# Korrektheit von Datenflüssen in Asynchronen Verteilten Systemen – Modellprüfung und Synthese[1]

Manuel Gieseking[2]

**Abstract:** Die Korrektheit von informationstechnischen Systemen spielt aufgrund deren wachsenden Einbindung in unser alltägliches Leben eine zunehmend wichtige Rolle und ist nicht zuletzt für sicherheitskritische Systeme entscheidend. Insbesondere *asynchrone verteilte Systeme* sind, aufgrund ihrer Vielzahl von unabhängig voneinander agierenden Komponenten, ohne computergestützte Analyseverfahren schwer korrekt zu implementieren. In der Dissertation führen wir, basierend auf *Petri-Netzen* bzw. *Petri-Spielen* und CTL$^*$, neue Modellierungs- und Spezifikationsformalismen ein, die es ermöglichen, Anforderungen an den unbeschränkten lokalen Datenfluss in asynchronen verteilten Systemen zu stellen. Wir stellen Lösungsalgorithmen für die entsprechenden *Modellprüfungs-* und *Syntheseprobleme* zur Verfügung, zeigen deren Korrektheit und Komplexität und bieten Werkzeugunterstützung für Teilklassen dieser Probleme.

## 1 Einleitung

Die Verbreitung und gesellschaftliche Abhängigkeit von Computersystemen hat in den letzten Jahrzehnten rapide zugenommen. Die Systeme haben sich von spezialisierten Werkzeugen für einige wenige zu einem unverzichtbaren Teil unseres alltäglichen Lebens entwickelt. Wir benutzen die Technologien zum Beispiel ganz selbstverständlich in unseren Handys oder verlassen uns auf ihre Funktionalität, wenn wir Hochgeschwindigkeitszüge benutzen. Die *Korrektheit* solcher Systeme ist entscheidend und eine äußerst anspruchsvolle Aufgabe. Während ein defektes Handy bestenfalls ärgerlich ist, kann ein Fehler im Notbremssystem des Zuges einen immensen Schaden anrichten.

Aufgrund der ständigen Verfügbarkeit von Netzwerken und des immer geringeren Platzbedarfs leistungsfähiger Geräte setzen sich moderne Computersysteme zunehmend aus einer Vielzahl vernetzter Computer zusammen. Auch wenn das System selbst als eine einzige kohärente Einheit erscheint, agieren die Komponenten eines solchen *verteilten Systems* häufig autonom. Um eine ständige Kommunikation jeder einzelnen Komponente mit einer zentralen Steuerung zu vermeiden, werden die Systeme immer öfter dezentralisiert aufgebaut. Dies hat jedoch den Preis, dass die Komponenten nur noch ein unvollständiges Wissen über den Gesamtzustand des Systems haben. Im Gegensatz zu einem *synchronen* System, wo die einzelnen, unabhängigen Komponenten in einem gemeinsamen festen Rhythmus

[1] Englischer Titel der Dissertation: “Correctness of Data Flows in Asynchronous Distributed Systems – Model Checking and Synthesis”

[2] Carl von Ossietzky Universität Oldenburg, gieseking@informatik.uni-oldenburg.de

voranschreiten, schreitet in einem *asynchronen* verteilten System jede Komponente in ihrem eigenen individuellen Tempo voran (möglicherweise unterbrochen durch Synchronisationen mit anderen Komponenten). Dies macht es besonders schwierig, Algorithmen für asynchrone verteilte Systeme korrekt zu implementieren, da die Komponenten zwischen den Synchronisationen nicht die Zustände der anderen Komponenten kennen.

Die zunehmende Größe und Komplexität dieser Systeme macht es für den Menschen nochmals schwieriger, Steuerungen für die einzelnen Komponenten *korrekt* zu implementieren. Folglich ist die Nachfrage nach Computerunterstützung für die Entwicklung solcher Systeme in den letzten Jahren deutlich gestiegen. Die *Modellprüfung* (*Model Checking*) und die *Synthese* stellen dabei zwei etablierte und vollautomatische Ansätze zur Entwicklung von korrekten Implementierungen aus mathematisch präzisen und eindeutigen formalen Modellen und Spezifikationen dar. Bei der Modellprüfung wird eine gegebene Implementierung daraufhin überprüft, ob sie die Korrektheitsspezifikation erfüllt, wohingegen der Syntheseansatz aus einer gegebenen Spezifikation automatisiert eine korrekte Implementierung erzeugt.

In der Dissertation [Gi22] führen wir neue Modellierungs- und Spezifikationssprachen für die Modellprüfung und die Synthese von asynchronen verteilten Systemen ein. Wir entwickeln mehrere Algorithmen für die zugehörigen Verifikations- und Syntheseprobleme und beweisen deren Korrektheit. Im Synthesefall waren wir in der Lage, erstmalig entscheidbare Lösungsalgorithmen zu erarbeiten, die ermöglichen das zugrunde liegende Modell um Spezifikationen zu erweitern, die über Sicherheitsanforderungen (die Vermeidung von unerwünschten Situationen) hinaus gehen. Die Implementierungen der Algorithmen und zugehörigen Benchmarks sind quelloffen und haben die Artifakt-Evaluations-Badges der Konferenzen CAV, TACAS und ATVA erhalten. Ein Webinterface[3] ermöglicht einen einfachen und interaktiven Zugang zu einer Auswahl an Funktionalitäten der im Rahmen der Arbeit entwickelten Kommandozeilen-Werkzeuge.

## 2 Hintergrund

### 2.1 Modellprüfung (Model Checking)

Die Modellprüfung ist ein Verfahren zur Verifikation von nebenläufigen Systemen mit endlichem Zustandsraum, das in den frühen 1980er Jahren von Clarke und Emerson [CE81] sowie von Queille und Sifakis [QS82] unabhängig voneinander entwickelt wurde. Es basiert auf einem *Modell*, welches beschreibt, *wie* sich das System *tatsächlich* verhält, und einer *Spezifikation*, die vorschreibt *was* das *gewünschte* Verhalten des Systems sein soll. Mit diesen Eingabeparametern kann ein Programm, der *Modellprüfer*, daraufhin vollautomatisch feststellen, ob das Modell die Spezifikation erfüllt. Falls eine Anforderung verletzt ist, kann

[3] `http://adam.informatik.uni-oldenburg.de`

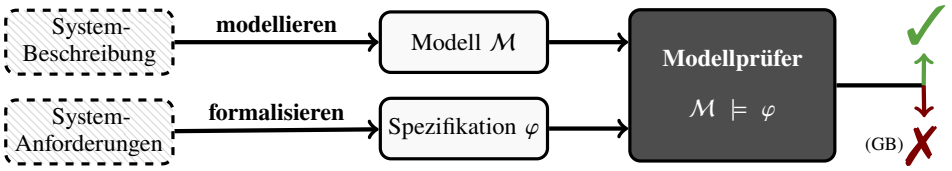


Abb. 1: Schematischer Überblick über das Verfahren der Modellprüfung. Zunächst wird ein mathematisch präzises Modell aus dem tatsächlichen System erstellt und die Anforderungen an das System werden in einer mathematisch eindeutigen Spezifikation formalisiert. Bei Ausführung prüft der Modellprüfer vollautomatisch, ob das Modell die Spezifikation erfüllt. Im negativen Fall wird ein Gegenbeispiel (GB) erstellt, welches das spezifikationsverletzende Verhalten beschreibt.

ein Gegenbeispiel automatisch geliefert werden. Dies ermöglicht eine einfachere Diagnose des fehlerhaften Verhaltens. Dieser Ablauf ist in Abb. 1 schematisch dargestellt.

Die Modelle dieser Arbeit basieren auf *Petri-Netzen*, einem etablierten Modell für asynchrone verteilte Systeme. Ein Petri-Netz ist ein gerichteter bipartiter Graph mit Plätzen, die den aktuellen Zustand eines Prozesses im System speichern, und Transitionen, die die Zustandsänderungen ausführen. Marken stellen die nebenläufigen Prozesse des Systems dar und werden durch die Transitionen von Platz zu Platz bewegt.

Als Spezifikationssprache werden häufig temporale Logiken verwendet. Eine temporale Logik erweitert die klassische Aussagenlogik um Operatoren, die sich auf das Verhalten des Systems über die Zeit beziehen. Dadurch können wir Eigenschaften angeben wie „Eine unerwünschte Situation tritt nie ein“ oder „Unter bestimmten Umständen tritt ein bestimmtes Verhalten unendlich häufig auf“. Die Spezifikationssprachen dieser Dissertation bauen auf der *branching-time temporal logic (CTL$^*$)* [EH86] und ihren Fragmenten, die *linear-time temporal logic (LTL)* [Pn77] und der *computation tree logic (CTL)* [CE81] auf. Eine große Herausforderung bei der Definition von Spezifikationen besteht darin, alle Anforderungen, die das System erfüllen soll, abzudecken.

## 2.2 Synthese

Während Verifikationsansätze die Korrektheit einer gegebenen Implementierung in Bezug auf eine gegebene Spezifikation *prüfen*, *leiten* Syntheseansätze die Implementierung automatisch aus einer gegebenen Spezifikation ab oder stellen ihre Nichtimplementierbarkeit fest. Dies führt zu Implementierungen, die „correct-by-construction“ sind. Mit dem Syntheseansatz können sich Entwickler*innen also darauf konzentrieren, *was* die Merkmale des Systems sind, und nicht darauf, *wie* diese Merkmale durch die tatsächliche Implementierung realisiert werden.

Bei der *spielbasierten* Synthese wird das Syntheseproblem als unendliches Spiel zwischen zwei Spielern über einem endlichen Graphen formuliert. Eine Spielerin, welche das *System* repräsentiert, versucht die gegebene Spezifikation zu erfüllen, während die andere Spielerin,

welche die *Umgebung* repräsentiert, versucht die Spezifikation zu verletzen. Die Zustände des Spiels sind Knoten in einem Graphen, der als *Spielarena* bezeichnet wird. Jeder Knoten ist eindeutig entweder der Umgebung oder dem System zugeordnet. Ein Spiel auf der Arena läuft in Runden ab. In jeder Runde wählt die Spielerin, der der aktuelle Knoten zugeordnet ist, einen Nachfolgeknoten. Die Spielerin kann diese Entscheidung mit vollständigem Wissen über die Spielarena und alle bisherigen Spielzüge tätigen. In *reaktiven* Systemen, also in Systemen, wo es eine permanente Interaktion mit der Umgebung gibt, sind solche Spiele im Allgemeinen von unendlicher Dauer. Die hier betrachteten Spiele sind also unendlich im Sinne der Spielabläufe, aber endlich im Sinne des Zustandsraums, d. h. der Spielarena, auf der sie gespielt werden. Eine *Strategie* für die Systemspielerin, das Spiel gegen alle möglichen Züge der Umgebung zu gewinnen, entspricht einer Implementierung, die die Spezifikation erfüllt.

Für *synchrone* verteilte Systeme haben Pnueli and Rosner gezeigt, dass das Syntheseproblem im Allgemeinen unentscheidbar ist [PR90]. Finkbeiner und Schewe identifizieren „information forks“ in der Kommunikationsarchitektur als hinreichendes Kriterium für die Unentscheidbarkeit solcher Systeme [FS05]. Die Komplexität für entscheidbare Architekturen, dazu gehören z.B. „pipelines“ [PR90], „two-way pipelines“ und „one-way ring“-Architekturen [KV01], ist *nicht elementar* [FS05].

Für *asynchrone* verteilte Systeme baut der Syntheseteil der Arbeit auf *Petri-Spielen* [FO17] auf. Petri-Spiele sind ein Mehrspieler-Spielmodell mit einer lokalen Sicherheitsgewinnbedingung und einem *kausalitätsbasierten* Speichermodell, d.h. anfänglich wissen die Spieler nichts voneinander, nur während einer gemeinsamen Synchronisation tauschen sie ihre gesamte kausale Vergangenheit aus. Die Spieler des Spiels sind die Marken in einem klassischen Petri-Netz. In Petri-Spielen gibt es zwei Teams: die Umgebungsspieler und die Systemspieler. Das gemeinsame Ziel der Systemspieler ist es, so zusammenzuarbeiten, dass jeder Spieler seine lokale Sicherheitsspezifikation erfüllt und das gegen alle möglichen Verhaltensweisen der Umgebungsspieler. Eine erfolgreiche Strategie der Systemspieler besteht aus einer korrekten lokalen Steuerung für jeden Systemspieler. Die Suche nach solchen Lösungen ist nicht trivial, da die Spieler zu jedem Zeitpunkt des Spiels einen unterschiedlichen Informationsstand über die anderen Spieler und den globalen Zustand des Systems haben können. Darüber hinaus kann sich dieser Informationsstand, abhängig von den Entscheidungen der Spieler für Synchronisationen, im Laufe der Zeit dynamisch ändern. Obwohl das Problem im Allgemeinen unentscheidbar ist, gibt es Unterklassen von Petri-Spielen, bei denen das Problem der verteilten Synthese entscheidbar ist [FO17]. Bei Petri-Spielen mit einem Umgebungsspieler, einer begrenzten Anzahl von Systemspielern und einer lokalen Sicherheitsbedingung ist das Problem *EXPTIME-vollständig* [FO17].

## 3 Beitrag der Arbeit

Die Arbeit trägt zu der Entwicklung von asynchronen verteilten Systemen und deren Korrektheitsanalyse mit Hilfe von Modellprüfungs- und Syntheseverfahren bei. Zum einen

wird ein Modell und eine Spezifikationssprache für den lokalen Datenfluss von Prozessen in asynchronen verteilten Systemen entwickelt und Modellprüfungsalgorithmen zur Verifikation der Korrektheit solcher Systeme bereitgestellt. Zum anderen werden, aufbauend auf dem vorherigen Modell und der Spezifikationssprache, Eingabesprachen für die Synthese von asynchronen verteilten Systemen mit kausalem Speicher entwickelt. Des Weiteren werden Lösungsalgorithmen zur automatischen Generierung korrekter Implementierungen von lokalen Steuerungen für die jeweiligen Prozesse im System eingeführt. Die praktische Nutzbarkeit wird durch Werkzeugunterstützung für beide Teile untermauert.

### 3.1 Modellprüfung lokaler Datenflüsse

Wir führen ein neues Verfahren zur Modellprüfung von asynchronen verteilten Systemen mit lokalen Datenflüssen ein. Das Hauptziel besteht darin, eine übersichtliche Modellierungstechnik zu haben, die die globale Konfiguration des Systems und den lokalen Datenfluss der Prozesse im System trennt und gleichzeitig praktikable Modellprüfungsalgorithmen mit vertretbarer Komplexität bereitstellt.

Die neu eingeführten *Petri-Netze mit Transits* [Fi19] ermöglichen diese Trennung, indem sie die Flussrelation von klassischen Petri-Netzen um eine sogenannte *Transitrelation* erweitern. Die globale Konfiguration des Systems und dessen Steuerung werden mit einem klassischen Petri-Netz spezifiziert, während der lokale Datenfluss der Prozesse mit den Transits spezifiziert wird. Betrachten wir zum Beispiel ein Rechnernetzwerk mit Switches, die Datenpakete gemäß einer bestimmten Routing-Konfiguration weiterleiten. Der Routing-Prozess und die möglichen Aktualisierungen seiner Konfiguration stellen die globale Steuerung des Systems dar, während der Weiterleitungsprozess der unbegrenzten Anzahl von Paketen, die zu jederzeit in das Netz eintreten können, den Datenfluss darstellt.

Die neu eingeführte temporale Logik *Flow-CTL*$^*$ [Fi20b] erlaubt es uns, mit LTL das globale Verhalten des Systems zu spezifizieren, z. B. um Maximalitäts- und Fairnessannahmen zu definieren. Außerdem kann CTL$^*$ genutzt werden, um die Korrektheit des lokalen Datenflusses der entsprechenden Läufe zu spezifizieren. Schlussendlich ist eine Spezifikation aus einzelnen Anforderungen an den Datenfluss zusammensetzbar, die dabei abhängig von verschiedenen Kontrollläufen sein können. Im Netzwerkszenario können sich Entwickler*innen beispielsweise auf unterschiedliche nebenläufige Aktualisierungsroutinen für den Routing-Prozess mit separaten Spezifikationen konzentrieren. Die Modellprüfungsalgorithmen können dann z. B. sicher stellen, dass das Ausrollen dieser Aktualisierungen keine Paketverluste oder Routing-Schleifen im Netz verursacht.

Wir stellen spezielle Modellprüfungsalgorithmen für die verschiedenen Fragmente von Flow-CTL$^*$ zur Verfügung. Der Algorithmus für das Linear-Time-Fragment *Flow-LTL* [Fi19] hat eine *einfach-exponentielle* Zeitkomplexität. Viele Anforderungen realer Probleme lassen sich durch Spezifikationen mit linearer Zeit ausdrücken. Zum Beispiel werden die gewünschten Anforderungen des Netzwerkszenarios in der Arbeit mit Flow-LTL

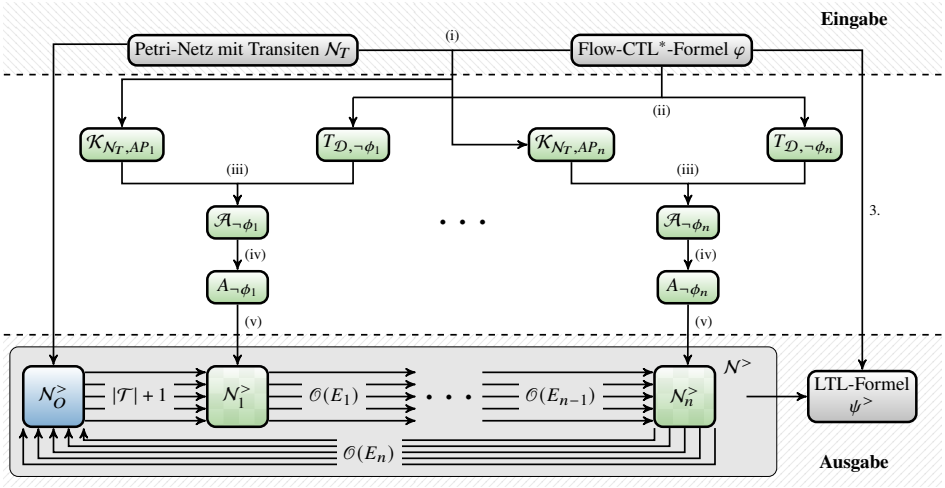


Abb. 2: Überblick über das Modellprüfungsverfahren: Für ein gegebenes sicheres Petri-Netz mit Transits $\mathcal{N}_T$ und eine Flow-CTL*-Formel $\varphi$, wird ein klassisches Petri-Netz $\mathcal{N}^>$ und eine LTL-Formel $\psi^>$ erstellt: Für jede Datenfluss-Teilformel $\mathbb{A}\,\phi_i$, wird *(i)* die beschriftete Kripke-Struktur $\mathcal{K}_{\mathcal{N}_T,AP_i}$ und *(ii)* der alternierende Baumautomat $T_{\mathcal{D},\neg\phi_i}$ konstruiert, daraus wird der alternierende Wortautomat $\mathcal{A}_{\neg\phi_i} = \mathcal{K}_{\mathcal{N}_T,AP_i} \times T_{\mathcal{D},\neg\phi_i}$ erzeugt, und daraus *(iv)* der Büchi-Automat $A_{\neg\phi_i}$ mit Kanten $E_i$, welcher dann in ein Petri-Netz $\mathcal{N}_i^>$ umgewandelt wird. Diese Teilnetze werden so zu einem Gesamt-Petri-Netz $\mathcal{N}^>$ zusammengesetzt, dass sie für jede im originalen Netz gefeuerte Transition sequentiell aktiviert werden. Die konstruierte Formel $\psi^>$ überspringt für den globalen Teil von $\varphi$ diese sequentiellen Schritte und prüft die Akzeptanz des erratenen Datenflussbaums für jeden Automaten im entsprechenden Teilnetz. Das anfängliche Modellprüfungsproblem wird dann durch Prüfung von $\mathcal{N}^> \models_{\mathrm{LTL}} \psi^>$ gelöst.

spezifiziert. Die gewünschten Anforderungen für die andere in der Arbeit vorgestellte Anwendungsdomäne, die Zugangskontrolle für Gebäude, werden durch das Branching-Time-Fragment *Flow-CTL* [Fi20b] ausgedrückt. Hier stellt der Datenfluss die möglichen Wege von Personen in einem Gebäude dar. Der entsprechende Modellprüfungsalgorithmus hat eine *doppelt-exponentielle* Zeitkomplexität. Für die vollständige Flow-CTL*-Logik hat der bereitgestellte Algorithmus eine *dreifach-exponentielle* Zeitkomplexität.

Wir reduzieren das Problem, ob ein Petri-Netz mit Transits eine Flow-CTL*-Formel erfüllt auf das Problem, ob ein klassisches Petri-Netz eine LTL-Formel erfüllt. Für Flow-CTL* und Flow-CTL, basiert diese Reduktion auf einer Folge von Automatenkonstruktionen für jede einzelne Anforderung an den Datenfluss. Eine Übersicht über die Reduktion ist in Abb. 2 gegeben. Für Flow-LTL stellen wir zwei verschiedene Algorithmen zur Verfügung, die beide die komplexe Automatenkonstruktionen vermeiden. Bei der ersten Konstruktion handelt es sich um einen Algorithmus mit *einfach-exponentieller* Laufzeit, der die nötigen Teilnetze für jede einzelne Anforderung an den Datenfluss in einer sequentiellen Reihenfolge verknüpft [Fi19]. Die zweite Konstruktion verknüpft die Teilnetze parallel, was zu einem Algorithmus mit *doppelt-exponentieller* Laufzeit für Spezifikationen mit mehr als einer einzelnen Anforderung an den Datenfluss führt [Fi20a]. Der zweite Exponent ist jedoch nur

von der Anzahl der in der Formel verwendeten separaten Anforderungen an den Datenfluss abhängig. Bei den Beispielen aus dem Netzwerkszenario, deren Spezifikationen nur wenige solcher Anforderungen enthalten, ist dieser Ansatz in der Praxis – trotz schlechterer theoretischer Komplexität – immer noch deutlich schneller als der sequentielle Ansatz.

Schlussendlich reduzieren wir das Modellprüfungsproblem für sichere Petri-Netze und LTL mit Stellen und Transitionen als atomare Präpositionen auf ein Hardware-Modellprüfungsproblem, indem wir das Petri-Netz in einen Schaltkreis kodieren [Fi19]. Auf diese Weise können wir die aktuellen Algorithmen und Werkzeugboxen, die es in diesem Umfeld gibt, nutzen um unser anfängliches Modellprüfungs-Problem zu lösen. Die Korrektheit aller Reduktionen ist mit Hilfe ausführlicher Beweise nachgewiesen. Die Implementierung der Algorithmen in dem Tool AdamMC [Fi20a; GHY21] ermöglichte es, Beispielspezifikationen für nebenläufige Aktualisierungsroutinen in Netzwerktopologien zu verifizieren und falsifizieren. Das Tool konzentriert sich auf die Ansätze bezüglich der Flow-LTL-Spezifikationen und bietet aktuell für Flow-CTL-Spezifikationen nur Algorithmen in einem frühen Entwicklungsstadium an.

### 3.2 Synthese von Verteilten System mit lokalen Bedingungen

Auf der Grundlage der Ergebnisse des vorherigen Teils, geht dieser Teil von der Verifikation zur Synthese über. Wir stellen ein neues Verfahren für die Synthese von asynchronen verteilten Systemen mit kausalem Speicher und lokalen Datenflüssen vor. Das Modell, genannt *Petri-Spiele mit Transits*, kombiniert die Eigenschaften von Petri-Spielen [FO17] und Petri-Netzen mit Transits [Fi19]. In Petri-Spielen tragen die Marken beim Fließen durch das Netz Informationen über ihre kausale Vergangenheit mit sich. Diese Informationen können von den Spieler*innen verwendet werden, um sich für die nächsten Züge zu entscheiden. In Petri-Spielen mit Transits definiert der Datenfluss eine weitere Ebene des Informationsflusses. Diese Ebene wird verwendet, um die Anforderungen an die Korrektheit des Systems zu spezifizieren.

Für die neuen Gewinnbedingungen erlauben wir eine *existentielle* Sicht (*gibt* es im System einen Fluss, der das Ziel erfüllt?) und eine *universelle* Sicht (erfüllen *alle* Flüsse im System das Ziel?). Wir definieren Sicherheits-, Erreichbarkeits-, Büchi-, Co-Büchi- und Paritätsbedingungen jeweils in beiden Sichten. Außerdem beschränken wir die Spezifikationssprache Flow-LTL [Fi19] auf ihr lokales Fragment und erweitern sie um die existentielle Sicht.

Wir lösen das Syntheseproblem mit lokalen Spezifikationen für 1-beschränkte Petri-Spiele mit Transits, die einen Umgebungsspieler, eine begrenzte Anzahl von Systemspielern und keine gemischte Kommunikation haben. Letztere Eigenschaft bedeutet, dass eine Systemspielerin niemals eine Kommunikation mit der Umgebung anbieten und gleichzeitig ein selbständiges Fortschreiten mit anderen Systemspielerinnen erlauben darf. Dies ermöglicht es uns, erstmals Petri-Spiele mit Gewinnbedingungen zu lösen, die über Sicherheitsanforderungen hinausgehen. Darüber hinaus können in Petri-Spielen mit Transits eine

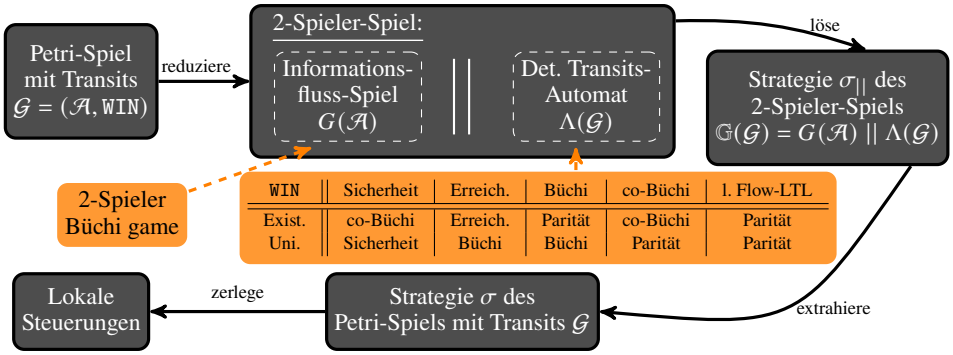


Abb. 3: Ein Überblick über das Entscheidungsverfahren für Petri-Spiele mit Transits und lokalen Gewinnbedingungen. Aus dem Petri-Spiel mit Transits $\mathcal{G} = (\mathcal{A}, \texttt{WIN})$ wird zunächst das Produkt aus einem Büchi-Spiel mit zwei Spielern $G(\mathcal{A})$, das die generellen Eigenschaften einer Strategie abbildet, und dem deterministischen Transits-Automaten $\Lambda(\mathcal{G})$, der die lokale Gewinnbedingung $\texttt{WIN}$ des Datenflusses abbildet, gebildet. Die unterschiedlichen Gewinnbedingungen von $\mathcal{G}$ führen zu unterschiedlichen Akzeptanzbedingungen von $\Lambda(\mathcal{G})$. Eine Strategie $\sigma_{||}$ für den Spieler 0 des Produktspiels kann (falls vorhanden) durch Anwendung von klassischen Spiellösungs-Verfahren ermittelt werden. Diese Strategie wird verwendet, um die Gewinnstrategie $\sigma$ für die Systemspieler von $\mathcal{G}$ zu ermitteln. Wenn in der Arena $\mathcal{A}$ jede Stelle zu einem passenden Prozess zugeordnet wurde, dann kann $\sigma$ leicht in die lokalen Steuerungen der einzelnen Prozesse zerlegt werden.

unbeschränkte Anzahl von Datenflüssen modelliert werden. Eine unbeschränkte Anzahl von Marken macht Petri-Spiele [FO17] (und damit auch Petri-Spiele mit Transits) dagegen unentscheidbar. Damit bieten Petri-Spiele mit Transits die Möglichkeit, unbeschränkte Merkmale zu modellieren und haben dabei trotzdem ein entscheidbares Syntheseproblem.

Das Syntheseproblem für Petri-Spiele mit Transits und lokalen Gewinnbedingungen wird auf das Syntheseproblem eines Zwei-Spieler-Spiels über einem endlichen Graphen mit vollständiger Information reduziert. Der Lösungsalgorithmus reduziert das komplizierte kausale Speichermodell der Spieler im Petri-Spiel mit Transits auf ein vollständiges Informationsmodell, das von der Gewinneigenschaft der Strategie losgelöst ist. Dies ermöglicht es uns, die allgemeine Existenz einer Strategie für die Systemspieler im Petri-Spiel mit Transits auf die Lösung eines Büchi-Spiels mit zwei Spielern zu reduzieren, während die Gewinneigenschaft der Strategie auf die Akzeptanz durch geeignet konstruierte Automaten reduziert wird. Diese Trennung und die Generalität der Beweise trägt zur Allgemeinheit des vorgestellten Lösungsalgorithmuses bei und ermöglicht eine einfachere Erweiterbarkeit auf weitere Gewinnbedingungen. Ein Überblick über die Reduktion ist in Abb. 3 dargestellt. Die Komplexität des Synthesealgorithmuses hängt von der Gewinnbedingung des Petri-Spiels mit Transits ab. Für die existentiellen und universellen Gewinnbedingungen ist das Syntheseproblem für Petri-Spiele mit Transits *EXPTIME-vollständig*. Für lokale Flow-LTL Gewinnbedingungen ist die Komplexität *einfach-* bzw. *doppelt-exponentiell* in der Größe des Petri-Spiels mit Transits und *doppelt-* bzw. *dreifach-exponentiell* in der Größe der

Formel. Dies ist abhängig davon, ob in der Formel eine Mischung aus existentiellen und universellen lokalen Datenflussspezifikationen verwendet wird.

Das Werkzeug AdamSYNT [FGO15; Fi17; GHY21] bietet BDD-basierte Algorithmen zum Lösen des verteilten Syntheseproblems für 1-beschränkte Petri-Spiele mit einem Umgebungsspieler, einer begrenzten Anzahl von Systemspielern, keiner gemischten Kommunikation und einer lokalen Sicherheitsbedingung. Darüber hinaus haben wir Algorithmen für High-Level-Petri-Spiele [GO21] (eine kompakte Darstellung von Mengen von Petri-Spielen) implementiert, welche die Symmetrien im System [GOW20; GW21] (die theoretischen Ergebnisse sind nicht Teil dieser Arbeit) ausnutzen. BDD-basierte Algorithmen für Teilklassen von Petri-Spielen mit Transits sind in einem frühen Entwicklungsstadium.

## 4 Schlussfolgerung

Diese Arbeit leistet einen Beitrag zur *korrekten* Entwicklung von *asynchronen verteilten Systemen* durch zwei sich ergänzende Ansätze: *Modellprüfung* und *Synthese*. Für beide Ansätze haben wir neue *Modellierungs-* und *Spezifikations*-Formalismen entwickelt, die es ermöglichen, Anforderungen an den unbeschränkten Datenfluss in asynchronen verteilten Systemen zu stellen. Die Arbeit stellt bewiesen korrekte *Lösungsalgorithmen* für die entsprechenden Modellprüfungs- und Syntheseprobleme zur Verfügung, die trotz der unbegrenzten Eigenschaften des Datenflusses und des unvollständigen Wissens der Systemkomponenten über die Umgebung in asynchronen verteilten Systemen eine angemessene Zeitkomplexität aufweisen. Die Implementierungen dieser Algorithmen behandeln das Problem der Zustandsraumexplosion, welches sich aus den unterschiedlichen Ausführungsreihenfolgen der asynchronen Komponenten in einem verteilten System ergibt, mit Hilfe einer Reduktion auf ein Hardware-Modellprüfungsproblem bzw. durch symbolische Algorithmen.

## Literaturverzeichnis

[CE81] Clarke, E. M.; Emerson, E. A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logics of Programs. S. 52–71, 1981.

[EH86] Emerson, E. A.; Halpern, J. Y.: “Sometimes” and “Not Never” revisited: on branching versus linear time temporal logic. J. ACM 33/1, S. 151–178, 1986.

[FGO15] Finkbeiner, B.; Gieseking, M.; Olderog, E.: Adam: Causality-Based Synthesis of Distributed Systems. In: Proc. of CAV. S. 433–439, 2015.

[Fi17] Finkbeiner, B.; Gieseking, M.; Hecking-Harbusch, J.; Olderog, E.: Symbolic vs. Bounded Synthesis for Petri Games. In: Proc. of SYNT. S. 23–43, 2017.

[Fi19] Finkbeiner, B.; Gieseking, M.; Hecking-Harbusch, J.; Olderog, E.: Model Checking Data Flows in Concurrent Network Updates. In: Proc. of ATVA. S. 515–533, 2019.

[Fi20a] Finkbeiner, B.; Gieseking, M.; Hecking-Harbusch, J.; Olderog, E.: AdamMC: A Model Checker for Petri Nets with Transits against Flow-LTL. In: Proc. of CAV. S. 64–76, 2020.

[Fi20b] Finkbeiner, B.; Gieseking, M.; Hecking-Harbusch, J.; Olderog, E.: Model Checking Branching Properties on Petri Nets with Transits. In: Proc. of ATVA. S. 394–410, 2020.

[FO17] Finkbeiner, B.; Olderog, E.: Petri games: Synthesis of distributed systems with causal memory. Inf. Comput. 253/, S. 181–203, 2017.

[FS05] Finkbeiner, B.; Schewe, S.: Uniform Distributed Synthesis. In: Proc. of LICS. S. 321–330, 2005.

[GHY21] Gieseking, M.; Hecking-Harbusch, J.; Yanich, A.: A Web Interface for Petri Nets with Transits and Petri Games. In: Proc. of TACAS. S. 381–388, 2021.

[Gi22] Gieseking, M.: Correctness of Data Flows in Asynchronous Distributed Systems - Model Checking and Synthesis, Diss., University of Oldenburg, 2022.

[GO21] Gieseking, M.; Olderog, E.: High-Level Representation of Benchmark Families for Petri Games. In: Model Checking, Synthesis, Learning. S. 115–137, 2021.

[GOW20] Gieseking, M.; Olderog, E.; Würdemann, N.: Solving high-level Petri games. Acta Informatica 57/3-5, S. 591–626, 2020.

[GW21] Gieseking, M.; Würdemann, N.: Canonical Representations for Direct Generation of Strategies in High-Level Petri Games. In: Proc. of PETRI NETS. S. 95–117, 2021.

[KV01] Kupferman, O.; Vardi, M. Y.: Synthesizing Distributed Systems. In: Proc. of LICS. S. 389–398, 2001.

[Pn77] Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science. S. 46–57, 1977.

[PR90] Pnueli, A.; Rosner, R.: Distributed Reactive Systems Are Hard to Synthesize. In: Proc. of FOCS. S. 746–757, 1990.

[QS82] Queille, J.; Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: International Symposium on Programming. S. 337–351, 1982.

**Manuel Gieseking** hat sein Studium an der Hochschule Bremen als Medieninformatiker begonnen. Die theoretischen Inhalte verlockten jedoch zu sehr, sodass er an die Carl von Ossietzky Universität nach Oldenburg wechselte, um dort Mathematik und Informatik zu studieren. Nach seinem Master in der Informatik hat er dort auch seine Promotion absolviert. Neben der großen Freude an der Lehre hat er während seiner Promotion die theoretischen Ergebnisse immer wieder gerne in Werkzeuge gegossen, die sich quelloffen gesammelt auf folgender GitHub-Seite befinden: `https://github.com/adamtool`.