



**Raimondas Sasnauskas**

**Symbolic Execution  
of Distributed Systems**

**Reports on Communications and Distributed Systems**

**Editor: Prof. Dr.-Ing. Klaus Wehrle**

**VOL 5**

# **Reports on Communications and Distributed Systems**

edited by  
Prof. Dr.-Ing. Klaus Wehrle  
Communication and Distributed Systems,  
RWTH Aachen University

Volume 5

**Raimondas Sasnauskas**

**Symbolic Execution of Distributed Systems**



Shaker Verlag  
Aachen 2013

---

# Symbolic Execution of Distributed Systems

Von der Fakultät für Mathematik, Informatik und  
Naturwissenschaften der RWTH Aachen University zur Erlangung  
des akademischen Grades eines Doktors der Naturwissenschaften  
genehmigte Dissertation

vorgelegt von

Dipl.-Inform.

**Raimondas Sasnauskas**

aus Kaunas, Litauen

Berichter:

Prof. Dr.-Ing. Klaus Wehrle  
Prof. Dr. Cristian Cadar

Tag der mündlichen Prüfung: 16.05.2013

---



**Bibliographic information published by the Deutsche Nationalbibliothek**  
The Deutsche Nationalbibliothek lists this publication in the Deutsche  
Nationalbibliografie; detailed bibliographic data are available in the Internet at  
<http://dnb.d-nb.de>.

Zugl.: D 82 (Diss. RWTH Aachen University, 2013)

Copyright Shaker Verlag 2013

All rights reserved. No part of this publication may be reproduced, stored in a  
retrieval system, or transmitted, in any form or by any means, electronic,  
mechanical, photocopying, recording or otherwise, without the prior permission  
of the publishers.

Printed in Germany.

ISBN 978-3-8440-2159-2  
ISSN 2191-0863

Shaker Verlag GmbH • P.O. BOX 101818 • D-52018 Aachen  
Phone: 0049/2407/9596-0 • Telefax: 0049/2407/9596-9  
Internet: [www.shaker.de](http://www.shaker.de) • e-mail: [info@shaker.de](mailto:info@shaker.de)

## Abstract

---

Automatism and high code coverage are the core challenges in testing distributed systems in their early development phase. Ideally, the testing process should cope with a large input space, non-determinism, concurrency, and heterogeneous operating environments to effectively explore the behavior of unmodified software. In practice, however, missing tool support imposes significant manual effort to perform high-coverage and integrated testing. One of the main testing challenges is to detect bugs that occur due to *unexpected inputs* or *non-deterministic events* such as node reboots or packet duplicates, to name a few. Often, these events have the potential to drive distributed systems into corner case situations, exhibiting bugs that are hard to detect using established testing and debugging techniques.

Recent advances in *symbolic execution* have proposed a number of effective solutions to automatically achieve high code coverage and detect bugs during testing of sequential, non-distributed programs. This attractive testing technique of unmodified code assists developers with concrete inputs to analyze distinct program paths. Being able to handle complex systems' software, these approaches only consider sequential programs and not their concurrent and distributed execution.

In this thesis, we present *symbolic distributed execution* (SDE)—a novel approach enabling symbolic execution of distributed systems. The main contribution of our work is three-fold. First, we generalize the problem space of SDE and develop methods to enhance symbolic execution for distributed software analysis. Second, we significantly optimize the basic execution model of SDE by eliminating *redundant execution paths*. The key idea is to benefit from the nodes' local communication and, thus, to minimize the number of states that represent a distributed execution. Third, we demonstrate the *practical applicability* of SDE with our tools KleeNet and SymNet, which are implemented as modular extensions of two popular symbolic execution frameworks.

With KleeNet, we realize an automated testing environment for self-contained distributed systems that generates test cases at high code coverage, including low-probability corner case situations before deployment. As a case study, we apply KleeNet to the Contiki operating system and show its effectiveness by detecting four insidious bugs in the  $\mu$ IP protocol stack. One of these bugs is critical and lead to the refusal of further TCP connections. Our second tool called SymNet provides a testing environment for unmodified software running in virtual machines that communicate in a real network setup. We combine *time synchronization* of virtual machines and *constraint synchronization* to explore distinct distributed execution paths. The application of our SymNet prototype to a HIP protocol implementation exposed a design bug and thereby demonstrates SymNet's effectiveness in automated testing of complex communication software.

## Kurzfassung

---

Automatismus und hohe Codeabdeckung sind die wesentlichen Herausforderungen beim Testen von verteilten Systemen in ihrer frühen Entwicklungsphase. Idealerweise sollte der Testprozess sowohl alle möglichen Eingaben, als auch Nichtdeterminismus, Nebenläufigkeit und heterogene Einsatzumgebungen der Software berücksichtigen. In der Praxis führen jedoch fehlende Werkzeuge und Methodiken zu hohem Zeit- und Kostenaufwand beim Testen. Insbesondere ist das Auffinden von Softwarefehlern schwierig, welche aufgrund *unerwarteter Eingaben* oder *nichtdeterministischer Ereignisse* (z.B. Neustart, Paketduplikate) auftreten. Diese kritischen Fehler führen verteilte Systeme oft zu Grenzfällen ihrer Ausführung und sind mit den etablierten Testverfahren schwer auffindbar.

*Symbolische Ausführung* ist bereits seit einigen Jahren eine sehr effektive Methode, um möglichst alle Ausführungspfade eines gegebenen Programms dynamisch abzuschreiten. Damit erreicht die Methode eine hohe Codeabdeckung und generiert überdies automatisch Testfälle, welche zu den einzelnen Programmpfaden führen. Aufgrund der Komplexität verteilter Systeme wurde die symbolische Ausführung bislang nur für sequentielle Programme eingesetzt.

Diese Arbeit stellt die Idee der *symbolisch verteilten Ausführung* vor, die symbolische Ausführung von verteilten Systemen ermöglicht. Das entwickelte Konzept besteht in der gleichzeitigen Ausführung mehrerer miteinander kommunizierender Softwareinstanzen und der Berücksichtigung der dabei möglichen Interaktionen zwischen diesen Systemen. Aus erkannten Fehlerfällen können automatisch Testfälle für die verteilte Ausführung eines Systems abgeleitet werden. Hierdurch lassen sich frühzeitig unvorhergesehene Grenzfälle der verteilten Ausführung erkennen und gezielt analysieren. Ein wesentlicher Beitrag hierbei sind die entwickelten Algorithmen zur Erkennung und Vermeidung *redundanter Zustände*, um eine effiziente Ausführung von kommunizierenden Systemen zu erreichen.

Das erarbeitete Konzept zur symbolischen Ausführung verteilter Systeme wurde in Form des Werkzeugs KleeNet realisiert und erfolgreich auf drahtlose Sensornetze angewendet. Bei den Tests des verbreiteten  $\mu$ IP-Protokollstapels im Contiki-Betriebssystem hat KleeNet kritische Fehler identifiziert, die jahrelang zu Problemen geführt haben, deren Ursache jedoch unentdeckt blieb. Ein großer Vorteil von KleeNet ist die Möglichkeit des *transparenten* Testens, ohne die Software vorher modifizieren zu müssen. Die Flexibilität des Ansatzes wurde mit dem zweiten Werkzeug SymNet demonstriert, welches über ein Netzwerk kommunizierende virtuelle Maschinen symbolisch ausführt. Hierfür werden die zu testenden Systeme *zeitsynchronisiert*, während symbolische Daten über das Netzwerk *serialisiert* übertragen werden. Die Anwendung des SymNet-Prototyps auf die Linux-Implementierung des HIP-Protokolls hat automatisch einen wichtigen Designfehler entdeckt.

## Acknowledgments

This thesis concludes an exiting chapter of my life spent at ComSys in Aachen. Thank you Klaus for the challenging research environment, countless *opportunities*, numerous “ganz kurz”, and continuous support in both my symbolic PhD career and personal endeavors. You have been a motivating mentor and reliable friend. I learned a lot! I hope that we will meet in Kaunas again or sail to the Curonian Spit someday. I would also like to thank Cristian Cadar for inspiring me with his research on symbolic execution, for sharing the early version of KLEE, and for acting as second opponent during my defense. I always enjoyed your hospitality and our discussions during my short visits at Imperial.

The joint work with my students was the main motivating factor during my time at ComSys. Thank you Alex, Andrius, Benjamin, Christian, Johannes, Oscar, Philipp, Russ, and Xinyu for the fruitful discussions during our weekly KleeNet meetings, your dedication, and the Glühwein at the Christmas market. In particular, I want to express my gratitude to Oscar for his 24/7 commitment, the state mapping madness, and for teaching me advanced C++. It was great working with you and for me you are an example of a rising research star. In addition, I want to thank Carsten Weise for many inspiring discussions and Fredrik Österlind for the joint efforts on making Contiki more reliable.

The genesis of this thesis began with a number of whiteboard discussions with Olaf. With addition of Hamad’s writing skills, Jó’s TinyOS mastery, and Tobi’s comments it was worth to give it a try and to “depart” from existing research. During the time period of six years, my colleagues significantly contributed to both my research and the shaping of my Lithuanian mind. Georg proved me that Germans can be funny whereas Elias convinced me that there is indeed order in chaos. Despite his roots, Ismet is for me an ideal of a friendly, heartfelt, and calm person. Also, I still strongly believe that Florian can defeat Usain Bolt in indoor sprints. Dirk is for me an example of a dedicated teacher with exceptional recovery capabilities. Finally, my big thanks goes to Ben, Henrik, Hanno, Janosch, Martin, Mirko, Mónica, Matteo, Nico, René, Uta, and Stefan for the great moments at ComSys. In particular, thank you Petra and Ulrike for the administrative and personal support. Also, my special thanks goes to Kai, Rainer, Alex, and Wladi for the reveling afternoons when I needed them.

This thesis is dedicated to the people I love the most: My parents Jūratė and Algirdas for exciting my curiosity with science, my brothers Mindaugas and Vytautas, Regina, Gintautas, and Reda for their support, my son Adas for making my life complete, and to my beautiful wife Gintarė for her belief, sacrifice, patience, encouragement, and endless love that have made me what I am today.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Challenges in Distributed Systems Testing . . . . .	2
1.2	Problem Statement . . . . .	3
1.3	Basic Concept and Research Challenges . . . . .	4
1.4	Contributions . . . . .	5
1.5	Outline . . . . .	6
<b>2</b>	<b>Background and Related Work</b>	<b>7</b>
2.1	Introduction to Testing . . . . .	7
2.2	Dynamic Symbolic Execution . . . . .	11
2.2.1	KLEE . . . . .	12
2.2.2	Selective Symbolic Execution . . . . .	17
2.3	Testing Distributed Systems . . . . .	20
2.3.1	Testing Before Deployment . . . . .	20
2.3.2	Testing After Deployment . . . . .	25
2.4	Conclusions . . . . .	29
<b>3</b>	<b>Formal Model of Symbolic Distributed Execution</b>	<b>31</b>
3.1	Symbolic Execution . . . . .	31
3.1.1	Concrete State Transition . . . . .	32
3.1.2	Symbolic State Transition . . . . .	33
3.1.3	Complexity . . . . .	34
3.1.4	Implementation . . . . .	35
3.1.5	Summary . . . . .	37
3.2	Symbolic Distributed Execution . . . . .	37
3.2.1	Distributed State Transition . . . . .	37

3.2.2	Symbolic Distributed State Transition . . . . .	40
3.2.3	Complexity . . . . .	40
3.2.4	Implementation . . . . .	41
3.2.5	Summary . . . . .	44
3.3	Conclusions . . . . .	44
<b>4</b>	<b>Design of an Efficient SDE</b>	<b>47</b>
4.1	Representation of Distributed States . . . . .	47
4.1.1	Problem Analysis . . . . .	47
4.1.2	Copy on Branch . . . . .	49
4.1.3	Copy on Write . . . . .	53
4.1.4	Super Distributed State . . . . .	57
4.1.5	Summary . . . . .	66
4.2	Packet Cache . . . . .	67
4.2.1	Problem Analysis . . . . .	67
4.2.2	Solution . . . . .	68
4.2.3	Implementation . . . . .	70
4.2.4	Summary . . . . .	71
4.3	Distributed Constraints . . . . .	71
4.3.1	Problem Analysis . . . . .	72
4.3.2	Solution . . . . .	73
4.3.3	Implementation . . . . .	77
4.3.4	Summary . . . . .	78
4.4	Distributed Assertions . . . . .	78
4.4.1	Problem Analysis . . . . .	79
4.4.2	Solution . . . . .	80
4.4.3	Implementation . . . . .	81
4.4.4	Summary . . . . .	82
4.5	Test Case Generation . . . . .	83
4.5.1	Problem Analysis . . . . .	83
4.5.2	Solution . . . . .	83
4.5.3	Implementation . . . . .	85
4.5.4	Summary . . . . .	86

4.6	Cluster-based Search and Parallelization . . . . .	87
4.6.1	Execution Model of SDE . . . . .	87
4.6.2	Clusters . . . . .	89
4.6.3	Cluster-based Search . . . . .	91
4.6.4	Cluster-based Parallelization . . . . .	94
4.6.5	Summary . . . . .	96
4.7	Conclusions . . . . .	97
<b>5</b>	<b>Implementation and Evaluation</b>	<b>99</b>
5.1	Symbolic Input Model . . . . .	99
5.1.1	Symbolic Input . . . . .	100
5.1.2	Node Model . . . . .	100
5.1.3	Network Model . . . . .	101
5.1.4	Implementation . . . . .	102
5.1.5	Summary . . . . .	103
5.2	KleeNet . . . . .	104
5.2.1	Design Overview . . . . .	104
5.2.2	Bridging KleeNet with COOJA . . . . .	108
5.2.3	Performance . . . . .	111
5.2.4	Bug Discovery Using KleeNet . . . . .	124
5.2.5	Summary . . . . .	130
5.3	SymNet . . . . .	131
5.3.1	Basic Concept . . . . .	132
5.3.2	Design Overview . . . . .	134
5.3.3	Prototype and Preliminary Evaluation . . . . .	139
5.3.4	Summary . . . . .	140
5.4	Conclusions . . . . .	140
<b>6</b>	<b>Discussion and Conclusion</b>	<b>143</b>
6.1	Results and Contributions . . . . .	143
6.2	Limitations of SDE . . . . .	144
6.3	Future Work . . . . .	145
6.3.1	Symbolic Time . . . . .	145

6.3.2	State Merging . . . . .	146
6.3.3	Test Case Analysis . . . . .	146
6.3.4	KleeNet Extensions . . . . .	147
6.3.5	SymNet Extensions . . . . .	148
6.4	Final Remarks . . . . .	149
Glossary		151
Bibliography		153

# 1

## Introduction

Implementing and testing new communication protocols and distributed applications is well recognized to be a difficult task for software developers. Ambiguous protocol definitions, distributed execution, node and network level non-determinism make the testing process very labor-intensive. In addition, achieving high code coverage, especially in testing for exceptions, is hard, often resulting in insufficient testing.

One typical testing issue in a distributed setting is low test coverage of concurrent applications [BFM<sup>+</sup>05]. Consequently, even small changes in the test setup may reveal new problems with the software. Moreover, critical bugs (e.g., node crash or wrong protocol state) appear only after prolonged operations, making it difficult to explain and narrow down their root causes since the information about the distributed state is lost [KAJV07].

A further challenge remains: Testing the interoperability with other, possibly even proprietary, implementations of the system. For example, a strong evidence of missing tool support is notable in the *Internet Engineering Task Force (IETF)* standardization community. To become an Internet Standard, each protocol draft requires at least two independent and interoperable implementations [Bra96]. Consequently, the developers spend significant amounts of manual effort and time during conformance testing sessions after each update of the draft. Nonetheless, even these tests do not provide an assurance that the implementations fully conform to (often ambiguous) draft details and manifest the same, interoperable behavior.

In the *Wireless Sensor Network (WSN)* community, reliable software is crucial because nodes are envisioned to be deployed in the absence of permanent network infrastructure and in environments with limited or no human accessibility [SPMC04, TPS<sup>+</sup>05]. Operating complex distributed protocols over lossy links and potentially unreliable nodes, WSNs demand extensive testing and debugging before deployment. After deployment, bugs are difficult to detect, costly to fix, and can potentially cause an operational outage.

For example, a recent WSN deployment in the Swiss Alps [BISV08] experienced sporadic packet loss on all GSM nodes simultaneously. This was caused by a bug in

the GPRS drivers of the WSN sink node used for collecting measurements. The bug prevented it from reconnecting to the cellular network after connection loss. It did not occur during testing before deployment as the test site had a very good GSM connectivity. Similarly, a bug in the flash driver of Deluge [HC04] caused a three-day network-outage during a deployment on an active volcano in Ecuador [WALJ<sup>+</sup>06]: Due to this bug, rebooting after remote reprogramming failed, breaking the network for three days until each node was manually reprogrammed on the volcano. Overall, examples of bugs detected during deployments [BISV08, LBV06, TPS<sup>+</sup>05, WALJ<sup>+</sup>06] indicate that bugs are often revealed in corner cases, that were not tested sufficiently before deployment.

Motivated by the observed and widely accepted necessity of rigorous pre-deployment testing, this thesis strives to develop efficient and effective testing methods for unmodified communication software. In the following section, we begin with the following question: Which properties of distributed systems make the testing process so difficult?

## 1.1 Challenges in Distributed Systems Testing

In contrast to programs that take an initial input and calculate the result solely based on that input, distributed systems are *reactive systems*. These systems run continuously and react on external events that are repeatedly received from the surrounding environment. Together with the distributed nature, distributed system implementations have the following main properties:

**Distributed Execution:** All asynchronous execution entities, i.e., programs in a network, may run concurrently. The distributed execution is driven and coordinated by both user input and communication among the nodes in a network.

**Unreliable Operating Environments:** The data communication medium is unreliable, being source of a number of non-deterministic failures such as data loss and corruption. Additionally, the hardware of a program instance is unreliable as well. Consequently, nodes in a network can experience reboots and complete outages.

**Heterogeneous Implementations:** The prevalent distributed systems are implemented in a variety of programming languages and run on different platforms/*Operating Systems (OSs)*. Hence, the ever-growing number of complexity of heterogeneous devices and networks puts high demands on interoperability.

From the testing point of view, the execution of a distributed system is triggered by numerous inputs that are categorized in Table 1.1. The boundaries between the selected categories are not strict since they may influence each other (e.g., time uncertainty and network inputs).

Initially, each software instance in a network is set up separately, potentially leading to diverging settings and thereof resulting behavior. After bootstrapping, the distributed execution is driven, for example, by preconfigured applications (e.g., data

Input class	Short Description
Settings	Settings of each software instance in a network
<b>User inputs</b>	Events with associated data, triggered by users
<b>Network inputs</b>	Data packets from peers in a network
<b>Environment</b>	OS, hardware, and environmental events
Time uncertainty	The main source of different temporal behavior

**Table 1.1** An overview of the main influence factors of a distributed system’s execution. The highlighted input classes encompass the main focus of this thesis.

collection from the environment) or active user participation (e.g., content request). The coordination among the nodes boils down to the exchange and processing of data packets that actively steer the control flow of applications (e.g., protocol state machine transitions). At the same time, non-deterministic environmental events such as hardware failures or packet loss can occur at any time and drive the distributed system execution into further, potentially unforeseen corner cases. Finally, time uncertainty with respect to all latter described events and their associated inputs leads to a potentially infinite number of event interleavings. Handling this input complexity for real-world, complex software implementations remains to be an open research and engineering problem.

In this thesis, we focus on user/network inputs and non-deterministic environmental events to rigorously analyze implementations of distributed systems. At the same time, we address the challenge of automated interoperability testing with the goal of specification-compliant and interoperable implementations. After defining the core challenges, we will now elaborate on the selected problem space in more detail.

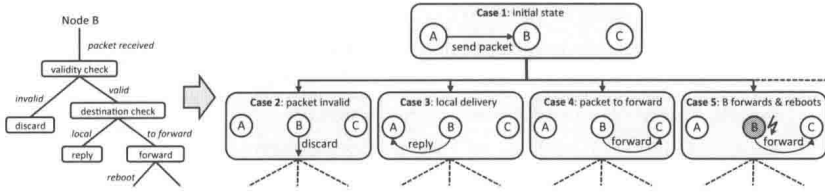
## 1.2 Problem Statement

Our considered problem scope addresses the challenge of efficient test case generation for unmodified distributed systems running arbitrary implementations. From this challenge, we derive the following three problem statements.

**Lack of High Input Coverage:** Complex distributed execution, node and network level non-determinism make prevalent testing techniques suffer from poor execution path coverage. For example, simulation and testbeds analyze merely single execution runs or slight modifications of them. Hereby, achieving high data input coverage is labor intensive, resulting in often insufficient manual testing. The latter challenge is crucial for corner case analysis since distributed systems’ execution is primarily data-flow driven.

**State Explosion:** Naively exercising the whole input space for typical distributed systems inevitably leads to exponential state growth and is not feasible. For example, exploring each value of a 20 byte long network packet would require  $2^{160}$  different runs. Therefore, more sophisticated methods are necessary to





**Figure 1.1** The key idea of our testing approach. The initial execution path ( $A$  broadcasts a packet to  $B$ ) splits after packet reception at node  $B$  into multiple execution paths, four of which are shown here. Following these execution paths, SDE provides a high program coverage including non-deterministic failures such as reboots and packet losses.

consider only those inputs that lead to different system behavior. Most important, any proposed solution should scale with growing network sizes and be applicable to real-world, complex communication software.

**Interoperability Issues:** Interoperability testing of distributed software such as communication protocols involves different, even closed-sourced implementations. Ideally, this process should cope with a large input space, non-determinism, concurrency, and heterogeneous operating environments to effectively explore the diverging execution paths. In practice, however, the missing tool support imposes a significant amount of manual effort to perform integrated conformance and interoperability testing of protocol implementations. These manual integration tests usually come late and may reveal bugs and misinterpretations of the protocol specification [SKJW12].

The presented problems raise a number of research and engineering challenges. In the following section, we first highlight the key ideas of our approach, called *Symbolic Distributed Execution (SDE)*, using a simple example scenario. Second, we define the main research questions of this thesis.

### 1.3 Basic Concept and Research Challenges

Consider a scenario with three communicating nodes, successively placed such that each of them is in the transmission range of its one-hop neighbors (Case 1 in Figure 1.1). Assume that node  $A$  initiates the communication by broadcasting a data packet to its neighbor  $B$ . Upon receiving the packet, node  $B$  first determines the validity of the packet, i.e., it calculates the header checksum. As the packet can contain arbitrary data, the validation branches into two execution paths, namely “*packet invalid*” and “*packet valid*”. Hence, SDE follows both program execution paths at node  $B$  separately. In case of an invalid packet,  $B$  discards it (see Case 2 in Figure 1.1). While manually testing the correct handling of arbitrary program input is time-consuming and challenging, our approach explores such distributed execution paths automatically.

In the case of a valid packet,  $B$  next checks the destination of the packet. Here,  $B$  again splits the execution path to “*packet to forward*” and “*local delivery*”. Case 3 in