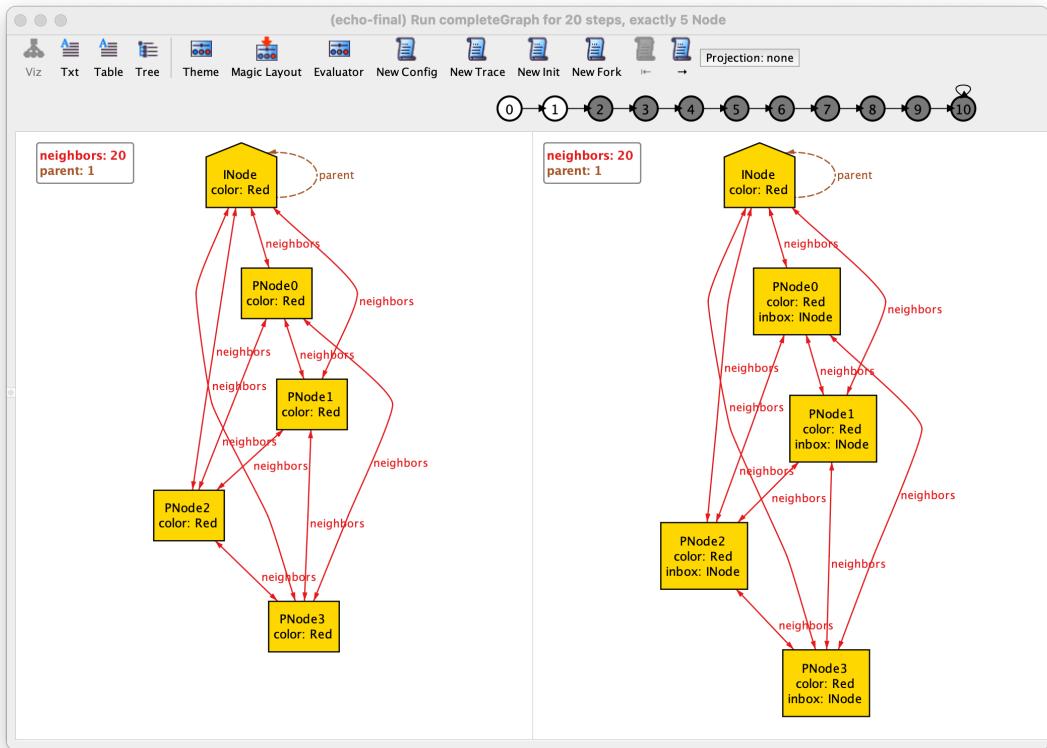


Abbildung 21: Der vollständige Graph mit 5 Knoten

Wir bauen mal absichtlich einen Fehler in die Spezifikation ein. Beim Echo eines Knotens fügt er seine eigene Id dem Posteingang des Elternknotens hinzu. Nun wollen wir diese Bedingung ändern, so dass der Posteingang des Elternknotens *nur* die Id des echoenden Knotens enthält.

```
// bisher in echo: inbox' = inbox ++ n.parent->(n.parent.inbox + n)
// jetzt:
inbox' = inbox ++ n.parent->n

check EchoComesBack
```

Nun erhalten wir ein Gegenbeispiel. Abb. 22 zeigt den Schritt, bei dem die `inbox` von `PNode1` nicht ergänzt wird durch `PNode0`, sondern `INode` aus dem Posteingang verschwindet.

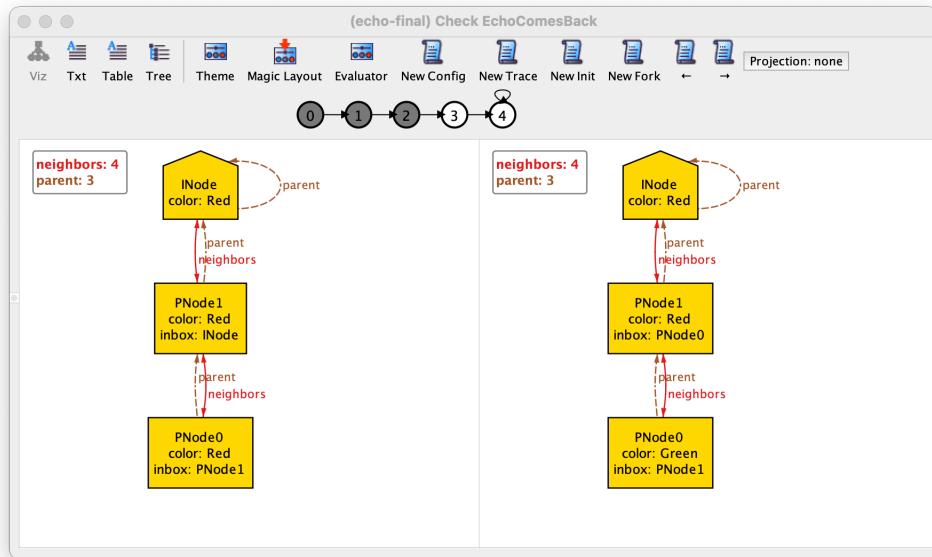


Abbildung 22: Gegenbeispiel bei fehlerhafter Spezifikation

Zurück zu unserer ursprünglichen Spezifikation. Wir erhielten bei der Überprüfung kein Gegenbeispiel. Freilich kennen wir die Limitationen der Analyse in Bezug auf die Zahl der Knoten (3) und die Zahl der Schritte (10). Wir können die Zahl der Knoten erhöhen:

```
check EchoComesBack for 5 Node
```

Auch für Graphen mit bis zu 5 Knoten bekommen wir kein Gegenbeispiel. Aber wie wir oben gesehen haben, reichen 10 Schritte für die Ausführungspfade bei 5 Knoten gar nicht aus. In unserem Beispiel könnten wir nun die Prüfung mit der Zahl von 11 Schritten wiederholen, weil wir ermittelt haben, dass 11 Schritte ausreichen für 5 Knoten.

Man kann in Alloy 6 aber auch die Analyse für eine unbegrenzte Zahl von Schritten durchführen. Dies ist möglich, wenn man für die Analyse nicht einen SAT-Solver verwendet, sondern einen Model Checker wie NuSMV oder nuXmv. Stellt man also im Menüpunkt Options>Solver die Variante Electrod/nuXmv ein, dann können wir folgende Überprüfung machen:

```
check EchoComesBack for 5 Node, 1.. steps
```

Dieses Kommando führt dazu, dass nuXmv alle (nicht-isomorphen) Graphen mit bis zu 5 Knoten für eine beliebige Anzahl von Schritten überprüft.

Nachdem wir uns nun versichert haben, dass das Echo beim Initiator ankommt. können wir die zweite Frage überprüfen, nämlich ob tatsächlich ein Spannbaum des Graphen aufgebaut wurde. Dazu müssen wir checken, dass die Relation `parent` ohne die Schleife bei `INode` ein Baum ist und ob alle Knoten im Baum auch tatsächlich vorkommen. Dazu definieren wir folgende Annahme:

```
assert SpanningTree {
    let pt = parent - INode->INode |
        always (INode.color = Green
            implies (tree[~pt] and parent.Node = Node)
    }

check SpanningTree
```

Im Modul `util/graph` gibt es bereits ein Prädikat `tree`, allerdings wird dort vorausgesetzt, dass die Kanten des Baums vom Eltern- auf den Kindknoten zeigen. Deshalb müssen wir für die Prüfung die transponierte Relation nehmen. Der Ausdruck `parent.Node` ist die Projektion der binären Relation `parent` auf die erste Komponente. Ist diese Projektion gerade `Node` selbst, bedeutet die Bedingung, dass jeder Knoten einen Elternknoten hat.

Die Überprüfung mit dem Solver `Sat4j` ergibt, dass kein Gegenbeispiel gefunden wurde, und auch mit `nuXmv` ist das so.

Nebenbei bemerkt: Man kann im Alloy Analyzer Ausdrücke wie `parent.Node`, die Projektion der binären Relation `parent` auf die erste Komponente leicht inspizieren, in dem man den Menüpunkt `Evaluator` anwählt. Im *Alloy Evaluator* kann man ein bestimmtes Modell zu einem bestimmten Zeitpunkt inspizieren und Ausdrücke von Alloy auswerten. In Abb. 23 wird der Evaluator gezeigt nach dem 6. Schritt des Algorithmus für den vollständigen Graphen mit 3 Knoten. Beispielhaft sieht man links die Projektion der binären Relation `parent` auf die erste und auf die zweite Komponente der Relation.

Damit ist sollte unser erstes Rendez-vous mit Alloy 6 nach meinem ursprünglichen Vorhaben eigentlich beendet sein. Ich denke, das Beispiel hat eine ganze Reihe der Möglichkeiten gezeigt, die man im interaktiven Spezifizieren und Verifizieren mit Alloy hat. Es gibt allerdings noch mehr zu entdecken.

*

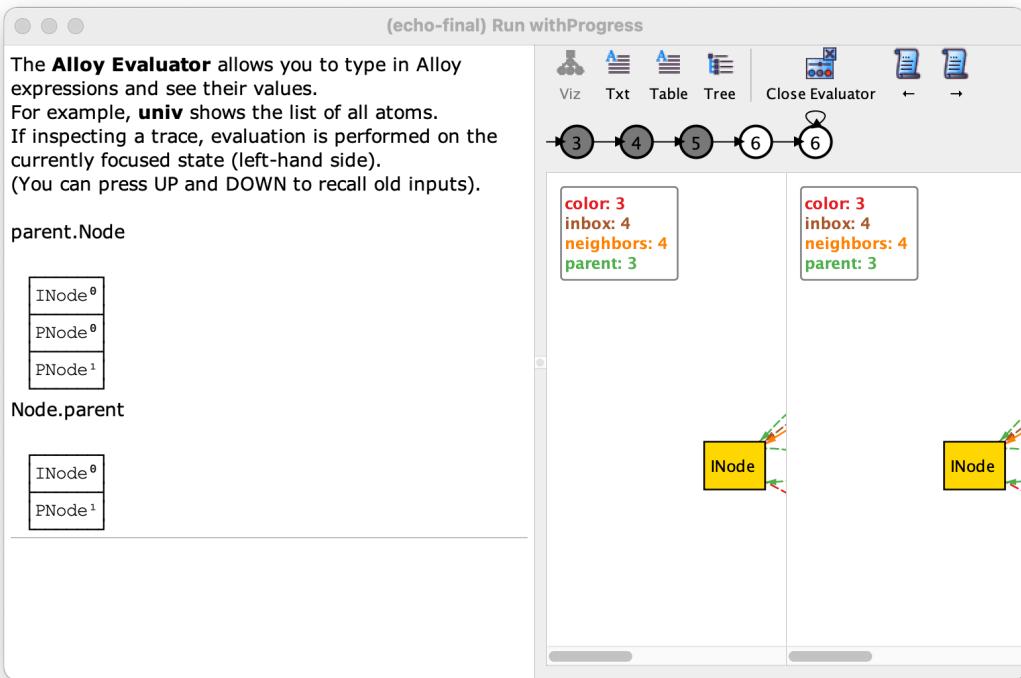


Abbildung 23: Der *Alloy Evaluator* – ein Beispiel

Die Fortsetzung des Spaziergangs folgt den Wegweisern, die Julien Brunel, David Chemouil, Alcino Cunha und Nuno Macedo in [Formal Software Design with Alloy 6](#) [Bru+21] insbesondere im Abschnitt [Alloy modelling tips](#) aufgestellt haben.

Zuerst werden wir neue Typen einführen, mit denen wir in der Visualisierung der Ausführungspfade in jedem Schritt sehen können, welches der möglichen Prädikate für den Zustandsübergang verantwortlich war.

Danach machen wir eine Variante unserer Spezifikation, bei der der Initiator nicht mehr als dezidierter Knoten auftaucht, sondern vom Alloy Analyzer bei der Generierung von Ausführungspfaden gewählt wird.

Diese Variante werden wir dann im abschließenden Abschnitt dazu verwenden, um ein konkretes *Szenario* durchzuspielen, und zwar genau das Beispiel in Abb. 1 und Abb. 2 aus Abschnitt 1.

5 Visualisierung der Transitionsprädikate

Die Spezifikation des Echo-Algorithmus beruht darauf, dass wir eine Bedingung `init` angeben, die den Initialzustand festlegt und dann für jeden möglichen Zustandsübergang ein *Transitionsprädikat*, das angibt unter welcher Bedingung sich welche Eigenschaften im Folgezustand unterscheiden und welche unverändert bleiben. Dieser Stil der Spezifikation von Dynamik wird als *events-as-predicates* bezeichnet. Und wir haben gesehen, dass der Alloy Analyzer Ausführungspfade des Systems visualisiert und uns die Möglichkeit gibt, sie zu inspizieren. In diesem Abschnitt werden wir sehen, dass es außerdem möglich ist, auch noch zu visualisieren, *welches* der Transitionsprädikate für den jeweiligen Zustandsübergang verantwortlich ist. Diese Möglichkeit wird gern als *event reification idiom* bezeichnet. Es geht also um die „Vergegenständlichung“ (was Reifikation laut Duden bedeutet) der Transitionsprädikate: sie werden selbst Objekte in unserer Spezifikation und können deshalb im Alloy Analyzer angezeigt werden. Es folgt nun, durch welche Tricks man dies erreicht:

Wir bauen auf unsere bisherige Spezifikation auf. Der erste Schritt besteht darin, dass wir eine Enumeration für die Transitionsprädikate definieren, also Singleton-Objekte, die diese Prädikate später in der Visualisierung repräsentieren sollen.

```
enum TransPred { Initiate, Forward, Echo, Stutter }
```

Nun müssen wir im zweiten Schritt sicherstellen, dass diese Reifikation auch mit den Zustandsübergängen gekoppelt wird. Dazu verwenden wir Funktionen von Alloy. Funktionen haben in Alloy im Prinzip denselben Aufbau wie Prädikate, aber sie ergeben nicht den Wahrheitswert eines Ausdrucks, sondern geben die Objekte zurück, die den Ausdruck erfüllen. Die Funktionen, die wir definieren, verbinden nun die `TransPred`s mit den Prädikaten, die die jeweiligen Zustandsübergänge kontrollieren:

```
fun initiate: TransPred {
    { tp: Initiate | initiate }
}
fun forward: TransPred -> Node -> Node {
    { tp: Forward, n, msg: Node | forward[n, msg] }
}
fun echo: TransPred -> Node {
    { tp: Echo, n: Node | echo[n] }
}
fun stutter: TransPred {
    { tp: Stutter | stutter}
}
```

```
fun transPreds: set TransPred {
    initiate + forward.Node.Node + echo.Node + stutter
}
```

Die Definition dieser Funktionen sieht auf den ersten Blick selbstbezüglich aus: In der Definition der Funktion `initiate` etwa kommt `initiate` selbst vor. Allerdings handelt es sich einmal um die *Funktion* und einmal um das *Prädikat*. Jedes dieser Konzepte hat seinen eigenen Namensraum oder anders ausgedrückt: Funktionen und Prädikate können überladen werden (siehe [Jac12, S. 127]). Wir werden gleich sehen, wie geschickt man diese Eigenschaft der Sprache verwenden kann.

Betrachten wir nun etwa die Funktion `initiate`. Wollen wir erfahren, ob das zugehörige Prädikat den Zustandsübergang tatsächlich ausgelöst hat, können wir nun einfach `some initiate` verwenden, das Prädikat, das angibt, dass die Funktion `initiate` ein Ergebnis hatte. Für Funktionen, die Argumente benötigen, kann man das auch tun, indem man sie auf die erste Komponente des Ergebnisses projiziert, die ja gerade das jeweilige `TransPred` enthält. Auf diese Weise definiert die Funktion `transPreds` die Menge aller möglichen Zustandsübergänge unseres Systems. Das bedeutet, dass wir nun die Transitionen so spezifizieren können:

```
fact trans {
    init
    always {
        some transPreds
    }
}
```

Um mit diesen Änderungen eine hübsche Visualisierung zu bekommen, müssen wir noch das *Theme* der Darstellung anpassen:

- Die Signatur `TransPred` soll in der Farbe `Blue` angezeigt werden. Ihre Objekte sollen nicht angezeigt werden, d.h. wir setzen `Show` auf `Off`.
- Aber die Menge `transPreds` soll schon angezeigt werden, denn sie enthält ja jeweils die zutreffenden Transitionsprädikate.
- Die Mengen `initiate` sowie `stutter` brauchen nicht als Labels gezeigt werden.
- Die Relationen `echo` und `forward` sollen als Attribute angezeigt werden.

Es lohnt sich, verschiedene Einstellungen der Visualisierung auszuprobieren, denn man kann durch `Apply` den Effekt unmittelbar sehen und so eine übersichtliche, aussagekräftige Darstellung erreichen. Mit den eben vorgenommenen Einstellungen erhalten wir

zum Beispiel für den vollständigen Graphen mit 3 Knoten die Darstellung in Abb. 24 an der wir ablesen können, dass der erste Schritt von Zustand 0 zu Zustand 1 durch das Transitionsprädikat `initiate` spezifiziert wurde und der zweite Schritt durch `forward` des Knotens `PNode1` mit dem künftigen Elternknoten `INode`.

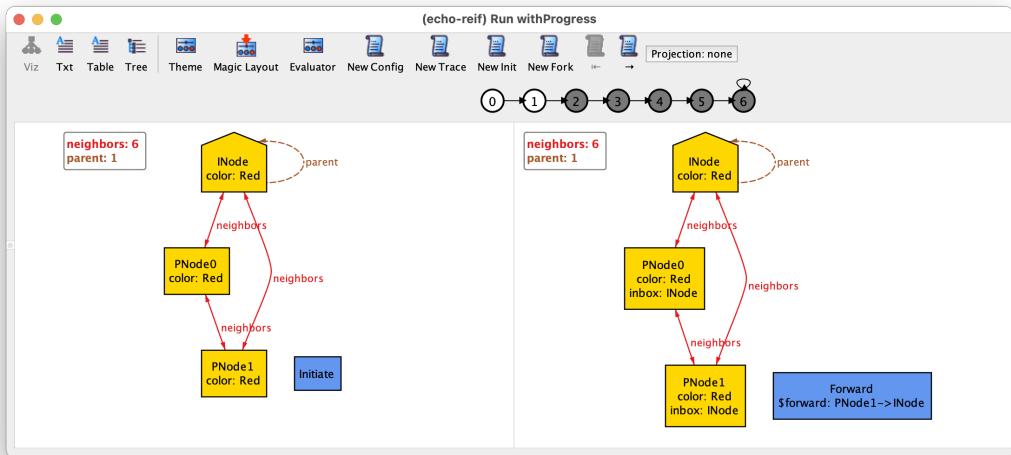


Abbildung 24: Visualisierung der Transitionsprädikate

Nebenbei: man kann die Idee Funktionen zur Verdeutlichung der Visualisierung einzusetzen auch dafür verwenden, die Knoten des Graphen grün anzusehen, die ihr Echo bereits geschickt haben. Man sieht dann wie schrittweise immer mehr Knoten grün werden.

Dazu führen wir folgende zusätzliche Funktion ein

```
fun finished: set Node {
    { n: Node | n.color = Green }
}
```

und im `Theme` stellen wir ein, dass die Menge `$finished` grün dargestellt wird.

6 Variante der Spezifikation

Für die Spezifikation des Echo-Algorithmus muss man nicht unbedingt einen *dezidierten* Initiator in der Struktur der Graphen vorsehen. In diesem Abschnitt wollen wir eine Spezifikation entwerfen, bei der der Initiator erst beim ersten Schritt des Algorithmus festgelegt wird.

Wir starten deshalb mit der Spezifikation von zusammenhängenden ungerichteten Graphen und sehen dabei schon die Relationen vor, die wir für die Ausführung des Algorithmus benötigen:

```
open util/graph[Node]

let unchanged[r] { (r)' = (r) }

enum Color {Red, Green}

abstract sig Node {
    neighbors: set Node,
    var parent: lone Node,
    var inbox: set Node,
    var color: Color
}

fact {
    noSelfLoops[neighbors]
    undirected[neighbors]
    stronglyConnected[neighbors]
}
```

Der Unterschied besteht zunächst nur darin, dass wir nicht wie bisher einen speziellen Initiator `INode` in der Definition der Graphen vorgesehen haben.

Das Transitionsprädikat `stutter` wird unverändert übernommen, ebenso wie das Versenden von Nachrichten durch `broadcast`.

```
pred stutter {
    unchanged[parent]
    changed[inbox]
    unchanged[color]
}
```

```

pred broadcast[n, fp: Node] {
    all q: n.neighbors - fp | q.inbox' = q.inbox + n
    all u: Node - n.neighbors + fp | u.inbox' = u.inbox
}

```

Da wir nun keinen vorgegebenen Initiator haben, müssen wir das Prädikat `initiate` ändern: es wird parametrisiert durch den zu wählenden Knoten, der den Algorithmus startet.

```

pred initiate[n: Node] {
    parent = n -> n
    no inbox
    color = Node -> Red
    unchanged[parent]
    #Node = 1 implies {
        color' = n -> Green
        unchanged[inbox]
    } else {
        broadcast[n, n]
        unchanged[color]
    }
}

```

Das Transitionsprädikat `initiate` spezifiziert nun den ersten Schritt des Algorithmus. Der übergebene Knoten wird der Initiator, kenntlich daran, dass er sich selbst als Elternknoten referenziert. Für den Zustandsübergang selbst unterscheiden wir den Fall, dass der Graph nur aus einem Knoten besteht von den größeren Graphen.

Die Transitionsprädikate `forward` und `echo` bleiben im Prinzip unverändert, wir müssen nur in `echo` den Initiator durch die Schleife der Relation `parent` identifizieren:

```

pred forward [n: Node, msg: Node] {
    no n.parent and msg in n.inbox
    parent' = parent + n->msg
    broadcast[n, msg]
    unchanged[color]
}

pred echo [n: Node] {
    one n.parent and n.inbox = n.neighbors and n.color = Red
    unchanged[parent]
    n.parent = n implies
        inbox' = inbox
}

```

```

    else
        inbox' = inbox ++ n.parent->(n.parent.inbox + n)
        color' = color ++ (n->Green)
}

```

Fast fertig. Die Besonderheit besteht jetzt darin, dass `initiate` bereits den ersten Schritt des Algorithmus durchgeführt hat, weshalb wir die weiteren Schritte erst *nach* dem ersten Schritt vorsehen dürfen, deshalb das `after` in der Spezifikation der Transitionen. Der temporale Operator `after` wird in der LTL als \circ geschrieben. $\circ \phi$ bedeutet, dass die Formel ϕ im Folgezustand gilt.

```

fact trans {
    some n: Node | initiate[n]
    after always {
        some n, msg: Node | forward[n, msg] or
        some n: Node | echo[n] or
        stutter }
}

```

Kleine Anpassungen ergeben den Rest der Variante:

```

pred fairness {
    all n, msg: Node {
        (eventually always (no n.parent and msg in n.inbox))
            implies (always eventually forward[n, msg])
        (eventually always (one n.parent and n.inbox = n.neighbors
                            and n.color = Red))
            implies (always eventually echo[n])
    }
}

run makeProgress {fairness}

assert echoComesBack {
    fairness implies (eventually all n:Node | n.color = Green)
}

check echoComesBack

assert SpanningTree {
    always color = Node->Green
        implies tree[~(parent-iden)])
}

```

7 Szenarien

In diesem Abschnitt werden Szenarien mit konkret gegebenen Graphen betrachtet. Damit schließt sich der Kreis zum Anfang, denn wir werden als Beispiel den Graphen mit 7 Knoten aus Abb. 1 als Beispiel nehmen und ein Szenario konstruieren, bei dem der aufspannende Baum wie in Abb. 2 entsteht.

Der erste Ansatz für Szenarien könnte sein, dass man mit Singletons arbeitet, d.h. man gibt explizit 7 Knoten vor.

```
open util/graph[Node]

let unchanged[r] { (r)' = (r) }

enum Color {Red, Green}

abstract sig Node {
    neighbors: set Node,
    var parent: lone Node,
    var inbox: set Node,
    var color: Color
}

one sig N0, N1, N2, N3, N4, N5, N6 extends Node {}
```

Die Transitionsprädikate kann man unverändert aus der Variante in Abschnitt 6 übernehmen:

```
pred stutter {
    unchanged[parent]
    unchanged[inbox]
    unchanged[color]
}

pred broadcast[n, fp: Node] {
    all q: n.neighbors - fp | q.inbox' = q.inbox + n
    all u: Node - n.neighbors + fp | u.inbox' = u.inbox
}

pred initiate[n: Node] {
    parent = n -> n
    no inbox
    color = Node -> Red
```

```

unchanged[parent]
#Node = 1 implies {
    color' = n -> Green
    unchanged[inbox]
} else {
    broadcast[n, n]
    unchanged[color]
}
}

pred forward [n: Node, msg: Node] {
    no n.parent and msg in n.inbox
    parent' = parent + n->msg
    broadcast[n, msg]
    unchanged[color]
}

pred echo [n: Node] {
    one n.parent and n.inbox = n.neighbors and n.color = Red
    unchanged[parent]
    n.parent = n implies
        inbox' = inbox
    else
        inbox' = inbox ++ n.parent->(n.parent.inbox + n)
    color' = color ++ (n->Green)
}

```

Nun definieren wir den *konkreten* Graphen aus Abb. 1 explizit als `fact`:

```

fact {
    neighbors = N0->N2 + N0->N3 +
                N1->N2 + N1->N3 +
                N2->N0 + N2->N1 + N2->N3 + N2->N4 +
                N3->N0 + N3->N1 + N3->N2 + N3->N4 + N3->N5 +
                N4->N2 + N4->N3 + N4->N6 +
                N5->N3 + N5->N6 +
                N6->N4 + N6->N5
}

```

Jetzt können wir die Schritte so durchführen, wie sie in der Beschreibung des Beispiels zum Spannbaum in Abb. 2 geführt haben:

```
run Scenario {
```

```

initiate[N2]
; forward[N0, N2]
; forward[N4, N2]
; forward[N3, N4]
; forward[N1, N3]
; forward[N6, N4]
; forward[N5, N3]
; echo[N5]
; echo[N6]
; echo[N1]
; echo[N3]
; echo[N4]
; echo[N0]
; echo[N2]
; always stutter
} for 7 Node, 15 steps

```

Hierbei kommt ein neuer Operator der Sprache von Alloy 6 ins Spiel, das Semikolon ; , das für den *Sequenzoperator* steht: mit jedem ; geht es in den nächsten Zustand. In unserem Beispiel werden also die Schritte in der angegebenen Reihenfolge ausgeführt und ganz am Ende lässt jeder Übergang alles unverändert.

Um dieses Szenario auszuprobieren musste man die Eigenschaften des Graphen gar nicht explizit spezifizieren, weil wir ja einen konkreten Graphen vorgegeben haben, der ungerichtet und zusammenhängend ist. Ebenso konnten wir auf ein Fakt verzichten, das die Transitionen festlegt, denn wir haben sie ja im Szenario explizit angegeben.

Allerdings hat dieser erste Ansatz einen großen Nachteil. Wir sind gewissermaßen statisch in diesem Ansatz, also festgelegt auf exakt diesen Graphen. Deshalb eignet sich dieser Ansatz eigentlich nur zum Ausprobieren von Transitionsprädikaten an einem Beispiel. Das kann hilfreich bei der Entwicklung der Spezifikation sein, später hilft er aber nicht mehr viel.

Es wäre viel besser, wenn wir Szenarien auf Basis unserer Spezifikation machen könnten. Hierbei ist das Modulkonzept von Alloy hilfreich. Wir können in die Spezifikation unseres Szenarios die in Abschnitt 6 beschriebene Spezifikation des Echo-Algorithmus als Modul einbinden und dann den konkreten Graphen definieren:

```

open echo_var

pred exampleGraph [N0, N1, N2, N3, N4, N5, N6: Node] {
    Node = N0 + N1 + N2 + N3 + N4 + N5 + N6
}

```

```

neighbors = N0->N2 + N0->N3 +
           N1->N2 + N1->N3 +
           N2->N0 + N2->N1 + N2->N3 + N2->N4 +
           N3->N0 + N3->N1 + N3->N2 + N3->N4 + N3->N5 +
           N4->N2 + N4->N3 + N4->N6 +
           N5->N3 + N5->N6 +
           N6->N4 + N6->N5
}

```

Diesen so definierten Graphen können wir nun in unserem Szenario verwenden:

```

run Scenario {
    some disj N0, N1, N2, N3, N4, N5, N6: Node {
        exampleGraph[N0, N1, N2, N3, N4, N5, N6]

        initiate[N2]
        ; forward[N0, N2]
        ; forward[N4, N2]
        ; forward[N3, N4]
        ; forward[N1, N3]
        ; forward[N6, N4]
        ; forward[N5, N3]
        ; echo[N5]
        ; echo[N6]
        ; echo[N1]
        ; echo[N3]
        ; echo[N4]
        ; echo[N0]
        ; echo[N2]
        ; always stutter
    }
} for 7 Node, 15 steps

```

Das Schlüsselwort `disj` legt fest, dass die Knoten paarweise verschieden sind. Auf diese Weise können wir nun unseren Beispielgraphen verwenden, ohne die Spezifikation des Algorithmus selbst mit diesem konkreten Graphen zu vermischen.

Diese zweite Möglichkeit, Szenarien zu definieren, eignet sich also sowohl zum Ausprobieren während der Entwicklung der Spezifikation, wie auch für Testszenarien, an denen man Änderungen der Spezifikation leicht überprüfen kann.

Mit diesem Beispiel ist unsere erste Begegnung mit Alloy 6 fast beendet. Weitere interessante Softwareartefakte warten darauf in Alloy interaktiv spezifiziert zu werden.

Im folgenden Abschnitt werfen wir noch einen Blick auf die Laufzeit der Analyse und lernen verschiedene Strategien der Analyse kennen.

8 Laufzeitmessungen

In diesem Abschnitt wird von einigen Laufzeitmessungen berichtet.

Die Spezifikationen von Alloy 6 bestehen aus statischen Strukturen, den Konfigurationen, die sich auf einem Ausführungspfad nicht ändern einerseits, sowie den möglichen Ausführungspfaden pro Konfiguration andererseits.

Das Kommando `run {} for 4 Node` führt in unserem Beispiel dazu, dass alle Ausführungspfade bis zu 10 Schritten analysiert werden für alle Graphen (modulo Isomorphie) mit bis zu 4 Knoten. Diese Graphen sind die Konfigurationen, für die es dann jeweils verschiedene Abläufe der Schritte des Algorithmus geben kann.

Für die Analyse kann man das Problem demzufolge zerlegen in (1) das Finden der Konfigurationen und (2) das Generieren der möglichen Ausführungspfade pro Konfiguration. Wegen dieser Zerlegung ist es möglich, die Analyse von Spezifikationen zu *parallelisieren*.

Der Alloy Analyzer hat drei mögliche Strategien für die Zerlegung der Analyse. Diese Strategien kann man im Menüpunkt `Options > Decompose strategy` einstellen. Die Möglichkeiten sind:

batch Die Analyse wird nicht zerlegt, sondern als Ganzes an einen Solver übergeben.

parallel Die Analyse wird zerlegt in das Finden von Ausführungspfaden pro Konfiguration und dieses wird parallel durchgeführt.

hybrid Bei dieser Strategie werden die Ausführungspfade der verschiedenen Konfigurationen parallel berechnet, aber zusätzlich bearbeitet ein Thread parallel dazu die unzerlegte Spezifikation.

Die hybride Strategie wird zusätzlich angeboten, weil es sein kann, dass es sehr viele zulässige Konfigurationen gibt, aber nur wenige erlaubte Ausführungspfade. In diesem Fall kann die Strategie der Parallelisierung zu einer langen Laufzeit führen, während die unzerlegte Spezifikation eventuell schneller analysiert werden kann [Mac+21].

Daraus ergibt sich folgende Faustregel:

- Wenn man erwartet, dass die Analyse zu einem Modell führt, eignet sich die parallele Strategie, denn sie kann schneller zu einem Ergebnis führen.
- Erwartet man jedoch, dass es kein Modell gibt, etwa dass bei einem `check` kein Gegenbeispiel gefunden werden kann, dann eignet sich die Strategie `batch` besser.

Selbstverständlich hängt die Laufzeit einer Analyse nicht nur von der Wahl der Strategie, sondern insbesondere von der Wahl des Solvers ab.

Im Folgenden werden Laufzeiten aufgelistet. Die Messungen wurden auf meinem MacBook Pro mit einem 2,3 GHz Quad-Core Intel Core i7 mit 32 GB Hauptspeicher durchgeführt. Analysiert wurde die Spezifikation in `echo.als` mit:

- Solver: `nuXmv` und `Sat4j`
- Aufgabe: `makeProgress`, `echoComesBack` und `SpanningTree`
- Strategie: `batch`, `hybrid` und `parallel`
- Konfigurationen: Graphen mit bis zu 5 Knoten

Solver `nuXmv`

Tabelle 1: Ausführungszeit von `makeProgess` mit `nuXmv`

Anzahl Knoten	batch	hybrid	parallel
= 3 Node	3,3 Sek.	2,4 Sek.	2,2 Sek.
= 4 Node	35 Sek.	3,9 Sek.	3,7 Sek.
= 5 Node	23 Min.	2 Min.	12,4 Sek.

Tabelle 2: Ausführungszeit von `echoComesBack` mit `nuXmv`

Anzahl Knoten	batch	hybrid	parallel
≤ 3 Node	3,4 Sek.	2,6 Sek.	2,3 Sek.
≤ 4 Node	38 Sek.	27 Sek.	21 Sek.
≤ 5 Node	3 Min. 35 Sek.	5 Min.	7 Min.

Tabelle 3: Ausführungszeit von `SpanningTree` mit `nuXmv`

Anzahl Knoten	batch	hybrid	parallel
≤ 3 Node	0,5 Sek.	0,5 Sek.	0,5 Sek.
≤ 4 Node	2,2 Sek.	2,6 Sek.	4,4 Sek.
≤ 5 Node	5,5 Sek.	6,6 Sek.	1 Min. 37 Sek.

Solver `Sat4j`

Tabelle 4: Ausführungszeit von `makeProgress` mit `Sat4j`

Anzahl Knoten	batch	hybrid	parallel
= 3 Node	0,5 Sek.	0,6 Sek.	0,4 Sek.
= 4 Node	6 Sek.	1,7 Sek.	1,7 Sek.
= 5 Node	6 Min. 30 Sek.	14 Sek.	10 Sek.

Tabelle 5: Ausführungszeit von `echoComesBack` mit `Sat4j`

Anzahl Knoten	batch	hybrid	parallel
≤ 3 Node	6,7 Sek.	2,6 Sek.	2,4 Sek.
≤ 4 Node	38 Sek.	27 Sek.	75 Sek.
≤ 5 Node	3 Min. 35 Sek.	1 Std. 14 Min.	1 Std. 1 Min.

Tabelle 6: Ausführungszeit von `SpanningTree` mit `Sat4j`

Anzahl Knoten	batch	hybrid	parallel
≤ 3 Node	0,2 Sek.	0,2 Sek.	0,2 Sek.
≤ 4 Node	0,2 Sek.	0,3 Sek.	1 Sek.
≤ 5 Node	0,3 Sek.	0,5 Sek.	7,6 Sek.

Literatur

- [Asm10] Nils Asmussen. *Ansätze zur Modellierung von Dynamik in Alloy*. 2010. URL: <https://esb-dev.github.io/mat/alloy-dynamik.pdf> (besucht am 01.11.2021).
- [Bru+18] Julien Brunel u. a. „The electrum analyzer: model checking relational first-order temporal specifications“. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Hrsg. von Marianne Huchard, Christian Kästner und Gordon Fraser. ACM, 2018, S. 884–887.
- [Bru+21] Julien Brunel u. a. *Formal Software Design with Alloy 6*. 2021. URL: <https://haslab.github.io/formal-software-design/index.html> (besucht am 23.11.2021).
- [Cim+02] Alessandro Cimatti u. a. „NuSMV 2: An OpenSource Tool for Symbolic Model Checking“. In: *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Hrsg. von Ed Brinksma und Kim Guldstrand Larsen. Bd. 2404. Lecture Notes in Computer Science. Springer, 2002, S. 359–364.
- [Fok18] Wan Fokkink. *Distributed algorithms: an intuitive approach*. Cambridge, MA: MIT Press, 2018.
- [Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. Revised edition. Cambridge, MA: MIT Press, 2012.
- [Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA: Addison-Wesley, 2002.
- [Mac+21] Nuno Macedo u. a. „Pardinus: A temporal relational model finder“. In: *Journal of Automated Reasoning* to appear (2021).
- [RA09] Burkhardt Renz und Nils Asmussen. *Kurze Einführung in Alloy*. 2009. URL: <https://esb-dev.github.io/mat/alloy-intro.pdf> (besucht am 01.11.2021).
- [Ren21] Burkhardt Renz. *Logik und formale Methoden: Vorlesungsskript*. 2021. URL: <https://esb-dev.github.io/mat/lfm.pdf> (besucht am 01.11.2021).