Efficient Verification of Fault-Tolerant Message-Passing Protocols

Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
eines Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Péter Bokor

aus Budapest, Ungarn

Referenten:
Prof. Neeraj Suri, Ph.D.
Prof. Péter Csaba Ölveczky, Ph.D.

Datum der Einreichung:
15. November 2011
Datum der mündlichen Prüfung:
29. November 2011

Darmstadt 2011
Hochschulkennziffer: D17

Apunak
(For my dad)

Summary

This thesis deals with efficient formal verification of fault-tolerant distributed protocols. The main focus is on protocols that achieve fault-tolerance using replication in distributed systems [AW04]. In addition, communication in the distributed system is abstracted using message-passing. However, most of the concepts and solutions discussed in the thesis apply beyond replication-based message-passing protocols.

The outcome of verification is two-fold: Either the verification proves that the system (protocol) satisfies its specification or it returns (or claims the existence) of a counterexample that witnesses that the system (protocol) violates the specification. Both the development and application of fault-tolerant message-passing protocols can benefit from verification. Firstly, these protocols can be complex conceptual designs and hard to implement due to (i) a rich variety of (even malicious [LSP82]) faults that should be tolerated by the protocol and (ii) the concurrency in the distributed system executing the protocol. Therefore, counterexamples returned by verification in the phase of development can help developers to get the conceptual protocol and its implementation right. Secondly, fault-tolerant messages-passing protocols specify strong guarantees such as atomic broadcast [JRS11] or diagnosis of malicious faults [SBS+11]. Therefore, the verification of such protocols can avoid failures of highly available systems that build upon these protocols.

The complexity of verification strongly depends on the size and nature of the system. Therefore, verification should be *efficient* in terms of space, time, and human interaction (ranging from full automation to requiring intuitive human guidance). Due to (i) and (ii), the verification of fault-tolerant message-passing protocols faces with a large problem space. Hence, straightforward verification approaches are inefficient. The thesis enables efficient verification of fault-tolerant message-passing systems in several ways.

New Models of Message-Passing Systems The input of verification is a *model* of the system. The model represents the system and can be of varying resolution, e.g., source code, binaries, or high-level executable pseudocode. The efficiency of verification can significantly differ for different models of the same system. In the first part of the thesis, models of fault-tolerant message-passing protocols are proposed to enable efficient verification.

The proposed models address different aspects of fault-tolerant message-passing protocols. These aspects include (A1) *message traffic* – an equivalence is shown between different models of sending, delivering, and consuming messages, which allows to select the model that is most amenable to efficient

verification; (A2) *fault-model* – a surprisingly simple and sound model of the widely-applied *crash* fault assumption is proposed allowing efficient verification of systems with crash faults; (A3) *symmetries* – an approach for finding symmetries in the model is introduced, which can be exploited by symmetry reduction [MDC06]; and (A4) *partial orders* – an equivalent translation of models is proposed to improve on the efficiency of partial-order reduction [God96, Val98, CGP99]. Symmetry and partial-order reductions are general tools for efficient verification whose efficiency strongly depends on the model in use.

New Algorithms for Efficient Verification The second part of the thesis improves on existing verification techniques to enhance their efficiency.

The first technique is partial-order reduction, whose performance is limited by the poor *flexibility* and *usability*, and high *time overhead* of existing implementations. A new implementation of partial-order reduction is presented, called *LPOR*, that improves on all these features at the same time. To demonstrate the features of LPOR, it is applied to general message-passing systems. Experiments with representative fault-tolerant message-passing protocols justify LPOR's improvements. In addition, LPOR is implemented as an open-source Java library. This implementation is applied in MP-Basset, a model checker for general message-passing systems developed in the context of this thesis.

The second verification technique is automated induction [MP95, dMRS03]. The efficiency of induction is limited by its general *incompleteness*, i.e., induction might return spurious (wrong) counterexamples for systems that satisfy the specification. A solution against spurious counterexamples is using *lemmas* [MP95]. However, the discovery of lemmas is, in general, hard to automate. The thesis proposes a *classification* of lemmas in fault-tolerant message-passing protocols to enable their automated discovery. An example protocol is verified using classified lemmas and machine-checked induction proofs. Finally, a new approach, called *strengthened transitions*, is proposed to combat spurious counterexamples. An application of strengthened transitions for general multi-process systems is also presented.

Kurzfassung

Diese Dissertation behandelt effiziente, formale Verifikation von fehlertoleranten, nachrichtenbasierten Protokollen. Der Schwerpunkt liegt dabei auf Protokollen, die Fehlertoleranz mittels Replikation in verteilten Systemen [AW04] erreichen. Die Kommunikation in verteilten Systemen wird als Versenden von Nachrichten abstrahiert. Jedoch gelten die meisten, in dieser Dissertation vorgestellten Konzepte und Techniken auch über replizierte, nachrichtenbasierte Systeme hinaus.

Verifikation gibt zwei Arten von Ergebnissen aus: Verifikation beweist entweder, dass das System (Protokoll) seine Spezifikation erfüllt oder zeigt ein Gegenbeispiel (bzw. behauptet dessen Existenz) als Nachweis, dass die Spezifikation verletzt wird. Verifikation findet sowohl in der Entwicklung als auch in der Anwendung fehlertoleranter, nachrichtenbasierter Protokolle Verwendung. Erstens sind diese Protokolle aufgrund (i) der großen Bandbreite von (möglicherweise bösartigen [LSP82]) Fehlern, die vom Protokoll toleriert werden müssen, und (ii) der Nebenläufigkeit im verteilten System, das zur Ausführung des Protokolls verwendet wird, häufig komplex. Daher können die von der Verifikation gelieferten Gegenbeispiele dabei helfen, Protokolle korrekt zu entwerfen und zu implementieren. Zweitens, werden in fehlertoleranten, nachrichtenbasierten Protokollen starke Eigenschaften wie Atomic-Broadcast [JRS11] oder die Diagnose von bösartigen Fehlern [SBS+11] spezifiziert. Somit kann Verifikation Ausfälle von hochverfügbaren Systemen vermeiden, die auf solchen Protokollen basieren.

Die Komplexität von Verifikation hängt stark von der Größe und Art des Systems ab. Deshalb ist die *Effizienz* von Verifikation hinsichtlich Speicherbedarf, Zeit und menschlicher Interaktion (von Vollautomatisierung bis hin zu intuitivem Eingreifen eines Menschen) besonders wichtig. Die oben genannten Gründe (i) und (ii) implizieren einen sehr großen Problemraum für die Verifikation fehlertoleranter, nachrichtenbasierte Protokolle. Einfache, naheliegende Verifikations-Ansätze sind somit häufig ineffizient. Diese Dissertation ermöglicht effiziente Verifikation fehlertoleranter, nachrichtenbasierter Protokolle auf verschiedenen Wegen.

Neue Modelle von nachrichtenbasierten Systemen Verifikation benötigt als Eingabe ein *Modell* des Systems. Das Modell repräsentiert das System mit unterschiedlichem Detaillierungsgrad, wie zum Beispiel Quellcode, Maschinencode oder ausführbarer Pseudocode. Die Effizienz der Verifikation kann signifikant vom verwendeten Modell des Systems abhängen. Im ersten Teil der Dissertation werden Modelle von fehlertoleranten, nachrichtenbasierten Protokollen vorgeschlagen, die eine effiziente Verifikation ermöglichen.

Die vorgestellten Modelle behandeln verschiedene Aspekte von fehlertoleranten, nachrichtenbasierten Protokollen. Diese Aspekte umfassen: (A1) *Nachrichtenverkehr* - Es wird die Equivalenz von verschiedenen Modellen von Senden, Empfang und Verarbeitung von Nachrichten gezeigt, sodass jeweils das Modell, das für eine effiziente Verifikation am besten geignet ist, verwendet werden kann; (A2) *Fehler-Modell* - Es wird ein überraschend einfaches und korrektes Modell der häufig verwendeten Crash-Fehler-Annahme eingeführt, das eine effiziente Verifikation von Systemen mit solchen Fehlern erlaubt; (A3) *Symmetrien* - Es wird ein Verfahren vorgestellt um Symmetrien im Modell zu finden, die von der Symmetry-Reduction-Technik [MDC06] genutzt werden; (A4) *Partielle Ordnung* - Es wird eine equivalente Transformation von Modellen gezeigt, die die Effizienz von Partial-Order-Reduktion [God96, Val98, CGP99] verbessert. Symmetrie- und Partial-Order-Reduktion sind allgemeine Verfahren für effizientere Verifikation, deren Effizienz stark vom gewählten Modell abhängt.

Neue Algorithmen für effiziente Verifikation Der zweite Teil der Dissertation verbessert bestehende Verifikationstechniken hinsichtlich ihrer Effizienz.

Die erste Technik ist Partial-Order-Reduktion, deren Effizienz durch mangelnde *Flexibilität*, *Bedienbarkeit* und die hohe *Zeitkomplexität* der bestehenden Implementierungen beschränkt wird. Eine neue Implementierung von Partial-Order-Reduktion, genannt *LPOR*, wird präsentiert, die alle diese Eigenschaften gleichzeitig verbessert. Um die Eignung dieses Ansatzes zu demonstrieren, wird LPOR auf nachrichtenbasierte Systeme angewendet. Experimente mit repräsentativen, fehlertoleranten, nachrichtenbasierten Protokollen validieren die durch LPOR erzielten Verbesserungen. LPOR ist in Java implementiert und mit Quelltext frei verfügbar. Diese Implementierung ist in MP-Basset integriert, ein Model-Checker für nachrichtenbasierte Systeme, der im Rahmen dieser Dissertation entwickelt wurde.

Die zweite Technik ist automatisierte Induktion [MP95, dMRS03]. Die Effizienz von Induktion ist dadurch beschränkt, dass sie *unvollständig* ist, das heißt, Induktion kann für Systeme, die ihre Spezifikation erfüllen, falsche Gegenbeispiele ausgeben. Eine Lösung gegen falsche Gegenbeispiele sind *Lemmata* [MP95]. Das Auffinden von Lemmata ist allerdings nur schwer automatisierbar. Diese Dissertation schlägt eine *Klassifizierung* von Lemmata in fehlertoleranten, nachrichtenbasierten Protokollen vor, um automatisches Finden von Lemmata zu ermöglichen. Ein Beispielprotokoll wird durch klassifizierte Lemmata und automatisierte Induktion verifiziert. Zuletzt wird ein neuer Ansatz, genannt *gestärkte Transitionen*, vorgestellt, um falsche Gegenbeispiele zu vermeiden. Eine Anwendung von gestärkten Transitionen auf allgemeine Multi-Prozess-Systeme wird ebenfalls erläutert.

Acknowledgements

As I recall, the first time I really used my mind was when, as an undergrad at Technical University of Budapest, we had to sketch Lovász's proof of the Perfect Graph Theorem. Using my mind was fun! The next time I used my mind was, years after, during my time as a PhD candidate. Who knows, maybe I'd have never used my mind again if I hadn't taken this way. So I'm happy that I did so.

My dad was the greatest man I know. He also was a great engineer, who was sceptical about his son doing research in computer science. As time went by, he started accepting and respecting me doing so, even though I still didn't know how a transistor really worked. I wish he could be here to see me graduating. I know, he would be proud. My dear mom and great sister have always been there for me throughout these years. Thank you very much!

My advisor, Neeraj has always supported and believed in me, even at times when I didn't believe in myself. He always granted me freedom in research and, at the same time, requested quality of the work I do. As a result, I've learnt how to be an architect of my own ideas and how to turn them into a structured and well-defined project. I'm deeply appreciative of his support to make this happen.

Prof. András Pataricza has given me the first drive towards research. He's a great teacher and I'm thankful for him that I could participate in his lectures. In his group, I learned much from fellow graduate students and friends Áki, Dancsi, and István.

It was a once in a lifetime experience to get colleagues and close friends with Marco. He taught me about music, research, and life. Meeting him and his great girls, Ilaria and Simmy, has made my life in Darmstadt to be a great story.

I'm very thankful to the whole DEEDS group! The mothers of DEEDS, Sabine, Ute, and Birgit, and other colleagues Andreas, Piotr, Dinu, Stefan, Vinay, Arda, Jesus, and all others have been my spare family for every day work, funny lunches, barbecues, and Stammtisches.

My thesis would be a different one without Matthias and Dan, whose constructive criticism and insights about distributed computing were always an important and useful feedback for me. Similar applies to Johannes and Florian from Prof. Helmut Veith's group, whose background has shaped my view of formal methods. Our discussions alternated between work and fun during lunch, dinner, and party time in Darmstadt and across borders starting from Budapest, over Belgrad, Sarajevo, Munich, Elche, Granada, Barcelona, and Lisbon, to Princeton. It was also a great experience to work with Prof. Sandeep Shukla.

I'm very lucky to get to know my good friend Lorenz. I feel that our discussions and countless trips to Frankfurt are somehow part of my thesis. Just like my friends from Frankfurt, Jeca, Doro, Boki, who was my patient advisor in the science of techno, Daniel, and last but not least Mr. Robert Johnson and his ultra bass.

Finally, many thanks for Gyuri and Kati for the Hungarian jokes, superb dishes, and the subtle analysis of being Hungarian abroad.

Péter Bokor
Darmstadt, October 30, 2011

Contents

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Modeling Message-Passing Systems 3
1.2 Efficient Verification Algorithms 5
1.3 Thesis Structure and Resulting Publications 7

2 Preliminaries **11**
2.1 State Transition Systems and Invariants 12
2.2 Stuttering Equivalence 13
2.3 Consensus with Paxos 13
2.4 Symmetry Reduction with Scalarsets 15
2.5 Partial-Order Reduction with Stubborn Sets 15
2.6 k-Induction Proofs 16

3 Models of Message-Passing Systems **19**
3.1 Models of Message Traffic 21
3.1.1 Reference Model 21
3.1.2 Generalized Reference Model 23
3.1.3 The Proposed Model 23
3.1.4 Equivalence of Models 24
3.2 Models of Crash-Faulty Processes 26
3.2.1 A Model of Crash-Faulty Processes 27
3.2.2 Equivalence basis 28
3.2.3 Equivalence of Explicit and Implicit Models of Crash-Faulty Processes 29
3.3 Finding Symmetries with Decomposition 31
3.3.1 Motivating Example 31
3.3.2 Formalizing Decomposition and Replicas 32

3.3.3 Finding Symmetries via Decomposition 34
3.4 Quorum Transitions 37
3.5 Transition Refinement for Efficient POR 38
3.5.1 Generalized Transition Refinement 39
3.5.2 Strategy 1: Quorum Split 40
3.5.3 Strategy 2: Reply Split 42
3.6 Experiments 43
3.6.1 Symmetry Reduction with Murφ 43
3.6.2 Partial-Order Reduction with MP-Basset 47
3.7 Related Work 51
3.8 Conclusions 53

4 Local Partial-Order Reduction 55
4.1 Positioning LPOR 55
4.2 The LPOR Interface: Interfering Transitions 58
4.3 The LPOR Stubborn Set Algorithm 59
4.3.1 The Stubborn Set Algorithm 60
4.3.2 Optimizations and Possible Extensions 65
4.3.3 Preserving Temporal Logics with LPOR 66
4.4 A Message-Passing Instantiation of LPOR 67
4.4.1 Extended Syntax of Message-Passing System Models . 67
4.4.2 LPOR Relations for Message-Passing Systems 68
4.5 Java-LPOR: An LPOR Implementation 69
4.6 Evaluating LPOR 71
4.7 Related Work 76
4.8 Conclusions 77

5 Induction in Distributed Protocols 79
5.1 Classification and Discovery of Lemmas 80
5.2 Strengthened Transitions 85
5.2.1 Motivating Example 85
5.2.2 Strengthened Transitions: A General Framework 88
5.2.3 Transition Validated Strengthened Guards 89
5.3 Related Work 91
5.4 Conclusions 92

6 The MP-Basset Model Checker 93
6.1 Basic Architecture: The Basset Model Checker 94
6.2 Feature 1: Quorum Transitions 95
6.3 Feature 2: Integrating Java-LPOR 97
6.4 Conclusions 98

7 Conclusions **99**
7.1 Generalizations 99
7.2 Related Open Questions 100

A Models of Message-Passing Systems **103**
A.1 Proof of Equivalent Models of Message Traffic 103
A.2 Proof of Equivalent Models of Crash Faults 106

B Local Partial-Order Reduction **109**
B.1 Generalized LPOR 109
B.2 Proof of Generalized LPOR 111
B.3 Proofs of LPOR Relations for Message-Passing 114
B.4 Enriched Syntax of Message-Passing System Models 117

Bibliography **118**

Curriculum Vitae **125**

List of Figures

1.1 The inputs and outputs of verification. 2

2.1 Example stuttering equivalent paths. 13

3.1 Proof sketch of Theorem 1. 26
3.2 (a) Unrefined transition t. (b) Refined transitions t_1 and t_2. (c) Overly refined transitions. 39

4.1 Non-interfering transitions t_1 and t_2. States s_{12} and s_3 are deadlocks. 56
4.2 A Petri net example. 63
4.3 Illustration of the proof of Theorem 5. 65

5.1 Overview of proving agreement with induction. 82
5.2 Example message-flow with processes i, j, k and l in consensus with failure detectors. 83
5.3 SAL specification of the Bakery protocol. 86

6.1 MP-Basset architecture illustration. 94
6.2 Example quorum transition specified in MP-Basset: Phase 2(a) of Paxos consensus. 96

B.1 Illustration of the proof of Theorem 10 113
B.2 Illustration of the proof of Lemma 7. 116

List of Tables

3.1 Overview of different message traffic models. 23
3.2 Algorithm 1 in comparison with the naive approach using the Murφ model checker. 46
3.3 Quorum and reply split in comparison with unsplit transitions using the MP-Basset model checker. 49

4.1 Performance results of LPOR implemented within MP-Basset. 75

5.1 Description of lemmas used in an induction proof of agreement. 82

Chapter 1

Introduction

Verification is the process of proving that a system meets its specification (Figure 1.1). In this case, the system is said to be correct (incorrect otherwise). The specification is a set of requirements against (properties of) the system. Examples of specifications are fault-tolerance or performance. Optionally, verification can return a counterexample, which is a witness that the system is incorrect. In practice, the verification process requires as input an abstract representation of the system, which is called model. The model determines a state space of the system, which consists of states and transitions between states. Intuitively, a state is an instant snapshot of the system, where the details captured in the snapshot are determined by the model in use. For example, a state underlying assembly-level models contains the values of registers of the CPU. A system can change its state via executing transitions. For example, every tick of the CPU clock can correspond to a transition. A sequence of states is called a path if there is a transition from every state to the next in the sequence.

We expect from the verification to be sound, i.e., it never claims an incorrect system to be correct. To guarantee soundness, it is necessary for the verification to consider the entire state space. Even if the model is less detailed than assembly code, e.g., high-level pseudocode can be used as model instead, the state space can be huge [YCW+09, BCG11]. In practice, the large size of the state space can prevent the verification process from succeeding. For example, the verifier can run out of memory or time. This phenomenon is referred to as *state space explosion*.

The thesis proposes solutions to mitigate state space explosion of *fault-tolerant message-passing protocols*. The main focus is on protocols that achieve fault-tolerance using replication in distributed systems [AW04]. In addition, message-passing is assumed as an intuitive abstraction of communication in distributed systems. However, most of the concepts and solutions

Figure 1.1: The inputs and outputs of verification.

discussed in the thesis apply beyond replication-based message-passing protocols.

Fault-tolerant message-passing protocols are especially susceptible to state space explosion due to at least two reasons. Firstly, they are concurrent programs with transitions (corresponding to local computations and sending of messages within a process) that are executed simultaneously while the processes interact with each other via messages. If there is no coordination between these transitions, a large number of paths will correspond to their indeterminate executions. Secondly, a system that is vulnerable to faults contains more indeterminacy than a fault-free system, which further increases the state space size.

Both the development and application of fault-tolerant message-passing protocols can benefit from verification. These protocols can be complex conceptual designs and hard to implement due to the concurrency in the distributed system executing the protocol and also to a rich variety of (possibly malicious [LSP82]) faults that should be tolerated by the protocol. Therefore, counterexamples returned by verification in the phase of development can help developers to get the conceptual protocol or its implementation right. Also, fault-tolerant messages-passing protocols specifies strong guarantees such as atomic broadcast [JRS11] or diagnosis of malicious faults [SBS+11]. Therefore, the verification of such protocols can avoid failures of highly available systems that build upon these protocols. For example, unexpected service outages of Google's App Engine [Goo09] and Amazon's EC2 [Ama11] cause significant profit and prestige loss of these system.

Contributions Summary The thesis addresses state space explosion via *efficient* verification. Verification efficiency is considered in terms of verification space (memory) and time, and the required human interaction (ranging from full automation to intuitive human guidance).

The first group of contributions relate to *models* of fault-tolerant message-passing protocols to enable their efficient verification. Firstly, new models of such protocols are proposed to reduce the state space size entailed by existing models. In addition, techniques are presented to create models that are amenable to symmetry [ID96] and partial-order reductions [CGP99]. These

approaches are able to reduce the space and time resources of verification. Section 1.1 highlights these contributions in more details.

The second group of contributions relate to *verification algorithms* to implement efficient verification. Firstly, an improvement of partial-order reduction [God96] is proposed. Secondly, different techniques are presented to avoid spurious (wrong) counterexamples in induction-based verification [MP95]. Since the automated elimination of spurious counterexamples is a generally hard problem, they negatively effect the efficiency of verification. Section 1.2 gives an overview of the proposed verification algorithms.

1.1 Modeling Message-Passing Systems

The Basic Model with Crash Faults The thesis starts by considering an existing and intuitive model of message-passing systems [AW04]. It turns out that the modeling of *message traffic* can greatly affect the size of the state space depending on how the events of creation, sending, transmission, delivery, and consumption of a message are "mapped" to states and transitions. It is sound to model each of these events by a separate transition. However, such a model entails modeling lots of states that are irrelevant for the verification process. On the other hand, it is unsound to model all of these events by a single transition because a transition represents an indivisible event. Therefore, the thesis explores different models of message traffic between these two extremes. The models are shown to be equivalent with each other, in particular, with the reference model taken from [AW04]. The equivalence of these models implies that the model yielding the smallest state space can be used for verification.

The above model of message traffic can be used to model fault-tolerant systems too. A widely-used model is the *crash* fault-model, which assumes that a process might stop working (it crashes) but it runs as intended before crashing. It turns out that the model of concurrent *fault-free* systems is also a sound model of processes that are vulnerable to crash faults. In fact, the thesis shows that a path where a process crashes is indistinguishable from (thus, equivalent with) a path where the process is correct but its messages are ignored. An implication of this equivalence is that the model needs not contain transitions that (explicitly) represent the event of crashing, which results in smaller state spaces.

The proposed equivalence results hold under the assumption that the specification is from a general but restricted class of specifications. The thesis lists a variety of message-passing protocols whose specification falls into this class showing that the restrictions are not impractically strong.

Quorum Transitions A general class of systems achieves fault-tolerance under the assumption that the number of faulty processes lies below a given threshold and each correct process executes an instance of the same replicated protocol [AW04, Bir05]. The threshold assumption implies that a set of messages from a large enough subset (or quorum) of processes contains at least one message from a correct process. Therefore, a common technique in such systems is that an event, called quorum event, is triggered when a set of messages from a quorum (e.g., a majority) of processes is delivered. A quorum event can be modeled by a set of transitions such that each of them is responsible for consuming a *single* message from a process in the quorum. The verification of such a model can be inefficient because the execution of each of these transitions results in a new state.

We observe that a quorum event can be abstracted by a single transition, called *quorum transition*. A quorum transition models the consumption of *all* messages from all processes in the quorum in a single, indivisible step. Therefore, the use of quorum transitions results in smaller state space sizes.

Annotating Symmetries A tool for efficient verification is *symmetry reduction* [MDC06], which achieves efficiency by exploiting symmetries of the system. The idea of symmetry reduction is that it is sufficient for the verification to consider only one, representative state from an equivalence class of symmetric (thus, redundant) states.

A practical challenge of symmetry reduction is *finding* the symmetries of the system. Although the maintenance of representative states and symmetry classes during verification is another hard problem in general [ID96, CJEF96], implementations exist that have proven to be efficient in practice [Mur, BDH02]. The best-known solution for finding symmetries is arguably [MDC06] the "language approach" [ID96]. The idea is to restrict the syntax of the specification language such that models written in the language are symmetric by construction. The language approach is implemented by annotating certain data types as symmetric and the variables of symmetric types must not be involved in symmetry-breaking transitions.

The problem with the language approach is that the modeler is required to annotate a data type as symmetric. This might require a clear understanding of the formal notion of symmetry as well as expertise in the system's structure and functioning. Therefore, the thesis proposes a heuristic that enables an intuitive and efficient annotation of symmetries in models of fault-tolerant message-passing protocols. The observation behind the new heuristic is that fault-tolerant protocols are often defined in terms of independent sub-protocols. Every process can execute multiple sub-protocols si-

multaneously. The heuristic suggests that replicas executing the same *single* sub-protocol should be annotated as symmetric. These fine-grained symmetries result in more symmetric states (thus, more reduction) compared to the naive intuition where symmetric replicas might execute *multiple* sub-protocols. Following the naive intuition, several symmetries across replicas remain unrevealed because the decomposition of the system (thus, the notion of symmetry) is too coarse.

Transition Refinement Another technique for efficient verification is based on commutative transitions. Two transitions are said to be commutative if their subsequent execution leads to the same state irrespective which of the two transition is executed first. The concept of commutative transitions enables efficient verification, which has been generalized into different notions of so called *partial-order reduction* (POR) [God96, Val98, CGP99, FG05, KWG09a].

Independent of the notion of POR being used, the efficiency of POR can vary for different models of the *same* system [God96]. Models containing more transitions can result in more efficient POR. The technique of translating a model into another model that contains more transitions than the original model is called *transition refinement* [God96]. However, as shown by the thesis, overly refined transitions can worsen the efficiency of POR. Therefore, an appropriate tuning of transition refinement is required. The thesis proposes a heuristic for the transition refinement of fault-tolerant message-passing protocols. Experiments with representative protocol examples justify the ability of the proposed heuristic to enable efficient POR.

1.2 Efficient Verification Algorithms

The second part of the thesis deals with efficient algorithms for the verification of models, in particular, models introduced in Section 1.1.

The space dimension of verification efficiency corresponds to the memory required by the verification process. The time dimension specifies the execution time it takes for the verification process to terminate. Verification time can be an important factor, for instance, in debugging where the verification process is repeatedly executed.

Another dimension of verification efficiency considers the automation of verification. Automated versus space and time-efficient verification can be contradictory goals. Full automation can be traded for space and time efficiency if the resulting (semi-automated) verification procedure is intuitive-to-use for a human verifier. The proposed verification algorithms consider

such trade-offs.

Partial-Order Reduction Partial-order reduction (POR) can be used to reduce the time and space efficiency of verification. Existing POR approaches have in common that they assume certain complex conditions to hold guaranteeing the soundness of POR[1]. Given this complexity, existing implementations of POR restrict to certain classes of systems, e.g., [HP94, TLSD11, AM11], which provably satisfy these conditions. These implementations are inefficient if a system does not satisfy the restrictions that characterize the corresponding class. Therefore, one limitation of POR is that its application to new systems is tedious and requires expertise in POR.

POR exploits commutative transitions to detect redundant paths during verification. This inevitably involves some time overhead to decide if two paths are redundant. Intuitively, the more elaborate the process of detecting redundant paths, the larger its time overhead. Therefore, existing POR implementations trade space efficiency for time efficiency [God96].

On this background, the thesis proposes a new POR implementation called *LPOR* (Local POR), which addresses the above limitations. LPOR leverages an existing notion of POR called statically computed stubborn sets [Val98]. Firstly, LPOR defines an *intuitive* and *flexible* interface so that POR can be easily and efficiently applied to new systems. Through this interface LPOR hides the complexity of the conditions defined by stubborn sets. Secondly, LPOR enables time efficient discovery of redundant paths by using a novel *pre-computation* scheme. Pre-computation allows to discover redundancy in the preface of the verification (one-time overhead) and to re-use this redundancy as verification proceeds.

A prototype Java-implementation of LPOR is available. In addition, a *new model checker* has been developed that utilizes this prototype. The model checker enables efficient verification, as shown by an evaluation with various fault-tolerant message-passing protocols.

Verification with Induction The last part of the thesis deals with induction-based verification [MP95, dMRS03]. As induction can be automated using Boolean Satisfiability (SAT) and Satisfiable Modulo Theories (SMT) [dMRS03], the recent advances of SAT/SMT and and also their tool support motivate the use of induction.

The main efficiency burden of induction results from its incompleteness: It is possible that induction cannot verify correct systems. The possible out-

[1]These conditions have been formalized by various notions of POR, most notably, stubborn sets [Val98], persistent sets [God96], and ample sets [CGP99].

come of induction is three-fold: (1 – Proof) the correctness of the system can be proven; (2 – Counterexample) the system is incorrect and a counterexample is given; (3 – Inductive counterexample) induction terminates and returns a path of the system that is either (3a – Spurious counterexample) *not* a path of the system or (3b – Real counterexample) a real counterexample. In case of (3), it is *indecisive* whether the system is correct or incorrect. In general, deciding whether an inductive counterexample is spurious or real is as hard as the verification itself. Therefore, induction-based verification is hard to fully automate.

Given the complexity of automating verification using induction, the thesis aims at *semi-automated* verification satisfying the following features: (F1) If the induction returns an inductive counterexample, then it is easy for a human verifier to decide if the counterexample is spurious or real; (F2) in favorable cases and correct systems, induction proves the system correct.

Using lemmas is an approach to rule out spurious counterexamples and, as a result, to achieve (F1-2) [MP95]. The main drawback of lemmas is that it is hard to automate their discovery. Based on the induction proof of an example protocol, the thesis proposes a *classification* of lemmas. As the classification applies to a general class of fault-tolerant message-passing protocols, it can be used to automate (or partly automate) the discovery of lemmas in induction proofs of other protocols. The thesis also proposes a new approach, called *strengthened transitions*, that features (F1-2). Finally, an application of strengthened transitions for general multi-process systems (including message-passing systems) is shown.

1.3 Thesis Structure and Resulting Publications

In the following, the structure of the thesis is highlighted together with the publications resulting from it.

Preliminaries (Chapter 2):

- Existing foundations are reviewed, which are used throughout the thesis.

Models of message-passing systems (Chapter 3):

- Section 3.1: A model of message traffic is proposed.

Resulting publication: Péter Bokor, Marco Serafini, Neeraj Suri, "On Efficient Models for Model Checking Message-Passing Distributed Protocols", Proc. of IFIP Conf. on Formal Techniques for Distributed Systems (FMOODS & FORTE), pages 216-223, 2010.

- Section 3.2: A model of processes that are vulnerable to crashing is proposed.

  Resulting publication: Habib Saissi, Péter Bokor, Marco Serafini and Neeraj Suri, "To Crash or Not To Crash: Efficient Modeling of Fail-Stop Faults", Invited paper, Proc. of Workshop on Logical Aspects of Fault-Tolerance (LAFT, in assoc. with LICS), 2011, To appear.

- Section 3.3: A heuristic for finding symmetries of fault-tolerant message-passing protocols is proposed.

  Resulting publications:

  - Péter Bokor, Marco Serafini, Neeraj Suri, and Helmut Veith, "Brief Announcement: Efficient Model Checking of Fault-tolerant Distributed Protocols Using Symmetry Reduction", Proc. of the 23^{rd} Symposium on Distributed Computing (DISC) 2009, pages 289-290.
  - Péter Bokor, Marco Serafini, Neeraj Suri, Helmut Veith, "Role-Based Symmetry Reduction of Fault-tolerant Distributed Protocols with Language Support", Proc. of the 11^{th} Conf. on Formal Engineering Methods (ICFEM) pages 147-166, 2009.

- Section 3.4: The concept of quorum transitions introduced.

- Section 3.5: A heuristic for transition refinement in models of fault-tolerant message-passing protocols is proposed.

  Resulting publication (Sections 3.4-3.5): Péter Bokor, Johannes Kinder, Marco Serafini and Neeraj Suri, "Efficient Model Checking of Fault-Tolerant Distributed Protocols", Proc. of the 41^{st} IEEE Conf. on Dependable Systems and Networks (DSN-DCCS), pages 73-84, 2011.

Practical partial-order reduction (Chapter 4):

- A new partial-order reduction framework called LPOR is proposed.

  Resulting publication: Péter Bokor, Johannes Kinder, Marco Serafini and Neeraj Suri, "Supporting Domain-Specific State Space Reductions through Local Partial-Order Reduction", Proc. of the 26^{th}

IEEE/ACM Conf. on Automated Software Engineering (ASE), 2011, To appear.

Induction proofs of distributed protocols (Chapter 5):

- Section 5.1: A classification of lemmas is presented, which can be used to discover new lemmas in fault-tolerant message-passing protocols.

- Section 5.2: The concept of strengthened transitions is proposed and applied to the verification of distributed protocols.

  Resulting publication: Péter Bokor, Sandeep Shukla, András Pataricza and Neeraj Suri, "Strengthened State Transitions for Complete Invariant Verification in Practical Depth-Induction", Proc. of the 3^{rd} Workshop on Automated Formal Methods Workshop (AFM, in assoc. with CAV 2008), pages 31-41.

Model checking support for message-passing systems (Chapter 6):

- A new model checker for general message-passing systems is presented. The main features of the model checker is that it implements LPOR and quorum transitions.

  Resulting publications: The above DSN-DCCS and ASE conference papers.

Miscellaneous publications not covered by the thesis:

- Péter Bokor, Marco Serafini, Áron Sisak, András Pataricza , Neeraj Suri, "Sustaining Property Verification of Synchronous Dependable Protocols Over Implementation", Proc. of the 10^{th} High Assurance Systems Engineering Symposium (HASE) 2007, pages 169-178.

- Kohei Sakurai, Péter Bokor, Neeraj Suri, "Aiding Modular Design and Verification of Safety-Critical Time-Triggered Systems by use of Executable Formal Specifications", Proc. of the 11^{th} High Assurance Systems Engineering Symposium (HASE) 2008, pages 261-270.

- Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke, Neeraj Suri, "Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas", Proc. of the 40^{th} IEEE Conf. on Dependable Systems and Networks (DSN-DCCS) pages 353-362, 2010.

- Marco Serafini, Péter Bokor, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstaetter, Fulvio Tagliabo, Jens Koch, "Application-Level Diagnostic and Membership Protocols for Generic Time-Triggered Systems", IEEE Transactions on Dependable and Secure Computing (TDSC), 8(2), pp. 177-193, March 2011.

- Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, Neeraj Suri, "Eventually Linearizable Shared Objects", Proc. of the 29^{th} Symposium on Principles of Distributed Computing (PODC) pages 95-104, 2010.

Chapter 2

Preliminaries

In this chapter, we refresh existing work that the thesis builds upon. This also includes the definition of auxiliary formalisms used throughout the thesis.

The following list highlights the content of each section of this chapter. In case the reader is familiar with these foundations, reading the rest of the chapter can be skipped.

- *State transition systems (Section 2.1):* State transition systems are defined as (low-level) models of the system under verification. In addition, invariants are defined as a widely-applicable class of specifications.

  In this thesis, we use state transition systems to define the semantics of (high-level) models of message-passing systems. Invariants are used to specify the properties of example message-passing protocols.

- *Stuttering equivalence (Section 2.2):* The concept of stuttering equivalence is recalled, which is a well-established equivalence base in formal specification and verification [Lam83, CGP99].

  We use this concept to state and prove property preservation between different models of message-passing systems.

- *The Paxos protocol (Section 2.3):* We give a brief overview of the widely-used Paxos consensus protocol, which is also representative for a general class of message-passing protocols.

  Paxos is used as a running example throughout the thesis for its relative simplicity and highly concurrent nature.

- *Symmetry reduction with scalarsets (Section 2.4):* We repeat the formal definition of symmetry. In addition, we review scalarsets, a pragmatic approach to detect symmetries of the system.

Specifically, we use the scalarset approach to detect symmetries in fault-tolerant message-passing systems.

- *Partial-order reduction with stubborn sets (Section 2.5):* The definition of stubborn sets is given. A stubborn set is a general characterization of the concepts of partial-order reduction.

  Our partial-order reduction results build upon stubborn sets.

- *Induction proofs (Section 2.6):* We briefly discuss induction proofs as general verification procedures.

  Our contributions using induction proofs improve on the efficiency of these proofs.

2.1 State Transition Systems and Invariants

A *state transition system* (STS) [CGP99] is a triple (S, T, S_0) where S is the set of states, T is the set of transitions, and $S_0 \subseteq S$ is the set of initial states. Optionally, an STS can specify a labeling function $L : S \to 2^{AP}$ that, given a set of atomic propositions AP, assigns to each state a subset of AP. In this case, the STS is denoted by a five tuple (S, T, S_0, L, AP). Every transition $t \in T$ is a relation $t \subseteq S \times S$. A transition t is *enabled* in $s \in S$ iff there is an $s' \in S$ such that $(s, s') \in t$. Otherwise, t is disabled in s. The set of all enabled transitions in s is denoted by $enabled(s)$. A state $s \in S$ is called a *deadlock* if $enabled(s) = \emptyset$. We write $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s_n$ and say that there is a *path* from s_0 to s_n iff for every $0 \leq i < n$ we have that $(s_i, s_{i+1}) \in t_{i+1}$. If the transitions $t_1, t_2, ..., t_n$ are irrelevant or clear from the context, we might use the notation $s_0, s_1, ..., s_n$ or $s_0, s_1, ...$ if n is unbounded, i.e, the path is infinite. In this case, we say that s_n is *reachable from* s_0. If $s_0 \in S_0$, then we say that s_n is *reachable*. A transition t is said to be *in* a path $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s_n$ if t is among $t_1, t_2, \ldots, t_n$.

Invariants Given a state transition system and a Boolean formula P defined over AP, P is called *invariant* if P holds in every reachable states with respect to the labeling function. Formally, let $s \models P$ iff formula P holds in a state s. The relation $\models$ is defined inductively as follows [CGP99]:

- $s \models p$ iff $p \in L(s)$.
- $s \models P_1 \vee P_2$ iff $s \models P_1$ or $s \models P_2$.
- $s \models P_1 \wedge P_2$ iff $s \models P_1$ and $s \models P_2$.

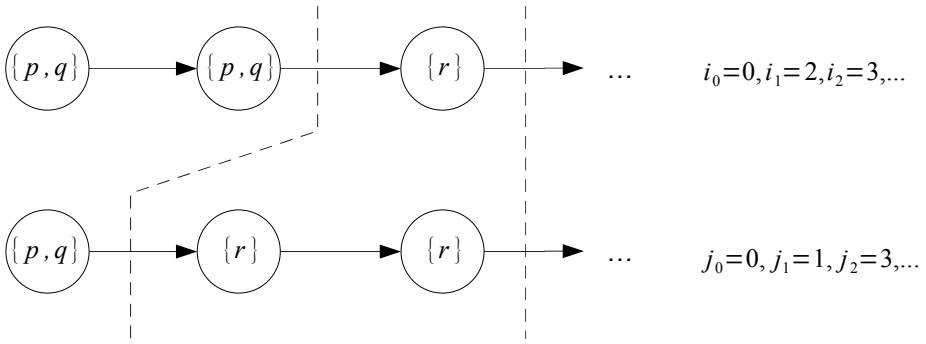

Figure 2.1: Example stuttering equivalent paths.

- $s \models \neg P$ iff $s \not\models P$.

Invariants are the simplest *temporal* formulas that can be defined in a state transition system. For details of temporal formulas, we refer to [CGP99].

2.2 Stuttering Equivalence

Two infinite paths $\sigma = s_0 \xrightarrow{t_0} s_1...$ and $\sigma' = s'_0 \xrightarrow{t'_0} s'_1...$ are *stuttering equivalent* [CGP99], $\sigma \approx_{st} \sigma'$ in short, if there are two infinite sequences of integers $0 = i_0 < i_1 < ...$ and $0 = j_0 < j_1 < ...$ such that for every $k \geq 0$, $L(s_{i_k}) = L(s_{i_k+1}) = ... = L(s_{i_{k+1}-1}) = L(s'_{j_k}) = L(s'_{j_k+1}) = ... = L(s'_{j_{k+1}-1})$. An example of two stuttering equivalent paths are shown in Figure 2.1.

Two state transition systems STS_1 and STS_2 are said to be stuttering equivalent if (a) for every path σ in STS_1 there is a path σ' in STS_2 such that $\sigma \approx_{st} \sigma'$, and vice versa, (b) for every path σ' in STS_2 there is a path σ in STS_1 such that $\sigma' \approx_{st} \sigma$. Paths in this definition are assumed to start from initial states.

2.3 Consensus with Paxos

The consensus problem Assume a collection of processes that can *propose* values. The (safety) requirements of consensus are that (1 – validity) only a value that has been proposed may be *chosen*, (2 – agreement) only a single value is chosen, and (3 – learning) a process never *learns* that a value has been chosen unless it actually has been. Additional (liveness) requirements can specify that eventually a value must be learnt. Examples of

possible applications of consensus are diagnosis or general service replication [Bir05].

Paxos consensus Paxos [Lam98, Lam01] is an algorithm that guarantees (1)-(3) to hold under the assumption that faulty processes can only fail by crashing. Processes communicate via messages assuming that messages cannot be corrupted. Paxos is a conceptual and widely applied consensus protocol for its practical assumptions (crash fault model) and system model (unreliable, possibly lossy channels, etc.) [Bir05].

The core Paxos algorithm specifies the following program run by two types of processes, called *proposers* and *acceptors*. It is assumed that proposer processes maintain disjoint sets of natural numbers, called proposal numbers.

Phase 1.

(a) A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.

(b) If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2.

(a) If the proposer receives a response to its prepare request (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

A proposal is chosen if a majority of acceptors have accepted it. Special processes, called *learners*, can learn about chosen proposals via messages from the acceptors.

2.4 Symmetry Reduction with Scalarsets

Notion of symmetry Given a state transition system (S, T, S_0), a *symmetry* [MDC06] is a permutation π acting on the states in S such that for all $s, s' \in S$ where $(s, s') \in t$ for some $t \in T$ it holds that $(\pi(s), \pi(s')) \in t'$ for some $t' \in T$.

Symmetry by construction Given a set of symmetries, symmetry reduction enables efficient verification [MDC06]. However, finding symmetries is hard in general because symmetry is a condition of the entire state space (see previous definition). One approach for finding symmetries efficiently is with *scalarsets* [ID96]. Given a general purpose specification language, a scalarset is a subrange $0, 1, ..., n-1$ of integers such that scalarsets *cannot* be involved in symmetry-breaking operations. This is formalized by the following restrictions:

C1 An array with a scalarset index type can only be indexed by a variable of exactly the same type.

C2 A term of scalarset type may not appear as an operand to + or any other operator in a term.

C3 Variables of scalarset type may only be compared using =.

C4 For all assignments $d := t$, if d is a scalarset variable, t must be a term of exactly the same scalarset type.

C5 Variables, elements of arrays and fields of records written by any iteration of a "for" statement indexed by a scalarset type must be disjoint from the set of variables, elements of arrays and fields of records referenced (read or written) by other iterations.

A permutation π acting on a scalarset $0, 1, ..., n-1$ determines a permutation σ acting on S where, given $s \in S$, $\sigma(s)$ is obtained by replacing in s every occurrence of a scalarset variable v with $\pi(v)$. If C1-5 holds, then π is provably a symmetry.

2.5 Partial-Order Reduction with Stubborn Sets

Given a state transition system (S, T, S_0) and a state $s_0 \in S$, a set $stub(s_0)$ of transitions from T is *(weakly) stubborn* if the two properties D1 and D2 are

satisfied [Val98]. D1 verifies the commutativity of transitions in the stubborn set with transitions outside the stubborn set. D2 ensures that there is at least one transition that cannot be disabled by transitions outside the stubborn set.

D1 $\forall t \in \mathit{stub}(s_0), \forall t_1, t_2, \ldots, t_n \in T \setminus \mathit{stub}(s_0), \forall s_n \in S$: if $s_0 \xrightarrow{t_1 t_2 \ldots t_n t} s_n$ then $s_0 \xrightarrow{t t_1 t_2 \ldots t_n} s_n$.

D2 If $\mathit{enabled}(s_0) \neq \emptyset$ then $\exists t \in \mathit{stub}(s_0), \forall t_1, t_2, \ldots, t_n \in T \setminus \mathit{stub}(s_0)$: if $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s_n$ then $t \in \mathit{enabled}(s_n)$. Such a transition t is called *key transition*.

A stubborn set is called *strong* if every $t \in \mathit{stub}(s_0) \cap \mathit{enabled}(s_0)$ is a key transition. Note that a key transition is always enabled in s_0.

Partial-order reduction is obtained by executing only the enabled transitions from $\mathit{stub}(s)$.[1] The stubborn set $\mathit{stub}(s)$ is called *trivial* if it contains all transitions from $\mathit{enabled}(s)$. Otherwise, the stubborn set is called *non-trivial*. If $t \in \mathit{stub}(s)$ and t is non-deterministic, then every s' with $(s, s') \in t$ is visited.

D1 and D2 guarantee that all *deadlocks* of the unreduced state transition system are contained in the reduced one. Therefore, partial-order reduction preserves deadlock-freedom, i.e., the unreduced state transition system contains no deadlock iff so does the reduced state transition system. In order to preserve properties other than deadlock-freedom, $\mathit{stub}(s_0)$ needs to satisfy additional constraints [Val98, KSV06]. Note that transitions in $\mathit{stub}(s)$ are not necessarily enabled in s.

Although disabled transitions cannot be executed, they can ease the design of stubborn set algorithms [God96] and even result in smaller stubborn sets when used to preserve certain temporal properties [Val98].

2.6 k-Induction Proofs

Induction can be used to verify invariants of state transition systems [MP95]. A generalized form of induction is called *k-induction* [dMRS03], which is defined as follows.

[1] Other notions of partial-order reduction characterized by ample sets [CGP99] or persistent sets [God96] can be seen as special cases of stubborn sets.

Basic induction Given a state transition system, a Boolean formula P defined over AP, and $k > 0$ called *(induction) depth*, P is an invariant if k-induction, i.e., the following two conditions, hold for all paths $s_0, s_1, ..., s_k$:

- *Base case:* If $s_0 \in S_0$, then $P(s_i)$ holds for all $0 \geq i > k$.
- *Inductive case:* If $P(s_i)$ for all $0 \geq i > k$, then $P(s_k)$ holds.

k-induction assumes that the system contains *no deadlock*. This assumption can be implemented by added a "dummy transition" t such that $(s, s) \in t$ for all states s.

A Boolean formula over AP is called *assertion*. An assertion P is called *inductive (invariant)* if k-induction, for some $k > 0$, holds, i.e., both the base and inductive steps are satisfied. Specifically, P is called *k-inductive (invariant)* if k-induction holds.

Strengthened induction It is possible that an assertion is an invariant but not an inductive invariant. To prove the invariance of such assertions, the following *strengthened* k-induction can be used. Given a state transition system, an assertion P, $k > 0$, and invariants $a_1, a_2, ..., a_l$, P is an invariant if the following two conditions hold for all paths $s_0, s_1, ..., s_k$:

- *Base case:* If $s_0 \in S_0$, then $P(s_i)$ holds for all $0 \geq i > k$.
- *Strengthened inductive case:* Let $P_s = (\bigwedge_{j=1}^{l} a_j) \wedge P$. If $P_s(s_i)$ for all $0 \geq i > k$, then $P(s_k)$ holds.

The invariants $a_1, a_2, ..., a_l$ and P_s are called *lemmas* and *strengthened assertion*, respectively. We say that P is *inductive relative to* $a_1, a_2, ..., a_l$ if strengthened k-induction, for some $k > 0$, holds. Specifically, P is *k-inductive relative to* $a_1, a_2, ..., a_l$ if strengthened k-induction holds.

Chapter 3

Models of Message-Passing Systems

In this chapter, we propose new models of message-passing systems and protocols executed within these systems. The proposed models enable efficient verification of general classes of message-passing protocols.

We now briefly summarize the proposed models along with the structure of the chapter:

- *Models of message traffic (Section 3.1):* Models of message traffic deal with modeling the sending, delivery, and receipt of messages. We first consider an existing model of message traffic and use it as a reference model [AW04], called **RM**. We then propose two new models: A generalization of the reference model, called **GRM**, and a simplification of **GRM**, called **M**.

  It turns out that the **RM** and **GRM** models can be inefficient for verification because they yield a significant number of states that are irrelevant for the specification of most message-passing protocols. We tackle this problem by showing an (stuttering) equivalence between the three models such that the equivalence preserves the protocol's specification. Since **M** models do not yield the irrelevant states, the equivalence result implies that **M** should be used for verification rather the models **RM** and **GRM**.

- *Models of crash-faulty processes (Section 3.2):* The crash fault model is a widely-used assumption in fault-tolerant computing [Bir05]. In this section, we deal with modeling crash faults so that the resulting model can be efficiently verified despite the additional modeling complexity of crash faults.

We start by defining an intuitive and arguably sound model of crash faults, which we call the explicit model. Then, we observe that a fault-free model is able to mimic most effects caused by these faults. Based on this observation, we formally define an equivalence between the explicit and fault-free models. Since the fault-free model yields significantly fewer states, its verification can be more efficient.

- *Finding symmetries with decomposition (Section 3.3):* Symmetry reduction [MDC06] is a powerful optimization of verification. However, symmetry reduction assumes that the symmetries of the system are known prior to verification.

  A general strategy is to declare and verify replica processes to be symmetric. We observe that replicas of common message-protocols can be decomposed into replicated sub-processes. This allows the general strategy to find more symmetries. In this section, we formalize decomposition-based symmetry reduction and develop it into a sound and efficient verification procedure.

- *Quorum transitions (Section 3.4):* We observe that message-passing fault-tolerant protocols often specify certain events that, conceptually, process multiple messages in an atomic step. We call such events quorum transitions.

  Although it is possible to model quorum transitions as a non-atomic sequence of events each of them processing a single message, we propose models using quorum transitions if the message-passing protocol explicitly mentions them. Such models are not only faithful, they also yield smaller models, as shown by our analysis.

- *Transition Refinement for Efficient POR (Section 3.5):* Partial-order reduction [CGP99] is another optimization of verification. We observe that the efficiency of partial-order reduction greatly depends on the set of transitions in the underlying state transition system. In particular, more transitions tend to improve the efficiency of partial-order reduction [God96], an informal technique called transition refinement.

  In this section, we formalize and generalize transition refinement. We also define a general strategy of refining transitions of fault-tolerant message-passing protocols. Although overly refined transition can worsen the efficiency of partial-order reduction, we show that the proposed strategy does not do so (see experiments in Section 3.6).

- *Experiments (Section 3.6):* We evaluate our symmetry detection and transition refinement strategies. Our experiments show that both strategies can significantly improve on the efficiency of verifying representative fault-tolerant message-passing protocols.

3.1 Models of Message Traffic

Three models of message traffic are presented (Sections 3.1.1-3.1.3). The first model is a reference model taken from a standard text book on distributed computing [AW04]. An equivalence of the three models is shown with respect to a general class of temporal logic specifications written in the widely-used LTL-X language (Section 3.1.4). The consequence of this equivalence result is that any of the three models can be used for verification. Since the third model yields considerably small state spaces than the first and second model, verification using the third model can be more efficient.

3.1.1 Reference Model

Message-passing system (adapted from [AW04]) A message-passing system consists of n processes. Each process i is associated with a set S_i of local states. A subset of S_i specifies the possible initial states of a process.

Processes communicate with each other by sending *messages* via directed *channels*. The set of all messages and channels are denoted by M and C, respectively. A channel is a point-to-point connection from one process to another. The topology of the system specifies for each pair (i, j) of processes if there is a channel $c_{i,j}$ from process i to j. The channel $c_{i,j}$ consists of an *output (and input) channel to (from) j (i)*. Formally, both output and input channels are a set of messages, respectively. A *(global) state* of the system is a tuple containing all channels and local states. S denotes the set of all states. The content of channel $c_{i,j}$ and process i's local state in s is referred to as $s(c_{i,j})$ and $s(i)$, respectively.

Processes send and receive messages and also perform local computations by executing *(local) transitions*. Formally, each process i is associated with a set $T_i \subseteq T$ of transitions where T denotes the set of all transitions. Furthermore, every transition t is associated with a true/false condition g_t (called *guard*), which is a function of a set of messages and a local state of the process. A transition $t \in T_i$ is *enabled* in a state s if, for some subset X of the union of all input channels of i, the condition $g_t(X, s(i))$ is true. In this case, the set X is called *accessible* for t in s. If t is enabled in s, it can be *executed* with accessible set X for t in s, and the resulting state s' is identical with s

except for the following: (1) the messages in X are removed from the input channels of i, (2) depending on $t, s(i)$ and X, local state $s'(i)$ is a state from S_i, and (3) zero or more messages are added to every output channel of i. If $m \in s(c_{j,i}) \cap X$ for some message $m \in M$ and channel $c_{j,i} \in C$, then we say that t *consumes* m from process j. If $m \in s'(c_{i,j}) \setminus s(c_{i,j})$ for some message $m \in M$ and channel $c_{i,j} \in C$, then we say that t *sends* m to process j. We use the notation $s \xrightarrow{t(X)} s'$ or simply $s \xrightarrow{t} s'$ if X is clear from the context or irrelevant. By assumption, a process sends no message to itself and, given X and $s(i)$, transitions are deterministic. If $t \in T_i$, $id(t) = i$ denotes the process executing t.

Message traffic We call the subsequent model of message traffic *reference model* (**RM**). Let process i and j be two processes such that there is a channel from process i to j. The output (input) channel to (from) process j (i) is modeled by an output (input) *buffer* denoted by $outbuf_i$ ($inbuf_j$). Message delivery is modeled by *delivery events*, which move messages from output to input buffers. Given a message m, a delivery event is denoted by a tuple $del(i, j, m)$. Given a state s, $del(i, j, m)$ is said to be enabled in s if $m \in outbuf_i$, disabled otherwise. If $del(i, j, m)$ is enabled, its execution in s results in a new state that is identical with s except that $m \notin outbuf_i$ and $m \in inbuf_j$.

This model requires *clean-channels*, i.e., every transition must remove all messages that are in the input channels of the process executing the transition.

Definition 1. *A model of a message-passing system is* with clean channels *iff, for every state s, and transition $t \in T_i$, and set X of messages such that X is accessible for t in s, X equals the union of all input channels of process i in s. Otherwise, the model is* with unclean channels.

Semantics & specification The semantics of **RM** is given by a state transition system (see Section 2.1) $(S^{RM}, T^{RM}, S_0^{RM}, L^{RM}, AP^{RM})$ where S^{RM} is the set of all global states, T^{RM} equals $\cup_{\forall i \in \{1,\ldots,n\}}(T_i \setminus \{(s, s') | \exists t \in T_i : s \xrightarrow{t} s' \wedge \exists c_{j,i} \in C : s'(c_{j,i}) \neq \emptyset\}) \cup_{\forall c_{i,j} \in C \forall m \in M} \{del(i, j, m)\} \cup \{(s, s) | s \in S^{RM} \wedge enabled(s) = \emptyset\}$, and S_0^{RM} contains global states where every process i assumes an initial local state from S_i and all buffers (output and input) are empty. Intuitively, each transition in T^{RM} corresponds to either a local transition of a process or to a delivery event. Note that state transitions (s, s') are excluded from $t \in T$ that do not respect that the model is with clean channels. Also, the existence of "idle" transitions containing (s, s) is

Model	Channel model	Clean Channels
RM	Input/output buffers	Yes
GRM	Input/output buffers	No
M	Single buffer	No

Table 3.1: Overview of different message traffic models.

assumed for every deadlock state s. This is a practical assumption used in widely-used formal specification languages [CGP99].

The specification of the model can be written using the set of atomic propositions AP^{RM}, the labeling function L^{RM}, and standard temporal logics [CGP99]. The specification language is the same for the models introduced in Sections 3.1.2 and 3.1.3.

3.1.2 Generalized Reference Model

We generalize **RM** and introduce the *generalized* reference model (**GRM**). This model is equivalent to **RM** except that **GRM** is with unclean channels. Intuitively, the **GRM** model introduces a new source of non-determinism via unclean channels: Given a state s, and a transition $t \in T_i$, and the union U of all input channels of process i, t can be executed with various accessible sets $X \subseteq U$, which are (strict or not) subsets of U.

Example 1. Consider a state s and a channel $c_{i,j}$ such that $s(c_{i,j}) = \{m_1, m_2\}$. Let $t \in T_i$ be a transition whose guard is always true. Then, the sets $\{m_1\}$, $\{m_2\}$, and $\{m_1, m_2\}$ are all accessible sets for t in s. In other words, it is non-deterministic which of m_1 and m_2 is first consumed processed by t or whether the two messages are processed "in parallel". (End of Example 1)

Semantics & specification The **GRM** model determines a state transition system $(S^{GRM}, T^{GRM}, S_0^{GRM}, L^{GRM}, AP^{GRM})$ where $S^{GRM} = S^{RM}$, $T^{GRM} = T \cup_{\forall c_{i,j} \in C \forall m \in M} \{del(i,j,m)\} \cup \{(s,s) | s \in S^{GRM} \wedge enabled(s) = \emptyset\}$, and $S_0^{GRM} = S_0^{RM}$. Note that no state transitions from $t \in T$ are excluded because the model with unclean channels does not restrict t.

3.1.3 The Proposed Model

Message traffic We introduce a new model of message traffic, denoted by **M**. In **M**, a channel $c_{i,j}$ is modeled by a *single* set (buffer) of messages. This means that the set $c_{i,j}$ of messages models the output channel to process j as well as the input channel from process i (unlike in the models **RM** and

GRM, where a channel is modeled by a pair of output and input buffers). Therefore, we define no delivery events in **M**.

In addition, the model is with unclean channels. Table 3.1 compares the models **RM**, **GRM**, and **M**. As we show in Section 3.1.4, despite the simplicity of **M**, a strong equivalence holds between these three models.

Semantics & specification The **M** model determines a state transition system $(S^M, T^M, S_0^M, L^M, AP^M)$ where S^M (and S_0^M) is the set of global (initial) states, and $T^M = T \cup \{(s,s) | s \in S^M \wedge \mathit{enabled}(s) = \emptyset\}$.

3.1.4 Equivalence of Models

The equivalence across the models **RM**, **GRM**, and **M** is based on the observation that the specification of message-passing protocols is often specified in terms of local process states. Therefore, the specification is independent of messages that have been sent but not yet processed. We call such systems process-labeled, which we formally define below. In Section 3.6, we show representative examples of such systems.

Definition 2. *A message-passing system is* process-labeled *if for all* $s_{RM} \in S_{RM}$, $s_{GRM} \in S_{GRM}$, $s_M \in S_M$, $s_{RM}(i) = s_{GRM}(i) = s_M(i)$ *for all* $1 \leq i \leq n$ *implies* $L_{RM}(s_{RM}) = L_{GRM}(s_{GRM}) = L_M(s_M)$.

The idea behind the equivalence is that for every path in one model there is a "similar" path in the other model such that these paths are indistinguishable by the specification (Theorem 1).

Similarity is formalized in terms of *stuttering equivalence* (see Section 2.2). In the following, Lemmas 1, and 2, show stuttering equivalence between **M** and **RM**, and **GRM** and **M**, respectively. Then, the equivalence between **RM** and **GRM** follows via the transitivity of stuttering equivalence (Lemma 3). The chain of our arguments can also be followed in Figure 3.1. The full proof of each lemma can be found in Appendix A.1.

Lemma 1. *Given a process-labeled message-passing system and a path* σ *of the model* ***M****, there is a path* σ' *of* ***RM*** *such that* $\sigma \approx_{st} \sigma'$.

Proof sketch. We construct σ' as the sequence of all local transitions in σ and, in addition, delivery events between these transitions. If a local transition t in σ consumes messages, then in σ' a delivery event is executed directly before the execution of t corresponding to each of these messages. The relative order of these delivery events in σ' is arbitrary. Since only the messages are delivered that are consumed by t, σ' respects clean channels.

Because σ and σ' start from the same initial states and every local transition t consumes the same set of messages in both paths, the result of these transitions are states where the local states of the processes are the same in σ and σ'. In addition, since the system is process-labeled, delivery events do not change the label of the state. Therefore, the labels of states in σ' follow those σ with a "stuttering" of the delivery events between two local transitions. □

Lemma 2. *Given a process-labeled message-passing system and a path σ of the model **GRM**, there is a path σ' of **M** such that $\sigma \approx_{st} \sigma'$.*

Proof sketch. We construct σ' as the sequence of all local transitions in σ. The order of the transitions in σ is preserved in σ'. In addition to the transitions in σ', path σ can contain delivery events between two local transitions. The messages that are delivered by these events are, by construction, in the channels of model **M**. Since model **M** is with unclean channels, if a local transition in σ is enabled (in a state), then it is also enabled in σ' (in the corresponding state).

Given the above construction of σ' and because the system is process-labeled, σ and σ' are stuttering equivalent. The argumentation is similar to the proof of Lemma 1. □

Lemma 3. *Stuttering equivalence is transitive, i.e., given a state transition system and three paths $\sigma, \sigma', \sigma''$, $\sigma \approx_{st} \sigma'$ and $\sigma' \approx_{st} \sigma''$ imply that $\sigma \approx_{st} \sigma''$ also holds.*

Proof sketch. From $\sigma \approx_{st} \sigma'$, there is partitioning of σ and σ' such that states in the corresponding partitions are labeled the same. There might be multiple such partitioning σ and σ'. We consider a *maximal* partitioning, where states in adjacent partitions have different labels. This property and $\sigma' \approx_{st} \sigma''$ imply that the partitioning of σ for stuttering equivalence with σ' is also partitioning for stuttering equivalence with σ''. □

LTL-X (Linear Temporal Logic without the next operator) [CGP99] is a rich language, which is suited for specifying the properties of most concurrent systems [Lam83]. Our equivalence result states that the truth of specifications of process-labeled systems written in LTL-X is preserved across the different models of message traffic.

Theorem 1. *Given a process-labeled system and an LTL-X formula f, f holds in the **RM** model iff it holds in the **GRM** model, and f holds in the **GRM** model iff it holds in the **M** model.*

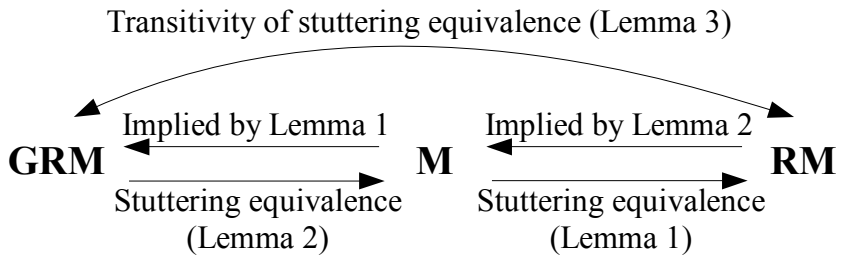

Figure 3.1: Proof sketch of Theorem 1.

Proof. It is known that, given two stuttering equivalent state transition systems STS_1 and STS_2, any LTL-X formula holds in STS_1 iff it is holds in STS_2 [CGP99]. We show that the **GRM** and **M**, and that **M** and **RM** models of process-labeled systems are stuttering equivalent. Therefore, and from the transitivity of stuttering equivalence (Lemma 3), we know that the models **RM**, **GRM**, and **M** are pairwise stuttering equivalent, which implies the state property preservation.

First we show that **GRM** and **M** are stuttering equivalent. The proof is illustrated in Figure 3.1. Lemma 2 shows that for every path in **GRM** there is a path in **M** such that the two paths are stuttering equivalent. The other direction is implied by Lemma 1 (which shows that for every path in **M** there is path in **RM** such that the two paths are stuttering equivalent) because every path in **RM** is also a path in **GRM**.

The stuttering equivalence of **M** and **RM** can be shown similarly. One direction is directly shown by Lemma 1. The other direction is implied by Lemma 2 because paths in **RM** are also paths in **GRM**. □

3.2 Models of Crash-Faulty Processes

In this section, we define a model of message-passing systems where processes are susceptible to *crash* faults (Section 3.2.1). In the crash fault-model, a process can stop receiving, processing, and sending messages, and it remains doing so forever. If the process crashes during the execution of an event, it executes the event as in the fault-free case except that it sends a subset of the messages that it is supposed to send [AW04].

We establish a general equivalence basis for preserving a general class of LTL (Linear Temporal Logic) specifications [CGP99] across state transition systems (Section 3.2.2). We apply these foundations to prove, maybe surprisingly, that the proposed model of crash faults is equivalent with the

fault-free model (Section 3.2.3).

3.2.1 A Model of Crash-Faulty Processes

Given a (fault-free) model of a message-passing system, we define another model where processes can fail by crashing. We call this model the *crash model*. Both the original and the resulting models are in terms of Section 3.1.1.

The fault-free and crash models share the same sets of channels and messages. In the crash model, every local state of process i is a tuple (s_i, a_i), where $s_i \in S_i$ and $a_i \in \{\bot, \top\}$. The variable a_i is called the *crash flag* of process i. Intuitively, the value $\bot$ means that process i is crashed, otherwise the flag assumes $\top$. The state (s_i, a_i) is an initial state of the process i in the crash model iff s_i is an initial state of process i in the original model and $a_i = \top$. Let S^c be the set of all states in the crash model.

The local transitions of the crash model are defined by the following definition.

Definition 3. *The set T_i^c of local transitions of process i is defined as $\{t' | q \xrightarrow{t'(X)} q'$ if there is $q, q' \in S^c, t \in T_i, s, s' \in S, X \subseteq M, a_i \in \{\top, \bot\}$ such that*

(1) (Fault-free transition) $s \xrightarrow{t(X)} s'$ and

(2) (Process up) $q(i) = (s(i), \top)$ and

(3) (Accessible messages) $X \subseteq \cup_{\forall c_{j,i} \in C} q(c_{j,i})$ and

(4) (Consistent local state update) $q'(i) = (s'(i), a_i)$ and

(5) (Consistent outgoing messages) $\forall c_{i,j} \in C : q'(c_{i,j}) \subseteq s'(c_{i,j})$.

}

Intuitively, transitions in the crash model consume the same set of messages and their execution results in the same local states as in the fault-free model, cf. (1-4). In addition, for a transition in the crashed model to be enabled, the executing process must not be crashed, cf. (2). Furthermore, the set of messages sent in the crash model is a subset of those in the fault-free model, cf. (5), because processes can crash during sending.

3.2.2 Equivalence basis

We define an equivalence relation between paths of state transition systems (STSs). Intuitively, two paths are equivalent if the i^{th} states in both paths are labeled the same.

Definition 4. *Given two STSs* $STS_1 = (S, T, S_0, AP, L)$ *and* $STS_2 = (S', T', S'_0, AP, L')$*, a path* $\sigma = s_0, s_1, ...$ *in* STS_1 *is said to be* label-equivalent *with another path* $\sigma' = s'_0, s'_1, ...$ *in* STS_2 *iff for every* $i = 0, 1, ...,$ $L(s_i) = L'(s'_i)$*. In this case, we write* $\sigma \approx_{AP} \sigma'$*.*

The previous definition can be naturally generalized to label-equivalence of two STSs.

Definition 5. *Given two STSs* $STS_1 = (S, T, S_0, AP, L)$ *and* $STS_2 = (S', T', S'_0, AP, L')$*, they are said to be* label-equivalent *iff the following two conditions hold:*

- *For every path* $\sigma = s_0, s_1, ...$ *in* STS_1 *where* $s_0 \in S_0$*, there exists a path* $\sigma' = s'_0, s'_1, ...$ *in* STS_2 *where* $s'_0 \in S'_0$ *and* $\sigma \approx_{AP} \sigma'$*.*

- *For every path* $\sigma' = s'_0, s'_1, ...$ *in* STS_2 *where* $s'_0 \in S'_0$*, there exists a path* $\sigma = s_0, s_1, ...$ *in* STS_1 *where* $s_0 \in S_0$ *and* $\sigma \approx_{AP} \sigma'$*.*

The next corollary follows from the above definitions and the semantics of LTL [CGP99]. It says that the truth of an arbitrary LTL formula is indistinguishable in label-equivalent STSs. The notation $STS \models \phi$ means that the LTL formula ϕ holds for every path of the state transition system STS.

Corollary 1. *[CGP99] Given two label-equivalent STSs* STS_1 *and* STS_2 *and a LTL formula* ϕ*, the following holds:*

$$STS_1 \models \phi \text{ iff } STS_2 \models \phi \ .$$

Proof. The $\Rightarrow$ direction: Assume that $STS_2 \not\models \phi$. Therefore, there must be a path $\sigma' = s'_0, s'_1, ...$ in STS_2 such that $\sigma' \not\models \phi$. From the semantics of LTL it follows that $s'_0 \in S'_0$. Since STS_1 and STS_2 are label-equivalent, there is a path $\sigma = s_0, s_1, ...$ in STS_1 where $s_0 \in S_0$ such that σ and σ' are label-equivalent. This implies that $\sigma \not\models \phi$ [CGP99], a contradiction.

The reverse direction can be proven similarly. □

3.2.3 Equivalence of Explicit and Implicit Models of Crash-Faulty Processes

We now prove an equivalence between the fault-free model of a message-passing system and its crash model. We call the fault-free (and crash) model the *implicit* (*explicit*) model because, as we will see, the fault-free model implicitly models crash faults.

The following definitions formalize the condition of the equivalence between the implicit and explicit models. Firstly, the equivalence result assumes that the specification is independent of the crash flag of each process and the content of the channels.

Definition 6. *A message-passing system with labeling functions* $L : S \to AP$ *and* $L^c : S^c \to AP^c$ *is* process/crash-labeled *if for all* $s \in S, s^c \in S^c$, $s^c(i) = (s(i), a_i)$ *for all* $1 \leq i \leq n$ *implies* $L(s) = L^c(s^c)$.

Secondly, the equivalence result assumes that the system is idling, which is formalized by the following definition. The idling property requires the existence of a "dummy" transition, which is always enabled and which does not change the state of the system.

Definition 7. *A message-passing system is* idling *if for all* $s \in S$ *there is a* $t \in T$ *such that* $(s, s) \in t$.

Note that the implicit and explicit models cannot be equivalent, in general, for systems that are *not* idling. For example, liveness properties that hold in the implicit (fault-free) model might not hold in the explicit model because processes can crash. If the system is not idling, it can be made idling by adding a dummy transition. It has to be established that this transformation preserves the truth of the target properties, e.g., LTL formulas. For example, the transformation always preserves invariants.

The following theorem together with Corollary 1 imply that an LTL formula holds for the implicit model of crash faults iff it holds for the explicit model. The equivalence is proven for the **M** model of message-passing systems. Given the equivalence stated by Theorem 1, the following result applies to the models **RM** and **GRM** too. The full proof of Theorem 2 can be found in Appendix A.2.

Theorem 2. *Given a process/crash-labeled and idling message-passing system, the state transition systems determined by the implicit and explicit* ***M*** *models of crash faults are label-equivalent.*

Proof sketch. Let σ and σ^c be paths in STS^M and STS^{M^c}, the state transition systems determined by the implicit and explicit models of crash faults, respectively. The proof is by induction on the length of the prefixes of σ and σ^c. Given a prefix of σ (and σ^c), we construct a prefix of a path in STS^{M^c} (in STS^M) such that label-equivalence holds for these prefixes. Then, label-equivalence between σ (and σ^c) and the constructed path follows by induction.

Given σ, we construct σ^c such that the transitions executed in σ^c correspond to those in σ and all processes are non-crashed. If t is a transition executed in σ, then there is a transition t^c of the explicit model that mimics t and keeps the crash flags unchanged. This follows from Definition 3. Since the message-passing system is process/crash-labeled, σ and σ^c are label-equivalent.

For the reverse direction, given σ^c, we construct a path σ such that the transitions executed in σ correspond to those in σ^c except that processes do not crash. This construction is always possible because the system is idling. If t^c is a transition executed in σ^c, then there is a transition t in the fault-free model that mimics t^c with respect to local state updates and send/recieve operations. This follows from Definition 3. The only difference is that t might send more messages, see condition (5), because the process executing t does not crash. However, since $\mathbf{M}$ is with clean channels, these additional messages can be ignored by transitions in the implicit model. Again, since the message-passing system is process/crash-labeled, σ and σ^c are label-equivalent. □

Finite channels Our initial model of message-passing systems from Section 3.1.1 assumes *infinite* channels, i.e., the size of the channel (a set of messages) is unbounded. We call a channel *finite* if there is an upper bound (capacity) of the number of messages residing in the channel. Finite channels can interfere with the above equivalence result, which we briefly discuss in the following.

We consider two models of finite channels. In the first model, new messages can be sent via a full channel, i.e., a channel that has reached its capacity. Sending a new message m via a full channel results in *replacing* a message in the channel with m. It turns out that Theorem 2 does *not* apply with this model of finite channels. To see this, assume that, in course of executing a transition t, a set Y of messages are replaced in full channels. In the explicit model, it is possible that only a subset of Y is replaced because the process executing t crashes during the send operations. In this case, a receiver process of the messages sent by t can see inconsistent channel content

because the explicit model contains messages that are not in the channel of the implicit model. As a result, the process in the explicit model might enter a local state that is unreachable in the implicit model.

In the second model no new message can be sent via a full channel: A transition t is only enabled if none of the channels that t sends a message to is full. Therefore, if a message is in the channel of the explicit model, then it is also in the channel of the implicit model and the result of Theorem 2 applies.[1]

3.3 Finding Symmetries with Decomposition

Symmetry reduction exploits symmetries of the system to enable efficient verification [MDC06]. Unfortunately, finding symmetries, a prerequisite of symmetry reduction, is a hard problem in general. An efficient approach for finding symmetries is using special data types, called scalarsets (see Section 2.4), in the specification of models. The use of scalarsets is restricted such that symmetry cannot be broken.

Despite the general strength of the scalarset approach, its use is not always intuitive even if the system contains obvious symmetries (Section 3.3.1). After establishing the formal foundations (Section 3.3.2), a new heuristic is proposed for an efficient discovery of symmetries in fault-tolerant message-passing protocols (Section 3.3.3).

3.3.1 Motivating Example

The main challenge of the scalarset approach is *where* in the specification to use scalarsets. In the context of distributed systems, an initial strategy is to index "identical" processes (replicas) with values from the same scalarset. This naive strategy turns out to be inefficient leaving intuitive symmetries undetected, as demonstrated by the following example.

Example 2. Imagine a system running the Paxos consensus algorithm (see Section 2.3). Assume that every process acts both as a proposer and an acceptor. Let s and s' be states where

- $s(1)$ =(**"Acceptor: initial state"**, "Proposer: Phase 1(a) message with $n = 35$ sent to process 4").

[1] Theorem 2 is a proof-of-concept result, which is applicable for different models of message-passing systems with possible adaptations as needed. Formal proofs and the exploration of the boundaries of applicability are beyond the scope of this thesis.

- $s(2) =$(*"Acceptor: Phase 1(b) promise with $n = 27$ sent to process 3"*, "Proposer: initial state").
- $s'(1) =$(*"Acceptor: Phase 1(b) promise with $n = 27$ sent to process 3"*, "Proposer: Phase 1(a) message with $n = 35$ sent to process 4").
- $s'(2) =$(**"Acceptor: initial state"**, "Proposer: initial state").

A permutation π with $\pi(s) = s'$ is a symmetry (see Section 2.4) because acceptors execute the same algorithm and the only difference between s and s' is the ID of the acceptor that has sent a Phase 1(b) message to process 3.

As the processes of this system are replicas (containing an acceptor and a proposer each), selecting the IDs of different processes from a scalarset respects the scalarset restrictions. Therefore, according to the scalarset approach, permuting the local states of the processes yields symmetries. Applying this to s results in s'' with

- $s''(1) =$(*"Acceptor: Phase 1(b) promise with $n = 27$ sent to process 3"*, "Proposer: initial state").
- $s''(2) =$(**"Acceptor: initial state"**, "Proposer: Phase 1(a) message with $n = 35$ sent to process 4").

The scalarset restrictions imply that there is a symmetry that maps s to s''. However, permuting the local states of processes in s does not yield s' because of the local states of the proposers. As a result, s and s' are considered "non-symmetric" although π is a symmetry. (End of Example 2)

3.3.2 Formalizing Decomposition and Replicas

The heart of our approach is the concept of *decomposition*. Intuitively, a decomposition of a model results in an other model that models the same system using more processes. The formal definition of decomposition is followed by its informal explanation.

Definition 8. *Given a message-passing system, let S, S^r, C, C^r, and T, T^r be the set of states, channels, and transitions of two models with n and $n+1$ processes, respectively. Then, the second model is a* decomposition with respect to process i *of the first model if the following holds:*

- *(Product state space) $S_i = S_i^r \times S_{n+1}^r$, and*
- *(Consistent topology) $C^r = C \cup \{c_{k,j}^r | j \in \{i, n+1\} \wedge c_{k,i} \in C\} \cup \{c_{j,k}^r | j \in \{i, n+1\} \wedge c_{i,k} \in C\}$, and*

- *(Bi-simulation)* $\exists s_1, s_2 \in S, t \in T : s_1 \xrightarrow{t} s_2$ *iff* $\exists s_1^r, s_2^r \in S^r, t \in T^r : s_1^r \xrightarrow{t} s_2^r$ *where* $\forall p \in \{1, 2\}$:
  - *(Composed state space)* $\forall j \in \{1, ..., n\} \setminus \{i\}$:
    - *(Matching local states)* $s_p(j) = s_p^r(j)$, *and*
    - *(Matching channel content)* $\forall c_{k,l} \in C \cap C^r : k \neq i \wedge l \neq i$ *implies* $s_p(c_{k,l}) = s_p^r(c_{k,l})$, *and*
  - *(Decomposed process state space)* $s_p(i) = (s_p^r(i), s_p^r(n+1))$, *and*
  - *(Decomposed channel content)* $\forall c_{j,i}, c_{i,j} \in C$:
    - *(Disjoint channels)* $s_p^r(c_{j,i}) \cap s_p^r(c_{j,n+1}) = \emptyset \wedge s_p^r(c_{i,j}) \cap s_p^r(c_{n+1,j}) = \emptyset$, *and*
    - *(Content decomposition)* $s_p(c_{j,i}) = s_p^r(c_{j,i}) \cup s_p^r(c_{j,n+1}) \wedge s_p(c_{i,j}) = s_p^r(c_{i,j}) \cup s_p^r(c_{n+1,j})$.

According to Definition 8, the decomposition of process i means that the local state space of process i is considered as a product of the local state spaces of a pair of sub-processes (product state space). Without the loss of generality, we assume the IDs of these processes in the decomposed model to be i and $n + 1$. The new processes have an incoming (outgoing) channel from (to) another process if and only if the original process i has one (consistent topology). This means duplicating the channels of process i. The decomposed model bi-simulates the transitions of the original model (bi-simulation): The processes other than i behave identically with the decomposed model regarding both local states and channel content (composed state space). The decomposed processes i and $n + 1$ also follow the state transition of the original model (decomposed process state space) and, in addition, the messages consumed and sent by them are disjoint (decomposed channel content).

Next, we formalize what it means for two processes to be *replicas*. Again, we go through discussing the formal definition afterwards.

Definition 9. *Given a model of a message-passing system, two processes* $i \neq j$ *are* replicas *if the following holds:*

- *(Channel replicas between* i *and* j*)* $c_{i,j} \in C$ *iff* $c_{j,i} \in C$.
- *(Channel replicas)* $\forall k \in \{1, ..., n\} \setminus \{i, j\}$:
  - *(Outgoing channels)* $c_{i,k} \in C$ *iff* $c_{j,k} \in C$, *and*
  - *(Incoming channels)* $c_{k,i} \in C$ *iff* $c_{k,j} \in C$, *and*

- *(Bi-simulation of local transitions)* $\exists s_1, s_2 \in S, t \in T_i : s_1 \xrightarrow{t} s_2$ *iff* $\exists s'_1, s'_2 \in S, t \in T_j : s'_1 \xrightarrow{t} s'_2$ *such that for all channels* c *of process* i *and the corresponding channel replica* c' *of process* j :

  - *There is a bijection* b *from* $s_1(c)$ *to* $s'_1(c')$ *(and* $s_2(c)$ *to* $s'_2(c')$*), if channel* c *is an incoming (outgoing) channel of process* i*, and*
  - *A message* $m \in s_1(c)$ *(and* $m \in s_2(c)$*) is consumed (sent) by* t *iff* $b(m) \in s'_1(c')$ *(*$b(m) \in s'_2(c')$*) is consumed (sent) by* t'*.*

According to Definition 9, replicas maintain a channel to/from the same set of processes (channel replicas). Also, there is a one-to-one correspondence (bi-simulation) between local transitions of the replicas such that the content of incoming and also outgoing channels of the replicas are identical modulo renaming (bijection). This means that in course of every transition, the first replica consumes/sends some messages and the second replica consumes/sends the renamed counterparts of these messages.

3.3.3 Finding Symmetries via Decomposition

A heuristic for finding symmetries Algorithm 1 shows our approach to find symmetries using decomposition and replica processes. The algorithm takes an initial model of a message-passing system as input and returns a model (possibly different from the input model) and verified symmetries of this model. These symmetries can be exploited by symmetry reduction.

Algorithm 1 proceeds through the following steps:

1 (Decomposition: lines 1.2-1.6) The input model is decomposed with respect to some process until there is such a process. The intuition behind this step is that symmetries are better exposed in fine-grained decompositions of the system. We call Algorithm 1 without lines 1.2-1.6 the *naive approach*.

2 (Scalarsets to index replicas: lines 1.7-1.8) Replica processes are assumed to be symmetric and, thus, are identified using scalarset types.

3 (Verification of scalarset restrictions: lines 1.9-1.11) If the scalarset restrictions can be verified (line 1.9), then the model with scalarset types is returned (line 1.10). Otherwise the algorithm returns the input model without scalarset symmetries (line 1.11).

Algorithm 1: Finding symmetries via decomposition and replica processes.

Input : Model M of message-passing system
Output: Decomposed and symmetric model

```
1.1  M_0 ← M;
1.2  while true do
1.3      if decomposition M' of M with respect to some process exists then
1.4          M ← M';
1.5          continue;
1.6      break;
1.7  forall the replica processes i and j in M do
1.8      Be i and j from the same scalarset;
1.9  if scalarset restrictions can be verified for M then
1.10     return M;
1.11 return M_0;
```

Example 3. Following up on Example 2, we show that Algorithm 1 is able to find that there is a symmetry mapping s to s'. The original model can be decomposed with respect to every process, where the resulting model contains a new acceptor and proposer process. Therefore, the states corresponding to s and s' in the decomposed model look like this:

- $s(1)$ =(**"Acceptor: initial state"**).
- $s(2)$ =(*"Acceptor: Phase 1(b) promise with* $n = 27$ *sent to process 3"*).
- $s(3)$ =("Proposer: Phase 1(a) message with $n = 35$ sent to process 4").
- $s(4)$ =("Proposer: initial state").
- $s'(1)$ =(*"Acceptor: Phase 1(b) promise with* $n = 27$ *sent to process 3"*).
- $s'(2)$ =(**"Acceptor: initial state"**).
- $s'(3)$ =("Proposer: Phase 1(a) message with $n = 35$ sent to process 4").
- $s'(4)$ =("Proposer: initial state").

Since every pair of acceptors (and proposers) are replicas, in the model returned by Algorithm 1, the IDs of acceptors (proposers) are from the same scalarset. It can be shown that these scalarsets satisfy the scalarset restrictions [BSSV09]. The permutation of the scalarset values swapping the IDs

1 and 2 yields s'. Therefore, there is indeed a symmetry that maps s to s'. (End of Example 3)

Soundness The soundness of Algorithm 1 is guaranteed by the verification of the scalarset restrictions (line 1.9). Therefore, symmetries returned by the algorithm are indeed symmetries even in case the system is wrongly decomposed or two processes are falsely identified as replicas. In fact, the formal definitions of decomposition (Definition 8) and replica processes (Definition 9) specify intuitive guidelines, whose implementation can be done (automatically or manually) with varying precision.

Note that, given that the scalarset restrictions are of syntactic nature, Step 3 of the algorithm can be automated by simple parsing of the model specification. As an optimization, if Step 3 fails, the algorithm can re-try from Step 2 with different scalarsets.

Reduction analysis Algorithm 1 is a heuristic, which in worst-case returns the input model and no symmetries. Also, there is no guarantee that the model contains no symmetries in addition to the returned ones. However, we show examples of message-passing protocols in Section 3.6, for which Algorithm 1 finds more symmetries than the naive approach.

Theoretically, Algorithm 1 can find exponentially more symmetries than the naive approach. Given a state s, the maximum number of symmetries π detected by Algorithm 1 such that s and $\pi(s)$ are different (symmetric) states is $F = \prod_{i=1\ldots R} n_i!$ where R is the number of scalarsets (i.e., replica classes) and n_i is the size of the i^{th} scalarset (i.e., the number of replicas in the i^{th} replica class). This is because the local states of replicas can be permuted arbitrarily independent of the local states of other replicas. Therefore, if $|S|$ is the number of states in the unreduced model, then the symmetry reduced state space contains at least $|S|/F$ states. Let $|S_r|$ denote the actual number of states contained in the symmetry reduced state space. The fraction $\frac{|S|/F}{|S_r|}$ (or expressed as percentage) is called the *efficiency* of the scalarset approach. Note that the efficiency is always ≤ 1 because $|S|/F$ is the maximum theoretical gain and, thus, $|S|/F \leq |S_r|$.

Assuming that every composed process contains one from each replica class ($n = n_i$ for all i), the number of symmetries detected by the naive approach is at most $n!$ where n is the number of processes. This is $(n!)^{R-1}$ times less than the number achieved with Algorithm 1.

3.4 Quorum Transitions

A general pattern in fault-tolerant message-passing protocols is that a transition consumes multiple messages by a single execution. We call such transitions *quorum transitions* and advocate their use in the modeling of message-passing protocols.

We start our discussion by formally defining quorum transitions.

Definition 10. *Given a model of a message-passing system, a transition $t \in T$ is called a* quorum transition *if there is $s, s' \in S, X \subseteq M$ such that $s \xrightarrow{X} s'$ and $|X| > 1$. Otherwise, t is called a* single-message *transition.*

As a general example of quorum transitions, consider systems that guarantee reliability under the assumption that the number of faulty processes lies below a given threshold and each correct process (replica) executes an instance of the same replicated service [AW04, Bir05]. The threshold assumption implies that a set of messages from a large enough subset (or quorum) of processes contains at least one message from a correct process. Therefore, a common technique in such systems is that the execution of an event is triggered when a set of messages from a quorum (e.g., a majority) of processes is received.

Sound modeling with quorum transitions Although it is possible to represent quorum transitions via a sequence of single-message transitions, quorum transitions enable a natural modeling of certain systems, as shown in the below example. An important difference between single-message and quorum transitions is that while a single-message transition results in a new state after consuming every single message, a quorum transition can consume multiple messages by one, indivisible state transition. This difference in the semantics can be relevant for the precise modeling of a system.

Example 4. Phase 2(a) of the Paxos algorithm requires that a proposer receives a response to a prepare request from a majority of acceptors. This can be modeled by a transition t such that if a set X of messages is accessible for t in some state, then $|X| \geq \lceil n_a/2 \rceil$, where n_a is the number of acceptors.

It is possible to model Phase 2(a) of Paxos using single-message transitions. For example, a proposer can maintain a *counter*, which is incremented by a single-message transition every time the proposer receives a response message to its prepare request. If enough ($\geq \lceil n_a/2 \rceil$) response messages have been received, Phase 2(a) can be executed. However, a counter is not part of the specification of Paxos. It is an auxiliary concept to implement the Paxos algorithm. Using quorum transitions, Paxos can be directly modeled without auxiliary mechanisms. (End of Example 4)

If it is more appropriate to model the system (or a sub-system) with quorum transitions than with single-message transitions, then it is possible to do so even if quorum transitions are implemented via a sequence of single-message deliveries. In this case, the model of the system can be divided into high-level and low-level sub-models. The high-level sub-model contains quorum transitions that are modeled via, for instance, single-message transitions at low-level. Such composite modeling not only result in modular models, they might also enable efficient verification where the different levels of the model can be verified independently of each other.

State space implication Using quorum transitions or, instead, a sequence of single-message transitions can greatly effect the size of the state space. To see this, consider a model M_1 of a message-passing system and transitions $t_1, ..., t_k$ that are enabled in some state s. Depending on the order of execution, the number of different states resulting from executing $t_1, ..., t_k$ is at most $k!k$. Let t be a quorum transition that is enabled in s for a set X and $s \xrightarrow{t(X)} s'$ for some s'. Assume another model M_2 of the same system that contains only single-message transitions. The shortest path from s to s' in M_2 contains at least $|X| = l$ transitions, because any transition in M_2 can consume at most a single message. If these transitions are enabled in s, then the number of states is at most $(k+l)!(k+l)$, which is at least $(k+l)^2$ times more states than $k!k$ in M_1. We know that $k \leq |T|$ where $|T|$ is the number of all transitions in M_1. If we assume that $l \leq n$, i.e., t consumes at most one message from each process, then M_2 can have $(|T| + n)^2$ times more states than M_1.

3.5 Transition Refinement for Efficient POR

Partial-order reduction (POR) [Val98, God96, CGP99] is a powerful optimization of state space exploration. POR is defined over the transitions of a state transition systems. It is known that the efficiency of POR can be improved if the state transition system specifies "small transitions" [God96]. Therefore, transitions can be *refined* into smaller transitions if this transition refinement preserves the properties of the original system. Unfortunately, overly refined transitions can achieve the negative effect, namely, worsening the efficiency of POR.

In this section, we develop transition refinement strategies that are applicable for a general class of message-passing protocols. To prove that the proposed strategies are property preserving, we generalize and formalize transition refinement (Section 3.5.1) and show our strategies as special cases thereof

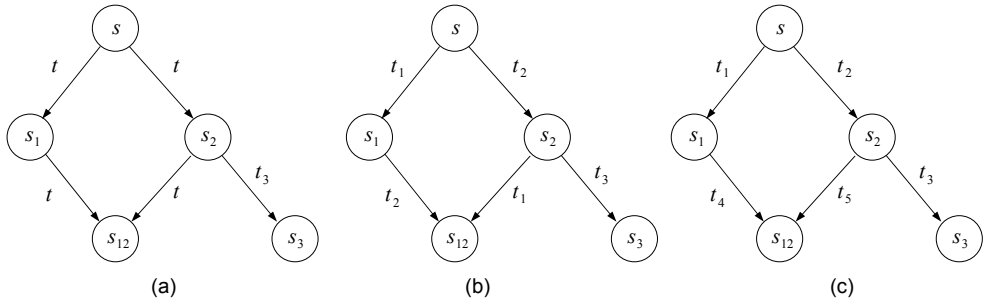

Figure 3.2: (a) Unrefined transition t. (b) Refined transitions t_1 and t_2. (c) Overly refined transitions.

(Sections 3.5.2 and 3.5.3). The experiments in Section 3.6 show significant improvements of POR using these strategies.

Before introducing the proposed approach, we demonstrate transition refinement by examples.

Example 5. In state transition system Figure 3.2(a), no POR is possible, i.e., all enabled transitions must be executed. To see this, consider for example the definition of stubborn sets (Section 2.5), a general notion of POR. Refining transition t into transitions t_1 and t_2 results in state transition system (b) where POR allows executing only t_2 in s. However, if t_1 and t_2 are further refined into t_4 and t_5, then, again, no POR is possible. (End of Example 5)

3.5.1 Generalized Transition Refinement

Transition refinement is a transformation of a state transition system into another one such that the state space remains intact. Note that the following general definition does not require that the original transition set contains fewer transitions than the refined one, although this case is the usual application of transition refinement.

Definition 11. *Given state transition systems* $STS = (S, T, S_0)$ *and* $STS' = (S, T', S_0)$*,* STS *is a* transition refinement *of* STS' *if for all* $s_1, s_2 \in S$ *the following holds:* $\exists t \in T : s_1 \xrightarrow{t} s_2$ *iff* $\exists t' \in T' : s_1 \xrightarrow{t'} s_2$.

The following theorem proves that POR and transition refinement preserves the specification of the system. The conditions of this property is that

(a) the specification is preserved by POR, and (b) the specification is interpreted over sequences of states (not over transitions). A general class of such specifications is the subset of CTL* that is preserved by POR [Val98, KSV06].

Theorem 3. *Let STS_1 and STS_2 be two state transition systems and φ a CTL* formula that is preserved by POR. Then, if STS_2 is a transition refinement of STS_1 and STS_2^R is the partial-order reduction of STS_2, then φ holds in STS_1 iff it holds in STS_2^R.*

Proof. Assume that φ holds in STS_1 but not in STS_2^R. Since STS_2 is a transition refinement of STS_1, there is a path in STS_1 iff this is a path in STS_2. Furthermore, because the semantics of CTL* is over sequences of states, φ also holds in STS_2. From the property preservation of POR we know that φ holds in STS_2^R, a contradiction. The reverse can be proven similarly. □

3.5.2 Strategy 1: Quorum Split

Our first transition refinement strategy for message-passing systems is called *quorum split*. The idea of quorum split is to define a new transition for each set of processes from which a transition can consume a message.

We start by defining a special class of transitions, called *exact quorum transitions*, for which quorum split is eligible. Exact quorum transitions are quorum transitions where the number of processes from which the transition consumes a message is fixed. Note that an exact quorum transition can be a single-message transition.

Given two states s, s', a transition $t \in T_i$, and a set X of messages such that $s \xrightarrow{t(X)} s'$, we define an auxiliary notation $senders(X)$ to be the set of processes that have sent a message in X, i.e., $\{j \mid m \in X \cap s(c_{j,i})\}$.

Definition 12. *Given a message-passing system, a transition t is an* exact quorum transition *with threshold q_t if $s \xrightarrow{t(X)} s'$ implies $|senders(X)| = q_t$ for all $s, s' \in S$ and set X of messages.*

Next, we define the quorum split transition refinement strategy.

Definition 13. *Given a message-passing system M_1 and an exact quorum transition t with threshold q_t, a* quorum split *of M_1* via *t is a message-passing system M_2 derived from M_1 by replacing t with transitions $t_1, t_2, \ldots, t_m$, for $m = \binom{n}{q_t}$, such that $s \xrightarrow{t_k(X)} s'$ iff $s \xrightarrow{t(X)} s' \wedge senders(X) = Q_k$, where $s, s' \in S$ and Q_k is the k^{th} of the m sets of process IDs of size q_t.*

Note that in principle every transition t can be split by adding a new transition t_Q for every subset Q of processes, even if t is not an exact quorum transition. This would mean adding 2^n extra transitions for every t (n is the number of all processes). However, an exponentially number of new transitions can prohibitively increase the overhead of POR.

The following theorem states that quorum split yields a transition refinement.

Theorem 4. *Given a message-passing system M_1, let STS_1 be the state transition system determined by M_1, t an exact quorum transition in M_1, M_2 a quorum split of M_1 via t, and STS_2 the state transition system determined by M_2. Then, STS_2 is a transition refinement of STS_1.*

Proof. Indirectly, assume that STS_2 is not a transition refinement of STS_1. Let T_1 and T_2 be the sets of transitions of STS_1 and STS_2, respectively. By construction, STS_1 and STS_2 share the same set of states (and initial states). Therefore, the indirect assumption is only possible if there are states s and s' such that (a) there is transition $t_1 \in T_1$ with $(s, s') \in t_1$ but for no $t_2 \in T_2$ holds that $(s, s') \in t_2$ or (b) there is transition $t_2 \in T_2$ with $(s, s') \in t_2$ but for no $t_1 \in T_1$ holds that $(s, s') \in t_1$. We consider (a) first. In model M_1, let X be a set of messages such that $s \xrightarrow{t_1(X)} s'$. If $t_1 \neq t$, then $t_1 \in T_2$, a contradiction. Since t is an exact quorum transition, it must be that $|senders(X)| = q_t$. M_2 is a quorum split of M_1 via t, so there is $t_k \in T_2$ such that $s \xrightarrow{t_k(X)} s'$ where $Q_k = senders(X)$, a contradiction. The reverse, case (b), can be shown similarly. □

An optimization The number of new transitions can be reduced by identifying a process i that never sends messages consumed by the refined transition t. In this case, if t is executed in a state with a set X of messages, then i cannot be in $senders(X)$, thus, t_k with $i \in Q_k$ can be excluded from the quorum split. The automatic detection of all possible $senders(X)$ sets can be done using static analysis, otherwise we conservatively assume that i can be in such a set.

Finally, we show an example quorum split of a model of the Paxos protocol.

Example 6. Consider the quorum transition t from Example 4. t is an exact quorum transition with threshold $\lceil n_a/2 \rceil$ assuming that Phase 2(a) is executed after the proposer has received a response message from a *minimum* majority of the acceptors. Assume a system with three acceptor processes ($n_a = 3$). Let 1, 2, and 3 be the IDs of these acceptor processes. The model

resulting from the quorum split of the original model via t contains new transitions t_1, t_2 and t_3 where $Q_1 = \{1, 2\}, Q_2 = \{1, 3\}$ and $Q_3 = \{2, 3\}$. For example, t_2 models Phase 2(a) where the proposer receives response messages from acceptors 1 and 3 (but not 2). Note that no transitions t_k need to be added where Q_k contains IDs of non-acceptor processes. This is because a proposer only receives messages from acceptors. (End of Example 6)

3.5.3 Strategy 2: Reply Split

The second proposed transition refinement strategy is an application of quorum split to a special class of transitions, called *reply transitions*. A reply transition is when a process receives one or more messages and sends messages only to the senders of these messages (e.g., acknowledgement). We call a quorum split via a reply transition *reply split*. The formal definition of reply transitions is as follows.

Definition 14. *Given a message-passing system, $t \in T_i$ is a* reply transition *if for all $s, s' \in S$ and for all sets X of messages, $s \xrightarrow{t(X)} s'$ implies that $\{j \mid s(c_{i,j}) \subset s'(c_{i,j})\} = senders(X)$.*

The reason that we discuss reply split as a new strategy is because a quorum split via reply transitions is differently advantageous for POR than a quorum split via general (non-reply) transitions. This has to do with "interfering" transitions, a concept that affects the efficiency of POR. There are different notions of interfering transitions [God96, CGP99]. For example, two transitions of different processes are interfering if the first transition sends a message that is consumed by the second transition. Intuitively, the fewer interfering transitions the more efficient POR.

Quorum split is able to decrease the number of interfering transitions in two ways:

- *General quorum split.* Quorum split via any transition t decreases the number of transitions that *interfere with* t: While every transition t' that sends a message to t interferes with t, after quorum split, t' only interferes with t_k (a new transition resulting from the quorum split via t) if the process executing t' is in Q_k.

- *Reply split.* If quorum split is done via a reply transition t, then transition t_k that results from the reply split of t can only send messages to processes in Q_k. Therefore, although t might *interfere with* with all transitions, t_k only interferes with transitions executed by processes in Q_k.

We close our discussion with an example of the reply split strategy.

Example 7. Consider Phase 1(b) of the Paxos consensus algorithm (see Section 2.3). Phase 1(b) is executed by acceptor processes and it can be modeled by a single-message reply transition t that responds to a request of a proposer process. Before reply split, t interferes with the transitions of *any* proposer. Consider a reply split via t in a model with 2 proposers with IDs 1 and 2. In this case, t is replaced by new transitions t_1 and t_2 with $Q_1 = \{1\}$ and $Q_2 = \{2\}$. As a result of reply split, t_1 (and t_2) only interferes with the transitions of proposer 1 (proposer 2). (End of Example 7)

3.6 Experiments

We demonstrate the efficiency of our heuristic for finding symmetries (Section 3.6.1) and the quorum and reply split transition refinement strategies (Section 3.6.2). We start from natural language specifications of representative message-passing protocols and create (manually) a model of each protocol. As our experiments use explicit state model checkers that explore states one-by-one, we apply **M** models for their smaller state spaces compared to **RM** and **GRM**, as implied by the results of Section 3.1. This is sound because the example message-passing systems are process-labeled. Also, we utilize quorum transitions whenever possible, similarly to Example 4.

Verification efficiency is measured by the number of explored states and the time of state space exploration. The result of each experiment is "Verified" if the system is correct (no counterexample is found), "Out of mem." if the the model checker runs out of memory, and "CE" if the model checker returns a counterexample. We also evaluate the efficiency of the proposed techniques for debugging: Faulty versions of the example message-passing protocols as well as wrong specifications are used to trigger the model checker to find a counterexample. In our experiments, the search terminates after finding the *first* counterexample. In this case, the state space is not explored exhaustively, i.e, there might be other counterexamples of the specification.

3.6.1 Symmetry Reduction with Murφ

Murφ We use the Murφ model checker [Mur] to evaluate the efficiency of Algorithm 1. Murφ implements the scalarset approach and symmetry reduction. Algorithm 1 is not automated in our experiments. We manually decompose models, identify replicas, and verify the scalarset restrictions. Since Murφ supports the verification of invariants (see Section 2.1) only, we restrict the specification such that it is expressible via invariants.

Example message-passing protocols We consider the following two message-passing protocols:

- The Paxos consensus protocol, explained in Section 2.3. Paxos processes can be decomposed into proposer and acceptor processes. For fault-tolerance, both proposers and acceptors are replicated. We consider the safety requirements of consensus for Paxos because liveness cannot be expressed as an invariant. We assume a system with three acceptors and at most two proposers, each proposer proposing at most one proposal. Learners are not modeled.

  Applying the results from Section 3.2, we use an implicit model of crash faults because this yields a smaller state space than the explicit model. Since Paxos is process/crash-labeled, the equivalence result (Theorem 2) applies.

- The Oral Messages OM(1) consensus protocol [LSP82], which is able tolerate one Byzantine process in a synchronous environment (reliable channels). OM(1) conceptually differs from Paxos as (a) it does not assume crash faults and (b) it assumes synchrony. OM(1) processes can be decomposed into a commander process and multiple lieutenant processes.[2]

  In the context of OM(1), the (safety and liveness) requirements of consensus are often called Interactive Consistency (IC). IC assumes for OM(1) that the system contains at least three lieutenants. Thanks to synchrony, OM(1) solves consensus (unlike Paxos) after a bounded number of steps. Therefore, the liveness requirement of consensus can be expressed by invariants using "monitors", which remember fractions of a state and use these state fractions as a reference in other states.

  In our model of OM(1), the possible faulty (Byzantine) behavior is *exhaustively* modeled. The only restriction we make is that Byzantine-faulty processes send well-formatted messages. Otherwise, the message (and its sender) can be detected to be erroneous and the message is discarded. Our experiments show that this exhaustive model of Byzantine faults is feasible for verification. This can be attributed to the number of faulty processes being restricted to be at most one and that the proposed value in OM(1) is binary ("attack" or "retreat").

[2]The roles, commander and lieutenants are analogous with proposer and acceptors in Paxos.

Experiment setup The experiments run on DETERlab machines [DET], equipped with a Xeon processor and 4 GB memory, and running a Linux installation with Fedora 6 core. We utilize for symmetry reduction the (default) fast canonicalization heuristic of Murφ.

We consider the following heuristics for finding symmetries:

- *None*: No scalarsets are used; therefore, no symmetries are detected and symmetry reduction is switched off. The Murφ models of all example protocols (also the decomposed ones augmented with scalarsets) are available on-line under http://www.deeds.informatik.tu-darmstadt.de/peter/thesis/SR/.
- *Naive approach*: The previous "none" heuristic augmented with scalarsets to index processes.
- *Algorithm 1*: Following Algorithm 1 with inputs from the "none" heuristic models.

Debugging We consider the following faulty protocols and wrong specifications:

- *Faulty Paxos*: An acceptor always accepts a proposal in Phase 2(b) even if an earlier promise forbids to do so.
- *Wrong safety*: A proposal in Paxos is considered to be chosen if it is accepted by an acceptor. This is a wrong specification because a proposal is chosen only if a majority of acceptors has accepted it.
- *Faulty OM(1)*: There are two faulty processes, which is a violation of OM(1)'s assumption that at most one process is faulty.

Results & discussion The results of our experiments are shown in Table 3.2. In addition to the number of visited states and exploration time, we derive two metrics from the number of visited states: (1) *gain* is the fraction of the number of visited states without and with symmetry reduction and (2) the efficiency of scalarsets (see Section 3.3.3).

The results of our experiments can be summarized as follows:

- *Significant symmetry reduction:* Symmetry reduction is able to significantly reduce the time and memory resources of both verification and debugging visiting at least one magnitude fewer states with Algorithm 1 than the unreduced search. In particular, OM(1) with 5 lieutenants

Protocol	Property	Result	Heur. (# proc.)	States	Gain	Effic.	Time
Paxos	Safety	Verified	None (3)	1,591,897	-	-	268 s
			Naive (3)	795,945	2x	33%	226 s
			Alg. 1 (5)	**136,915**	**12x**	**96%**	**32s**
	Wrong safety	CE	None (3)	649,301	-	-	61 s
			Naive (3)	325,074	2x	-	226 s
			Alg. 1 (5)	**57,677**	**11x**	-	**12 s**
Faulty Paxos	Safety	CE	None (3)	1114,891	-	-	126 s
			Naive (3)	562,298	2x	-	122 s
			Alg. 1 (5)	**101,239**	**11x**	-	**20s**
OM(1)	IC	Verified	None (3)	1,797	-	-	0.1 s
			Naive (3)	941	2x	31%	3 s
			Alg. 1 (4)	**345**	**5x**	**85%**	**0.1s**
	IC	Verified	None (4)	150,417	-	-	9.6 s
			Naive (4)	26,401	6x	24%	17 s
			Alg. 1 (5)	**6,999**	**22x**	**90 %**	**7.4s**
	IC	Verified	None (5)	Out of mem.	-	-	-
			Naive (5)	2,402,167	-	-	4 h
			Alg. 1 (6)	**490,839**	-	-	**2h**
Faulty OM(1)	IC (safety)	CE	None (3)	934	-	-	0.1 s
			Naive (3)	843	1.1x	-	2.9 s
			Alg. 1 (4)	**200**	**5x**	-	**0.1s**

Table 3.2: Algorithm 1 in comparison with the naive approach using the Murφ model checker.

could not be verified without symmetry reduction because the queue of unexplored states in Murφ runs out of memory. The reason why model checking time can show different trends than memory is the time overhead of symmetry reduction.

- *Algorithm 1 outperforms naive approach:* Algorithm 1 outperforms the naive approach in *all* experiments with respect to the number of visited states and model checking time (these experiments are depicted in bold). The gain of symmetry reduction with Algorithm 1 is up to six times more the with the naive approach. In addition, the time of model checking can be up to 30 times faster with Algorithm 1 than with the naive approach (OM(1) with 3 and 4 processes).

  Symmetry reduction with the naive approach can take longer than the unreduced search due to the overhead of symmetry reduction. This is not the case with Algorithm 1 because the increased state reduction adds up to an overall reduction of model checking time.

- *Algorithm 1 finds efficient symmetries:* Both protocols are almost optimally symmetric with respect to their decomposed replicas (approaching 100% efficiency). We observe that the difference between the achieved and maximum gain is within 3% of the size of the unreduced state space. Note that the efficiency cannot be computed for the

experiments with counterexamples or for verification of OM(1) with six processes because the size of the unreduced state space is unknown.

The efficiency of the naive approach is considerably lower. Even in the case of OM(1), where the theoretical maximum benefit is the same for Algorithm 1 and the naive approach, the efficiency of Algorithm 1 is significantly higher. This is because the local states of lieutenants in the composed models cannot be freely permuted because there is one process that contains both the commander and a lieutenant.

3.6.2 Partial-Order Reduction with MP-Basset

MP-Basset The quorum and reply split experiments are conducted with the MP-Basset model checker. MP-Basset is a model checker specifically for message-passing systems, which supports different partial-order reduction (POR) implementations. A detailed description of MP-Basset is given in Chapter 6. Similarly to Murφ, the current version of MP-Basset supports invariants only, which restricts our experiments to specifications that are expressible via invariants. The split (transition refined) models are created manually in these experiments.

Example message-passing protocols We consider the following protocols:

- The first protocol is the Paxos consensus protocol. The model and analyzed specification are as in Section 3.6.1 except that the model contains a learner. In this model, the agreement property of consensus can be specified as a process local property: The learner must not learn proposals with different values. In the context of POR, this model is more convenient to work with as some of the existing POR property preservation results [God96, FG05, SA06] relate to process local properties.

- The second example is a consistent *multicast* protocol called Echo Multicast [Rei94]. We consider the *agreement* property of consistent multicast that specifies that no two processes (called receiver) receive different messages sent (via multicast) by a process (called initiator). Echo Multicast implements agreement in a Byzantine environment [LSP82] where up to one third of the processes can fail arbitrarily and the remaining processes are correct (also called *honest*).

- Our third example is a *regular storage* protocol in the style of [ABND95]. The objective of (distributed) storage is to reliably store

data despite failures of (undistributed) base (storing) objects. A regular storage guarantees that a read operation (executed by processes called *readers*) returns a value not older than the one written by the latest preceding write operation (executed by processes called *writers*). The protocol assumes a crash-tolerant setting where a minority of all base objects might crash. Similar to Paxos, crash faults are modeled implicitly. Furthermore, this protocol is *single-writer*, i.e., there is exactly one writer process.

Byzantine fault-model The multicast protocol allows arbitrary faulty behavior. It turns out that an exhaustive model of faulty behavior (as in case of OM(1) from Section 3.6.1) is infeasible for the verification of this protocol. Therefore, we restrict the behavior of a Byzantine-faulty process. We consider "meaningful" faults to challenge the protocol's ability to guarantee agreement. We define the following attack strategies:

- A Byzantine initiator attempts to violate the agreement property by sending different messages to each of two groups of honest receivers.
- A Byzantine receiver sends invalid confirmations to an honest initiator and co-operates with a Byzantine initiator by confirming (signing) both of its messages.

Debugging We consider the following experiments where we expect the model checker to return a counterexample:

- *Faulty Paxos*: The learner does not compare the values in accepted proposals received from acceptor processes. This is contra the specification of Paxos where a learner learns a proposal if the *same* proposal (including proposal number and value) is accepted by a majority of acceptors.
- *Wrong agreement*: In Echo Multicast we exceed the threshold of the number of maximum Byzantine processes. In particular, we consider a setting with two honest receivers, one honest initiator, two Byzantine receiver, and one Byzantine initiator.
- *Wrong regularity*: We require that a read operation that completes after a write has to return the value written by the write even if the two operations are concurrent.

Protocol (# proc.)	**Property**	**Result**	**Heur.**	**States (POR)**	**Time (POR)**
Paxos (6)	Safety	Verified	None	2,822,764	9h37m
			Quorum	1,826,560	11h28m
			Reply	1,087,486	3h47m
			Combined	**548,061**	**3h30m**
Faulty Paxos (6)	Safety	CE	None	279	10s
			Quorum	279	10s
			Reply	105	8s
			Combined	**105**	**8s**
Echo Multicast (3,0,1,1)	Agreement	Verified	None	652	12s
			Quorum	232	12s
			Reply	652	12s
			Combined	**232**	**12s**
Echo Multicast (2,1,0,1)	Agreement	Verified	None	2787	31s
			Quorum	2787	31s
			Reply	1165	18s
			Combined	**1165**	**18s**
Echo Multicast (3,1,1,1)	Agreement	Verified	None	12,023,663	>48h
			Quorum	7,600,843	>48h
			Reply	>10,472,557	>48h
			Combined	**7,087,193**	**42h21m**
Echo Multicast (2,1,2,1)	Wrong agreement	CE	**None**	**48**	**6s**
			Quorum	48	9s
			Reply	48	7s
			Combined	48	9s
Regular storage (3,1)	Regularity	Verified	None	20,039	3m4s
			Quorum	18,451	4m31s
			Reply	18,451	3m13s
			Combined	18,451	4m32s
Regular storage (3,2)	Wrong regularity	CE	None	41,331	6m46s
			Quorum	29,877	9m51s
			Reply	**6,969**	**1m32s**
			Combined	6,987	2m34s

Table 3.3: Quorum and reply split in comparison with unsplit transitions using the MP-Basset model checker.

Experiment setup All experiments run on DETERlab machines [DET] with Xeon processors and 4 GB of memory. In case of the Paxos protocol, we assume a system with three acceptors, at most two proposers, and a single leader. We use different configurations for Echo Multicast and regular storage. For the former, a setting (HR,HI,BR,BI) specifies the number of honest receivers, initiators, Byzantine receivers and initiators, respectively. Every initiator sends at most one message via multicast. For regular storage, (B,R) gives the number of base objects and readers, respectively. The writer (and each reader) initiates a single write (at most one read) operation.

We consider the following transition refinement strategies:

- *None*: Transitions are unrefined as in the "natural" specification of the protocol. All models (unsplit and split) used in our experiments are available on-line [MP-].
- *Quorum split*: Quorum split via non-reply quorum transitions only.
- *Reply split*: Quorum split via reply transitions only.
- *Combined split*: Quorum split of all transitions.

The POR algorithm used in our experiments is the LPOR algorithm, which is discussed in details in Chapter 4. LPOR is a "static" implementation of POR, which resembles other static implementations. Therefore, we argue that the observed trends apply to other POR implementations too.

Results & discussion The results of our experiments are shown in Table 3.3. For each protocol and setting, we highlight in bold the search strategy (if any) if this outperforms the other strategies both in terms of the number of visited states and model checking time. We observe the following trends:

- *Significant reductions with splits:* Our split strategies achieve significant reductions in terms of both memory and time (up to 81% and 64% for Paxos) compared to the unsplit case. A reduction of model checking time can be achieved although the *throughput* of model checking (number of visited states per time unit) falls with transition refinement because of the additional time overhead of computing POR (due to the increased number of transitions). Despite this additional time overhead, the significant space reduction adds up to an overall time reduction. For example, in case of Paxos the throughput without splits is 90 states/second versus 43 states/second with combined split and, still, the experiment with combined split finishes earlier.

Note that the throughput is lower with quorum split than with reply split. For example, in the case of verified regular storage the experiments with quorum and reply splits visit the same number of states but the experiment with quorum split takes more than a minute longer. This is because quorum split via quorum transitions triggers a high time-overhead optimization of the LPOR algorithm (called necessary enabling transitions – details in Chapter 4).

- *Efficient debugging:* Quorum and reply splits can also be used to find bugs fast and using little memory. If the bug is "deep" in the search space (unlike in case of faulty Paxos or multicast with wrong agreement), we observe similar trends as for verification (see regular storage with wrong regularity).

- *Caveat:* The quorum and/or reply split strategies can be ineffective. In this case, the search strategy with the least overhead achieves the best result, for instance, in the experiment Echo Multicast (2,1,2,1). An example where reply split is ineffective is Echo Multicast (3,0,1,1), where there is a single initiator to which the receivers can reply. Also, quorum split makes no difference for Echo Multicast (2,1,0,1) because the quorum contains all receivers.

  A side effect of transition refinement can be that the order in which transitions are explored is changed.[3] As a result, the number of visited states and exploration time can also change even if transition refinement is ineffective. One possible reason for that is the imperfection of the state comparison mechanism in practical model checking.[4] For example, it can be shown that both splits are ineffective for regular storage (3,1). Still, the number of visited states is (slightly) different for the unsplit and split cases.

3.7 Related Work

Models of message traffic and crash faults Our results of comparing different models of a system and showing equivalences between these models relate to general reduction techniques such as symmetry and partial-order reductions [CGP99] and the corresponding property preservation results. For example, partial-order reduction preserves a class of temporal formulas that

[3] In MP-Basset, transitions with lexicographically smaller IDs are preferred.

[4] MP-Basset adopts the heuristics for state comparison from Java Pathfinder [JPF], the model checker underlying MP-Basset.

are indistinguishable in stuttering equivalent state transition systems. Theorems 1 and 2 can also be seen as reduction techniques preserving certain properties across different models of message-passing systems. The advantage of the proposed reductions compared to symmetry or partial-order reductions is that they apply to *any* message-passing system. Therefore, our reductions yield no time overhead during verification.

Similar to our result of stuttering equivalent models of message traffic, [CSCBM09] shows that a fine-grained model of message-passing algorithms can be reduced to a stuttering equivalent coarse-grained model. This result applies to a special class of message-passing algorithms characterized by (1) communication-closed rounds and (2) crash faults based on the Heard-Of model [CBS09]. The reduction shows that it suffices to model a path of the system as a sequence of synchronized rounds where in each round every correct process sends and receives messages and updates its local state. Our system model is a general one and it does not assume that a path is divided into communication-closed rounds (a message sent in a round must be delivered before the end of this round otherwise the message is considered to be lost), nor does it restrict to crash faults.

Symmetry detection There are various approaches targeting the detection of symmetries with the least user intervention possible [MDC06]. These approaches seem to share the scalarset idea: The syntax of a language used to specify the system is restricted such that the state space contains symmetries by construction. Applications of symmetry reduction to multi-process systems (beginning from [ID96, CJEF96]) assume basic symmetries across processes and do not consider systematic process decomposition to find more symmetries.

Promising applications of the proposed symmetry detection approach include parametric verification, e.g., [PXZ02], where the number of processes is unbounded. In fact, classical symmetry reduction [ID96, CJEF96], which is applied to finite systems, and parametric verification of multi-process systems have in common that they have to establish a notion of symmetry across processes before exploiting these symmetries in verification.

Another possible application is to integrate our symmetry detection heuristic into existing model checkers. For example, the TLC model checker [Lam02] already supports symmetry reduction but it requires the user to (manually) verify symmetries. In particular, TLC supports a language called +CAL [Lam06], which is especially suited for the high-level specification of distributed multi-process algorithms.

Model checking of self-stabilizing algorithms was proposed by using sym-

bolic techniques that are (under favorable conditions) insensitive to the large number of initial states [TNPK01]. It is also possible to apply symmetry reduction to such algorithms (powered by our symmetry detection heuristic) in which case different but symmetric initial states need not be explored. Combined with explicit state model checking, such an approach does not suffer from the drawbacks of symbolic model checking.

Quorum transitions Our formal models of message-passing systems starting with a reference model adapted from [AW04] can be seen as actor programs [AMST97]. A concept similar to quorum transitions appears as joint transitions of actors [FA95]. The actor model implements rich semantics, e.g., synchronization between actors or dynamic creation of actors, to make the formalism expressive. In contrast, the main motivation of introducing quorum transitions in this thesis is to enable efficient verification.

Transition refinement Partial-order reductions [God96, CGP99, FG05, KWG09b] specify sufficient conditions of preserving certain classes of properties *given* the transitions of the system. Therefore, transition refinement is orthogonal to these approaches. A concept similar to transition refinement is operation refinement from [God96]. There, operation refinement is discussed informally in the context of a proof-of-concept modeling language and its effect on the performance of partial-order reduction implementations is not studied. In addition, no generally applicable operation refinement strategy is proposed nor it is evaluated in practical verification.

3.8 Conclusions

We have proposed reductions of formal models of message traffic (Section 3.1), crash-faulty processes (Section 3.2), and quorum transitions (Section 3.4) into models that are more amenable to verification. We see the main strength of these contributions on the practical side. Our reduction results formally verify natural intuitions such as the soundness of using unclean channels or that crash events need not be modeled explicitly. As a result, the verification with the reduced models serves as a formal argument rather than just a "reasonable simplification". As natural extensions, systems with different message-passing characteristics (e.g., finite channels) or fault models (e.g., malicious processes [LSP82]) can be considered.

We have also proposed models of message-passing protocols to improve on the efficiency of symmetry (Section 3.3) and partial-order reduction (Section 3.5). Again, these contributions are more of practical relevance. Symmetry

and partial-order reductions have been researched extensively, however, under the assumption that a model of the system is given. However, none of these techniques achieves efficient verification if the model of the system is inappropriately specified. The proposed models result from systematic analysis of the characteristics of fault-tolerant message-passing protocols. Other efficient applications of symmetry and partial-order reductions are possible after similar analysis of the particular domain.

Chapter 4

Local Partial-Order Reduction

This chapter presents a new partial-order reduction (POR) implementation called LPOR. LPOR is a general POR implementation applicable for any system whose semantics can be given as a state transition system.

In Section 4.1, we motivate LPOR and structure it in the broad spectrum of the POR landscape. Section 4.2 presents LPOR's intuitive interface and Section 4.3 the LPOR algorithm that implements POR. Then, Section 4.4 shows an application example of LPOR for general message-passing protocols. A prototype implementation of LPOR is presented in Section 4.5, which is used in Section 4.6 to evaluate the efficiency of LPOR using the previous message-passing application and model checking representative fault-tolerant message-passing protocols. We close the chapter with discussing work related to LPOR in Section 4.7.

4.1 Positioning LPOR

Several notions implement the concept of POR [CGP99, Val98, God96], differing from each other in flexibility and efficiency. The commonality of these approaches is that the developer of a model checker is expected to verify complex conditions to guarantee soundness. This hurdle can prevent developers from implementing POR or even lead to erroneous implementations. Therefore, our main motivation is to propose an approach that *simplifies* the conditions to be verified, but gives up neither the *flexibility* nor the *efficiency* of POR. Next, we explain why previous notions of POR are difficult to use and how our approach improves on them.

The general concept of POR lies in the commutativity of non-interfering transitions. Conceptually, a transition is a mechanism to change the state of the system, e.g., a Java method, or the delivery of a message. POR is based

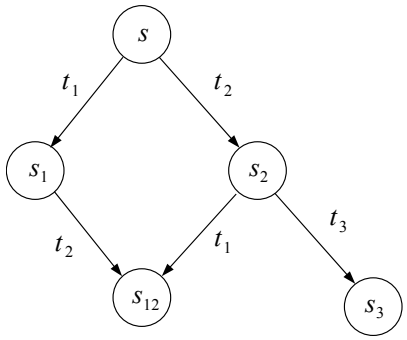

Figure 4.1: Non-interfering transitions t_1 and t_2. States s_{12} and s_3 are deadlocks.

on the simple observation that the execution of a pair of non-interfering transitions leads to the same state irrespective of which of the transitions is executed first. In Figure 4.1, t_1 and t_2 are non-interfering because both paths $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_{12}$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s_{12}$ lead to s_{12}. Therefore, it is sufficient to explore the execution of these transitions in a single representative order, reducing memory and time required for model checking.

POR is *sound* if no state is missed that is relevant for verifying the target property. For example, although t_1 and t_2 are non-interfering, it is an unsound reduction to explore only the path $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_{12}$ if the property states the reachability of s_3. Existing notions of POR define necessary conditions of soundness that are hard to check in general because they require global knowledge about the state graph, which limits the applicability of POR. This problem is usually addressed by fixing the application of POR to a particular specification language and computational model, such that soundness is guaranteed by construction. As a result, existing specification languages with POR support are few and restrictive in different ways: they consider restricted computational models, for example FIFO-based message-passing [HP94, Hol03], Petri nets or process algebras [Val98], they only allow models with deterministic transitions [God96, CGP99] or acyclic state graphs [FG05, SA06], they preserve only invariants [GFYS07, FG05], or they only support bug finding [KWG09a].

We present a novel take on POR, to ease its application to rich specification languages. We call our approach *local* POR (LPOR) because locality is key to simplify the verification of POR conditions for designing new model checkers; in fact, the simplicity of LPOR allows an easy development of new PO reductions. LPOR consists of an input interface (accessible by the user

of LPOR) and a POR algorithm (hidden from the user).[1] At the interface of LPOR, the user defines locally "interfering" transitions, whose soundness can be verified more easily than the global (path-based) soundness conditions in other POR approaches. This local information is sufficient for our LPOR algorithm to efficiently compute sound partial-order reductions. In the example of Figure 4.1, the user can define and verify the following local interferences: t_2 can enable t_3 (when executed in s), and t_1 is dependent on (is disabled by) t_3 (when executed in s_2). Based on this information, the LPOR algorithm knows that t_1 and t_2 are non-interfering and can establish that exploring only the paths $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s_{12}$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_3} s_3$ preserves all deadlock states, a fundamental preservation property used by LPOR to preserve more complex properties.

Our contributions around LPOR are the following:

- **LPOR stubborn set algorithm** LPOR's interface (Section 4.2) contains two intuitive relations between transitions, namely *can-enable* and *dependency*. Each of these relations is local, i.e., they are defined given paths of at most length two. Transitions that are not included in these relations are considered to be non-interfering and are used by LPOR to achieve reduction. The user has to prove the non-interferences correct, but it is sound to declare transitions as interfering even when they are not. An important feature of LPOR is that non-interfering transitions are completely configurable, while other approaches conservatively assume certain transitions to be interfering, e.g., transitions executed by the same process [CGP99]. LPOR also supports *necessary enabling transitions*, which we generalize from [God96]. Although the definition of such transitions does involve paths, they naturally appear in high-level languages.

  The LPOR algorithm (Section 4.3) computes stubborn sets statically [Val98] (see Section 2.5) and supports general transition systems without assumptions about the state graph or transitions. Intuitively, a stubborn set is a large enough subset of the transitions enabled in the current state, e.g., $\{t_2\}$ in s in Figure 4.1, such that no deadlock state remains unvisited if only transitions in stubborn sets are executed. LPOR leverages stubborn sets to preserve properties in the temporal logic CTL^*_{-X}. LPOR is fast thanks to a novel *pre-computation* scheme, which allows to compute information needed by LPOR once, before model checking, and then to repeatedly use it in every new state.

[1] In the remainder of this chapter, by *user* we mean the user of LPOR and not necessarily the end-user of the model checker.

- **Applying LPOR to message-passing** We instantiate the relations at LPOR's interface for general message-passing systems (Section 4.4). This example also shows that the use of LPOR is straightforward for domain experts.

  We briefly discuss two additional LPOR application examples. First, we use a Petri net example in explaining the LPOR algorithm (Section 4.3.1). Second, we show how the POR approach used in the SPIN model checker can be expressed in LPOR terms (Section 4.7).

- **Experiments and comparison with DPOR** We implement LPOR as an openly available Java library called Java-LPOR (Section 4.5) that easily integrates with existing model checkers. As an example use case of Java-LPOR, we implement our message-passing instantiation of LPOR in the Java Pathfinder-based model checker MP-Basset (Chapter 6).

  We evaluate the efficiency of LPOR using message-passing examples. Our experiments show that LPOR achieves significant (up to 94%) time and space reductions for model checking real-world fault-tolerant message-passing protocols (Section 4.6). Furthermore, countering current notions of dynamic POR being superior to static POR [FG05], we also show that LPOR (implementing static POR) competitively improves upon dynamic POR without entailing the constraints of dynamic POR.

4.2 The LPOR Interface: Interfering Transitions

The typical application scenario of LPOR is adding POR to the analysis of systems written in some specification language. Assume that a model checker implementing the LPOR algorithm (Section 4.3) is available for this language. We will show in Chapter 6 how we support the integration of LPOR into existing model checkers. Now, the user, an expert in the domain of the language, must provide two inputs at LPOR's interface. First, unless it is not already available, she must define the semantics of the language in terms of a state transition system. Second, based on her domain-specific knowledge, she defines and proves two intuitive relations containing pairs of interfering transitions. These relations are local considering paths of length at most two. LPOR leverages a third optional relation, which is not strictly local, but naturally appears in high-level languages.

Can-enable relation A transition t *can enable* another transition t', if in at least one state where t' is disabled, executing t results in a state where t' is enabled. We say that a relation is *can-enabling* if it is a superset of all pairs (t, t') of transitions such that t can enable t'.

Definition 15. *A relation* **ce** $\subseteq T \times T$ *is* can-enabling *iff* **ce** $\supseteq \{(t,t') \mid \exists s, s' \in S : s \xrightarrow{t} s' \wedge t' \notin \mathit{enabled}(s) \wedge t' \in \mathit{enabled}(s')\}$.

Dependency relation We define that t' is *dependent* on t if both t and t' are enabled in some state (t and t' are co-enabled) and either (a) t can disable t' or (b) their subsequent execution in different orders results in different states (t and t' do not commute). By convention, t is not dependent on itself. We say that two transitions are dependent (independent) if one (none) of them is dependent on the other. Note that the following relation is not necessarily symmetric.

Definition 16. *A relation* **dep** $\subseteq T \times T$ *is a* dependency *relation, iff* **dep** $\supseteq \{(t,t') \mid t \neq t' \wedge \exists s, s' \in S : t, t' \in \mathit{enabled}(s) \wedge s \xrightarrow{t} s'$ *and either (a)* $t' \notin \mathit{enabled}(s')$ *or (b)* $\exists s'' \in S : s \xrightarrow{tt'} s''$ *and not* $s \xrightarrow{t't} s''\}$.

NET relation Next, we define a relation that contains a pair of transitions t and t' if t' is a *necessary enabling transition (NET)* for t, i.e., t' must be executed at least once for t to be enabled (adapted from necessary enabling sets [God96]). Note that this relation is based on paths. It is purely optional though as it is sound to not include pairs of transitions in a NET relation or, in particular, to define an empty one. Similarly, it is always sound to include a pair of (even non-interfering) transitions in can-enabling and dependency relations.

Definition 17. *A relation* **net** $\subseteq T \times T$ *is a* necessary enabling transition (NET) *relation, iff* **net** $\subseteq \{(t,t') \mid \forall s_0 \in S_0, \forall s \in S, \forall t_1, \ldots, t_n \in T :$ *if* $s_0 \xrightarrow{t_1 t_2 \ldots t_n} s \wedge t \in \mathit{enabled}(s)$, *then* $t' = t_i$ *for some* $1 \leq i \leq n\}$.

Note that the transitive closure of every NET relation is also a NET relation. Every user-provided NET relation can thus be extended to its closure.

4.3 The LPOR Stubborn Set Algorithm

Now we present LPOR, our local partial-order reduction algorithm. Formally, LPOR computes stubborn sets [Val98], which are subsets of $\mathit{enabled}(s)$

in a state s such that it is sufficient to explore transitions in such a subset. Stubborn sets are formalized via two constraints called D1 and D2 (see Section 2.5). LPOR can be configured to preserve properties from simple deadlock-freedom to arbitrary LTL_{-X} and CTL^*_{-X} specifications. LPOR can be adapted to similar POR semantics such as ample [CGP99] or persistent sets [God96]. We chose stubborn sets because they allow the most relaxed system model. For example, both persistent and ample sets assume deterministic transitions.

LPOR is a *static* POR algorithm, i.e., given a state s of the system, LPOR outputs a stubborn set in s without further exploration (as opposed to dynamic POR [FG05]). Therefore, LPOR can be implemented in stateful (even parallel [SD01]) explicit-state model checking. We present a simplified variant of the LPOR algorithm that assumes that the search path, i.e., a path from an initial state to s, is available. The search path can be obtained by depth-first search. However, a generalized form of LPOR makes no assumption about the search path and is compatible with both depth and breadth-first search. Therefore, it is amenable to symbolic (Binary Decision Diagram-based) implementations [ABH+01] as well. The generalized LPOR algorithm is presented Appendix B.1.

We first present the core LPOR algorithm and sketch its correctness, i.e., LPOR indeed computes stubborn sets (Section 4.3.1). Then, we discuss some optimizations of LPOR (Section 4.3.2) and the preservation of general temporal properties (Section 4.3.3).

4.3.1 The Stubborn Set Algorithm

As stated before, the use of NET in LPOR is optional. We therefore start out by explaining the LPOR algorithm (Algorithm 3) without the NET optimization where $\mathbf{net} = \emptyset$ (the third of LPOR's three input relations – see Section 4.2).

Forward enable sets LPOR uses two helper functions $\mathit{FwdEnableSetIdx}(t, t')$ and $\mathit{FwdEnableSet}(t)$ (Algorithm 2), whose return values can be pre-computed (before model checking), because they are independent of the state. The first function returns true if t can be the first in a sequence of enabling transitions that enables another transition t'' on which t' is dependent (lines 2.2-2.4). *FwdEnableSetIdx* is defined based on the *forward enable set* $\mathit{FwdEnableSet}(t)$ of t, which contains those transitions that can be enabled through a sequence of enabling transition starting with t (lines 2.5-2.15). More precisely, the set contains all transitions t' such that (t, t') is in the transitive closure of a can-enabling relation **ce**. The set

Algorithm 2: $FwdEnableSet(t)$ and $FwdEnableSetIdx(t, t')$ are precomputed for every $t, t' \in T$.

```
2.1  function FwdEnableSetIdx(t, t')
2.2    forall the (t'', en) ∈ FwdEnableSet(t) do
2.3      if (t'', t') ∈ dep then return true;
2.4    return false;
2.5  function FwdEnableSet(tr)
2.6    Tr' ← {(tr, ∅)};
2.7    do
2.8      Tr ← Tr' ;
2.9      forall the t_1 ∈ T do
2.10       forall the (t, en) ∈ Tr do
2.11         if (t, t_1) ∈ ce then
2.12           en_1 ← en ∪ {t_2 | (t_1, t_2) ∈ net};
2.13           Tr' ← Tr' ∪ {(t_1, en_1)};
2.14   while Tr ≠ Tr';
2.15   return Tr;
```

contains tuples of the form (t, en) where t is a transition and en is a set of transitions, which is used in the NET-optimized version of LPOR. If the NET relation is empty, en is also empty (line 2.12). We now explain how LPOR uses these two functions to compute stubborn sets.

Stubborn set computation In addition to the relations **ce**, **dep**, and **net**, LPOR has three parameters: (1) a transition $t_I \in enabled(s)$, called *initial transition*, which is in the stubborn set, (2) the current state s, and (3) the search path $\tau \in T^*$ (for Algorithm 3, it suffices that τ is a set containing $t_1, \ldots, t_n$). From D2, no stubborn set in s can be empty unless $enabled(s) = \emptyset$. Conceptually, LPOR proceeds, similarly to other static POR algorithms, by applying different rules of the form "if t is in the stubborn set, then transitions $t_1, t_2, \ldots$ must also be in the set". In this case, we say that $t_1, t_2, \ldots$ are added *on behalf of* t. LPOR maintains two sets of transitions: *Stub*, which represents the stubborn set (line 3.1) and *Trans*, which contains a transition t in *Stub* such that new transitions might be added to *Stub* on behalf of t (line 3.2). Therefore, LPOR adds transitions to *Stub* until *Trans* is empty (lines 3.3-3.14) and *Stub* is returned (line 3.15). We now explain how transitions are added on behalf of a transition t in *Trans*.

First, we add those enabled transitions t_1 that t is dependent on (lines 3.7-3.9). We add t_1 if either t_1 and t do not commute (disallowed by D1)

Algorithm 3: The LPOR(t_I, s, τ) stubborn set algorithm for a state $s \in S$, an initial transition $t_I \in \mathit{enabled}(s)$, and a current search path $\tau \in T^*$.

```
3.1  Stub ← {t_I};
3.2  Trans ← {t_I};
3.3  while Trans ≠ ∅ do
3.4      choose t ∈ Trans;
3.5      Trans ← Trans \ {t};
3.6      forall the t_1 ∈ enabled(s) \ Stub do
3.7          if (t_1, t) ∈ dep then
3.8              Stub ← Stub ∪ {t_1};
3.9              if dep is non-transitive then Trans ← Trans ∪ {t_1};
3.10         else if FwdEnableSetIdx(t_1, t) then
3.11             if ∃(t_dep, en) ∈ FwdEnableSet(t_1) : (t_dep, t) ∈ dep
3.12             ∧(en = ∅ ∨ ∀t_2 ∈ en : (t_2 ∉ Stub ∨ t_2 ∈ τ)) then
3.13                 Stub ← Stub ∪ {t_1};
3.14                 Trans ← Trans ∪ {t_1};
3.15 return Stub;
```

or it can disable t (which can violate D2). Note that **dep** does not have to be symmetric as D1 allows that t and t_1 do not commute. We will show an example of this case in a message-passing instance of LPOR (Section 4.4).

There is another way to violate the stubborn set conditions: an enabled transition t_1 outside the stubborn set can start a sequence of enabling transitions that enables another transition on which t is dependent (D1). This can only happen if $\mathit{FwdEnableSetIdx}(t_1, t)$ is true (line 3.10). In this case, we add t_1 to the stubborn set (line 3.13). Note that the condition in lines 3.11-3.12 is trivially true if LPOR is run without NET optimization because the *en*-sets are empty.

In both previous cases, t_1 is added to *Trans* (line 3.9 and 3.14) so that LPOR can verify whether new transitions must be added on behalf of t_1. We discuss the optimization for transitive dependency relations (line 3.9) in Section 4.3.2.

NET optimization Stubborn set computation can benefit from the NET relation if more than one transition t_2 is necessary for some transition t_1 to be enabled. In this case, a stubborn set does not need to contain *all* such t_2 but only *one* that has not been executed yet. The NET optimization

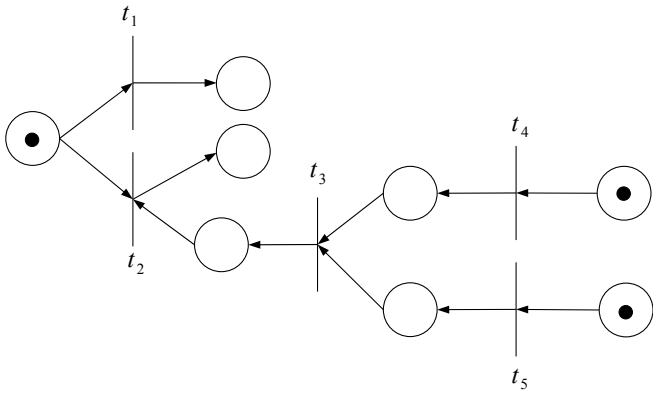

Figure 4.2: A Petri net example.

cannot be fully pre-computed as the check whether "a transition has not been executed yet" can only be carried out during the search. However, we can store these t_2 transitions in the *en*-field associated with t_1. It is key to our NET optimization that the content of *en*-fields is propagated along the can-enabling relation, i.e., if t can enable t_1 and (t, en) and (t_1, en_1) are in a forward enable set, then $en \subseteq en_1$ (line 2.12). This is because the transitions necessary to be executed for t to be enabled are, transitively, also necessary to be executed for t_1 to be enabled.

Then, using the notation of Algorithm 3, if some t_2 is in the *en*-field associated with a transition $t_{\mathbf{dep}}$, we can verify, given the current state s, that "t_2 has not been executed yet". Assume that $(t_{\mathbf{dep}}, en)$ is in the forward enable set of t_1 and the conditions in lines 3.10-3.11 are true. Then, we only add t_1 to the stubborn set if either t_2 is not in the stubborn set or t_2 has already been executed, i.e., is contained in the model checker's current search path τ (line 3.12). Note that, for some transition t, (t, en) can be in a forward enable set multiple times with different *en*. This is possible if t can be enabled by different sequences of transitions.

Example We illustrate the LPOR algorithm on a simple Petri net example (Figure 4.2). For this net, $\mathbf{ce} = \{(t_3, t_2), (t_4, t_3), (t_5, t_3)\}$, $\mathbf{dep} = \{(t_1, t_2), (t_2, t_1)\}$, $\mathbf{net} = \{(t_4, t_3), (t_5, t_3)\}$ are valid enabling, dependency, and NET relations, respectively. Note that we omit the possible (t_3, t_2), (t_4, t_2), and (t_5, t_2) from **net** for this example. Figure 4.2 depicts the initial token marking s; the set of enabled transitions in s is $\{t_1, t_4, t_5\}$. Consider a run of LPOR in s with initial transition t_1, i.e., LPOR$(t_1, s, ())$. As t_2 is disabled in s, no transition is added to the stubborn set in lines 3.7-3.9. Supposed

that transitions are processed by ascending index, t_4 is added to the stubborn set because $\mathit{FwdEnableSet}(t_4) = \{(t_4, \emptyset), (t_3, \{t_4, t_5\}), (t_2, \{t_4, t_5\})\}$, $(t_2, t_1) \in \mathbf{dep}$, and t_4 and t_5 are both not in the stubborn set. However, thanks to the NET optimization t_5 is not added because $\mathit{FwdEnableSet}(t_5) = \{(t_5, \emptyset), (t_3, \{t_4, t_5\}), (t_2, \{t_4, t_5\})\}$, t_4 already is the stubborn set, and τ is empty. As a result, $\mathrm{LPOR}(t_1, s, ()) = \{t_1, t_4\} \subset \mathit{enabled}(s)$.

Correctness The next theorem states that LPOR indeed generates stubborn sets. The proof of the theorem can be found in Appendix B.2. A sketch of the proof is given below.

Theorem 5. *Let (S, T, S_0) be an STS and* **ce**, **dep**, *and* **net** *a can-enabling, dependency, and NET relation, respectively. Then, for all $s \in S$, $t_I \in \mathit{enabled}(s)$, and $\tau \in T^*$ with $\exists s_0 \in S_0 : s_0 \xrightarrow{\tau} s$, $\mathrm{LPOR}(t_I, s, \tau)$ is a stubborn set.*

Proof sketch. A key property of LPOR is that, when executed in a state $s = s_0$, every transition t in $\mathrm{LPOR}(t_I, s, \tau)$ is independent of all transitions $t_1, t_2, \ldots, t_n$ that are in a path starting from s and that are outside $\mathrm{LPOR}(t_I, s, \tau)$. To show that D1 and D2 hold, consider the paths starting from s_0, as illustrated in Figure 4.3.

We first show that t is a key transition (D2). Indirectly, assume that t_i for some $1 \leq i \leq n$ can disable t, i.e., $t \notin \mathit{enabled}(s_i)$. Therefore, t must be dependent on t_i, a contradiction by the previous property.

As t is a key transition, $t \in \mathit{enabled}(s_i)$ for every $1 \leq i \leq n$. Let s'_n be a state such that $s_{n-1} \xrightarrow{t_n} s_n \xrightarrow{t} s'_n$. From the above property, t is independent of t_n, so there exists s'_{n-1} such that $s_{n-1} \xrightarrow{t} s'_{n-1} \xrightarrow{t_n} s'_n$. Repeating this rule n times, we obtain a path $s \xrightarrow{t} s' \xrightarrow{t_1} s'_1 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} s'_{n-1} \xrightarrow{t_n} s'_n$, which proves D1. □

Worst-case complexity Algorithm 3 is guaranteed to terminate (proof in Appendix B.2) and has worst-case time complexity $O(|T|^3 2^{|T|})$ with and $O(|T|^2)$ without NET optimization. Despite the worst-case exponential overhead of the NET optimization, our experiments show that LPOR with NET can achieve significant reductions of model checking time (Section 4.6).

We now sketch the idea behind the above complexity results. Assume that checks for set inclusion and adding/removing elements to/from sets take constant time. The basic quadratic time complexity in $|T|$ is due to (1) *Trans* containing at most $|T|$ transitions (line 3.3), and (2) adding at most $|T|$ transitions to the stubborn set on behalf of every transition in *Trans* (line 3.6). Note that every transition in *Trans* is also in *Stub* and

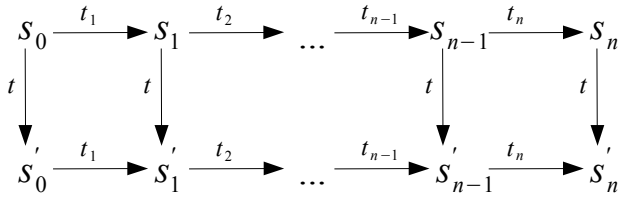

Figure 4.3: Illustration of the proof of Theorem 5.

no transition is ever removed from *Stub*. Therefore, the condition in line 3.6 and that *enabled*(s) is fixed throughout an execution of Algorithm 3 guarantee that every transition is added at most once to *Trans*. Without NET optimization, the condition in lines 3.11-3.12 is always true. Therefore, no computation overhead is added in this case. With NET optimization, the condition requires to range through possibly each element in a forward enable set and check if this element is in the stubborn set. As elements of the forward enable set are tuples of a transition and a subset of transitions, the maximum size of such a set is $|T|2^{|T|}$.

4.3.2 Optimizations and Possible Extensions

First, if the dependency relation is transitive, then the enabled transition t_1 does not have to be added to *Trans* (line 3.9). This is sound because all transitions that would be added to the stubborn set on behalf of t_1 are also added on behalf of t.

LPOR is a non-deterministic algorithm with three main sources of non-determinism, each of them possibly affecting the size of the stubborn set: (1) the selection of the initial transition, (2) the selection of t in line 3.4, and (3) the order in which **forall** iterates through the transitions in line 3.6. The tuning of these parameters in such a way that they result in small stubborn sets depends on the analyzed system.

We improve the NET-optimization by making it state-conditional, i.e., t' is a NET for t in a state s if t is not enabled in s and t' must be in any path starting from s before t can be enabled. The details of this optimization can be found in Appendix B.1. While state-conditionality can increase the achieved state-reduction, it also increases the time-overhead by limiting the possibilities for pre-computation.

The NET optimization can be generalized to necessary enabling *sets*, i.e., for each transition t a set T' of transitions such that at least one transition in T' must be executed for t to be enabled. This gives more flexibility compared to LPOR where T' contains at most one transition.

LPOR computes *strong* stubborn sets (see Section 2.5), which implies that all transitions that can disable a transition t in the stubborn set are also included in the set. In general, it is possible that a transition is removed from a strong stubborn set such that the resulting set is stubborn in the weak but not in the strong sense. However, an algorithm computing weak stubborn sets can incur a higher time overhead; in LPOR, this would require to refine dependency in terms of "can disable" and "might not commute" relations.

4.3.3 Preserving Temporal Logics with LPOR

Constraints D1 and D2 suffice to preserve deadlocks but they are too weak for preserving *invariants*, a simple but useful property in program analysis. For the preservation of invariants, stubborn sets must satisfy *proviso* and *visibility*, which can be implemented independently of the semantics of the transitions [God96, NG97, Val98, CGP99]. Proviso solves the ignorance problem and visibility guarantees that states missed in the reduced state graph do not interfere with the property.

Preserving LTL$_{-X}$ with LPOR If transitions are deterministic and the stubborn sets satisfy proviso and visibility, then the reduced graph preserves any property expressible in LTL$_{-X}$ (Linear Temporal Logic without the next time operator). If transitions can be non-deterministic, then an additional constraint called D3 must be satisfied [Val98]. Intuitively, D3 guarantees that infinite paths are preserved by POR. If transitions are deterministic, then D1 and D2 imply D3. However, D3 has to be separately established if transitions can be non-deterministic: let $\sigma = s \xrightarrow{t_1 t_2 \ldots}$ be an infinite path in the unreduced graph and t a key transition in s in a stubborn set $stub(s)$. Assume that $t_1, t_2, \ldots$ are outside $stub(s)$. Since t is a key transition, there is a path $s \xrightarrow{t_1 t_2 \ldots t_n t} s_n$ for every $n > 0$. It is possible that although for every $n > 0$ there is path $\sigma_n = s \xrightarrow{t t_1 t_2 \ldots t_n} s_n$ (from D1), every such σ_n proceeds through *different* states depending on n. Therefore, a cycle in σ might not correspond to a cycle in σ_n and, thus, there exists no infinite path $s \xrightarrow{t t_1 t_2 \ldots}$.

If the dependency relation is symmetric, LPOR returns stubborn sets which satisfy D3 (the proof of this property can be found in Appendix B.2). The key to this result is the following property of LPOR (which is also used to prove D1 and D2) – we call this property *commutativity*: For all transitions in σ_n it holds that $(t_i, t) \notin \mathbf{dep}$, i.e., t is independent of t_i. If **dep** is symmetric, then t_i is also independent of t, which implies that t cannot disable t_i. As a result, t_{n+1} is enabled in s_n and there is $\sigma_{n+1} = s \xrightarrow{t t_1 t_2 \ldots t_n} s_n \xrightarrow{t_{n+1}} s_{n+1}$.

Now, σ_n and σ_{n+1} proceed through the same prefix of states resulting in the preservation of infinite paths.

Preserving CTL$^*_{-X}$ with LPOR For the preservation of CTL$^*_{-X}$ (Computational Tree Logic without the next operator) a restrictive condition called NB is needed [Val98]. If all transitions are deterministic, NB requires that non-trivial stubborn sets contain exactly one "invisible" transition. In case of non-deterministic transitions, NB additionally requires that transitions in such stubborn sets are *super-deterministic*. Informally, a super-deterministic transition is deterministic, cannot be disabled by other transitions, and commutes with the execution of other transitions. The commutativity property of LPOR directly implies that stubborn sets computed by LPOR that contain a single *deterministic* and invisible transition satisfy NB. As a result, if not all transitions are non-deterministic, then LPOR can achieve reduction whilst preserving properties written in CTL$^*_{-X}$.

4.4 A Message-Passing Instantiation of LPOR

Assume that message-passing systems are modeled in terms of **M** models (Section 3.1.3). As the semantics of these models is given as a state transition system, they are amenable for LPOR. First, we extend the syntax of the models to simplify the discussion (Section 4.4.1 – a detailed formalization of this extension can be found in Appendix B.4) and define the LPOR relations from Section 4.2 for message-passing systems (Section 4.4.2). The simplicity of these definitions shows that the use of LPOR is indeed straightforward for domain experts.

4.4.1 Extended Syntax of Message-Passing System Models

Every transition t can be associated with $t.M_I$ (and $t.M_O$), the set of messages possibly received (sent) by t, and $t.I$ (and $t.O$), the set of processes that t can receive (and send) messages from (to). We assume the local state of a process to be an assignment of values to *local variables*. Given a variable x, t is a *write* transition with respect to x and we write $x \in W(t)$ if t can change the value of x in some state. Similarly, t is called a *read* transition ($x \in R(t)$) if the guard of t depends on the value of x. As a special case, a write transition t is an *increment* transition ($x \in Inc(t)$) if t always increases the value of x.

Increment transitions are relevant in the context of *timestamp-compare* read transitions t ($x \in \mathit{CompTS}(t)$), a class of transitions common in concurrent systems, e.g., Paxos. Such a transition t uses x to store a "timestamp" and compare it with the timestamps of incoming messages. The guard of t can be true only if the timestamp of the message is greater or equal than the current value of x.

The sets $R(t)$, $W(t)$, $\mathit{Inc}(t)$, and $\mathit{CompTS}(t)$ can be conservatively determined by lightweight static analysis. Note that it is always sound to exclude variables from these sets.

4.4.2 LPOR Relations for Message-Passing Systems

Can-enable relation We say that a transition t can *locally* enable another transition t' of the same process if t is a write and t' is a read transition with respect to some common variable x. An exception to this rule is if t is an increment and t' is a timestamp-compare transition with respect to x. In this case t cannot enable t' because a process sends no new message to itself and the timestamp x is increased by t. Formally, can-local-enable $= \{(t, t') \mid id(t) = id(t') \wedge \exists x \in W(t) \cap R(t') : x \notin \mathit{Inc}(t) \cap \mathit{CompTS}(t')\}$, where $id(t)$ denotes the process executing transition t.

A transition t can *remotely* enable a transition t' if it may send messages that can be received by t'. A necessary condition for this to happen is that t and t' are executed by different processes ($id(t) \neq id(t')$), that transition t can send a message to the process executing t' ($id(t') \in t.O$), that transition t' can receive a message from the process executing t ($id(t) \in t'.I$), and that t can send a message that can be received by t' ($t.M_O \cap t'.M_I \neq \emptyset$). Therefore, we define that can-remote-enable $= \{(t, t') \mid id(t) \neq id(t') \wedge id(t') \in t.O \wedge id(t) \in t'.I \wedge t.M_O \cap t'.M_I \neq \emptyset\}$.

Definition 18. *Given a message-passing system,* MP-can-enable $=$ can-remote-enable $\cup$ can-local-enable.

Dependency relation A transition t' is dependent on t if both are executed by the same process or if t can remotely enable t'. The intuition is that local transitions may change the state of the same process and, if t can remotely enable t', then t can send a message that is processed by t'. Our dependency relation can be refined by excluding pairs of transitions that are executed by the same process and access a disjunct set of variables. This is a refinement that we do not consider in this thesis. Note that the following relation can be asymmetric, which enables LPOR to compute smaller stubborn sets.

Definition 19. *Given a message-passing system,* MP-dependency $= \{(t, t') \mid t \neq t' \wedge id(t) = id(t')\} \cup$ can-remote-enable.

NET relation The following NET relation is based on the observation that a transition t with $t.I \neq \emptyset$ cannot be enabled unless a process sends a message to process $id(t)$. For example, imagine that t represents a function that requires input from a majority of processes. This implies that $|t.I| = \lceil \frac{n}{2} \rceil$, i.e., a majority of the number of all processes n. Then, t can be enabled only after each of these processes has sent a message to process $id(t)$.

Note that we have to check two additional conditions to make sure that a transition is indeed a NET for t. Firstly, t is required to be *input-deterministic*, i.e., t always consumes a message from every process in $t.I$. Otherwise, t can possibly be enabled even if a process in $t.I$ sends no message to process $id(t)$. Secondly, it is possible that $i \in t.I$ and process i has multiple transitions, say t' and t'', that can enable t (formally, $id(t'') = id(t') \wedge t'' \neq t' \wedge \{(t', t), (t'', t)\} \subseteq$ can-remote-enable). In this case, neither t' nor t'' is necessarily a NET for t.

The NET relation is defined below. In Appendix B.1, an example is shown of how the content of the channels can be used to make this relation state-conditional.

Definition 20. *Given a message-passing system,* MP-NET $= \{(t, t') \mid t$ *is input-deterministic* $\wedge\ id(t') \in t.I \wedge \forall (t'', t) \in$ *can-remote-enable:* $t'' = t' \vee id(t'') \neq id(t')\}$.

The next theorem states that the above relations are indeed LPOR relations as of Section 4.2, a task that must be carried out by the user. The proof of this theorem can be found in Appendix B.3.

Theorem 6. *Given a message-passing system,* MP-can-enable, MP-dependency *and* MP-NET *are can-enabling, dependency, and NET relations, respectively.*

4.5 Java-LPOR: An LPOR Implementation

We implement LPOR in a Java library, called Java-LPOR. Java-LPOR can be integrated into any explicit state model checker. The source code of Java-LPOR is available for download[2].

[2]http://www.deeds.informatik.tu-darmstadt.de/peter/Java-LPOR.jar

The LPOR algorithm currently implemented by Java-LPOR computes stubborn sets satisfying D1, D2, and the additional constraint regarding *visible* transitions [Val98] (cf. Section 4.3.3). Visible transitions are transitions that might interfere with the target property; the visibility constraint prevents non-trivial stubborn sets from including visible transitions.

The main steps of integrating Java-LPOR are as follows. As a running example, we show how we used Java-LPOR to implement message-passing LPOR from Section 4.4.

1) Specifying the transitions: Before the search can start, all transitions of the system must be provided as Java classes.

2) Implementing the LPOR relations: Java-LPOR exports LPOR's relations via the following interface. This generic interface is parametric in the class `T` of transitions.

```
public interface LPORRelations<T> {
    public boolean dep(T t1,T t2);
    public boolean canEnable(T t1,T t2);
    public boolean net(T t1,T t2);
}
```

For example, the following snippet shows the implementation of our dependency relation for message-passing systems (compare with Definition 19). The method `t1.isLocal(t2)` returns true iff $id(t_1) = id(t_2)$.

```
public boolean dep(TransitionMP t1,TransitionMP t2){
    return !t1.equals(t2) &&
         t1.isLocal(t2) || canRemoteEnable(t1, t2);
}
```

3) Setting up LPOR: For the preservation of invariants, Java-LPOR requires to identify visible transitions. In our current implementation, the user is required to annotate visible transitions using the following interface.

```
public interface VisibilityChecker<T> {
    public boolean isVisible(T t);
}
```

Given the list of all transitions `trans`, the LPOR relations `rel`, and a class `vis` for checking visible transitions, an LPOR utility instance can be created. Its constructor is responsible for pre-computing the forward enable sets. The instance of `LPORUtil` can then be used to compute stubborn sets for a particular state by invoking the `LPOR` method. As arguments, the method requires an initial transition and the list of enabled transitions. Transitions are identified by their index in `trans`.

```
public class LPORUtil<T>{
    public LPORUtil(List<T> trans,
                    LPORRelations<T> rel,
                    VisibilityChecker<T> vis){
            this.trans=trans;
            this.rel=rel;
            this.vis=vis;
            precompute();
    }
    public int[] LPOR(int t_I, int[] enabledTrans){
        ...
    }
    ...
}
```

4) Computing stubborn sets: Finally, the following snippet shows how the set of transitions that must be executed in a state is pruned by a call to the `LPOR` method of an `LPORUtil` instance.

```
enabledTrans=lporUtil.LPOR(initTrans, enabledTrans);
```

4.6 Evaluating LPOR

We evaluate LPOR by model checking the fault-tolerant message-passing protocols from Section 3.6.2. The transitions of the models of these protocols are refined using the combined split strategy. We use the LPOR relations for message-passing systems (Section 4.4) and the Java-LPOR implementation of LPOR (Section 4.5), all integrated within the MP-Basset model checker (see Chapter 6).

Comparison with Dynamic POR We compare LPOR with dynamic POR (DPOR) [FG05]. We explain how DPOR differs from static POR (SPOR) in Section 4.7. The benefit of DPOR is that it can be less conservative about the selection of paths that are explored in the reduced search. However, our experiments show the efficiency of LPOR over DPOR, improving on the reductions of a message-passing DPOR implementation.

Like any SPOR algorithm, LPOR can be soundly combined with DPOR for further reduction [FG05]. This must respect the restrictions imposed by DPOR, however. For example, DPOR assumes the absence of cycles in the state space. We only consider protocol examples with acyclic state spaces for a fair comparison.

We compare LPOR with the original DPOR algorithm by Flanagan and Godefroid [FG05] because this preserves the specifications (properties) of our example protocols. For example, the DPOR variant in [SA06] only guarantees that every transition executed in the unreduced search is also executed in the reduced one, which is too weak of a preservation given the example properties.

We express the specifications of the example protocols using invariants. To preserve invariants, we use LPOR computed stubborn sets with the additional constraint regarding visible transitions (discussed in Section 4.5). This constraint can also be implemented in DPOR such that if a visible transition is executed in a state during the search, then all enabled transitions in this state will be executed. For comparing LPOR with DPOR, we utilize the Basset model checker [LDMA09], which implements an adaptation of Flanagan and Godefroid's DPOR algorithm for actor programs. The actor semantics used in Basset is similar to our model of message-passing except that quorum transitions are not supported. Therefore, we extended Basset's DPOR implementation with quorum transitions: When a process executes a quorum transition, the vector clock of the process will be updated to be the maximum of (1) its current value and (2) the values of the vector clocks of the senders of the messages, where the values correspond to the time of sending the message. In original Basset, this computation involves one sender as every transition consumes a single message.

Experiment setup We run our experiments in a DETERlab testbed [DET] on 2GHz Xeon machines. We compare LPOR with the unreduced models and DPOR, our extension of Basset's DPOR implementation as explained above. We integrated both this DPOR algorithm and LPOR within MP-Basset.

For fair comparison, both of our POR implementations use the same heuristic for initial transitions.[3] In this heuristic, transitions are preferred that either start a new instance of the protocol or, if there is no such transition, complete no ongoing instance.[4] Intuitively, the execution of such a transition "delays" the completion of an instance, because a completed instance usually interferes with the specification, thus, disabling partial-order reductions. This heuristic shows good performance in our LPOR experiments. Surprisingly, our heuristic suggests the opposite of the *transaction* strategy proposed in [BGG06]. We speculate that the difference lies in that our target protocols allow more concurrency than the cache coherence pro-

[3]This heuristic is used for the experiments in Section 3.6.2 too.

[4]An instance of Paxos consists of Phases 1(a)-2(b) including learning, the multicast of a message in Echo Multicast, and a read or write operation in regular storage.

tocol analyzed in [BGG06], where the processing of further client requests is blocked until the centralized cache controller (assumed to be fault-free) completes the ongoing instance of the protocol started by another client.

Each protocol experiment is run with each of the the following POR algorithms:

- *None*: Unreduced search (no POR is used).
- *DPOR*: Flanagan and Godefroid's DPOR algorithm [FG05] augmented with quorum transitions and the visible transitions constraint. DPOR is run as stateless search because DPOR can be unsound if state comparison is used [FG05].
- *LPOR stateless*: The generalized LPOR algorithm run without state comparison in MP-Basset.
- *LPOR-NET*: The generalized LPOR algorithm without NET optimization (with state comparison).
- *LPOR*: The generalized LPOR algorithm (with NET optimization and state comparison).

In addition to these POR settings, we conduct the following experiments:

- *On-line/pre-computed forward enable sets*: Each LPOR experiment is run with on-line computed versus pre-computed forward enable sets (see Section 4.3.1). If pre-computation is used, the measured model checking time includes the time of pre-computation.
- *Model Java Interface (MJI)*: We tune the efficiency of our LPOR implementation by using (or not using) MJI within MP-Basset. The use of MJI is discussed in details in Section 6.3.

Main results & trends The results of our experiments are shown in Table 4.1. Similarly to previous experiments, we write "Verified" if the model checker finds no bug, otherwise (in case of faulty protocols or wrong specifications) a counterexample ("CE") is returned. The best result and the corresponding POR algorithm is written in bold for each protocol experiment. In buggy instances the search is stopped after finding the first bug. Therefore, the number of visited states depends on the order in which transitions are executed in a state. This schedule can be different in DPOR and LPOR. The times where forward enable sets are computed on-line (no pre-computation) are written in parentheses.

We expand our protocol experiment setup compared to Section 3.6.2:

- A new version of Paxos is added, called Faulty Paxos II, which contains a subtle bug: An acceptor does not remember the highest-numbered proposal it has ever accepted but only the *last* proposal it has accepted. We remark that this bug manifests itself with at least three proposals, as it was successfully validated by MP-Basset.

- New configurations of Echo Multicast are added. In addition, we use a modified attack model in the configuration Multicast (2,1,2,1) to further challenge the model checker: A Byzantine-faulty process is non-deterministically changing between correct and faulty versions of the protocol. As a result, the bug can only be found after longer search (compare the results in Tables 3.3 and 4.1).

We observe the following trends:

- *Fast bug finding with POR:* The POR-based search finds bugs faster than unreduced search and there is no clear winner between DPOR and LPOR.

- *Efficient LPOR:* LPOR is highly efficient as shown by the exhaustive search results ("Verified") reducing the number of states by up to to 94% and search time by up to 90% – see register example.

- *Efficient NET optimization:* Although the additional online checks in the NET optimization slow down LPOR (as discussed in Section 4.3.1), e.g., 74 states/sec versus 59 states/sec for exhaustive Multicast (3,1,1,0), the additional state reduction adds up to reducing the total model checking time. In fact, the NET optimization can be very efficient by achieving additional space and time reductions of up to 87% – see Multicast (4,1,1,0) exhaustive search result.

- *LPOR outperforming DPOR:* LPOR outperforms DPOR in *all* exhaustive search experiments, even in stateless search where the benefit of LPOR is not biased by the stateful optimization. In addition, LPOR proves to be more time efficient than DPOR, i.e., the time overhead of LPOR is smaller. For example, the stateless exhaustive runs of Register (3,1) visit the same number of states but LPOR is faster.

  We observe that a high number of concurrent quorum transitions increases LPOR's advantage over DPOR. Protocol settings where different executions of one or more quorum transitions of the same process can be co-enabled in a state contain significantly more sources of concurrency. In these cases, LPOR is more efficient than DPOR because

Protocol (# proc.)	Property	Result	POR alg.	MJI	States	Time (on-line)
Paxos (2,3,1)	Safety	Verified	None	-	>38mil	>192h
			DPOR	No	3,305,752	22h53m
			LPOR stateless	Yes	1,118,341	6h14m (6h19m)
				No		8h51m (28h32m)
			LPOR-NET	Yes	1,130,234	6h59m (7h1m)
				No		8h51m (24h10m)
			LPOR	**Yes**	**548,061**	**3h18m** (3h21m)
				No		4h45m (18h52m)
Faulty Paxos (2,3,1)	Safety	CE	None	-	238,790	1h34m
			DPOR	**No**	**2,028**	**50s**
			LPOR stateless	Yes	3489	1m16s
			LPOR-NET	Yes	3,489	1m43s
			LPOR	Yes	3,415	1m40s
Faulty Paxos II (3,3,1)	Safety	CE	None	-	>16mil	>192h
			DPOR	**No**	**21,177**	**12m31s**
			LPOR stateless	Yes	175,725	1h24m
			LPOR-NET	Yes	173,414	1h25m
			LPOR	Yes	173,414	1h28m
Register (3,1)	Regularity	Verified	None	-	287,638	47m
			DPOR	No	27,763	6m50s
			LPOR stateless	Yes	27,763	5m57s (5m59s)
				No		6m17s (9m23s)
			LPOR-NET	**Yes**	**18,451**	**4m32s** (4m32s)
				No		5m3s (7m1)
			LPOR	Yes	18,451	4m36s (4m36s)
				No		4m55 (7m52s)
Register (3,1)	Wrong regularity	CE	None	-	7,619	1m52
			DPOR	**No**	**2,344**	**40s**
			LPOR stateless	Yes	4,654	1m4s
			LPOR-NET	Yes	3,497	55s
			LPOR	Yes	3,497	58s
Register (3,2)	Wrong regularity	CE	None	-	24,939,222	181h
			DPOR	No	11,235	3m56s
			LPOR stateless	Yes	11,235	3m37s
			LPOR-NET	**Yes**	**6,987**	**2m32s**
			LPOR	Yes	6,987	2m34s
Multicast (3,1,1,0)	Agreement	Verified	None	-	7,279	1m34s
			DPOR	No	7,945	1m46s
			LPOR stateless	Yes	2,674	38s (38s)
				No		1m2s (1m47s)
			LPOR-NET	Yes	6,607	1m29s (1m30s)
				No		2m7s (2m7s)
			LPOR	**Yes**	**2178**	**37s** (37s)
				No		59s (1m46s)
Multicast (4,1,1,0)	Agreement	Verified	None	-	102,058	28m13s
			DPOR	No	183,265	44m45s
			LPOR stateless	Yes	24,382	6m12s (6m17s)
				No		7m8s (11m2s)
			LPOR-NET	Yes	94,186	26m26s (26m37s)
				No		29m32s (31m15s)
			LPOR	**Yes**	**12,494**	**3m34s** (3m39s)
				No		5m4s (9m32)
Multicast (2,1,2,1)	Wrong agreement	CE	None	-	7,543	3m32s
			DPOR	No	4,890	2m8s
			LPOR stateless	Yes	4,890	1m57s
			LPOR-NET	**Yes**	**2,139**	**1m4s**
			LPOR	Yes	2,139	1m47s

Table 4.1: Performance results of LPOR implemented within MP-Basset.

concurrently executed quorum transitions within a process leads to the addition of backtracking points in DPOR [FG05, SA06].

- *Varying pre-computation gains:* The benefit of pre-computation is significant when our LPOR implementation is *not* using MJI calls. The pros and cons of using MJI are discussed in Section 6.3. We observe a higher relative gain of using pre-computation in NET optimized LPOR. The reason is that forward enable sets containing non-empty *en*-fields (in the NET optimized case) tend to be larger, thus, their computation takes longer.

  The reason why the MJI implementation does not greatly benefit from pre-computation for our particular protocol examples is two-fold: Firstly, lines 3.10-3.14 in LPOR (Algorithm 3) are executed in a relatively small number of states. Secondly, the body of the do-while loop in the forward enable set computation (Algorithm 2) is executed only a few (1-2) times during an average invocation of *FwdEnableSet*.

4.7 Related Work

The basic structure of the LPOR algorithm is similar to Godefroid's stubborn (and persistent) set algorithms [God96], which start with a transition and keep adding new transitions using the dependency and can-enabling relations until the current set of transitions is not stubborn. An application of these algorithms to a new language is only possible after a translation of this language into a specific language used in [God96], which specifies processes communicating via shared objects. Transitions in this language are assumed to be deterministic. Furthermore, the algorithms in [God96] do not support pre-computation. The ample set algorithms in [CGP99, HP94, Hol03] also restrict to process-based systems and deterministic transitions. Moreover, they conservatively assume that a non-trivial ample set consists of all enabled transitions of a particular process.

Promela is a general language with explicit support for multi-process systems and message-passing. SPIN is a widely-used model checker for specifications written in Promela [Hol03]. SPIN supports a specific form of POR, which is based on the observation that transitions t_1 and t_2 are independent if they are from different processes, and t_1 is the only transition writing to (or reading from) a FIFO channel (exclusive write or read, respectively) [HP94, Hol03]. Such interferences can be easily expressed in LPOR by excluding (t_1, t_2) and (t_2, t_1) from the dependency relation. We note that in the description of [HP94], t_1 and t_2 are considered "independent" only in states

where the channel is non-empty (non-full). This is because their definition of dependency includes that a transition can enable another transition. In the sense of Definition 16, t_1 can enable read (send) transitions but t_1 and t_2 are (always, i.e., state-unconditionally) independent.

It is possible to give a graph theoretic implementation of LPOR as proposed in [Val98]. In this approach, the vertices of the graph are transitions and t is connected to t_1 if t_1 needs to be added to the stubborn set on behalf of t. Then, certain vertices of this graph, e.g., included in properly selected strongly connected components, correspond to stubborn sets.

Dynamic POR (DPOR) [FG05] is a POR implementation which computes a persistent set in some state s gradually while the successors of s are explored. In this way the persistent set algorithm can learn about interfering transitions and needs not to guess them as in static POR. In other words, DPOR explores future paths instead of guessing them. However, DPOR also makes static assumptions about co-enabled dependent transitions. Furthermore, DPOR is inherently a depth-first search, it needs to know the sequence of transitions in the current path (which is not straightforward in parallel model checking [SD01]) and can be unsound with stateful model checking [YCGK08].

Operation refinement from [God96] (and, in general, transition refinement from Section 3.5) translates from one transition system to another in order to improve on the efficiency of POR. Such a translation is orthogonal to LPOR, which requires a state transition system at its input.

The input relations of LPOR can be partly or entirely derived automatically using a SAT solver, an approach similar to [BGG06]. Moreover, SAT-based bounded model checking can be used to compute more accurate enabling sequences than our forward enable sets. For example, given transitions t_1, t_2, t_3, it is possible that t_1 can enable t_2, and t_2 can enable t_3, but t_2 cannot enable t_3 if t_2 was enabled by t_1.

4.8 Conclusions

We have proposed LPOR, a framework for easy-to-use, flexible, and efficient POR implementations. While existing POR implementations trade flexibility for ease-of-use and efficiency, e.g., SPIN's POR is limited to exclusive write/read FIFOs or DPOR prohibits cycles, the strength of LPOR is that it provides these features at the same time.

It is to be seen whether state-conditional can-enabling and dependency relations can improve on LPOR's reductions. For example, a state-conditional can-enabling relation can be used to rule out transitions t_1 in line 3.10 of

Algorithm 3 that cannot enable any transition in the current state. Another possible extension is to add symmetry reduction to LPOR. Although partial-order and symmetry reductions are compatible in theory [EJP97], no implementation of their combination is available nor the efficiency of such combination has been tested on real examples.

Chapter 5

Induction in Distributed Protocols

This chapter deals with the verification of invariants using induction proofs [MP95, dMRS03] (reviewed in Section 2.6). Invariants cover a general and practical class of specifications. If it is possible to express the specification as an invariant, there is no need for expensive verification procedures that are able to verify general temporal formulas [CGP99].

The application of induction proofs is motivated by at least two arguments. Firstly, an efficient automation of induction proofs is possible by the recent advances of Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers. Secondly, induction proofs can verify systems with infinitely many states.

Despite the efficiency of induction proofs in particular case studies [SBS+11], an important practical limitation of the induction proof is its general incompleteness: It is possible that the induction step of a proof fails for an assertion P that is an invariant of the system. Such invariants are called *non-inductive* invariants.

In an attempt to prove the invariance of an assertion P, the possible outcome of induction is three-fold: (1 – Inductive invariant) induction holds and P is proven to be an invariant; (2 – Counterexample) the base case of the induction fails and it returns a path $s_0, s_1, ..., s_k$ such that s_0 is an initial state and P does not hold in s_k; (3 – Inductive counterexample) induction step fails and returns a path $s_0, s_1, ..., s_k$ such that either (3a – Spurious counterexample) s_k is *not* reachable and it violates P or (3b – Reachable state violating P) s_k is reachable and it violates P. In case of (3), it is indecisive whether P is an invariant or not because the induction does not determine if s_k is reachable. A sufficient condition for s_k to be reachable is that s_0 is reachable. Note that, in general, deciding the reachability of a

state is as hard as the verification itself.

Given the dilemma of inductive counterexamples, we propose two techniques with the following features.

(F1) If the outcome of the induction is (3), then the decision between (3a) and (3b) is easier for a human verifier.

(F2) In favorable cases, invariants can be proven that are non-inductive otherwise.

Feature F1 of the proposed techniques is a side-effect of their ability to eliminate spurious counterexamples (to achieve F2). As a result, these spurious counterexample are not among the possible inductive counterexamples, which makes the reachability analysis easier (cf. discussion in Section 5.2.3).

Our contributions can be summarized as follows.

- *Classification of lemmas (Section 5.1):* Using lemmas [MP95] is a well-known approach to achieve (F1-2) – see Section 2.6 for more details. The main drawback of lemmas is that it is hard to automate their discovery. Firstly, based on the induction proof of a message-passing consensus protocol, we propose a *classification* of lemmas. The proposed classification applies for a general class of message-passing protocols. Secondly, we discuss how the classification can be used to automate the discovery of lemmas.

- *Strengthened transitions (Section 5.2):* We propose a novel approach for achieving (F1-2), called *strengthened transitions*. We also discuss how this technique can be used for induction proofs of general multi-process protocols.

5.1 Classification and Discovery of Lemmas

We define a possible classification of lemmas used in induction proofs of message-passing algorithms. The different lemma classes are explained based on an example message-passing consensus protocol. Although the definition of the lemma classes is informal, it does not interfere with the soundness of verification because every classified lemma is verified using induction. A use of the proposed lemma classes is to guide the automated discovery of new lemmas.

The example consensus protocol The consensus protocol assumes that processes fail by crashing and that every process is equipped with a *perfect failure detector* [CT96].[1] The protocol specifies the following transitions for each of the replicated processes:

- *Sending proposal:* Initially, process i broadcasts a "propose" message with its proposed value p_i (a binary value, for simplicity).
- *Receiving proposal:* Process i receives a propose message with p_j from process j and stores p_j as the proposed value of process j.
- *Sending suspicion:* Process i broadcasts a "suspect" message with p_j (or default value if not available – empty message) upon i's failure detector suspecting j.
- *Receiving suspicion:* Process i receives a non-empty suspect message with p_k from process j and stores the fact of suspicion and p_k as the proposed value of process k.
- *Decision:* Process i *decides* value $min\{p_j\}$ if it has received p_j or a suspicion for every process j.

We consider the agreement property of consensus, which specifies that no two processes decide different values.[2]

Induction proofs automation with SAL We use the SAL verification suite [SAL] for its automation of induction proofs. The example consensus protocol is specified in SAL's input language. This SAL model of the protocol is available on-line at http://www.deeds.informatik.tu-darmstadt.de/peter/thesis/kInd/cons.sal.

The model of the system can greatly influence whether an assertion is inductive or not. We emphasize that the SAL model we use is not "optimized" for verification. It is used as a modeling exercise for teaching purposes and it follows a clean and intuitive structure.

It turns out that agreement with four processes is not inductive for this model or, by increasing k, k-induction times out. SAL also contains a symbolic model checker using BDDs (Binary Decision Diagrams). SAL's symbolic model checker is unable to prove agreement due to prohibitively large BDDs.

[1] We use a consensus protocol from the lectures of Prof. Christof Fetzer at Technische Universität Dresden (Department of Computer Science, Institute for System Architecture).

[2] We do not consider liveness for this analysis.

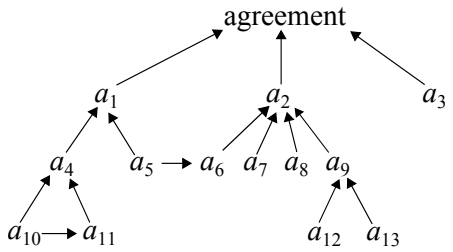

Figure 5.1: Overview of proving agreement with induction.

Name	Type	Inductive	Description
a_3	D	Yes	Decision function
a_8	D	Yes	Local condition of decision
a_7	SM	Yes	# of crashed processes is below the threshold
a_{10}	SM	Yes	Maximum # of messages not exceeded
a_{12}	SM	Yes	Failure detector only suspects crashed processes
a_5	MF	Yes	Propose message carries proposed value
a_{13}	MF	Yes	Suspect messages cannot be forged
a_4	MF	No	Suspect messages carry unforged proposed values
a_6	MF	No	Empty suspect message is dismissed
a_9	MF	No	Suspect messages carry unforged suspicions
a_{11}	MF	No	Proposed values are disseminated via messages
a_1	SV	No	Proposed values cannot be forged
a_2	SV	No	Decision only if all proposed values are known

Table 5.1: Description of lemmas used in an induction proof of agreement.

Strengthened induction proof of consensus We find that agreement can be proven using 13 lemmas denoted as $a_1, ..., a_{13}$. Figure 5.1 shows the structure of a possible proof: An acyclic, directed graph is shown whose nodes correspond to agreement and the lemmas and edges have the following meaning: Given a node n, let $n_1, ...n_k$ be the nodes from which there is an edge to n. Then, n is inductive relative to $n_1, ..., n_k$. For example, agreement is inductive relative to a_1, a_2, a_3. The proof is minimal in the sense that every lemma is inductive only relative to all lemmas pointing to it.

The proof can be replayed using the above SAL model, which contains all lemmas. Spurious counterexamples can be generated by not using all necessary lemmas, e.g., a_9 is not inductive relative to a_{12}. The time of all proofs in a Cygwin-emulated SAL installation with the Yices SAT/SMT solver running on a 2.8 GHz AMD processor is less than 25 minutes. We remark that faster (or slower) proofs are possible using different lemmas.

Proposed classification of lemmas We classify the lemmas into the following types (see Table 5.1):

- (D) *Definition lemmas* are invariant correlations between state variables. For instance, a_3 follows from the definition of the decision func-

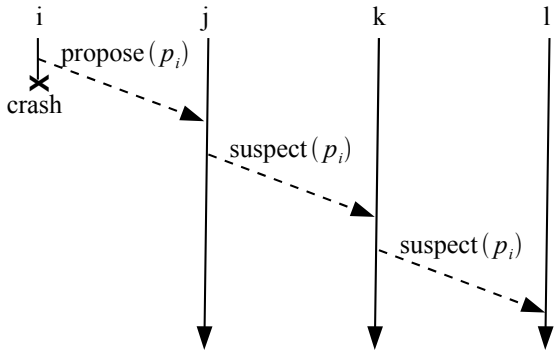

Figure 5.2: Example message-flow with processes i, j, k and l in consensus with failure detectors.

tion, which specifies for each process the correlation between the decision value, suspicions, and proposed values of other processes.

- (SM) *System model lemmas* relate to the computation and communication environment, e.g., a_7 disallows that the number of crashed processes exceeds a threshold.
- (MF) *Message-flow lemmas* correspond to *message-flows* [TT08], i.e., patterns in the message traffic of the protocol. For example, a_4 is based on a flow describing how the proposed value of a crashing process is propagated to other processes via a chain of suspect messages (see Example 8).
- (SV) *State validity* lemmas constrain the local states across multiple processes. For example, a_1 expresses the integrity of proposed values across the system.

Message-flow lemmas is an important class of lemmas, as shown by our example with 6 out of 13 lemmas being message-flow lemmas. The following example explains a message-flow, which lemma a_4 is based on.

Example 8. Figure 5.2 shows an example message-flow in the style of [TT08]. Four processes i, j, k and l and the following scenario are depicted: Process i crashes after sending its proposed value p_i to process j. After receiving this message, process j's failure detector suspects process i, which triggers process j to send a suspect message with p_i to process k (and to all other processes – not depicted). After receiving process j's message, process k's failure detector also suspects process i and sends a suspect message with p_i (which process k knows from process j) to process l (and to all other processes – not depicted).

This message-flow suggests the invariant (cf. lemma a_4) that if a process (k or l) learns about the proposed value of another process i *via a suspect message*, then this value is indeed the (unforged) proposed value of process i. (End of Example 8)

Systematic lemma discovery The manual discovery of $a_1, ..., a_{13}$ and their general classification suggest strategies to systematically discover lemmas in a fully (or partly) automated manner. Note that since every lemma must be proven to be an invariant, discovering wrong lemma candidates (non-invariants) cannot jeopardize the overall soundness of the verification.

- D type lemmas can be discovered "by construction" if definitions are annotated in the model of the protocol. For example, the SAL language implements constructs called `DEFINITIONS` for specifying invariant correlations between state variables.

- SM type lemmas relate to message-passing systems but they are independent of the particular message-passing protocol. Therefore, given a general model of message-passing systems, a pool of SM lemma candidates can be defined once for all.

- Originally, message-flows (the source of MF lemmas) are provided by a protocol expert [TT08]. An automated discovery of message-flows from the specification of the protocol can also be built upon annotated model specifications. For example, the user of the MP-Basset model checker for message-passing systems (Chapter 6) specifies message types (such as "suspect") that explicitly appear in send operations as well as in the name of the transition that consumes the message. These syntactic restrictions enable the discovery of simple message flows by parsing the code of the model and matching message types in send operations with transition names.

- The automated discovery of SV lemmas is possibly the most challenging. One strategy is to design heuristics that are guided by the particular characteristics of a given class of protocols. For example, every consensus protocol specifies agreement. If the specification assumes a non-malicious fault model where messages cannot be corrupted, then a possible lemma candidate requires that a process i can learn about the proposed value of another process j only if process i has indeed proposed this value (cf. lemma a_1).

Automated proofs Besides the automated discovery of lemmas, we also aim at automating the proofs of non-inductive lemmas. As the simplest case, a lemma candidate is inductive without lemmas (see D and SM lemmas in Table 5.1).[3] In cases where induction fails, the induction depth can be increased depending on the available resources of verification. Given inductive lemmas, these can be used to prove the non-inductive ones. The exploration of the possible combinations depends on the available resources too. To guide the exploration, the proofs can be assisted by spurious counterexamples returned by the SAT/SMT solver: Lemmas that do not hold for a spurious counterexample returned for n should be among $n_1, ..., n_k$.

Note that there is no harm in using more lemmas: If an assertion n is inductive relative to the invariants $n_1, ..., n_k$ and n_{k+1} is another invariant, then n is also inductive relative to $n_1, ..., n_{k+1}$. However, using too many (including unnecessary) lemmas can negatively affect the efficiency of the proof.

5.2 Strengthened Transitions

Strengthening transitions is a new way of thinking of spurious counterexamples in induction proofs: Instead of considering unreachable *states* as root causes of spurious counterexamples, our approach aims at *transitions* being executed in these states. The main idea of strengthened transitions is to have fewer transitions such that (1) invariants are preserved (soundness) and (2) there are no spurious counterexamples.

We structure the discussion as follows. In Section 5.2.1, we explain and motivate the idea of strengthened transitions based on a mutual exclusion protocol. The general approach is presented in Section 5.2.2. Finally, Section 5.2.3 shows an application of strengthened transitions for general multi-process systems.

5.2.1 Motivating Example

Bakery mutual exclusion Figure 5.3 shows a simplified specification of the Bakery *mutual exclusion* protocol with two processes, taken from the SAL distribution [SAL]. The protocol executed by a process is specified by `bprocess` between lines S3-21. The process maintains three variables:

- A *program counter* `pc` (line S7) taking a value from `sleeping`, `trying`, and `critical` (line S2). Initially the program counter assumes

[3] Note that D type lemmas are always 1-inductive.

```
S1 bakery: CONTEXT = BEGIN
S2  PC: TYPE = {sleeping, trying, critical};

S3  bprocess: MODULE =
S4    BEGIN
S5      INPUT y2 : NATURAL
S6      OUTPUT y1 : NATURAL
S7      LOCAL pc : PC
S8      INITIALIZATION
S9                  pc = sleeping;
S10                 y1 = 0
S11    TRANSITION
S12      [ pc = sleeping --> y1' = y2 + 1;
S13                          pc' = trying
S14        []
S15        pc = trying AND (y2 = 0 OR y1 < y2)
*S16 AND y1 > 0          %STRENGTHENED GUARD
S17             --> pc' = critical
S18        []
S19          pc = critical --> y1' = 0;
S20                          pc' = sleeping ]
S21  END;

S22  system: MODULE =
S23   bprocess
S24   []
S25   RENAME y2 TO y1, y1 TO y2 IN bprocess

S26  mutex:THEOREM system
S27   |- G(NOT(pc.1 = critical AND pc.2 = critical));
S28 END
```

Figure 5.3: SAL specification of the Bakery protocol.

`sleeping` (line S9).

- A *local ticket number* `y1`, which is a natural number. Initially, the local ticket number assumes 0 (line S10).
- A *remote ticket number* `y2` (line S5), which is a reference to the local ticket number of the other process.

A SAL transition consists of a *guard* and list of *assignments*, separated by `-->`. The assignments are used to update the value of the variables. Transitions in SAL are separated by `[]`. This model defines three transitions:

- *Sleeping → trying (lines S12-13):* If the program counter is `sleeping` (the process is sleeping), then set it to `trying` and "take the next

available ticket number".

- *Trying → critical (lines S15-17):*[4] If the program counter is `trying` (the process is trying) and it is this process's "turn", then enter the critical section.

- *Critical → sleeping (lines S19-20):* If the program counter is `critical` (the process is in the critical section), reset the local ticket number and return to sleeping.

A system containing two processes and executing the Bakery protocol is specified by instantiating `bprocess` two times and renaming the corresponding variables (lines S22-25). The mutual exclusion `mutex` property of the protocol (lines S26-28) specifies that it is never the case that both processes reside in the critical section.

Non-inductive mutex It turns out that the mutual exclusion property is not inductive with (the default) induction depth 10 – recall that line *S16 is ignored. The spurious counterexample returned by SAL's k-induction implementation is a path starting from a state where the first process assumes `pc=trying` and `y1=0`. This is not a reachable state because `y1` is a non-negative integer and the process increments `y1` in course of executing the transition 'sleeping → trying'.

Note that the mutual exclusion property is not k-inductive for any number of k. This is because, for instance, the state where both processes are trying and `y1=0` and `y2=1` is a recurring one after the second process enters the critical section and returns to local state trying.

Strengthened transitions & inductive mutex Consider adding line *S16 in Figure 5.3. This means restricting the guard of transition 'trying → critical'. As a result, the transitions of this model of the system contain fewer pairs of states than the original model. We call such transformations of the system model *strengthening transitions* (or strengthened transitions). It turns out that mutex is *inductive* in the model augmented with line *S16.[5]

Strengthened transitions can rule out spurious counterexamples. In the above case, the unreachable state leading to a spurious counterexample is where the first process is trying and `y1=0`. Thanks to strengthening the transition 'trying → critical', this transition cannot be executed in this state

[4] Ignore line *S16 for the moment.

[5] Mutex is k-inductive for all $k > 2$.

because its guard is false (`y1`$\not>$`0`). As a result, a k-long path causing the induction step of k-induction to fail does not exist.

Note that in any reachable state where the first process is trying it holds that `y1>0`. This is because transition 'sleeping $\rightarrow$ trying' updates `y1` to be `y2+1` and `y2` is always non-negative. Therefore, the proposed strengthened transitions in the model of the Bakery protocol preserves the set of reachable states; thus, it preserves the soundness of induction.

5.2.2 Strengthened Transitions: A General Framework

Strengthening transitions is a transformation from one state transition system into another. The intuition is that the original transitions subsume the strengthened ones so that the set of reachable states is preserved.

Definition 21. *Given state transition systems $STS = (S, T, S_0)$ and $STS' = (S, T', S_0)$, STS' is* strengthening transitions *of STS (or simply T' in STS' are* strengthened transitions*) if (1) for all $s, s' \in S, t' \in T'$ with $(s, s') \in t'$ implies that there is $t \in T$ with $(s, s') \in t$ and (2) the sets of reachable states in STS and STS' are the same.*

A corollary of condition (2) in Definition 21 is that strengthening transitions preserves the soundness of verifying the invariance of assertions: If the set of reachable states is the same in two systems, then an invariant in the first system is also an invariant in the second.

Corollary 2. *Given state transition systems STS and STS', and an assertion P, if STS' is strengthening transitions of STS, then P is an invariant in STS iff it is an invariant in STS'.*

The next theorem states that strengthening transitions results in a state transition system that preserves the set of invariants that are inductive in the original system. Under favorable circumstances, there are additional invariants that are inductive in the system with strengthened transitions – for instance, in case of the Bakery protocol in the previous section.

Theorem 7. *Given state transition systems STS and STS', an invariant P (in both STS and STS'), and $k > 0$, if STS' is strengthening transitions of STS and P is k-inductive in STS, then P is also k-inductive in STS'.*

Proof. The proof is indirect: Assume that P is not k-inductive in STS'. The contradiction directly follows from the fact that a k-long path in STS' is also a path in STS. Formally, let $s_0, s_1, ..., s_k$ be a a path in STS'. By definition,

we know that, for all $0 \leq i < k$, $(s_i, s_{i+1}) \in t'$ for some transition t' in STS' implies that there is a transition t in STS such that $(s_i, s_{i+1}) \in t$. Therefore, $s_0, s_1, ..., s_k$ is also a path in STS. This implies that if induction fails in STS' due to a path (in either the base or the inductive case), then this path is also considered in the induction in STS and this induction fails too, a contradiction. □

Caveat Note that it is possible that the shortest path between two reachable states decreases after strengthening transitions. This is because some state transitions might be missing in the state transition system with strengthened transitions. As a result, the base case of the induction might find counterexamples with larger depths. Therefore, if the base case of the induction is used for debugging, strengthening transitions can be avoided.

In Section 5.2.3, we show an example of strengthening transitions where, given two reachable states s and s', $(s, s') \in t$ for some transition in the original model iff $(s, s') \in t'$ for some strengthened transition. In this case, the base case of the induction returns counterexamples of the same length.

5.2.3 Transition Validated Strengthened Guards

We informally discuss a general application of strengthening transitions. The proposed strategy is applicable to models with guarded transitions and where the execution order of transitions is constrained by the specification. As we argue onwards, these models contain the models of a general class of message-passing protocols.

We observe that the idea of strengthened transitions in the Bakery protocol from Section 5.2.1 can be generalized based on two simple concepts:

- *Transition validated assertions [MP95]:* The simple observation of transition validated assertions is that *any* state that is a result of executing a transition reflects the state updates specified by the transition. In the example of the Bakery protocol, an assertion validated by the transition 'sleeping → trying' is `y1=y2+1`.

- *Sequential execution plans:* We focus on protocols specifying "sequential" processes. Formally, a sequential process i has a *sequential execution plan*. A sequential execution plan is a sequence of transitions $t_1, t_2, ...$ such that this sequence equals the sequence of transitions in any path σ of the system where all transitions of processes other than i are removed from σ. Intuitively, a sequential execution plan is determined by a process that executes transitions in a deterministic, pre-defined order.

In the Bakery protocol example, the sequence 'sleeping → trying', 'trying → critical', 'critical → trying', 'trying → sleeping', ... is a sequential execution plan.

The proposed strengthening scheme Given a multi-process system specified with guarded transitions, a process i with a sequential execution plan, a transition t of this process can be strengthened using the following scheme:

(1) *Obtain candidate TVA:* Assume that t' is the only transition that (immediately) precedes t in process i's sequential execution plan. Let a be a transition validated assertion (TVA) validated by t'.

In the example of the Bakery protocols, 'trying → critical' plays the role of t, 'sleeping → trying' the role of t', and `y1=y2+1` the role of a.

(2) *Restrict TVA:* If a "interferes" with transitions of processes other than i, restrict a such that it interferes only with transitions of process i. Denote a' the resulting assertion.

The semantics of interference depends on the particular system (or class of systems). In the example of the Bakery protocols, the restriction of `y1=y2+1` results in `y1>0`.

(3) *Strengthen guard via TVA:* If g_t is the guard of t, then change the guard to $g_t \wedge a'$ (cf. line *S16 in Figure 5.3). Intuitively, this corresponds to strengthening transitions because pairs (s, s') are excluded from t where a' does not hold for s.

Detecting spurious counterexamples It is possible that an invariant is non-inductive even in a model with strengthened transitions. In this case, strengthening transitions with transition validated assertions makes it easier to decide whether or not the returned inductive counterexample is spurious or not (cf. feature F1). This is because the inductive counterexample never contains state transitions that contradicts with the transition validate assertions. As a result, inductive counterexamples resemble more a real path of the system. Note that lemmas have a similar effect on inductive counterexamples.

Execution plans in message-passing Again, the existence of a sequential execution plan means that the corresponding process executes transitions in a deterministic, pre-defined order. In this case, the source of nondeterminism is due to the concurrent execution of different processes.

Note that all example message-passing protocols considered in this thesis specify sequential execution plans: Paxos consensus (Section 2.3), Echo Multicast [Rei94], regular register [ABND95], and OM Byzantine consensus [LSP82]. In addition, other protocols such as diagnosis [WLS97, SBS+11] or atomic broadcast [JRS11] and many other protocols also specify sequential execution plans.

Lemmas or strengthened transitions? Using transition validated assertions to strengthen transitions or to obtain lemmas are similar concepts. However, their implementation can affect the efficiency of the induction proofs. The advantage of lemmas is that they structure the proof, whereas strengthened transitions modify (and, arguably, complicate) the model. On the other hand, lemmas need to be proven before they can be used in strengthened induction proofs. This might result in significant time overhead.

For example, a transition validated lemma for the Bakery protocol is that if process one is trying, then `y1>0`. This lemma is an inductive invariant. The proof of the transition strengthened Bakery protocol takes approximately the same time (0.5 seconds) as with the above transition validated lemma. In addition, the proof of the lemma takes approximately the same amount of time.[6]

5.3 Related Work

Approaches against spurious counterexamples can be categorized as bottom-up and top-down [MP95]. Bottom-up approaches consider the state transition system only, whereas top-down ones are guided by the goal assertion. In their current form, both the classification of lemmas and strengthened transitions are bottom-up approaches. Bottom-up and top-down approaches can be combined. Also, the proposed classification and strengthened transitions can be made more effective using top-down approaches such as strengthened [dMRS03] or disjunctive invariants [Rus00].

Another bottom-up approach is message-flows induced lemmas [TT08, OTT09]. Our classification builds on message-flows using them as one possible source (class) of lemmas. We remark that the (parametric) properties considered in [TT08, OTT09] cannot be proven using message-flow induced lemmas only.

[6] This is a similar trade-off as between lemmas (incremental proof) and strengthened assertions [MP95].

An alternative approach to induction is using interpolants [McM03]. Proving the invariance of an assertion using interpolants is also based on over-approximating the set of reachable states and it is, in general, an incomplete verification procedure. Lemmas, derived by any method, e.g., as a result of our classification, can be used for more precise over-approximations. Intuitively, an interpolant is a side-product of the proof of (symbolic) insatisfiability, which is not always supported. This limits the general usability of interpolant-based techniques.

Finally, the proposed approaches are as complete as induction proofs. In general, completeness guarantees require strong assumptions such as finite state systems and the feasibility of induction with depth of the longest loop-free paths of the system [dMRS03].

5.4 Conclusions

We have proposed two techniques against spurious counterexamples in induction proofs. While the first technique, a classification of lemmas (Section 5.1), relate to fault-tolerant message-passing protocols, the second technique, strengthened transitions (Section 5.2) is applicable for general systems. These contributions set the stage for the automation of paper proofs that are still the current practice in the design of fault-tolerant message-passing protocols, e.g., [JRS11]. These paper proofs are usually split into "lemmas" to structure and, thus, ease the proof for the human verifier. These lemmas directly correspond to lemmas in induction proofs. Given the long history of proving complex message-passing protocols by hand [AW04], established proof structures can be used to derive new heuristics to discover lemmas.

After promising preliminary results of applying lemma classification and strengthened transitions for automated induction proofs, more application examples are needed to tune their efficiency for a wider range of systems. In addition, the proposed approaches being bottom-up techniques, an interesting extension would be to combine them with existing and new top-down approaches. For example, lemmas can be classified depending on the specification, i.e., target assertion. Even though specific to the specification of interest, such top-down classification would be still generally applicable given that certain specifications such as consensus, register properties, broadcast, etc. are shared by various systems.

Chapter 6

The MP-Basset Model Checker

We have developed a model checker called MP-Basset for the debugging and verification of general message-passing systems. MP-Basset builds on Basset [LDMA09], a model checker for actor programs. The source code, application examples, and documentation of MP-Basset is available at http://www.deeds.informatik.tu-darmstadt.de/peter/mp-basset/.

We summarize the main goals and corresponding design decision of MP-Basset:

- *Expressive specification (Section 6.1):* MP-Basset borrows Basset's architecture [LDMA09] to enable implementation/modeling of message-passing systems written in Java. This powerful feature is a result of utilizing Java Pathfinder, a model checker for general Java programs [JPF], which Basset is built upon.

- *Intuitive specification (Section 6.2):* We observe that quorum transitions are widely-used in the specifications of faulty-tolerant message-passing algorithms (see Section 3.4). Therefore, MP-Basset explicitly supports the specification of such transitions (not supported by Basset).

- *Efficient model checking (Section 6.3):* Partial-order reduction (POR) [CGP99] is a powerful technique to enable efficient verification of general and especially multi-process systems such as message-passing systems. As part of MP-Basset, the Basset model checker implements dynamic POR (DPOR) [FG05]. For a combination with DPOR and also to circumvent applications where DPOR is ineffective, MP-Basset implements LPOR from Chapter 4, an implementation of static POR [Val98].

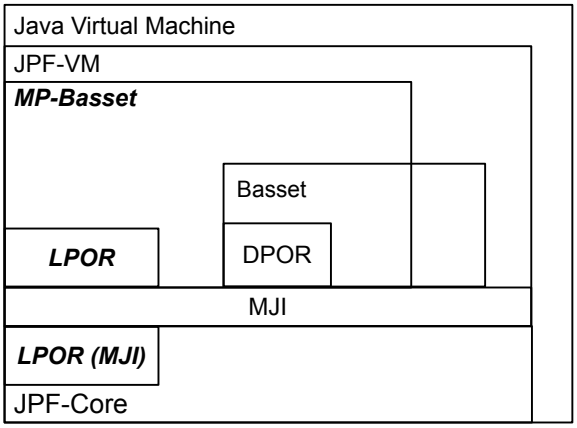

Figure 6.1: MP-Basset architecture illustration.

6.1 Basic Architecture: The Basset Model Checker

The architecture of MP-Basset is illustrated in Figure 6.1. The intuition is that the inclusion of a box denotes that it is "subsumed" by the outer boxes. As MP-Basset is an extension of Basset, we can also explain Basset's architecture based on Figure 6.1.

Basset runs on top of the Java Pathfinder (JPF) model checker [JPF]. JPF is written in Java itself and, as such, is executed by a Java Virtual Machine (JVM). The execution of a Java program that is model checked by JPF is "modeled" within JPF's *model layer*. We call this layer JPF-VM to refer to its functional similarity with the host JVM. Basset is an ordinary Java program that runs within JPF-VM.

JPF defines a gateway called *Model Java Interface* (MJI) between the modeled program and the core of JPF (JPF-Core). JPF-Core implements the search functionalities of JPF such as storing/matching states or the computation of enabled transitions per state. The MJI gateway can be used to access the services of JPF-Core or even the operating system hosting the JVM, and also to speed up the execution of Java code outside the model layer (see discussion in Section 6.3).

By default, JPF assumes a fine-grained interleaving of Java threads. In order to prevent JPF from exploring unnecessary interleavings, Basset uses MJI (a) to implement the semantics of message-passing systems, e.g., the execution of a transition is an atomic event, and (b) to implement different DPOR algorithms [LDMA09].

6.2 Feature 1: Quorum Transitions

As Basset does not support quorum transitions, MP-Basset implements quorum transitions by extending Basset's concept of "enabled message" into "enabled set of messages". More precisely, a set X of messages in the current state s is enabled if there is a transition t and a state s' such that $s \xrightarrow{t(X)} s'$. Basset's enabled messages can be seen as a special case with $|X| = 1$.

Note that computing these sets is time-expensive, as demonstrated in the following example; in worst case, the computation involves going through the powerset of *all* messages in the current state, which is an exponential overhead compared to the single-message case. Therefore, using quorum transitions can only reduce verification time if the memory reduction can compensate for the increased per state time overhead.

Example 9. Consider a state s and process i where messages m_1, m_2 and m_3 are in the input channels of process i in s. In order to find the enabled sets of messages, MP-Basset generates every set X in the powerset of $\{m_1, m_2, m_3\}$ to check if X is accessible for some transition $t \in T_i$ in s. If there is such t, then X is an enabled set of messages in s. The number of possible sets X is 2^3 compared to only three messages that need to be considered in a model containing only single-message transitions. This is the price we pay for the memory gain with quorum transitions – cf. discussion in Section 3.4. (End of Example 9)

Syntax-by-example Next, we explain the syntax of quorum transitions via an example. The syntax is intuitive and thanks to the expressiveness of Java it imposes no practical restrictions on the system being specified. For a complete reference of MP-Basset, we refer to its website: http://www.deeds.informatik.tu-darmstadt.de/peter/mp-basset/.

Example 10. The snippet in Figure 6.2 shows how Phase 2(a) of Paxos can be modeled as a quorum transition. MP-Basset transitions are specified by a pair of Java methods, the guard annotated by `@guard` and with Boolean return value, and the transition itself annotated by `@message`. The name of a guard must be that of the associated transition prefixed with _. The arguments of transition and guard are identical containing (1) the class of the messages this transition can consume and (2) an array `messages` corresponding to the sets of messages accessible for the transition.[1] The order of messages in `messages` is irrelevant. In this example, the messages consumed

[1]By convention, parameters are not read or written. The message class can be used to cast messages in the `messages` array.

```
@guard
public boolean _Phase2a(Phase2aParam m, Object[] messages){
    // guard: replies from a majority of N acceptors
    return messages.length==(Math.ceil((double)(N+1)/2));


@LPORAnnotation(
                senders=paxos.actor.Acceptor.class,
                recipients=paxos.actor.Acceptor.class,
                messageOut="Phase2b",
                isQuorumTransition=true,
                priority=3,
                quorumSize=2
)
@message
public void Phase2a(Phase2aParam m, Object[] messages){
    ... // select highest Phase2a message among 'messages'
    Phase2bParam proposal=new Phase2bParam(propNo, readReplHighest.val);
    for (ActorName a : acceptors)
        send(a, "Phase2b", proposal);
}
```

Figure 6.2: Example quorum transition specified in MP-Basset: Phase 2(a) of Paxos consensus.

by transition `Phase2a` are of class `Phase2aParam`. This Java class contains the information sent by an acceptor process in Phase 1(b) (not depicted).

In this example, the guard requires (i.e., returns true) that `messages` contains a message from a majority of the `N` acceptors. In the transition's code the proposer sends to all acceptor processes the highest-numbered proposal among the Phase 2(a) messages in `messages`. As in Basset, all processes of an MP-Basset specification are inherited from `ActorName`. A message can be sent using the pre-defined `send` operation, which takes the recipient and the message as arguments. A message consists of (i) the message's name (a `String`) and (ii) the message class. By convention, a transition can only consume a message if their names are matching.

MP-Basset implements the LPOR instantiation for message-passing systems from Section 4.4. The required syntax extensions presented in Section 4.4.1 are (partly) implemented in MP-Basset via the Java annotation called `LPORAnnotation`. `LPORAnnotation` is used to annotate the transition's method. Currently, `LPORAnnotation` is filled by the user. The annotation field `senders` (and `recipients`) corresponds to $t.I$ (and $t.O$). In this example, a proposer in Phase 2(a) consumes/sends messages from/to acceptors. The value of field `messageOut` corresponds to $t.M_O$ and contains the name of the messages sent by this transition.[2] The field `isQuorumTransition` is auxiliary for MP-Basset and it indicates that `messages` is indeed a message

[2] $t.M_I$ is implicitly given by the name of the transition.

array.

In addition to the extended syntax, `LPORAnnotation` is used to specify the initial transition heuristic via the `priority` field: High priority transitions are preferred to be used as initial transitions. Also, the annotation is used to implement our split transition refinement strategies. For example, the threshold of exact quorum transitions can be specified in `quorumSize`. In the example, the number of acceptor processes is three, which means that two acceptors are a majority. (End of Example 10)

Syntax disclaimer The syntax explained in Example 10 corresponds to the current version of MP-Basset and is, partly, determined by MP-Basset (and Basset) internals. For example, the specification of each process (including its transitions) is interpreted as native Java code by the compiler. Therefore, the guard and the transition must be well-formed Java methods. As such, they are not allowed to share the same name even if the type of the return value is different (`boolean` for the guard and `void` for the transition). It is a legacy from Basset that messages and transitions are matched via their names. Also, a current limitation of a quorum transition is that every message in `messages` must be of the same class.

6.3 Feature 2: Integrating Java-LPOR

MJI trade-off with LPOR As implied by the architecture of Basset, MP-Basset can run Java code either in the model layer or in the host JVM accessible via the Model Java Interface (MJI). Due to the indirection of the model layer, execution in this layer is slower than in the host JVM. The modeled Java program can always execute code in the host JVM using MJI. However, as there is a speed penalty of using MJI, time efficient JPF applications should use MJI with care. One source of this time overhead is that MJI converts arguments of MJI method calls between the modeled and the host JVM's object model.

We explore this trade-off with Java-LPOR when integrating it into MP-Basset: We create two architectures, one where the LPOR algorithm runs in the model layer and another one where it runs in the host JVM (cf. Figure 6.2). We compare the time performance of the two architectures in Table 4.1: MJI "No" and "Yes" experiments show model checking time with the implementation in the model and the host JVM layer, respectively. We depict model layer times only for the exhaustive search experiments because the "CE" results show similar trends.

Results: MJI wins In our experiments, the MJI-based implementation is faster. This meets our expectations for (state-unconditional) LPOR without NET because no state information is passed (and thus converted) to LPOR, whereas in (state-conditional) full-fledged LPOR, the NET relation is a function of a small fraction of the current state (see Section 4.4.2). Although, for our message-passing instantiation of LPOR, the MJI overhead turns out to be more time efficient than executing the LPOR algorithm in the model layer, this does not necessarily generalize. In other LPOR applications, particularly where the entire state has to be converted for MJI, the execution time penalties may trade off differently.

Optimizing MJI calls The straightforward approach to call LPOR via MJI would be to serialize, pass, and de-serialize transitions as primitive types. To avoid this time-expensive and tedious task, we instantiate an exact *copy* of each transition within JPF-Core. As a result MJI calls can simply address transitions and the result of the queries is passed through primitive types. In particular, a stubborn set returned by Java-LPOR in MP-Basset is an integer array where a non-zero value in the i^{th} position means that the i^{th} transition is in the stubborn set.

6.4 Conclusions

We have extended the Basset model checker for message-passing systems [LDMA09] with LPOR's message-passing instantiation (Section 4.4) and quorum transitions (Section 3.4). The implementation of these techniques is efficient, as shown by a number of experiments with fault-tolerant message-passing protocols (Section 4.6).

A natural extension of MP-Basset is adding symmetry reduction to it. This is also motivated by the soundness of combining partial-order and symmetry reductions [EJP97]. While the current version of MP-Basset uses Java Pathfinder (JPF) as a "black box", an extension with symmetry reduction requires to look into JPF. Intuitively, JPF implements the following stateful model checking algorithm: "In every state s, expand s unless s has already been expanded".[3] Symmetry reduction modifies the algorithm like this: "In every state s, expand s unless *s or another state that is symmetric with s* has already been expanded". To implement this additional check, a thorough understanding of JPF is required.

[3]Expansion of a state means executing the transitions that are enabled in this state.

Chapter 7

Conclusions

We have proposed, discussed, and implemented different techniques to enable efficient verification of fault-tolerant message-passing protocols. We started the thesis by classifying its contributions into system models and verification procedures (cf. Figure 1.1 repeated below).

We conclude by discussing how the these contributions are applicable beyond the boundaries of the thesis (Section 7.1) and how they can be extended for other interesting application domains (Section 7.2).

7.1 Generalizations

Models The models proposed in Chapter 3 are specific to model-passing communication. While replicas (Section 3.3) and quorum transitions (Sections 3.4-3.5) typically appear in fault-tolerant settings, the models of message traffic (Section 3.1) and crashing processes (Section 3.2) are applicable for general message-passing systems. In addition, the concept of decomposing a system into replicas can be generalized into a multi-process model where message-passing is used as one possible communication abstraction.

In contrast to the previous models that are restrictive to message-passing, LPOR (Chapter 4) and strengthened transitions (Section 5.2) assume a very general model, namely, state transition systems. Therefore, these verification techniques are applicable way beyond the specialities of message-passing systems.

Verification procedures The experiments reported in the thesis have been conducted using explicit-state model checking with depth-first search (Section 3.6 and 4.6) and using a Boolean satisfiability (SAT) solver (Chapter 5). Murϕ [Mur] and MP-Basset (Chapter 6) are explicit-state model checkers supporting both depth-first and breadth-first searches. We have used depth-first search as the default setting.[1] In our induction experiments with the SAL verification suite [SAL], we have used the default Yices SAT solver.

These are example implementations that are not assumed by the proposed verification techniques. Both symmetry and partial-order reductions can be implemented with breadth-first search or with Binary Decision Diagrams [ABH+01, CJEF96]. Also, induction proofs can be efficiently implemented with Satisfiable Modulo Theories (SMT) [SAL] or even manually [MP95]. Further possible implementations include symbolic execution [GKS05, CDE08] or assisted theorem proving [WLS97].

7.2 Related Open Questions

Combining reductions We have evaluated symmetry and partial-order reductions in isolation (Section 3.6 and 4.6, respectively). They can be considered as orthogonal optimizations, as they are shown to be sound when used together [EJP97].

In addition, partial-order reduction can also be combined with a new optimization called dynamic interface reduction [GWZ+11]. Therefore, the efficiency of the proposed symmetry and partial-order reductions can be further improved through combinations with other reductions. However, the achieved efficiency of such combined reductions is yet to be seen, especially given the increased time overhead that might result from the complex reduction logic.

Direct verification Murφ and SAL specify simple input languages, which is key to the efficiency of these verifiers. Although MP-Basset is able to model check Java code, the input syntax of MP-Basset restricts the Java language (see Chapter 6). There are just few approaches to verify the actual (direct) implementation of distributed systems, which are limited to unexhaustive bug finding [YCW+09] or to small systems [GWZ+11]. The combination of reductions can be a key to enable direct verification.

[1] The reported gains of symmetry and partial-order reductions are similar using breadth-first search.

Verifying infinite systems The systems verified in this thesis are finite-state systems, i.e., the state space of their models contains a finite number of states.

This is not an inherent limitation of the proposed techniques. One form of infinite-state systems is parametric systems, where the model of the system depends on some parameters, e.g., the number of processes. The verification of parametric systems can be done by showing that the system is symmetric with respect to its parameters [TT08, GNRZ08]. Therefore, our symmetry detection strategy (Section 3.3) can be used in combination with parametric verification techniques.

Another form of infinite-state systems is where the system specifies infinite domains, e.g., real numbers. SMT solvers, e.g., [SAL], can be leveraged for the verification of such systems. For example, using SMT solvers together with classified lemmas or strengthened transitions (Chapter 5) can be used to verify systems with infinite domains.

Linearizability We have considered temporal logic [CGP99] to specify the desired properties of the system. These properties are verified or refuted by the verification process. In the context of distributed computing, a widely-used correctness condition is linearizability [HW90]. Unfortunately, general linearizability cannot be directly expressed in temporal logic.[2] It is yet to be explored how symmetry and partial-order reductions and induction proofs can help the efficient verification of general linearizability.

[2]There are special cases where this is possible though. For example, the specification of regular storage is also a form of linearizability that, in the models we consider (Sections 3.6 and 4.6), can be expressed by simple invariants.

Appendix A

Models of Message-Passing Systems

A.1 Proof of Equivalent Models of Message Traffic

We prove Lemmas 1, 2 and 3, which imply Theorem 1.

Proof in Appendix 1. *Lemma 1 holds.*

Proof. Construction of σ'. Without loss of generality, let σ be the path $s_0 \xrightarrow{t_0} s_1...$, where s_0 is an initial state of **M**. The path $\sigma' = s'_0 \xrightarrow{t'_0} s'_1...$ is constructed as follows. Firstly, be s'_0 an initial state in **RM** such that $s'_0(i) = s_0(i)$ for all $1 \leq i \leq n$. This is possible because the processes modeled by **M** and **RM** specify the same set of local states. For each $i = 0, 1, ...$, let X be the set of messages such that $s_i \xrightarrow{t_i(X)} s_{i+1}$. For each $1 \leq j \leq n$ and $m \in X \cap c_{j,id(t_i)}$ let a delivery event $del(j, id(t_i), m)$ be in σ'. These delivery events in σ' are executed directly after each other in an arbitrary order, and they are followed by t_i. In addition, $t_1, t_2, ...$ are executed in σ' in the same order and in σ. Note that t_i is enabled in σ iff it is enabled in σ' because (a) the messages in X are the only messages in the input channels of process $id(t_i)$, and thus, the model is with clean channels, and (b) the local states of processes are identical in both paths.

Stuttering equivalence of σ and σ'. The proof is an induction on the number of local transitions in σ and σ'. Consider the first local transition t_0. Since channels are empty in initial states, t'_0 is not a delivery event and $t'_0 = t_0$. This and $s'_0(i) = s_0(i)$ for all $1 \leq i \leq n$ imply that $s'_1(i) = s_1(i)$ for all i. Therefore, $L_{RM}(s'_0) = L_M(s_0)$ and $L_{RM}(s'_1) = L_M(s_1)$ follow from

the assumption that the system is process-labeled, and the integer sequences $i_0 = 0, i_1 = 1$ and $j_1 = 0, j_1 = 1$ show stuttering equivalence of σ and σ' for the base case of the induction.

Assume that $\sigma \approx_{st} \sigma'$ for the first k local transitions in σ and σ'. Furthermore, $s_{i_k}(i) = s'_{j_k}(i)$ for all $1 \leq i \leq n$. Therefore, $L_M(s_{i_k}) = L_{RM}(s'_{j_k})$. Let i_{k+1} be $k+1$. Furthermore, let j_{k+1} be $j_k + l + 1$, where $l = |X|$ and $s_{i_k} \xrightarrow{t_{i_k}(X)} s_{i_{k+1}}$. By construction, $t'_{j_k}, t'_{j_k+1}, ..., t'_{j_k+l-1}$ are delivery events. Therefore, $s_{i_k}(i) = s'_{j_k+l}(i)$ for all $1 \leq i \leq n$ and, from the assumption that the system is process-labeled, $L_{RM}(s'_{j_k}) = L_{RM}(s'_{j_k+1}) = ... = L_{RM}(s'_{j_k+l})$. Also by construction, $t'_{j_k+l} = t_{i_k}$ and $s'_{j_k+l} \xrightarrow{t'_{j_k+l}(X)} s'_{j_k+l+1} = s'_{j_{k+1}}$. Now, $s_{i_k}(i) = s'_{j_k+l}(i)$ for all $1 \leq i \leq n$ yields that $s_{i_{k+1}}(i) = s'_{j_k+l}(i)$ for all i. Therefore, and because the system is process-labeled we have that $L_M(s_{i_{k+1}}) = L_{RM}(s'_{j_{k+1}})$. □

Proof in Appendix 2. *Lemma 2 holds.*

Proof. Construction of σ'. Without loss of generality, let σ be the path $s_0 \xrightarrow{t_0} s_1...$, where s_0 is an initial state of **GRM**. The path $\sigma' = s'_0 \xrightarrow{t'_0} s'_1...$ is constructed as follows. Firstly, be s'_0 an initial state in **M** such that $s'_0(i) = s_0(i)$ for all $1 \leq i \leq n$. This is possible because the processes modeled by **GRM** and **M** specify the same set of local states. Furthermore, let $t'_1, t'_2, ...$ be all local transitions in σ'. In addition, $t'_1, t'_2, ...$ are executed in the same order in σ and σ'. For each local transition t_i in σ, let X be the set of messages such that $s_i \xrightarrow{t_i(X)} s_{i+1}$. Then, t_i is executed with accessible set X in σ' too. Note that a transition in σ' is enabled iff it is enabled in σ because the content of channels and the local states of processes are identical in both paths.

Stuttering equivalence of σ and σ'. The proof is an induction on the number of local transitions in σ and σ'. The base case of the same as can be shown as for Lemma 1.

Assume that $\sigma \approx_{st} \sigma'$ for the first k local transitions in σ and σ'. Furthermore, $s_{i_k}(i) = s'_{j_k}(i)$ for all $1 \leq i \leq n$. Therefore, $L_{GRM}(s_{i_k}) = L_M(s'_{j_k})$. Let i_{k+1} be $i_k + l + 1$, where l is the number of delivery events that are executed between the k^{th} and $k+1^{th}$ local transition in σ. Therefore, and because the system is process-labeled, $L_{GRM}(s_{i_k}) = L_{GRM}(s_{i_k+1}) = ... = L_{GRM}(s_{i_k+l})$. Furthermore, let j_{k+1} be $k+1$. Because delivery events change no local state of any process, $s_{i_k+l}(i) = s'_{j_k}(i)$ for all $1 \leq i \leq n$. By construction, $t'_{j_k} = t_{i_k+l}$ and $s'_{j_k} \xrightarrow{t'_{j_k}(X)} s'_{j_k+1} = s'_{j_{k+1}}$, where X is a set of messages such that $s_{i_k+l} \xrightarrow{t_{i_k+l}(X)} s_{i_k+l+1} = s_{i_{k+1}}$. Now, $s_{i_k+l}(i) = s'_{j_k}(i)$ for all $1 \leq i \leq n$

yields that $s_{i_{k+l}}(i) = s'_{j_{k+1}}(i)$ for all i. Therefore, and because the system is process-labeled we have that $L_{GRM}(s_{i_{k+l}}) = L_M(s'_{j_{k+1}})$. □

Proof in Appendix 3. *Lemma 3 holds.*

Proof. The runs $\sigma, \sigma', \sigma''$ are sequences of configurations in the form $\sigma = c_0c_1..., \sigma' = c'_0c'_1...$, and $\sigma'' = c''_0c''_1....$ From $\sigma \approx_{st} \sigma'$, there must exist $e_0 = 0 < e_1 < ...$ and $f_0 = 0 < f_1 < ...$ such that $L(c_{e_k}) = L(c_{e_k+1}) = ... = L(c_{e_{k+1}-1}) = L(c'_{f_k}) = L(c'_{f_k+1}) = ... = L(c'_{f_{k+1}-1})$ for all $k = 0, 1, ...$ Also, from $\sigma' \approx_{st} \sigma''$, there must exist $g_0 = 0 < g_1 < ...$ and $h_0 = 0 < h_1 < ...$ such that $L(c'_{g_k}) = L(c'_{g_k+1}) = ... = L(c'_{g_{k+1}-1}) = L(c''_{h_k}) = L(c''_{h_k+1}) = ... = L(c''_{h_{k+1}-1})$.

We modify the integer sequences e_k, f_k, g_k, h_k to be *maximal*. For the brevity of the discussion, we define $k > 0$, $i \in \{e, f, g, h\}$, and $s \in \{c, c', c''\}$. For example, if $i = e$ and $s = c$ then we write s_{i_k} and mean c_{e_k}. For each k, i and s, we re-assign $i_{k+1} = i_{k+l+1}, i_{k+2} = i_{k+l+2}, ...$ if $L(s_{i_{k+1}-1}) = L(s_{i_{k+1}}) = L(s_{i_{k+2}}) = ... = L(s_{i_{k+l}})$ and $L(s_{i_{k+1}-1}) \neq L(s_{i_{k+l+1}})$.

First, we prove that the new assignment preserves stuttering equivalence. Without loss of generality, we consider e_k and f_k and the modified assignment. Let e'_k and f'_k denote the modified assignment. The proof is an induction on $k = 0, 1, ...$ For $k = 0$, we know that $L(c_{e_0}) = ... = L(c_{e_1-1}) = L(c_{e_1}) = ... = L(c_{e_2-1}) = ... = L(c_{e_l}) = ... = L(c_{e_{l+1}-1}) = L(c'_{f_0}) = ... = L(c'_{f_1-1}) = ... = L(c'_{f_{l'}}) = ... = L(c'_{f_{l'+1}-1})$ for some $l, l' > 0$ where $e'_1 = e_{l+1}$ and $f'_1 = f_{l'+1}$. We also know that $L(c_{e_0}) \neq L(c_{e_{l+1}})$ and $L(c'_{f_0}) \neq L(c'_{f_{l'+1}})$. Moreover, we know that $l = l'$. Otherwise, either $l' < l$ or $l' > l$ holds. However, $l' < l$ means $l' + 1 \leq l$ which would imply $L(c_{e_{l'+1}}) = L(c'_{f_{l'+1}}) = L(c_{e_0}) = L(c'_{f_0})$. In turn, this would contradict with $L(c'_{f_0}) \neq L(c'_{f_{l'+1}})$. Similarly, $l' > l$ leads to a contradiction. Note that $l = l'$ implies that $L(c_{e'_1}) = L(c'_{f'_1})$. We have proven that $\sigma \approx_{st} \sigma'$ for $k = 0$ with e'_k and f'_k. Now, assuming that the same holds for every k, we prove that it also holds for $k + 1$ (induction). By the induction hypothesis, we have that $L(c_{e'_{k+1}}) = L(c'_{f'_{k+1}})$ and there is $l > 0$ such that $e'_{k+1} = e_l$ and $f'_{k+1} = f_l$. We know from $\sigma \approx_{st} \sigma'$ that $L(c_{e_l}) = L(c'_{f_l})$. Now, the induction step follows similar to the base case.

Second, we show that the integer sequences e'_k and h'_k satisfy the definition of $\sigma \approx_{st} \sigma''$. We use another induction on k. From $\sigma \approx_{st} \sigma'$, we have $L(c_{e'_0}) = ... = L(c_{e'_1-1}) = L(c'_{f'_0}) = ... = L(c'_{f'_1-1})$. From $\sigma' \approx_{st} \sigma''$, we have $L(c'_{g'_0}) = ... = L(c'_{g'_1-1}) = L(c''_{h'_0}) = ... = L(c''_{h'_1-1})$. Since $e'_0 = f'_0 = g'_0 = h'_0 = 0$, we have that $L(c_{e'_0}) = L(c''_{h'_0})$. This implies $L(c_{e'_0}) = ... = L(c_{e'_1-1}) = L(c''_{h'_0}) = ... = L(c''_{h'_1-1})$, i.e., $\sigma \approx_{st} \sigma''$ for $k = 0$. We know that

$L(c_{e'_1}) = L(c'_{f'_1})$ and $L(c'_{g'_1}) = L(c''_{h'_1})$. This implies that $L(c_{e'_1}) = L(c''_{h'_1})$; otherwise, it must be that $L(c'_{f'_1}) \neq L(c'_{g'_1})$. However, this is impossible because f'_k and g'_k are maximal integer sequences of the same path, and thus, $f'_1 = g'_1$. In summary, we have shown that $\sigma \approx_{st} \sigma''$ for $k = 0$ with e'_k and h'_k and that $L(c_{e'_1}) = L(c''_{h'_1})$. The induction step can be shown similarly to the base case. □

A.2 Proof of Equivalent Models of Crash Faults

Proof in Appendix 4. *Theorem 2 holds.*

Proof. Let $STS^M = (S^M, T^M, S_0^M, L^M, AP^M)$ and $STS^{M^c} = (S^{M^c}, T^{M^c}, S_0^{M^c}, L^{M^c}, AP^{M^c})$ be the STS determined by $\mathbf{M}$ and the corresponding crash model, respectively. Without the loss of generality, the "dummy" transition is denoted by t_d (and t_d^c), i.e., $t_d \in T^M$ (and $t_d^c \in T^{M^c}$) such that for every state $s \in S^M$ ($s^c \in S^{M^c}$) it holds that $(s, s) \in t_d$ ($(s^c, s^c) \in t_d^c$).

Let $\sigma = s_0, s_1, ...$ and $\sigma^c = s_0^c, s_1^c, ...$ are runs of STS^M and STS^{M^c}, respectively. The proof is by induction on the length of the prefixes of σ and σ^c. Given a prefix of σ (and σ^c), we construct a prefix of a path in STS^{M^c} (in STS^M) such that label-equivalence holds for these prefixes. Then, label-equivalence between σ (and σ^c) and the constructed path follows by induction.

The $\Rightarrow$ direction. Consider the prefix s_0, s_1 of σ as the base case. By the assumption of label-equivalence, $s_0 \in S_0^M$. In our construction, let $s_0^c \in S_0^{M^c}$ be such that for all $1 \leq i \leq n$ we have that $s_0^c(i) = (s_0(i), \top)$. Such s_0^c exists by the definition of the crash model. Note that all channels in both s_0 and s_0^c are empty because these are initial states. Let $t \in T^M$ be a transition such that $s_0 \xrightarrow{t} s_1$. If $t = t_d$, then we construct s_1^c such that $s_1^c = s_0^c$. Otherwise, there is a $t^c \in T^{M^c}$ and s_1^c such that $s_0^c \xrightarrow{t^c} s_1^c$ and for all $1 \leq i \leq n$ it holds that $s_1^c(i) = (s_1(i), \top)$. This follows from Definition 3 and that t^c can only update the local state of process $id(t^c)$. Note that $s_0^c \xrightarrow{t^c(\emptyset)} s_1^c$ because all channels are empty in initial states. Since the message-passing system is process/crash-labeled, we have that $L^M(s_0) = L^{M^c}(s_0^c)$ and $L^M(s_1) = L^{M^c}(s_1^c)$. In addition, we assume that t^c is such that for all channels $c_{l,m} \in C$ it holds that $s_1(c_{l,m}) = s_1^c(c_{l,m})$. This follows from condition (5) in

Definition 3 and that t^c is allowed to send messages only via the outgoing channels of process $id(t^c)$.

By induction, assume that for all $1 \leq j \leq k$ it holds that $L^M(s_j) = L^{M^c}(s_j^c)$. Furthermore, we have for all $1 \leq i \leq n$ that $s_j^c(i) = (s_j(i), \top)$ and for all channels $c_{l,m} \in C$ that $s_k(c_{l,m}) = s_k^c(c_{l,m})$. The construction of s_{k+1}^c is analogous to that of s_1^c. The only difference is that if $s_k^c \xrightarrow{t^c(X)} s_{k+1}^c$, then X is not necessarily the empty set.

The $\Leftarrow$ direction. By assumption, $s_0^c \in S_0^{M^c}$. In our construction, let $s_0 \in S_0^M$ be such that for all $1 \leq i \leq n$ we have that $s_0^c(i) = (s_0(i), \top)$. Let $t^c \in T^{M^c}$ be a transition such that $s_0^c \xrightarrow{t^c} s_1^c$. If $t^c = t_d$, then we construct s_1 such that $s_1 = s_0$. Otherwise, there is a $t \in T^M$ and s_1 such that $s_0 \xrightarrow{t} s_1$ and for all $1 \leq i \leq n$ it holds that $s_1^c(i) = (s_1(i), a_i^1)$, for some $a_i^1 \in \{\top, \bot\}$. Since the message-passing system is process/crash-labeled, we have that $L^M(s_0) = L^{M^c}(s_0^c)$ and $L^M(s_1) = L^{M^c}(s_1^c)$. In addition, for all channels $c_{l,m} \in C$ it holds that $s_1(c_{l,m}) \supseteq s_1^c(c_{l,m})$.

By induction, assume that for all $1 \leq j \leq k$ it holds that $L^M(s_j) = L^{M^c}(s_j^c)$. Furthermore, we have that $1 \leq i \leq n$ $s_j^c(i) = (s_j(i), a_i^j)$, for some $a_i^j \in \{\top, \bot\}$, and for all channels $c_{l,m} \in C$ that $s_k(c_{l,m}) \supseteq s_k^c(c_{l,m})$. If $a_i^k = \bot$, then we know that $s_k^c = s_{k+1}^c$ and that $s_k^c \xrightarrow{t_d^c} s_{k+1}^c$. This is because there can be no $t^c \neq t_d^c$ such that $s_k^c \xrightarrow{t^c} s_{k+1}^c$ because (2) in Definition 3 is unsatisfiable with $a_i^k = \bot$. In this case, we construct that $s_{k+1} = s_k$. Since $s_{k+1} = s_k$ and $s_{k+1}^c = s_k^c$, the induction step trivially holds. Otherwise, if $a_i^k = \top$, let t^c be a transition in the crash model such that $s_k^c \xrightarrow{t^c(X)} s_{k+1}^c$. From Definition 3 we know that there is a $t \in T^M$ and s_{k+1} such that $s_k \xrightarrow{t(X)} s_{k+1}$ and for all $1 \leq i \leq n$ with $s_{k+1}^c(i) = (s_{k+1}(i), a_i^{k+1})$. Note that from $s_k^c(i) = (s_k(i), a_i^k)$ and $s_k(c_{l,m}) \supseteq s_k^c(c_{l,m})$ for all channels $c_{l,m} \in C$, X is accessible for t in s_k. Therefore, it holds that $L^M(s_k) = L^{M^c}(s_k^c)$ and $L^M(s_1) = L^{M^c}(s_1^c)$. In addition, for all channels $c_{l,m} \in C$ it holds that $s_k(c_{l,m}) \supseteq s_{k+1}^c(c_{l,m})$. □

Appendix B

Local Partial-Order Reduction

B.1 Generalized LPOR

Algorithm 5, called *generalized* LPOR, is an LPOR-like algorithm that is optimized for better space-reductions using state-conditional NET relations.

Generalized LPOR uses a modified forward enable set function (Algorithm 4) which is different from Algorithm 2 in that it stores *pairs* of transitions in the *en*-fields (line 4.12). The reason why we store (t_1, t_2) if (t_1, t_2) is in a NET relation (and not only t_2 as in Algorithm 2) is that we can verify in the current state whether or not t_2 must be executed in future paths before t_1 can be enabled (as opposed to the simple check of Algorithm 2 if t_2 was ever executed in the current path). Assume that $(t_{\mathbf{dep}}, en)$ is in the forward enable set of t_1 and the conditions in lines 5.10-5.11 are true. Using the notations of Algorithm 5, we only add t_1 to the stubborn set if either t_3 is not already in the stubborn set or t_3 needs not necessarily be executed for t_2, and thus, $t_{\mathbf{dep}}$ to be enabled (line 5.12). The later condition is expressed by a relation called NET-transition-to-fire relation (state-conditional NET).

Definition 22. *A relation* $\mathbf{nttf} \subseteq S \times T \times T$ *is* NET-transition-to-fire (Nttf) *if for all* $t, t' \in T, s \in S$, *if* $(s, t, t') \in \mathbf{nttf}$, *then* $t \notin enabled(s)$ *and* t' *is in* σ *for all paths* σ *from* s *to some* $s' \in S$ *such that* $t \in enabled(s')$.

The correctness of generalized LPOR (Algorithm 5) does not depend on the NET relation (state-unconditional NET) used in Algorithm 4. The purpose of this relation is to "guess" (before model checking) those transitions that can potentially be subject to NET optimization, and Nttf is used to "verify" (during model checking) the soundness of NET. For example, a possible heuristic was shown in Section 4.3, where NET contains those pairs (t, t') transitions where t can only be enabled if t' is executed at least once

Algorithm 4: Generalized $FwdEnableSet(t)$ and $FwdEnableSetIdx(t, t')$ are pre-computed for every $t, t' \in T$.

```
4.1  function FwdEnableSetIdx(t, t')
4.2    forall the (t'', en) ∈ FwdEnableSet(t) do
4.3      if (t'', t') ∈ dep then return true;
4.4    return false;
4.5  function FwdEnableSet(tr)
4.6    Tr' ← {(tr, ∅)};
4.7    do
4.8      Tr ← Tr';
4.9      forall the t_1 ∈ T do
4.10       forall the (t, en) ∈ Tr do
4.11         if (t, t_1) ∈ ce then
4.12           en_1 ← en ∪ {(t_1, t_2) | (t_1, t_2) ∈ net};
4.13           Tr' ← Tr' ∪ {(t_1, en_1)};
4.14   while Tr ≠ Tr';
4.15   return Tr;
```

before t. In this case, it is indeed correct that $(s, t, t') \in \mathbf{nttf}$ iff t' is not in the current search path τ. It is possible in general, however, that t' is in τ but it must be executed at least once more for t to be enabled.

We now show a message-passing example that utilizes the Nttf-optimization of generalized LPOR. Based on MP-NET we can define a NET-transition-to-fire relation. If t' is a NET for t, and in some state s t' has not yet sent a message needed for t to be enabled, then t' must be in any path from s to a state where t is enabled. Note that it is possible that t' is in τ. This can happen if there is a message m in the channel from the process executing t' to the process executing t such that $m \notin t.M_I$. Intuitively, t' can enable t because $t'.M_O \cap t.M_I \neq \emptyset$ but t' can also enable other transitions and m is not meant for t.

In Appendix B.3 we prove that the following relation is indeed an Nttf relation.

Definition 23. MP-NET-transition-to-fire $\subseteq S \times T \times T$ *is a relation such that* $(s, t, t') \in$ *MP-NET-transition-to-fire iff* $(t, t') \in$ *MP-NET* $\wedge (s(c_{id(t'),id(t)}) \cap t.M_I = \emptyset)$.

Theorem 5 is a special case of the following theorem.

Theorem 8. *Let* (S, T, S_0) *be an STS and* $\mathbf{net} \subseteq T \times T$ *an arbitrary relation, and* $\mathbf{ce}, \mathbf{dep}$ *and* $\mathbf{nttf}$ *a can-enabling, dependency, and NET-transition-to-*

Algorithm 5: Generalized LPOR(t_I, s) stubborn set algorithm for every $s \in S$, $t_I \in T$.

```
5.1  Stub ← {t_I};
5.2  Trans ← {t_I};
5.3  while Trans ≠ ∅ do
5.4      choose t ∈ Trans;
5.5      Trans ← Trans \ {t};
5.6      forall the t_1 ∈ enabled(s) \ Stub do
5.7          if (t_1, t) ∈ dep then
5.8              Stub ← Stub ∪ {t_1};
5.9              if dep is non-transitive then Trans ← Trans ∪ {t_1};
5.10         if FwdEnableSetIdx(t_1, t) then
5.11             if ∃(t_dep, en) ∈ FwdEnableSet(t_1) : (t_dep, t) ∈ dep
5.12             ∧(en = ∅ ∨ ∀(t_2, t_3) ∈ en : (t_3 ∉ Stub ∨ (s, t_2, t_3) ∉ nttf))
                 then
5.13                 Stub ← Stub ∪ {t_1};
5.14                 Trans ← Trans ∪ {t_1};
5.15 return Stub;
```

fire relation, respectively. Then, for all $s \in S, t \in enabled(s)$, LPOR$(t, s)$ *is a stubborn set.*

B.2 Proof of Generalized LPOR

First we prove that Algorithm 5 terminates.

Lemma 4. *Given the notation of Theorem 8,* LPOR(t, s) *terminates for all $s \in S, t \in T$. In addition, the worst-case time complexity of* LPOR(t, s) *is $O(|T|^3 2^{|T|^2})$ (and $O(|T|^2)$) using (not using) the Nttf-optimization.*

Proof. LPOR(t, s) terminates if *Trans* is empty (line 5.3). Assume that LPOR(t, s) runs forever, i.e., *Trans* is never empty at line 5.3. In every iteration of the while-loop (lines 5.3-5.14) exactly one transition is removed from *Trans* (line 5.5) and zero or more transitions are added to it (lines 5.9 and 5.14). Since the set of all transition (T) is finite, there must be a transition t' that is added multiple times to *Trans*. However, since t' is also added to *Stub* (line 5.8 and 5.13), no transitions are ever removed from *Stub*, and t' can only be added to *Trans* if $t' \notin$ *Stub* (lines 5.7 and 5.10), t' cannot be added to *Trans* multiple times, a contradiction.

Consequently, the body of the while-loop (lines 5.3-5.14) can be executed at most $|T|$ times. Within the body, transition t_1 from T might be added to *Trans* (lines 5.6-5.14). If the Nttf-optimization is used, every element in $FwdEnableSet(t_1)$ might be queried and compared with *Stub* (lines 5.11 - 5.12). Since the maximum size of $FwdEnableSet(t_1)$ is $|T|2^{|T|^2}$, the overall time complexity follows. We assume that the operations of set inclusion and adding/removing elements to/from a set can be done in constant time. □

Next, we prove that given the current state s and a transition t in a set T' of transitions returned by LPOR, no other transition on which t depends and disabled in s can be enabled only through executing transitions outside T'. This property is key to prove the soundness of our pre-computation based stubborn set algorithm.

Lemma 5. *Given the notation of Theorem 8, for all* $s \in S, t, t_1, t_2 \in T$ *if* $t_1 \in \text{LPOR}(t, s)$ *and* $t_2 \notin enabled(s)$ *and* $(t_2, t_1) \in \mathbf{dep}$*, then for any path* σ *from* s *to* $s' \in S$ *such that* $t_2 \in enabled(s')$ *there is a transition* $t_4 \in \text{LPOR}(t, s)$ *which is in* σ*.*

Proof. Let $s = s_0 \xrightarrow{t'_1} s_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_n} s_n = s'$ denote the path σ. We first show that there is an *enabling sequence ES of* t_2 of transitions $t'_{k_1}, t'_{k_2}, \ldots, t'_{k_l}$, $1 \leq k_1 < \ldots < k_l \leq n$, such that $t_{k'_1} \in enabled(s)$ and, for all $1 < i \leq l$, $t'_{k_i} \notin enabled(s)$ and, for all $1 \leq i < l$, $(t'_{k_i}, t'_{k_{i+1}}) \in \mathbf{ce}$ and $(t'_{k_l}, t_2) \in \mathbf{ce}$. Let $0 < i \leq n$ be such that $t_2 \in enabled(s_i)$ and, for all $0 \leq j < i$, $t_2 \notin enabled(s_j)$. Since **ce** is a can-enabling relation, $(t'_i, t_2) \in \mathbf{ce}$. If $t'_i \in enabled(s)$, then t'_i constitutes *ES* ($l = 1$). Otherwise, let $0 < j \leq i$ be such that $t'_i \in enabled(s_j)$ and, for all $0 \leq k < j$, $t'_i \notin enabled(s_k)$. Again, $(t'_j, t'_i) \in \mathbf{ce}$. If $t'_j \in enabled(s)$, then t'_j, t'_i constitute *ES* ($l = 2$). Continue this construction until $t'_{k_1} \in enabled(s)$. The construction always terminates as $l \leq n$.

The proof is indirect. Assume that none of $t'_1, t'_2, \ldots, t'_n$ is in $\text{LPOR}(t, s)$. First, either t_1 is added to $Trans$ or, if *dep* is transitive, another transition t_1^d such that $(t_1, t_1^d) \in \mathbf{dep}$ is added to $Trans$ (lines 5.7-5.9). In both cases, $FwdEnableSet(t'_{k_1})$ is queried in lines 5.10-5.11 because $t'_{k_1} \in enabled(s)$ and $t'_{k_1} \notin Stub$. Let $(t_2, en) \in FwdEnableSet(t'_{k_1})$ for some en. We know that such (t_2, en) exists because $t'_{k_1}, t'_{k_2}, \ldots, t'_{k_l}$ is an enabling sequence of t_2. Since there can be multiple such (t_2, en) in $FwdEnableSet(t'_{k_1})$ we select the one which is *stored along* the enabling sequence *ES*, i.e., $(t_{k'_1}, \emptyset) \in Tr'$ and $(t'_{k_1}, t'_{k_2}) \in \mathbf{ce}$ (lines 4.6 and 4.11) and (t'_{k_2}, en_{k_2}) is added to Tr' (lines 4.12-4.13), $(t'_{k_2}, en_{k_2}) \in Tr$ and $(t'_{k_2}, t'_{k_3}) \in \mathbf{ce}$ and (t'_{k_3}, en_{k_3}) is added to Tr', …, $(t'_{k_{l-1}}, en_{k_{l-1}}) \in Tr$

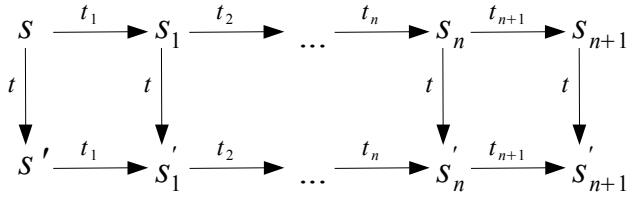

Figure B.1: Illustration of the proof of Theorem 10 .

and $(t'_{k_{l-1}}, t'_{k_l}) \in \mathbf{ce}$ and (t'_{k_l}, en_{k_l}) is added to Tr', and $(t'_{k_l}, en_{k_l}) \in Tr$ and $(t'_{k_l}, t_2) \in \mathbf{ce}$ and (t_2, en) is added to Tr'.

Next, the conditions in lines 5.10-5.11 hold because either $(t_2, t_1) \in \mathbf{dep}$ (if t_1 is added to $Trans$) or $(t_2, t_1^d) \in \mathbf{dep}$ (if t_1^d is added to $Trans$). The later is true because $(t_2, t_1) \in \mathbf{dep}$ and $(t_1, t_1^d) \in \mathbf{dep}$ and *dep* is transitive. The set *en* cannot be empty because $t'_{k_1} \notin Stub$ (line 5.12, first disjunct). Therefore, let $(t', t'') \in en$ such that $t'' \in Stub$ and $(s, t', t'') \in \mathbf{nttf}$. Again, if no such (t', t'') exists, then t'_{k_1} is added to *Stub*, a contradiction. We now show that t'' is among $t'_1, t'_2, \ldots, t'_n$, a contradiction. First, t' is among the transitions in *ES*. This is because (t_2, en) is stored along *ES* and every tuple in $FwdEnableSet(t'_{k_1})$ is written only once when added to the forward enable set (line 4.12). Say that $t' = t'_i$ for some $1 \leq i \leq n$. From $(s, t', t'') \in \mathbf{nttf}$ we know that t'' must be executed before t' can be enabled. Since we know that $t'_i \in enabled(s_{i-1})$, t'' must be among $t'_1, \ldots, t'_{i-1}$, the final contradiction.

□

A simple consequence of Lemma 5 is the following Corollary:

Corollary 3. *Given the notation of Theorem 8, for all* $s, s' \in S$, $t, t', t_1, ..., t_n \in T$ *such that* $t \in \text{LPOR}(t', s)$, $t_i \notin \text{LPOR}(t', s)$ *for all* $1 \leq i \leq n$, *and* $s \xrightarrow{t_1 \ldots t_n} s'$, *it holds that* $(t_i, t) \notin \mathbf{dep}$.

Proof. Indirectly, assume that $(t_i, t) \in \mathbf{dep}$ for some $1 \leq i \leq n$. If $t_i \in enabled(s)$, then t_i is added to $\text{LPOR}(t', s)$ in lines 5.7-5.9. Otherwise, Lemma 5 implies that there is $1 \leq j < i$ such that $t_j \in \text{LPOR}(t', s)$, a contradiction. □

Now we prove that generalized LPOR computes strong stubborn sets, which implies Theorem 8. In turn, Theorem 8 implies Theorem 5.

Theorem 9. *Given the notation of Theorem 8, for all* $s \in S, t' \in T$, $\text{LPOR}(t', s)$ *is a strong stubborn set.*

Proof. Given $s_1, s_2, ..., s_n \in S$, and $t_1, t_2, ..., t_n \in T$, let $s = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots s_{n-1} \xrightarrow{t_n} s_n$ be a path such that $t_1, t_2, ...t_n \notin \text{LPOR}(t', s)$. We first show that every $t \in \text{LPOR}(t', s)$ is a key transition (D2). Indirectly, assume that t_i for some $1 \leq i \leq n$ can disable t, i.e., $t \notin \mathit{enabled}(s_i)$. Therefore, $(t_i, t) \in \mathbf{dep}$ must hold, a contradiction by Corollary 3.

Next, we show that D1 holds. We know that every $t \in \text{LPOR}(t', s)$ is a key transition. Therefore, $t \in \mathit{enabled}(s_i)$ for every $1 \leq i \leq n$. Let s'_n be a state such that $s_{n-1} \xrightarrow{t_n} s_n \xrightarrow{t} s'_n$. From Corollary 3, $(t_n, t) \notin \mathbf{dep}$, so there exists s'_{n-1} such that $s_{n-1} \xrightarrow{t} s'_{n-1} \xrightarrow{t_n} s'_n$. As illustrated in Figure 4.3, repeating this rule n times, we obtain a path $s \xrightarrow{t} s' \xrightarrow{t_1} s'_1 \xrightarrow{t_2} ... \xrightarrow{t_{n-1}} s'_{n-1} \xrightarrow{t_n} s'_n$. □

Now we prove that stubborn sets generated by LPOR also satisfy D3 [Val98] if the **dep** relation used in LPOR is symmetric.

Theorem 10. *Given the notation of Theorem 8, if* **dep** *is symmetric, then for all* $s \in S, t' \in T$, $\text{LPOR}(t', s)$ *satisfies D3, i.e., for all* $t, t_1, t_2, ... \in T$ *such that* $t \in \text{LPOR}(t', s)$ *is a key transition,* $t_1, t_2, ... \notin \text{LPOR}(t', s)$, *and* $s \xrightarrow{t_1 t_2 ...}$ *is an infinite path, there is an infinite path* $s \xrightarrow{t t_1 t_2 ...}$.

Proof. Indirectly, assume that there is $n > 0$ such that for all $s'_n \in S$ where $s \xrightarrow{t t_1 t_2 ... t_n} s'_n$ it holds that $t_{n+1} \notin \mathit{enabled}(s'_n)$. As t is a key transition (note that every transition in $\text{LPOR}(t', s)$ is a key transition), t is enabled in $s_n \in S$ where $s \xrightarrow{t_1 t_2 ... t_n} s_n$ (illustrated in Figure B.1). Let $s'_n \in S$ be any state such that $s_n \xrightarrow{t} s'_n$. From Corollary 3, we know that $(t_{n+1}, t) \notin \mathbf{dep}$. In addition, as **dep** is symmetric, it also holds that $(t, t_{n+1}) \notin \mathbf{dep}$. Therefore, t cannot disable t_{n+1} and we have $t_{n+1} \in \mathit{enabled}(s'_n)$. Finally, Theorem 9 (D1) implies that there is a path $s \xrightarrow{t t_1 t_2 ... t_n} s'_n \xrightarrow{t_{n+1}}$, a contradiction. □

B.3 Proofs of LPOR Relations for Message-Passing

Lemma 6. *Given any MP protocol, MP-can-enable is a can-enabling relation.*

Proof. The proof is indirect. Assume that there are $s, s' \in S, t, t' \in T$ such that $t' \notin \mathit{enabled}(s)$ and $s \xrightarrow{t} s'$ and $t' \in \mathit{enabled}(s')$ and $(t, t') \notin$ MP-can-enable.

First, assume that t and t' are local. Let i be the process executing t (and t'). From $(t, t') \notin$ MP-can-enable we have that either $C = \emptyset$ or

$\forall x \in C : x \in Inc(t) \cap CompTS(t')$. From $t' \notin enabled(s)$ we know that $g_{t'}(X, s(i))$ is false for every $X \subseteq (\cup_{\forall j \in P} s(c_{j,i}))$. Since t and t' are local, $(\cup_{\forall j \in P} s(c_{j,i})) \supseteq (\cup_{\forall j \in P} s'(c_{j,i}))$ because process i sends no message to itself (it can consume messages though). Therefore, there is no $X \subseteq \cup_{\forall j \in P} s'(c_{j,i})$ such that $X \not\subseteq \cup_{\forall j \in P} s(c_{j,i})$. This implies that $s(i) \neq s'(i)$. Let $x_{i_1}, ..., x_{i_k}$ be the variables such that $s.x_{i_1} \neq s'.x_{i_1}, ..., s.x_{i_k} \neq s'.x_{i_k}$. From this we know that $x_{i_1}, ..., x_{i_k} \in W(t)$, and, from $C = \emptyset$, $x_{i_1}, ..., x_{i_k} \notin R(t')$. Now, let s_1 be a state which equals s except that $s_1.x_{i_1} = s'.x_{i_1}$. From $x_{i_1} \notin R(t')$, t' is disabled in s_1. Similarly, let s_2 be a state which equals s_1 except that $s_2.x_{i_2} = s'.x_{i_2}$. t' cannot be enabled in s_2 either. After repeating this rule k times, we have that s' equals s_{k-1} except the value of x_{i_k} and t' is disabled in s', a contradiction. Therefore, we assume for all $x_{i_1}, ..., x_{i_k} \in C$ that $x_{i_j} \in Inc(t) \cap CompTS(t')$ $(1 \leq j \leq k)$. Let s_1 be a state which equals s except that $s_1.x_{i_1} = s'.x_{i_1}$. From $x_{i_1} \in Inc(t)$ we know that $s_1.x_{i_1} > s.x_{i_1}$. This implies that t' is disabled in s_1 because $x_{i_1} \in CompTS(t')$. We can continue like this to show that t' is disabled in s_k. Note that s_k may differ from s' in that $s_k.x \neq s'.x$ for some $x \notin C$. We can obtain s' from s_k and show that t' is disabled in s' similarly to the above case where $C = \emptyset$ was assumed.

Second, assume that t and t' are not local. Again, from $t' \notin enabled(s)$ we know that $g_{t'}(X, s(id(t')))$ is false for every $X \subseteq (\cup_{\forall i \in P} s(c_{i,id(t')}))$. Since t and t' are not local, $s(id(t')) = s'(id(t'))$ because t can only change the local state of process $id(t)$. Therefore, there must be $X' \subseteq (\cup_{\forall i \in P} s'(c_{i,id(t')}))$ such that $g_{t'}(X', s'(id(t')))$ is true. From $(t, t') \notin$ MP-can-enable, we have three cases. First, $id(t') \notin t.O$. From $t' \notin enabled(s)$ and that t and t' are not local we know that $X' \not\subseteq (\cup_{\forall i \in P} s(c_{i,id(t')}))$, i.e., X' is not accessible for t' in s. Therefore, $s'(c_{id(t),id(t')}) \setminus s(c_{id(t),id(t')}) \neq \emptyset$ because $s(c_{i,id(t')}) = s'(c_{i,id(t')})$ for all $i \neq id(t)$. This contradicts with (3) in the definition of well-formedness. Second, $id(t) \notin t'.I$. We also know that $X' \cap s'(c_{id(t),id(t')}) \neq \emptyset$, which contradicts with (1). Third, $t.M_O \cap t'.M_I = \emptyset$. Let $m \in M$ be a message such that $m \in X'$ and $m \in (s'(c_{id(t),id(t')}) \setminus s(c_{id(t),id(t')}))$. We know that these two sets are non-empty. Further, there must be such a message m which is in both sets otherwise X' is accessible for t' in s. From (2) and $m \in X'$ we know that $m \in t'.M_I$. From (4) and $m \in (s'(c_{id(t),id(t')}) \setminus s(c_{id(t),id(t')}))$ we have that $m \in t.M_O$, a contradiction.

□

Lemma 7. *Given any MP protocol, MP-dependency is a dependency relation.*

Proof. The proof is indirect. Assume that there is $s, s', s'' \in S, t, t' \in T$

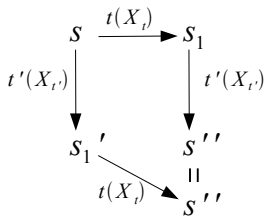

Figure B.2: Illustration of the proof of Lemma 7.

such that $t, t' \in \mathit{enabled}(s)$ and $s \xrightarrow{t} s'$ and either (a) $t' \notin \mathit{enabled}(s')$ and $(t, t') \notin$ MP-dependency or (b) $s \xrightarrow{tt'} s''$ and there is no $s \xrightarrow{t't} s''$ and $(t, t') \notin$ MP-dependency. We first consider (a). t' can be disabled in s' either because $s(id(t')) \neq s'(id(t'))$ or because there is $X \subseteq (\cup_{\forall i \in P} s(c_{i,id(t')}))$ which is accessible for t' in s but is not accessible for t' in s'. Since $(t, t') \notin$ MP-dependency, t and t' are not local. This means that $s(id(t')) = s'(id(t'))$ because t can only change $s(id(t))$ and $id(t) \neq id(t')$. Also, since t and t' are not local we have $(\cup_{\forall i \in P} s(c_{i,id(t')})) \subseteq (\cup_{\forall i \in P} s'(c_{i,id(t')}))$. This means that $X \subseteq (\cup_{\forall i \in P} s'(c_{i,id(t')}))$, i.e., X is accessible for t' in s'.

Let us now consider (b). The proof is illustrated in Figure B.2. Again, t and t' cannot be local. Let $id(t) = i$ and $id(t') = j$. Let s_1 be the state after the execution of t in s. Let s'_1 be the state after the execution of t' in s with the accessible set $X_{t'}$ such that t' is executed in s_1 with the same $X_{t'}$. Assume indirectly that $X_{t'}$ is not accessible for t' in s. This means that t sends a message m to process j such that $m \in X_{t'}$. Since t and t' are well-formed, we have that $j \in t.O$, $i \in t'.I$, $m \in t.M_O$, and $m \in t'.M_I$, which implies that $(t, t') \in$ can-remote-enable. However, this contradicts with $(t, t') \notin$ MP-dependency. Now, from $i \neq j$ we have $s(j) = s_1(j)$ and since t' is executed in s and s_1 with the same $X_{t'}$ we also have $s'_1(j) = s''(j)$.

One reason that no $s \xrightarrow{t't} s''$ exists can be that t is disabled in s'_1. However, this is impossible because process j cannot change the local state of process i and t' can only add messages to the input buffers of i. Formally, assume that $s \xrightarrow{t(X_t)} s_1$. We know that $s(i) = s'_1(i)$ and $X_t \subseteq \cup_{\forall k \in P} s'_1(c_{k,i})$ because $i \neq j$. Therefore, $g_t(X_t, s'_1(i))$ holds, i.e., t is enabled in s'_1. Let s''' be the state such that $s'_1 \xrightarrow{t(X_t)} s'''$. We now show that $s'' = s'''$, which leads to the final contradiction. We showed that $s'_1(j) = s''(j)$. This and $i \neq j$ imply that $s''(j) = s'''(j)$. Similarly, we can show that $s''(i) = s'''(i)$.

In addition, the content of all channels in s'' and s''' are identical too. For simplicity, assume that every message m contains the identifier of the channel where m resides. Therefore, it suffices to show that the unions C''

and C''' of all channels in s'' and s''' are the same. Let C denote the union of all channels in s. Let A_t ($A_{t'}$) denote the messages added by transition t (and t'), i.e., $A_t = \cup_{\forall k \in P} s_1(c_{i,k}) \setminus s(c_{i,k}) = \cup_{\forall k \in P} s'''(c_{i,k}) \setminus s'_1(c_{i,k})$ and $A_{t'} = \cup_{\forall k \in P} s'_1(c_{j,k}) \setminus s(c_{j,k}) = \cup_{\forall k \in P} s''(c_{j,k}) \setminus s_1(c_{j,k})$. We have that $C'' = (C \setminus X_t \cup A_t) \setminus X_{t'} \cup A_{t'}$ and $C''' = (C \setminus X_{t'} \cup A_{t'}) \setminus X_t \cup A_t$. Since $X_t \cap A_{t'} = \emptyset$, $C''' = C \setminus X_{t'} \setminus X_t \cup A_{t'} \cup A_t$. From $i \neq j$ we have $X_t \cap X_{t'} = \emptyset$, which implies $C''' = C \setminus X_t \setminus X_{t'} \cup A_t \cup A_{t'}$. Finally, $(t, t') \notin$ can-remote-enable implies $X_{t'} \cap A_t = \emptyset$, which implies $C'' = C'''$.

□

Lemma 8. *Given any MP protocol, MP-NET-transition-to-fire is a NET-transition-to-fire relation.*

Proof. The proof is indirect. Assume that there is $s \in S, t, t' \in T$ such that $(s, t, t') \in$ MP-NET-transition-to-fire and a path $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$ such that $t \in \mathit{enabled}(s_n)$ and $t_i \neq t'$ for every $1 \leq i \leq n$. Let $\mathit{id}(t) = i$ and $\mathit{id}(t') = j$. Since $t \in \mathit{enabled}(s_n)$ there must be an $X \subseteq M$ such that $s_n \xrightarrow{t(X)} s'$ for some $s' \in S$. From $(s, t, t') \in$ MP-NET-transition-to-fire we know that $(t, t') \in$ MP-NET, which implies that $t \in \mathit{ID}$. Also, t is well-formed. Therefore, $X \cap s_n(c_{k,i}) \neq \emptyset$ iff $k \in t.I$. Furthermore, from $(t, t') \in$ MP-NET we know that $j \in t.I$. Let m be a message in $X \cap s_n(c_{j,i})$. Let t'' be the transition that sends m to process i. From the well-formedness of t we have $m \in t.M_I$, and $s(c_{j,i}) \cap t.M_I = \emptyset$ implies that t'' must be among $t_1, t_2, \ldots, t_n$. Now, $(t'', t) \in$ can-remote-enable because t'' and t well-formed and by the definition of m. Since $\mathit{id}(t'') = j$ and $(t, t') \in$ MP-NET it must be that $t'' = t'$, contradiction. □

B.4 Enriched Syntax of Message-Passing System Models

A local state in $S_i \subseteq D_{i_1} \times ... \times D_{i_l}$ is an assignment $\langle x_{i_1} = v_{i_1}, ..., x_{i_l} = v_{i_l} \rangle$ of variables $x_{i_1}, ..., x_{i_l}$ to values from domains $D_{i_1}, ..., D_{i_l}$. A transition t is *local to* itself or to another transition t' if $\mathit{id}(t) = \mathit{id}(t')$, otherwise t is *remote to* t'.

We associate with every transition $t \in T$ a tuple (I, M_I, O, M_O) where $I, O \subseteq P$ are sets of process IDs and $M_I, M_O \subseteq M$ are sets of messages. The convention is that each *field* of the tuple is denoted by $t.\mathit{field}$. The transitions are *well-formed* which means that for every $X \subseteq M$, $s, s' \in S$ and $j \in P$ such that $s \xrightarrow{t(X)} s'$ the following holds: (1) $X \cap s(c_{j,\mathit{id}(t)}) \neq \emptyset$ implies $j \in t.I$

and (2) $X \subseteq t.M_I$. Furthermore, (3) $s'(c_{id(t),j}) \setminus s(c_{id(t),j}) \neq \emptyset$ implies $j \in t.O$ and (4) $s'(c_{id(t),j}) \setminus s(c_{id(t),j}) \subseteq t.M_O$.

We say that t is *input-deterministic* if t is in some $ID \subseteq T$ such that $ID \subseteq \{t \in T \,|\, \forall s, s' \in S, i \in P, X \subseteq M : s \xrightarrow{t(X)} s' \wedge i \in t.I$ implies $X \cap s(c_{i,id(t)}) \neq \emptyset\}$.

We write $x \in W(t)$ and say the t is a *write with respect to* a variable x if it might change the value of x. A set containing such variables is $W(t) \supseteq \{x| \exists s, s' \in S : s \xrightarrow{t} s' \wedge s.x \neq s'.x\}$. Supposed the operators $<, >$ and $=$ are defined for x, the set $Inc(t) \subseteq W(t)$ contains variables whose value can only be increased: $Inc(x) \subseteq \{x \,|\, \forall s, s' \in S : s \xrightarrow{t} s' \wedge s.x < s'.x\}$. In practice, x can be a *timestamp* which is incremented locally or upon receiving a message.

Further, a transition t is called a *read* transition *with respect* to a variable x if g_t depends on x. A set containing such variables is $R(t) \supseteq \{x \,|\, \exists s, s' \in S, X \subseteq M : (\forall y \neq x : s.y = s'.y) \wedge g_t(X, s(i)) \neq g_t(X, s'(i))\}$. As a special case, x in $CompTS(t) \subseteq R(t)$ if t cannot be enabled with the same accessible set by increasing x: $CompTS(t) \subseteq \{x \,|\, \forall s, s' \in S, X \subseteq M :$ if $g_t(X, s(i)) = \mathit{false} \wedge g_t(X, s'(i)) = \mathit{true} \wedge (\forall y \neq x : s.y = s'.y)$ then $s.x > s'.x\}$. In practice, x denotes a timestamp so that certain messages (with small timestamp) are discarded for higher values of x.

Note that it is always sound to remove transitions from ID and, given a transition t, to add (remove) variables to (from) $W(t), R(t)$ ($Inc(t)$, $CompTS(t)$).

Bibliography

[ABH+01] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design (FMSD)*, 18:97–116, 2001.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42:124–142, 1995.

[AM11] Sabina Akhtar and Stephan Merz. Partial-order reduction for verifying pluscal algorithms. In *Proceedings of Workshop on Automated Verification of Critical Systems (AVOCS)*, 2011. To appear.

[Ama11] http://aws.amazon.com/message/65648/, April 2011.

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley, 2004.

[BCG11] Radu Banabic, George Candea, and Rachid Guerraoui. Automated vulnerability discovery in distributed. In *Proceedings of Workshop on Hot Topics in System Dependability (HotDep)*, pages 188–193, 2011.

[BDH02] Dragan Bošnacki, Dennis Dams, and Leszek Holenderski. Symmetric spin. *International Journal on Software Tools for Technology Transfer (STTT)*, 4:92–106, 2002.

[BGG06] Ritwik Bhattacharya, Steven M. German, and Ganesh Gopalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *Proceedings*

of SPIN conference on Model Checking Software, pages 252–270, 2006.

[Bir05] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web services, and Applications*. Springer, 2005.

[BSSV09] Péter Bokor, Marco Serafini, Neeraj Suri, and Helmut Veith. Role-based symmetry reduction of fault-tolerant distributed protocols with language support. In *Proceedings of the International Conference on Formal Methods and Software Engineering (ICFEM)*, pages 147–166, 2009.

[CBS09] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 2009.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[CJEF96] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[CSCBM09] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A reduction theorem for the verification of round-based distributed algorithms. In *Proceedings of Workshop on Reachability Problems (RP)*, pages 93–106, 2009.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43:225–267, 1996.

[DET] http://www.isi.deterlab.net/.

[dMRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of Computer-Aided Verification (CAV)*, pages 14–26, 2003.

[EJP97] Allen E. Emerson, Somesh Jha, and Doron Peled. Combining partial order and symmetry reductions. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–34, 1997.

[FA95] Svend Frølund and Gul Agha. Abstracting interactions based on message sets. In *Proceeedings of Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems (ECOOP)*, pages 107–124, 1995.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 110–121, 2005.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *Proceedings of SPIN conference on Model Checking Software*, pages 95–112, 2007.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[GNRZ08] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards smt model checking of array-based systems. In *Proceedings of Joint Conference on Automated Reasoning (IJCAR)*, pages 67–82, 2008.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.

[Goo09] http://groups.google.com/group/google-appengine/msg/ba95ded980c8c179, July 2009.

[GWZ+11] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, pages 265–278, 2011.

[Hol03] Gerard Holzmann. *The Spin Model Checker*. Addison-Wesley Professional, 2003.

[HP94] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 197–211, 1994.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, 1990.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design (FMSD)*, 9:41–75, 1996.

[JPF] http://babelfish.arc.nasa.gov/trac/jpf.

[JRS11] Flavio Junqueira, Benjamin Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of IEEE/IFIP Conference on Dependable Systems and Networks (DSN/DCCS)*, pages 245–256, 2011.

[KSV06] Lars Michael Kristensen, Karsten Schmidt, and Antti Valmari. Question-guided stubborn set methods for state properties. *Formal Methods in System Design (FMSD)*, 29:215–251, 2006.

[KWG09a] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of Conference on Computer Aided Verification (CAV)*, CAV '09, pages 398–413, 2009.

[KWG09b] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 398–413, 2009.

[Lam83] Leslie Lamport. What good is temporal logic? In *Proceedings of Information Processing*, pages 657–668, 1983.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32:51–58, 2001.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[Lam06] Leslie Lamport. Checking a multithreaded algorithm with +cal. In *Proceeedings of Distributed Computing (DISC)*, pages 151–163. 2006.

[LDMA09] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *Proceedings of Conference on Automated Software Engineering (ASE)*, pages 468–479, 2009.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4, 1982.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Proceedings of Conference on Computer Aided Verification (CAV)*, pages 1–13, 2003.

[MDC06] Alice Miller, Alastair F. Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(8), 2006.

[MP-] http://www.deeds.informatik.tu-darmstadt.de/peter/mp-basset.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, 1995.

[Mur] http://www.cs.utah.edu/formal_verification/murphi/.

[NG97] Ratan Nalumasu and Ganesh Gopalakrishnan. A new partial order reduction algorithm for concurrent system verification. In *Proceedings of Conference on Hardware Description Languages and their Applications: Specification, Modelling, Verification and Synthesis of Microelectronic Systems (CHDL)*, pages 305–314, 1997.

[OTT09] John O'Leary, Murali Talupur, and Mark R. Tuttle. Protocol Verification using Flows: An Industrial Experience. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 172–179, 2009.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 107–122, 2002.

[Rei94] Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of Conference on Computer and Communications Security (CCS)*, pages 68–80, 1994.

[Rus00] John Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In *Proceedings of Conference on Computer Aided Verification (CAV)*, pages 508–520, 2000.

[SA06] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 339–356, 2006.

[SAL] http://sal.csl.sri.com/.

[SBS+11] Marco Serafini, Péter Bokor, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstätter, Fulvio Tagliabo, and Jens Koch. Application-level diagnostic and membership protocols for generic time-triggered systems. *IEEE Transactions on Dependable and Secure Computing*, 8(2):177 –193, 2011.

[SD01] Ulrich Stern and David L. Dill. Parallelizing the murφ verifier. *Formal Methods in System Design (FMSD)*, 18:117–129, 2001.

[TLSD11] Tian Huat Tan, Yang Liu, Jun Sun, and Jin Song Dong. Verification of computation orchestration system with compositional partial order reduction. In *Proceedings of Conference on Formal Engineering Methods (ICFEM)*, 2011. To appear.

[TNPK01] Tatsuhiro Tsuchiya, Shin'ichi Nagano, Rohayu Bt Paidi, and Tohru Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 12:81–95, 2001.

[TT08] Murali Talupur and Mark R. Tuttle. Going with the flow: Parameterized verification using message flows. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8, 2008.

[Val98] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, 1998.

[WLS97] Chris J. Walter, Patrick Lincoln, and Neeraj Suri. Formally verified on-line diagnosis. *IEEE Transaction on Software Engineering (TSE)*, 23(11):684–721, 1997.

[YCGK08] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of SPIN Workshop on Model Checking Software*, pages 288–305, 2008.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 213–228, 2009.

Curriculum Vitae

Personal data

Name:	Péter Bokor
Date of birth:	12/June/1980
Place of birth:	Budapest, Hungary
Current residence:	Darmstadt, Germany

Education

1999	High-school diploma
September/'99-July/'05	**Computer Science** program at Budapest University of Technology and Economics (**TU Budapest**)
October/'02-July/'03	**Scholarship** (Scherer Stipendium) at Technical University of Darmstadt (**TU Darmstadt**, Faculty: Computer Science)
July/'05	**Diploma** (equivalent with Master's degree) at TU Budapest tutored by Prof. András Pataricza (co-tutored by Prof. Neeraj Suri at TU Darmstadt), Topic: Model Checking of an On-line Diagnosis Algorithm, Result: **with honors**
September/'05 - Today	**Ph.D. candidate** at TU Budapest and TU Darmstadt (co-tutored by Prof. András Pataricza and Prof. Neeraj Suri), Topic: Efficient Verification of Fault-Tolerant Message-Passing Protocols
Date of graduation	29/November/2011

Technical experience

Teaching	Teaching assistant in lecture **Formal Methods** at BUTE (in English and Hungarian)
Project work	Participation in EU project **DECOS** (Dependable Embedded Components and Systems) and **HITACHI Research/TU Darmstadt** research collaboration
Research	Accepted research papers including the implementation of prototypes, and paper presentations at international conferences (see detailed list of papers below and also under http://www.deeds.informatik.tu-darmstadt.de/peter/)

Languages

German	fluency (top level state examination; UNICERT level III exam)
English	fluency
Hungarian	native

Journal publication

- Marco Serafini, Péter Bokor, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstaetter, Fulvio Tagliabo, Jens Koch, "Application-Level Diagnostic and Membership Protocols for Generic Time-Triggered Systems", IEEE Transactions on Dependable and Secure Computing (TDSC), 8(2), pp. 177-193, March 2011.

Conference/workshop publications

2011

- Péter Bokor, Johannes Kinder, Marco Serafini and Neeraj Suri, "Supporting Domain-Specific State Space Reductions through Local Partial-Order Reduction", Proc. of the 26^{th} IEEE/ACM Conf. on Automated Software Engineering (ASE), 2011, To appear.

- Habib Saissi, Péter Bokor, Marco Serafini and Neeraj Suri, "To Crash or Not To Crash: Efficient Modeling of Fail-Stop Faults", Invited paper, Proc. of Workshop on Logical Aspects of Fault-Tolerance (LAFT, in assoc. with LICS), 2011, To appear.

- Péter Bokor, Johannes Kinder, Marco Serafini and Neeraj Suri, "Efficient Model Checking of Fault-Tolerant Distributed Protocols", Proc. of the 41^{st} IEEE Conf. on Dependable Systems and Networks (DSN-DCCS), pages 73-84, 2011.

2010

- Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, Neeraj Suri, "Eventually Linearizable Shared Objects", Proc. of the 29^{th} Symposium on Principles of Distributed Computing (PODC) pages 95-104, 2010.

- Péter Bokor, Marco Serafini, Neeraj Suri, "On Efficient Models for Model Checking Message-Passing Distributed Protocols", Proc. of IFIP Conf. on Formal Techniques for Distributed Systems (FMOODS & FORTE), pages 216-223, 2010.

- Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke, Neeraj Suri, "Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas", Proc. of the 40^{th} IEEE Conf. on Dependable Systems and Networks (DSN-DCCS) pages 353-362, 2010.

2009

- Péter Bokor, Marco Serafini, Neeraj Suri, Helmut Veith, "Role-Based Symmetry Reduction of Fault-tolerant Distributed Protocols with Language Support", Proc. of the 11^{th} Conference on Formal Engineering Methods (ICFEM) pages 147-166, 2009.

- Péter Bokor, Marco Serafini, Neeraj Suri, and Helmut Veith, "Brief Announcement: Efficient Model Checking of Fault-tolerant Distributed Protocols Using Symmetry Reduction", Proc. of the 23^{rd} Symposium on Distributed Computing (DISC) 2009, pages 289-290.

2008

- Kohei Sakurai, Péter Bokor, Neeraj Suri, "Aiding Modular Design and Verification of Safety-Critical Time-Triggered Systems by use of Executable Formal Specifications", Proc. of the 11^{th} High Assurance Systems Engineering Symposium (HASE) 2008, pages 261-270.

- Péter Bokor, Sandeep Shukla, Andras Pataricza and Neeraj Suri, "Strengthened State Transitions for Complete Invariant Verification in Practical Depth-Induction", Proc. of the 3^{rd} Workshop on Automated Formal Methods Workshop (AFM, in assoc. with CAV 2008), pages 31-41.

2007

- Péter Bokor, Marco Serafini, Aron Sisak, Andras Pataricza , Neeraj Suri, "Sustaining Property Verification of Synchronous Dependable Protocols Over Implementation", Proc. of the 10^{th} High Assurance Systems Engineering Symposium (HASE) 2007, pages 169-178.

Miscellaneous publication

- Péter Bokor, Neeraj Suri, "Effective Model Checking of Message-Passing Distributed Protocols", DETERlab Newsletter, 2010 Fall.