Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 81
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 81
Espoo 2003 HUT-TCS-A81

EFFICIENT COMPUTER-AIDED VERIFICATION OF PARALLEL AND DISTRIBUTED SOFTWARE SYSTEMS

Marko Mäkelä

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 81
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 81
Espoo 2003 HUT-TCS-A81

EFFICIENT COMPUTER-AIDED VERIFICATION OF PARALLEL AND DISTRIBUTED SOFTWARE SYSTEMS

Marko Mäkelä

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering, for public examination and debate in Auditorium E at Helsinki University of Technology (Espoo, Finland) on the 28th of November, 2003, at 12 o'clock noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

© Marko Mäkelä

ISBN 951-22-6791-8
ISBN 951-22-6792-6 (PDF)
ISSN 1457-7615

Otamedia Oy
Espoo 2003

ABSTRACT: The society is becoming increasingly dependent on applications of distributed software systems, such as controller systems and wireless telecommunications. It is very difficult to guarantee the correct operation of this kind of systems with traditional software quality assurance methods, such as code reviews and testing. Formal methods, which are based on mathematical theories, have been suggested as a solution. Unfortunately, the vast complexity of the systems and the lack of competent personnel have prevented the adoption of sophisticated methods, such as theorem proving.

Computerised tools for verifying finite state asynchronous systems exist, and they been successful on locating errors in relatively small software systems. However, a direct translation of software to low-level formal models may lead to unmanageably large models or complex behaviour. Abstract models and algorithms that operate on compact high-level designs are needed to analyse larger systems.

This work introduces modelling formalisms and verification methods of distributed systems, presents efficient algorithms for verifying high-level models of large software systems, including an automated method for abstracting unneeded details from systems consisting of loosely connected components, and shows how the methods can be applied in the software development industry.

KEYWORDS: distributed systems, software systems, model checking, verification, reachability analysis

Contents

PREFACE

This dissertation is the result of studies and research at the Laboratory for Theoretical Computer Science of Helsinki University of Technology from 1998 to 2003. I am grateful to my supervisor Doc. Nisse Husberg, to Prof. Emeritus Leo Ojala and to Prof. Ilkka Niemelä, the current head of the laboratory, for providing me with an opportunity to work at the laboratory, and to my colleagues, especially Dr. Kimmo Varpaaniemi and Dr. Keijo Heljanko, for valuable feedback and discussions.

The work was funded by the Helsinki Graduate School on Computer Science and Engineering, the National Technology Agency of Finland (TEKES), the Nokia Corporation, Elisa Communications, the Finnish Rail Administration, EKE Electronics, Genera, the Academy of Finland (Project 47754) and by personal grants from Tekniikan Edistämissäätiö and Jenny and Antti Wihuri Fund. Without the funding and grants of these organisations, full-time studies and international visits would not have been possible.

Last but not least, I would like to thank my wife Heli and my friends and family for support and encouragement.

Otaniemi, 24th of October, 2003

Marko Mäkelä

List of Publications

[**P1**] M. Mäkelä. Optimising enabling tests and unfoldings of algebraic system nets. In José-Manuel Colom and Maciej Koutny, editors, *Application and Theory of Petri Nets 2001, 22nd International Conference, ICATPN 2001*, Newcastle upon Tyne, England, June 2001, *Lecture Notes in Computer Science* 2075, pages 283–302, Springer-Verlag, Berlin, Germany.

[**P2**] M. Mäkelä. MARIA: Modular reachability analyser for algebraic system nets. In Javier Esparza and Charles Lakos, editors, *Application and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002*, Adelaide, Australia, June 2002, *Lecture Notes in Computer Science* 2360, pages 434–444, Springer-Verlag, Berlin, Germany.

[**P3**] M. Mäkelä. Efficiently verifying safety properties with idle office computers. In Charles Lakos, Robert Esser, Lars M. Kristensen and Jonathan Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, Adelaide, Australia, June 2002, *Conferences in Research and Practice in Information Technology* 12, pages 11–16, Australian Computer Society Inc.

[**P4**] M. Mäkelä. Model checking safety properties in modular high-level nets. In Wil M.P. van der Aalst and Eike Best, editors, *Application and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003*, Eindhoven, The Netherlands, June 2003, *Lecture Notes in Computer Science* 2679, pages 201–220, Springer-Verlag, Berlin, Germany.

[**P5**] J. Järvenpää and M. Mäkelä. Towards automated checking of component-oriented enterprise applications. In Daniel Moldt, editor, *Second Workshop on Modelling of Objects, Components and Agents*, DAIMI report PB-561, pages 67–85. University of Århus, Denmark, August 2002.

The dissertation consists of the five publications listed above, and a dissertation summary. Publications [**P1**]–[**P4**] cover state space analysis methods, and they have been solely written by the current author.

The publication [**P5**] is joint work with Jukka Järvenpää, who came up with the idea of applying state space analysis to component-based software and business logic. The formal modelling principles and the experiment presented in [**P5**] were written by the current author, while Mr. Järvenpää covered more practical aspects, such as the prototype front-end for the MARIA tool, to be used in simulated application maintenance work.

1 INTRODUCTION

Until late 1980's, home and office computers used to be isolated systems that performed one task at a time. It was fairly easy to avoid and to correct programming errors, since the software was programmed to interact with at most one entity at a time—if the printer or the disk drive was slow, the user would have to wait for his turn.

Today, microprocessors or microcontrollers are replacing mechanical or electric control systems, and they have enabled entirely new application areas, such as mobile telecommunications. The computing nodes are networked with each other, and the software they execute responds to events arriving from independent sources, such as sensors, user interfaces, or other computing nodes.

Clearly, an asynchronously operating distributed system is more difficult to manage than an isolated appliance that performs one task at a time. On the other hand, a fault in an isolated system has negligible consequences when compared to a design flaw that may lock up a telecommunication network, allow trains to collide in a railway safety system, or discriminate some users in a congested situation.

Ensuring the correct operation of distributed software systems is a challenge. Due to the nondeterministic nature of asynchronous systems, it may be hard to reproduce faults. A timing-related error might not show up at all when the software has been compiled with support for debugging enabled, or when it is executed in a debugger. *Computer aided verification* has been suggested as a supplementary method to testing and code reviews. The idea behind it is to utilise the increasing computing capacity to ease the design of complex systems.

One of the most promising computer aided verification methods is *model checking* [33]. The basic idea is to construct a graph that contains the reachable states of the system (or a model of it) as its nodes and the state transitions as its edges. This graph is known as the *Kripke structure* of the system, if its nodes are labelled with *atomic propositions* that characterise the state of the system. The property we would like to verify can be specified in temporal logic [33, Chapter 3]. A model checking algorithm can determine whether the Kripke structure is a model of the logic formula, i.e., whether the system meets its specification, and reproduce execution traces that demonstrate detected violations.

Two decades ago, when model checking tools could handle up to tens of thousands of states, the method was considered impractical by some researchers [159]. Even though computers have evolved since then, there still is a practical problem: almost any system of interest has a huge number of reachable states. If the size of the system is characterised with a parameter, such as the number of processes or the capacity of message channels, the number of reachable states tends to grow exponentially in this parameter [150, Section 1]. *Symbolic model checking* [117] tries to solve the problem by representing sets of states with formulae. It has been successful in the verification of hardware designs, which use simple data types and operations. Alas, it is less likely to achieve similar reductions in software specifications whose logical structures are less regular [25].

In *explicit state space enumeration*, each node in the state graph is visited (and often stored) separately. This can require huge amounts of memory and processing time. The state explosion problem can be countered by applying *reduction methods*, which ignore some states that do not affect the property being checked, or by simplifying the model.

Abstract (simplified) models can be constructed at early stages of development, allowing principal design problems to be detected and corrected before any implementation work is started. Provided that the model reflects the software architecture and the modelling formalism supports the concepts of the implementation language, it is possible to keep the model up to date, in order to verify new features added to the implementation. However, it can be expensive and error-prone to maintain the model by hand. If the software architecture facilitates an automated translation from code to abstract models, model checking can reveal an order of magnitude more errors than testing [75].

Unfortunately, model checking is not yet a widespread technique in software engineering. This could be attributed to the lack of efficient computer tool support. Tools designed for interactive use may have problems with large automatically generated models due to inefficient use of memory. On the other hand, heavy-duty model checking tools may lack a simulator, which can be frustrating to someone who would like to experiment with a model without having any specific verification question in mind. Also, the formalism supported by the tool could be inefficient in certain tasks. For instance, modelling point-to-multipoint communications in a language based on processes communicating over point-to-point links can generate unnecessary overhead both in the model and its state space. A generic high-level formalism, such as the algebraic system nets [98] implemented in the MARIA tool [**P2**], allows a broad range of systems to be modelled in a compact and comprehensible way.

This research focuses on efficient techniques for modelling and model checking parallel and distributed high-level software systems.

1.1 Interfaces and Abstraction in Software Systems

The complexity of software systems can be managed by partitioning designs into *layers* containing replaceable *modules* that provide *services.* A classic example is the reference model for open systems interconnection [84], the "protocol stack" where each layer provides refined service to upper layers by making use of services provided by lower layers. The lowest layer is the physical communication medium, and the highest layer is the distributed application that provides service to real-world users.

Layers are inherent in many operating systems, programming languages and programming frameworks. Programming frameworks implement high-level interfaces to the system resources. The resources may be managed in pools in order to optimise performance or to provide automatic deallocation, so that the application code can be kept simpler.

The layers and service interfaces provide a nice form of *abstraction.* The same application program module can be executed in any environment that provides the required interfaces. When a system has been built of suitably isolated parts, one can reason about its modules without knowing the complete system in detail.

For instance, the designer of an Internet-based "shopping cart" application can assume that all underlying hardware and software components work properly. The components could include the TCP/IP stack [81] implemented in the participating devices, a database management system, and library routines for authenticating users and tracking sessions. To ensure the correct operation of the whole system, the designer can focus on the high-level application logic, as discussed in [**P5**].

At execution time, distributed or parallel software systems are partitioned into execution contexts called *processes* or *threads*, each of which represents a single flow of control. The processes can communicate with each other by passing messages or accessing shared memory areas. Depending on the application, the processes can perform a large number of internal computation steps between acts of communication. Distinguishing local actions from synchronising or communicating ones can lead to exponential savings [**P4**].

1.2 Objective and Methods

The aim of the research was to develop efficient computerised tools and methods for modelling and verifying parallel and distributed high-level software systems. Two places for improvement were identified:

1. The modelling formalism of a model checking tool should support the high-level concepts and operations of programming languages to avoid complicated or inefficient translations that are tedious and error-prone to carry out by hand or to implement in a compiler.

2. Semi-automated means of deriving abstractions could enable the use of model checking tools on more complex systems, even if little or no model checking expertise is available.

The workload of model checking experts can be minimised by developing translators from engineering documents and program code to verification models and formal requirements. The closer the modelling formalism is to the programming language, the simpler the translator can be made. In the extreme case, no translator is needed—the software can be verified by executing it in a special environment, as in [23].

There are multiple levels of abstractions. High-level programming languages hide many details, such as memory management, processor register allocation and instruction scheduling. Abstract data types hide the details of data structures. If the modelling tool directly supports the abstract concepts, systems that make use of them can be modelled more compactly and explored more efficiently than with a lower-level tool.

On a higher level, abstraction is provided by layered or modular software architectures. If there exist abstract descriptions of the components, the system can be inspected one component at a time, replacing the rest with abstract models. If not, modular state space exploration can create abstractions "on-the-fly" by exploring the components in isolation and considering only those global states where interactions occur.

A constructive research method was applied. The algorithmic ideas for efficient state space enumeration of high-level models were implemented in the tool MARIA, which was designed from the scratch by the current author. The ideas were presented and implemented for high-level Petri nets, but with the exception of [**P1**], it is straightforward to adapt them to other formalisms.

1.3 Contributions

This dissertation presents algorithms and methods that make the software reliability verification process more efficient. Publications [**P1**, **P3**] describe efficient algorithms for exploring large state spaces of high-level models. Publication [**P2**] shows how the usability of a model checking tool can be improved by combining interpreter and compiler techniques. Publications [**P4**, **P5**] present two methods for alleviating the state space explosion [150] of modular or component-based systems.

The main contributions of each of the publications are the following:

[**P1**] One of the most time-consuming tasks of state space enumeration tools for high-level Petri nets, determining the assignments under which transitions are enabled, is viewed as a unification problem. The presented algorithm, boosted with a pre-processing step for reordering the input, has been optimised for computing successor states while constructing the assignments. A variation of the algorithm is demonstrated to produce significantly smaller unfolded place/transition systems for certain instances of high-level nets than the canonical method [98, Section 4.1].

[**P2**] A freely available, extensible state space enumeration tool for a class of high-level Petri nets is presented. Unlike [10, 155], the tool has been designed for checking automatically translated models of software written in high-level languages.[1] The interpreter-driven mode of operation makes it easy to simulate large models, while compiled code speeds up exhaustive analysis tasks.

[**P3**] The distributed version of the state space enumeration algorithm in MARIA [**P2**] is presented. The algorithm is not designed for a dedicated computing cluster like [21, 69, 76, 82, 115, 124, 146], nor is it tightly coupled with any formalism like [26, 124, 137], but it is optimised for utilising the idle processing time of ordinary office computers. The central server simplifies management, allows dynamic load balancing and reduces the memory consumption at the computing nodes. Experiments made with up to twelve clients show over 90 % processor utilisation, depending on the average branching degree of the model, and the utilisation factor has been demonstrated to improve when state space reduction methods are applied.

[1]Translators for PROD [155] have problems with data types [78] or restrict the source language [17, Section 3]. The developers of a translator from SDL [89] into Design/CPN [119] suggest the development of a new model checker as future work [30, Section 8].

[**P4**] A simple algorithm is presented that checks safety properties in systems consisting of loosely coupled modules. Instead of generating a single state space of all possible interleaved executions of the modules, the algorithm constructs local state spaces for each module and only includes the synchronisation actions in the combined state space, which can greatly reduce the processing power requirements. The work is based on an earlier proposal [28], but the presented algorithm supports nested modules, uses simpler data structures and has been implemented—in a way compatible with [**P3**].

[**P5**] A method of modelling and model checking data-centric enterprise applications built on component-based frameworks is presented. The systems are formalised by combining automatically derived models of the high-level application logic with previously stored models of component behaviour. Related work has been reported on telephone switches [75, 76] and device drivers [112].

The Structure of the Dissertation The dissertation consists of five publications and this dissertation summary, structured as follows.

Section 2 is an introduction to verification. It compares the available formalisms and techniques for verifying the correct operation of parallel and distributed systems. Section 3 presents state space enumeration, the general-purpose technique for model checking high-level systems, and describes options that allow more complex systems to be explored. It provides the background information needed for understanding the three algorithms [**P1**, **P3**, **P4**] that have been implemented in the MARIA tool [**P2**]. Section 4 describes the workflow of verification and estimates the degree of automation that can be achieved with the use of different tools and methods. Finally, the publications are summarised in Section 5 and our conclusions and outline for future development are presented in Section 6.

2 VERIFICATION

Jörg Desel [43] defines three terms for determining the quality of a system, or a model thereof: *validation*, *verification* and *evaluation*. Validation answers the question "was the right system built?" by determining whether the system fulfills its intended purpose. Verification answers the question "was the system built right?" by proving that the system matches its specification. Evaluation captures other aspects of the system, such as usability and end user acceptance, maybe performance as well.

This section concentrates on verification, determining whether a system matches its requirements. Furthermore, it concentrates on nondeterministic systems, which are difficult to inspect by testing, as variations in timing can make them behave differently on the same input. To automate verification, both the system and the requirements must be formalised. Logic seems a natural choice for formulating requirements. For modelling computing systems, there are many *formalisms*, which differ in their preferred *level of abstraction*.

In *top-down* design methodologies, the system is first described at an abstract level. The abstract description is divided into components or modules that are gradually refined towards concrete implementations. Kurshan [102, Section 8.6] discusses the concept of *stepwise refinement*:

> [T]here is a circle of thinking which supports the view that one may start with a high-level (abstract) specification, refine it in a linear fashion down to a model which may be physically implemented (adding more detail at each step) and be done. It is thought that if each model is consistent with its more abstract specification (in the refinement succession), then one knows that the lowest-level model is true to each of its specifications.

High-level (abstract) models can also be thought as "rapid prototypes" that permit early "debugging." The top-down methodology is well suited for proof systems, such as the Temporal Logic of Actions [105]. According to Lamport [105, Section 9.5.1], the refinement steps could even be automated to some degree:

> Derivation of a program by a rigorous procedure that guarantees its correctness is preferable to post hoc verification. [...] Any method for proving that one program implements another can be used as the basis for program derivation. [...] Unfortunately, we know of no concurrent algorithm used in a real system that was systematically derived, not simply justified by post hoc verification.

In *bottom-up* design methodologies, the "low level" entities are defined first and used as building blocks of more and more complex entities. The "low level" entities can be readily available as the primitives of a programming language or the services of the operating system or existing software packages. They can also be constructed by the designer.

Many programmers have a pragmatic approach to problems. They construct small prototypes and experiment with them in order to become familiar with the problem at hand. The question is whether the produced code meets its requirements. In the ideal world, there would exist formal requirements and descriptions of each module. In practice, no detailed requirements might be available, and everyone working on the system could have a slightly different view. However, every system should fulfill some generic safety properties, such as the absence of deadlocks, failed assertions, arithmetic errors, data integrity violations or resource leakages. Such properties can be verified by model checking or state space enumeration, which is the topic of Section 3.

2.1 Formulating and Checking the Desired Properties

Dwyer et al. [49] motivate *specification patterns*—a high-level front-end to temporal logics—with the following example:

> For example, consider the following requirement for an elevator: *Between the time an elevator is **call**ed at a floor and the time it **open**s its door at that floor, the elevator can arrive **at** that **floor** at most twice.* To verify this property with a linear temporal logic (LTL) model checker, a developer would have to translate this informal requirement into the following LTL formula:
>
> $$\begin{aligned}&\Box((\mathbf{call} \wedge \Diamond\mathbf{open}) \rightarrow \\ &\quad((\neg\mathbf{atfloor} \wedge \neg\mathbf{open})\ \mathsf{U} \\ &\quad\quad(\mathbf{open} \vee ((\mathbf{atfloor} \wedge \neg\mathbf{open})\ \mathsf{U} \\ &\quad\quad\quad(\mathbf{open} \vee ((\neg\mathbf{atfloor} \wedge \neg\mathbf{open})\ \mathsf{U} \\ &\quad\quad\quad\quad(\mathbf{open} \vee ((\mathbf{atfloor} \wedge \neg\mathbf{open})\ \mathsf{U} \\ &\quad\quad\quad\quad\quad(\mathbf{open} \vee (\neg\mathbf{atfloor}\ \mathsf{U}\ \mathbf{open})))))))))\end{aligned}$$

In the automata-theoretic approach to verification, a desired property of a system is negated and translated into an automaton. Figure 1 illustrates such an automaton for our property. The leftmost state is the only initial state, and the rightmost one is an accepting, or a "bad" state. Also the state space of the model is interpreted as an automaton. A synchronised product automaton of the two automata is constructed. The system is in error if a "bad" state is reachable in the product. State space analysis tools look for such states and produce *error traces*—executions from an initial state of the two automata that lead to an error.

The automaton in Figure 1 contains only one "bad" state. The shortest path to this state is via the edge labelled $\mathbf{call} \wedge \neg\mathbf{open} \wedge \mathbf{atfloor}$ followed

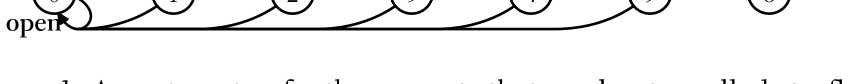


Figure 1: An automaton for the property that an elevator called at a floor may arrive at the floor at most twice without opening its door.

by horizontal edges. Informally, the first step—from state 0 to state 2 of the automaton—corresponds to the case where the elevator has already arrived at the floor when it is called. Then, keeping its door shut, the elevator leaves and arrives twice. After that, in state 6 of the property automaton, it does not matter how many more times the elevator might arrive. An error is signalled once the elevator door finally opens.

Does an elevator whose door never opens fulfill this property? Intuitively, one could believe "no," but the "bad" state in Figure 1 cannot be reached without the door being open at some point. This example shows that the desired properties have to be formulated carefully, and even the obvious requirements must be stated explicitly—computers are dumb.

We believe that a substantial amount of errors can be captured with assertions or invariants, stating that some condition must hold in each possible state of the system or whenever a given program statement is about to be executed. Such properties can be directly checked in the state space of the model—without computing a product automaton—since the property automaton has only one non-accepting, or "good" state.

Many properties can be automatically derived from engineering documents, such as the referential integrity rules for databases, or assertion statements in source code. Assertion statements of the form **assert**(q) can be formulated in linear temporal logic as $\Box(p \to q)$, where p indicates the point of execution. In Maria, these can be translated into **reject** statements or intentional evaluation errors [**P3**, Section 2.1].

Havelund and Skakkebæk [66, Section 2.1] and Holzmann [75, Section 6] propose temporal assertion statements, such as **assert_eventually**(q) for $\Box(p \to \Diamond q)$. As opposed to *safety* properties, these *liveness* properties require something to *eventually* happen, without setting any fixed bound. An automaton on infinite sequences is needed for expressing liveness. The liveness model checker [106, 107] in our toolset has been built on top of the safety model checker [**P3**].

2.2 Modelling Formalisms

Before a system can be verified, a model of it must be constructed using some formalism. Figure 2 shows the relations between some families of modelling formalisms. State spaces consist of the reachable states of a system and transitions leading from a state to another. They can be presented with transition systems or process algebra. Higher-level models can be expanded to transition systems, and high-level Petri nets can be unfolded to place/transition nets. There exist translations between some guarded command languages and Petri nets. There is some variation in each family. For instance, transition systems can be defined to include

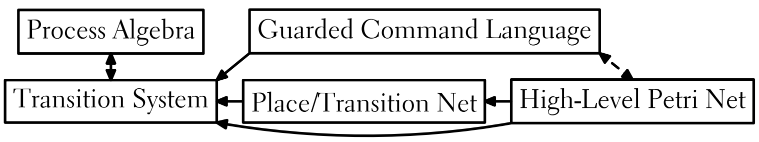


Figure 2: Some language families for describing computations.

state propositions [151], or the emphasis may be on the transitions or actions, as it often is in process algebra and in compositional modelling.

In software engineering, one use of models is to convey information in a more structured format than natural language, as in UML class diagrams [125, Part 5] or sequence diagrams [125, Part 7]. Often such models do not specify all details, or they lack exact semantics. For verification purposes, a computer must be able to interpret all of the model.

Timing can play a significant role especially in real-time systems. In untimed formalisms, the actions or transitions are not assigned any durations or timing constraints. Only the order in which events occur is observable—nothing exact can be deduced about the time. In timed formalisms, enabled transitions may be forced to occur within a predetermined time, and they may be assigned a probability for occurring.

This section presents some of the modelling formalisms that are supported by computerised verification tools. Emphasis is on untimed formalisms, since certain types of timed transitions can be simulated by quantising the time in discrete ticks, as in [20].

Guarded Command Languages

Guarded commands play a central role in Dijkstra's mini-language [45, Chapter 4]. A guarded command is defined as a sequence of statements that may be executed if a Boolean expression holds. The predicates of programs expressed in the mini-language are solely transformed by assignment statements. The only other statements are the repetition (**do**...**od**) and the choice (**if**...**fi**) of guarded command sets.

Dijkstra's original work is directed towards the development of sequential programs. Extensions for describing parallel systems include **parbegin**...**parend** blocks [44] and message passing [71].

One of the best known model checkers for communication protocols, Spin [74] of Bell Labs, is based on a process oriented guarded command language where processes may communicate via message channels.

Dijkstra's notation has also influenced Murφ [46], SMV [117] and the Basic Petri Net Programming Notation of PEP [13, Section 3.1].

Process Algebra

In the Calculus of Communicating Systems [120, 121] (CCS), the processes may have *internal actions* denoted by the τ symbol, and a pair of processes can engage in a synchronisation event where one process "sends" and the other "receives" a symbol, e.g., α and $\bar{\alpha}$. The Algebra of Communicating Shared Resources [32] extends CCS by allowing prioritised actions that may consume resources and time.

In 1978, Hoare introduced the Communicating Sequential Processes (CSP) using a guarded command language notation [71] inspired by Dijkstra [45, Chapter 4]. In this definition, processes communicated in a similar fashion to CCS, by sending and receiving messages via synchronised point-to-point channels. In 1985, Hoare defined CSP in the form it is known today in his famous book [72]. In [72, Section 7.4.1], he compares CCS and CSP. The major semantic difference is that CSP allows any number of processes to take part in a synchronisation.

The term "process algebra" does not only refer to algebraic notation for transition systems, but also to a method of verifying concurrent systems. Process algebra offers an alternative to model checking (Section 2.1): The model of the system can be shown to be equivalent with a simpler model where the property obviously holds.

Transition Systems

Transition systems are perhaps the simplest way of describing the behaviour of a computing system. As in Figure 1, they can be viewed as directed graphs of the possible state transitions of a system. They can also be represented textually in some process algebraic notation.

Transition system tools include CÆSAR/ALDÉBARAN [52] and the recently published Tampere Verification Tool [151] (TVT). User-definable rules for parallel composition allow TVT to serve as a verification backend for many process algebras, including LOTOS [85], which is also supported by CÆSAR/ALDÉBARAN. LOTOS features data types, variables and parameterised actions, and various synchronisations.

Transition systems are a simple formalism. To model complex behaviour, it may be practical to construct a higher-level model and translate the states spaces of its components into transition systems. Transition systems can be generated from Petri net models with MARIA [**P2**], and from other high-level specifications, such as an SDL [89] specification of a communication protocol [110].

Transition systems can be reduced or compared with respect to various equivalence relations, such as strong bisimulation [121, Definition 7.1.1]. This is useful when the system is modelled as a parallel composition of components. The composition can be made smaller if it is constructed incrementally, hiding actions and reducing the resulting intermediate systems whenever possible. Kaivola [93] presents efficient reduction and comparison algorithms for the NDFD equivalence relation [93, Definition 4.2.7], which preserves all properties expressed in the nexttime-less linear temporal logic [93, Section 3.2].

Transition systems can also be viewed as a simple formalism for describing state space reduction techniques. It is not necessary to explicitly construct transition systems in order to apply the techniques. Instead, the techniques can be directly implemented in state space enumeration tools, like the partial order reduction methods in SPIN [74, Section 3.3] and PROD [155] and the modular state space exploration in MARIA [**P4**].

Petri Nets

Computing systems consisting of multiple processing units usually communicate via shared variables (common storage that can be read or written by multiple processing units) or by exchanging messages over point-to-point or point-to-multipoint communication channels. In transition systems and process algebra, modelling shared storage easily requires large numbers of actions. Guarded command languages have a provision for shared data, but their message primitives are usually limited to point-to-point channels. It would be nice to have a more generic formalism that supports both shared data and arbitrary synchronisations.

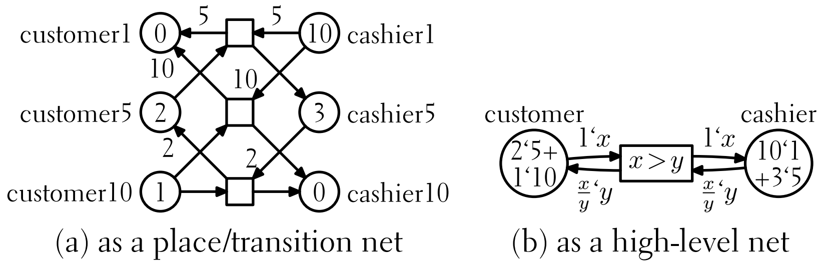


Figure 3: An algorithm for breaking money.

In 1962, the dissertation of Carl Adam Petri, *Kommunikation mit Automaten*, introduced a graphical formalism for modelling synchronisations and communications between automata. The draft standard [15] defines two classes of Petri nets: *place/transition nets* and *high-level nets*.

Place/transition nets [15, Clause 10.1] have *places* that contain a non-negative amount of *tokens*, which can be removed or added by the occurrences of *transitions*. A transition is connected to places via *input arcs* and *output arcs* whose *weights* indicate how many tokens the occurrence of the transition removes from the input places and adds to the output places. A transition is *enabled* (it may occur) if its each input place contains at least as many tokens as the weight of the arc indicates. The state (*marking*) is defined as the number of tokens in each place.

Let us consider a situation that could happen near a coin-operated machine. A customer comes to a cashier with a bank bill in his hand, asking "Could you break this for me?" The cashier then changes the money to an equivalent amount of money in smaller coins or bills. In Figure 3(a), the money owned by the two parties is represented by the marking of six places, denoted with circles. The marking is represented with numbers enclosed in the place symbols. In Figure 3(a), the customer has two coins or bills of value 5 and one of value 10.

The possible actions are modelled by transitions, drawn as rectangles. The input and output arcs are denoted by arrows leading to or from the transition symbols. Arcs are associated with positive integer weights. By convention, weights of 1 are omitted from diagrams. In our algorithm, the cashier always returns one type of money, e.g., ten small coins for one big bill, as in the transition in the center of Figure 3(a).

In high-level Petri nets [15, Clause 10.2], places and tokens are associated with data types (also known as *colours* or algebraic *sorts*). In other words, the tokens can represent data that is stored in the places. Figure 3(b) is a more compact version of Figure 3(a). The six places are *folded* to two places whose markings are *multi-sets* of coin or bill denominations. The rule for breaking money can now be specified with one transition that has two *variables*: x for the money offered by the customer and y for the change given by the cashier. These variables are constrained by the transition *guard* $x > y$. The available denominations are no longer hard-coded in the model, as in Figure 3(a).[2]

[2]To allow arbitrary positive integers as denominations, the guard must be refined so that x/y is an integer, since the cashier cannot return fractions of coins or bills.

High-level nets are better suited for detailed modelling of high-level software systems than place/transition nets, since the arithmetic operations in the software can be directly translated into arc inscriptions and transition guards. If the data domains of a high-level net are small enough, the net can be *unfolded* to a place/transition net of manageable size. Figure 3(a) depicts the unfolded counterpart of Figure 3(b).

Less expressive power often means more analysis power. Structural analysis makes it possible to prove certain types of properties for a large class of systems. Structural properties are much easier to derive for place/transition nets than for high-level nets. The structural properties of a place/transition net simply depend on its topological structure [123, Section VII].

The computer tools for analysing Petri net models range from research prototypes to commercial packages. PEP [13] and LoLA [136] operate on ordinary place/transition systems. They can analyse simple high-level net models via unfolding. Tools that operate directly on high-level nets include PROD [155] (which is based on predicate/transition nets [59]), Design/CPN [119] and CPN/Tools [10] (which support coloured nets [91]) and MARIA [**P2**] (for algebraic system nets [98]).

Originally, Petri nets were an untimed formalism. Extensions have been developed for performance modelling and analysis [123, Section IX]. Analysis tools include GreatSPN [27] for timed and stochastic Petri nets and SMART [31] for various stochastic modelling formalisms.

Petri nets may have a shortcoming when it comes to modelling complex software systems: they lack structuring capabilities. From the theoretical perspective, all transitions are equal, and the data and control flows are indistinguishable from each other. It may be easier to understand and maintain nets when they are divided into fairly isolated components or modules. In the Design/CPN editor, the net can be drawn on multiple pages, which are connected via shared places or substitution transitions—a special form of macro expansion. The textual modelling language of MARIA supports similar techniques. MARIA can preserve some structure during analysis, as it supports modular exploration of multiple nets that are synchronised via shared transitions [**P4**].

For modelling object-oriented software systems, more sophisticated structuring capabilities have been proposed. Proposed extensions include Object Petri Nets [103], Concurrent Object-Oriented Petri Nets [14] and Agent-Oriented Coloured Petri Nets [122] that have evolved into Reference Nets [101]. One reason why the tool support on these formalisms is directed toward modelling, simulation and executable code generation rather than verification is the large number of combinations in which identifiers can be assigned to resources or objects. Each object is assigned an abstract value that uniquely identifies it throughout its lifetime. If the pool of available identifier values is limited to n, at most n objects can coexist. For instance, the identifiers for objects representing the items of an n-element linked list can be chosen in $n! = n(n-1)\cdots 3 \cdot 2 \cdot 1$ ways from such a pool. That is, there are $n!$ equivalent combinations that are considered distinct. Techniques that address this problem are discussed in Section 3.2.

Logic

One of the most abstract modelling formalisms is logic. There, the possible states of a system are specified with a number of propositions or conditions. Verifying a property of the system is a matter of checking whether the formula stating the property can be derived from the formulae that describe the system by applying some proof rules.

Chandy and Misra [24] presented an abstract way of developing algorithms. Their Unity notation has no explicit control flow. Mapping Unity algorithms to hardware architectures is nontrivial, as the implementations of an abstract algorithm on shared or distributed memory can heavily differ from each other. Also Reisig [132] describes abstract distributed algorithms, using algebraic nets. Like Chandy and Misra, he presents a system for proving safety and a limited form of liveness.

The Z notation has recently become an international standard [87]. Several computerised type checkers and theorem proving tools are available. Z is related to B [3] and VDM [68, 88], which is closely related to Meta IV, a language for defining semantics. Extensions to Z include objects and CSP-style parallel composition [53].

Lamport's Temporal Logic of Actions [105] (TLA) is an untyped language based on linear temporal logic and a set of axioms and proof rules. The behaviour of the system is described with mathematical and logical equations—there is no concept of assignment. TLA is intended to be a tool for specifying concurrent algorithms at a very abstract level and proving that they satisfy the desired safety and liveness properties.

2.3 Specialised Verification Methods and Tools

There exist several methods and computerised tools for verifying formal specifications. Some are specialised in certain types of systems, while state space enumeration can be adapted for any kind of a finite system. The specialised techniques usually support a restricted set of built-in operations, and they can often manage very large or infinite state spaces.

If there exists a specialised technique for analysing a particular type of a system, it usually makes sense to use that technique. There are two possible problems. Sometimes, it may be difficult to translate the system description into the formalism used by the tool. If the tool makes use of computationally complex heuristic algorithms, it may exhaust the available computing resources even when checking a fairly simple model.

The remainder of this section presents some specialised verification methods and discusses their suitability for verifying software systems. Of these methods, symbolic model checking is closest to the generic technique of state space enumeration, which is covered in Section 3.

Theorem Proving

The strength of theorems is that they can assert that something works for a large or infinite number of parameter combinations. Proof systems are efficient for verifying systems consisting of little control logic, such as special-purpose arithmetic units, as in [16, Chapter 3]. Theorem proving also suits well for abstract specifications, like those written in DisCo [90], a specification language for reactive systems that is based on TLA [105].

If the system uses complex data structures or if its control logic consists of many computing steps—as the case often is with software—the logic formulae may become too complex for any theorem proving system. Due to the heuristic nature of theorem proving algorithms, most theorem provers require quite a bit of manual intervention. A small modification to the model can confuse the heuristics and lead the algorithm to an infinite loop.

If the property being verified does not hold, the theorem prover might not be able to clearly show what is going wrong. The problem can be diagnosed or the proof validated with an interactive animator or simulator, as in the DisCo tool set [96, Section 4], or with a model checker, as in the Symbolic Analysis Laboratory [12] (SAL).

Both DisCo and SAL delegate theorem proving to the Prototype Verification System [126] (PVS). Another theorem prover for typed higher-order logic (and the Z notation) is ProofPower [5], which is based on the extensible language Standard ML. Maude, a framework for building theorem provers, is a tool for rewriting logic with user-defined strategies. It can be used as a model checker or simulator [34, Section 8.1].

Rewriting Systems and Equivalence Tests

Most tools for process algebra or transition systems include some kind of a rewriting system for reducing or comparing models with respect to some equivalence relation that preserves the property being verified. VERSA [32] rewrites process algebraic models and tests transition systems for equivalence.

When the system is designed in a top-down fashion, equivalence tests can be used for verifying each refinement step, proving that the system implements its specification. However, the refinement steps in software system development are rarely presented in formal notation.

Since process algebras are action-oriented, it may be difficult to efficiently translate data-intensive software systems into them. Simply instantiating distinct "read" and "write" actions for each possible value of each variable could make the model too large to be processed in any tool.

Structural Analysis

Certain properties of a system can be checked without considering its dynamic behaviour. For instance, modern compilers for strongly typed high-level programming languages ensure that the data types are used in a consistent way. They may also warn about variables or procedures that have been defined but are never used.

Advanced compilers even issue warnings for procedure-local variables that *may* be used before they are initialised. In the general case, the variable can be referred to in conditional blocks, and the conditions can be affected by the environment of the procedure. Thus, the structural analysis performed by the compiler cannot be affirmative in all cases.

Some properties or analysis algorithms may apply to certain classes of models. With structural analysis, a model can be determined to belong to a particular class. Then all properties proven for that class will trivially hold in the model, and there might be a more efficient analysis method. For instance, Murata [123, Section VI] summarises the liveness, safeness and reachability criteria for different topological classes of place/transition nets. He then demonstrates some "short-cuts" for one special case, marked graphs [123, Section VII].

Place/transition nets can also be classified by their matrix representations [123, Section VIII]. From matrix equations, it is possible to derive net *invariants* that hold for all possible initial states of the system. Transition invariants indicate possible loops of the system, and place invariants can be used for reasoning about the possible states of the system.

We believe that structural analysis augments other verification techniques. In high-level software systems, a relatively simple static analysis tool can locate simple programming errors more efficiently than a full-blown theorem prover or model checker. However, static analysis generally cannot prove or disprove arbitrary properties of a system.

Symbolic Model Checking

Symbolic model checking methods manipulate data structures that represent sets of reachable states of the model. The data structure can be interpreted as a characterising function that maps state variables to truth values. The set consists of those states that the function maps to "true."

The actions of the model being checked perform operations on state variables. Symbolic methods translate these operations into operations on the data structures representing the sets of states. Thus, the effect of an action in the model can be computed in all previously discovered states of the system simultaneously. Therefore, symbolic model checking can explore very large, even infinite state spaces in limited time.

However, it can be nontrivial to map the operations on single states to operations on sets of states. Therefore, many symbolic model checking tools are limited to rather low-level description formalisms.

Decision Diagrams. Boolean Decision Diagrams [18] (BDD), which represent Boolean functions as directed acyclic graphs, have been particularly successful in verifying hardware designs, where the data types and operations tend to be simpler and more regular than in software [25].

Software systems make use of integers, arrays, and other high-level data types. In order to apply BDDs on such systems, the data types have to be mapped into Booleans. An integer variable must be translated into multiple Boolean variables, and any operations on it, such as addition or multiplication, have to be expressed with Boolean operators. Such a translation could greatly increase the size of the model. Predicate abstraction [41] is more viable, since it describes the state of the system with predicates, which are modelled with Boolean variables.

Data Decision Diagrams [39] generalise BDDs to arbitrary data types. The operations on the data types are not hard-coded in the algorithm, as with BDDs [18, Section 4], but they have to be formalised as homomorphisms on DDDs [39, Section 1.3]. This method could prove efficient on software systems, if some essential data types and operations were formalised and packaged in an extensible tool.

Complete finite prefixes. An interesting type of symbolic model checking that resembles structural analysis is the unfolding of finite-state place/transition systems into complete finite prefixes that describe their dynamic behaviour [118]. A prefix can in some cases be exponentially smaller than the reachable state space of the system, which makes the prefixes interesting for verification purposes.

In order to apply net unfoldings on software systems modelled with high-level Petri nets, one would have to unfold the models into place/ transition nets. In practice, this is often infeasible due to the complexity of the used data types. Recently, the technique has been extended to high-level nets [97]. Even though the use of place/transition nets can be avoided, the high-level nets should be constructed carefully. For example, if "read" operations atomically shift the contents of FIFO buffers, the resulting prefixes may become very large due to the number of distinct events, one for each possible buffer content.

Lossy FIFO systems. A distributed system that consists of processes communicating via unbounded channels can have an infinite number of reachable states. However, it is possible to prove safety properties of such systems with a backward reachability analysis algorithm [2].

If there is no abstract definition of the processes, it can be tedious to construct the state machines by hand. Leppänen and Luukkainen [110] extracted the state machines from a formal model of a communication protocol by partitioning the model and performing state space enumeration on the partitions.

A similar treatment could be given to program code, by repeatedly executing the code of each process for a bounded number of steps, simulating "receive" actions with nondeterministic choice of possible actions, as in the VeriSoft tool [23]. However, this would generate finite trees of observed actions rather than nonterminating automata. Folding the trees into cyclic graphs may require a considerable degree of insight and manual work.

Bounded Model Checking. In bounded model checking, a non-terminating system is searched for counterexamples to a given property for a bounded number of execution steps. The technique can be implemented in symbolic model checking as well as explicit state space enumeration, and it can also be encoded as a propositional satisfiability problem, which can be solved by a tool like Prover [141]. Amla et al. [4] compare these three approaches and conclude that BDDs perform best in deep counterexamples, while satisfiability solvers and random state space exploration are the winners for small bounds.

3 STATE SPACE ENUMERATION

State space enumeration refers to techniques that incrementally construct the set of reachable states in a model by considering the effect of enabled actions in each reachable state separately. The basic technique can be applied to any computing system whose states can be stored, restored and compared.

Appendix B.1 contains the basic state space enumeration procedure. Its three most important parameters are the initial state s_0 of the system, the state transformation rules successors, and a function error(s) that identifies erroneous states. We do not specify the details of the procedure reportError. It could display diagnostic information and abort the exploration, either unconditionally or based on some criteria, such as the severity of the error or the number of errors reported so far.

The search algorithm makes use of two data structures: the set of states it has explored, and a collection of states that are waiting to be processed. It must be possible to insert items into the set and to check if the set contains a given item. The collection must support insertion and removal of items, and emptiness check. If the collection is a queue, then the search proceeds *breadth first*, guaranteeing that a shortest path to an error is found first.

State space enumeration is a general-purpose tool with one practical limitation: the model being analysed must have a limited number of states and enabled actions. Otherwise checking all the reachable states one by one will consume too much time or memory. This limitation can be alleviated by the use of different state space reduction techniques.

The rest of this section discusses the benefits of state space enumeration, compares state space reduction techniques, and presents ways to reduce the execution time of state space enumeration.

3.1 Benefits of State Space Enumeration

Easy implementation. Unlike the specialised methods discussed in Section 2.3, state space enumeration can treat the function for computing successor states as a "black box." Thus, there is no need to model the system in detail, provided that the nondeterminism in it can be controlled by the model checking testbed. For instance, to check whether a computerised player of a two-player board game can lose a game it starts, it is sufficient to have a program interface to the function that computes the next move on a given board. The initial state s_0 would be the empty board, and the successor relation can be defined as the next move of the computerised player combined with all possible moves of its opponent, which would be simulated by our verification harness. The function error would identify those boards where the opponent has won.

Counterexamples. When there are millions of reachable states, merely providing the user with an erroneous state can be as frustrating as reporting "there is an error somewhere in your system." A *counterexample trace* is a sequence of actions that leads from the initial state to the error.

Unlike in proof systems, it is straightforward to construct error traces with the state space enumeration algorithm. PROD [155] extends the algorithm presented in Appendix B.1 by constructing a *reachability graph* that records the reachable states S as its nodes and the successor relation as its edges. When an erroneous state is found, a path to it can be found by graph traversal.

Not all edges of the graph need to be recorded in order to recover error traces. In fact, it is sufficient to maintain a mapping from each encountered state s' to its ancestor s, from which s' was first generated by successors. By repetitively applying this mapping, the trace can be obtained from the error to the initial state s_0. In order to conserve memory, MARIA writes this mapping sequentially into a disk file [**P3**, Section 2.4], similar to Murφ.

3.2 Alleviating the State Space Explosion

Almost any system of interest has a huge number of reachable states. If the size of the system is characterised with a parameter, such as the number of processes or the capacity of message channels, the number of reachable states tends to grow exponentially in this parameter [150, Section 1]. However, given a simple verification question, it is possible to throw some information away by creating a more abstract model or by applying some reduction in the state exploration algorithm, thus reducing the number of states that need to be explored [150, Section 4.1].

Eliminating Redundant Data

One way of deriving abstract models is removing data and program code. Automated slicing techniques [147] can eliminate those parts that cannot affect the property being verified. A formal treatment of property preserving abstractions for process algebra and temporal logic has been given in [114]. The model can be reduced further if the behaviour of the system is over-approximated or under-approximated [66, Section 3.2]. However, errors found in an over-approximated model might not correspond to errors in the original system. Lazy abstraction [70], which applies predicate abstraction [41], starts with a very abstract model and refines it gradually to rule out errors that lack counterparts in the concrete model.

Models may contain redundant data. Kaivola [93] exploited the theory of data-independence [109] and showed that several properties of a communication protocol can be verified by considering only three distinct data packets that represent all packets circulating in the system. Valmari [150, Section 7.1] discusses the elimination of remnant variable values—that is, resetting variables whose current values will not be read. This technique has been implemented in SPIN [74].

During state space exploration, it may be possible to identify some reachable states as redundant and ignore them. For instance, seven of the eight states shown in Figure 4 are redundant. *Symmetry reduction* techniques [83, 142] are essential in model checking certain types of systems, such as software systems that dynamically allocate resources. Junttila describes some approximative algorithms that can be applied to systems

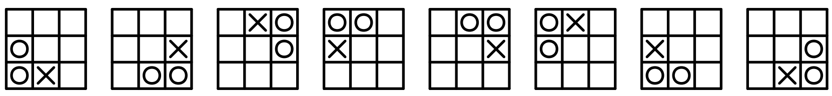
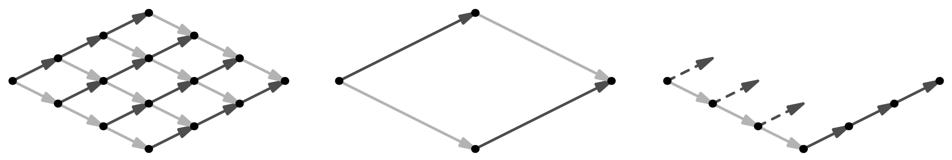


Figure 4: The cells of an $n \times n$ board can permuted in eight ways by rotation and reflection. The boards above look different but represent the same state. One of them could be chosen as a *canonical representative* of the group.



(a) interleaved actions (b) coarsened atomicity (c) dynamic reduction

Figure 5: Concurrent execution of k independent processes consisting of n sequential actions produces k^n possible states (a). Atomic processes produce 2^n states (b). Partial order reduction techniques reduce the number of choices in each state, and may thus explore only one path of kn states (c).

with a large number of dynamically allocated objects [92, Section 7].

Dealing with Independent Actions

A key contributor to the state space explosion is nondeterminism caused by concurrently enabled independent actions. Figure 5 shows an extreme case and illustrates two examples of *partial order reduction methods* [128], which were originally presented as an aid for proving properties of parallel programs [6, 113]. The **atomic** keyword of Spin [74] or a *resource token* [78] in Petri nets groups process-local actions into bigger indivisible steps. The result, in Figure 5(b), can be reduced further with *path compression* [**P3**, Section 4.1], which is a special case of a more generic method [95]. Both methods can be combined with partial order reduction, which ignores some enabled actions, as in Figure 5(c).

Compositional Verification

In *compositional reachability analysis* or *modular verification* [64, 160], the model is checked in multiple phases. It may be possible to transform the property being checked into something that can be checked on each component separately or on a composition of fewer components. Also, the full state space of the system can be composed incrementally, collapsing the sequences of internal actions in each intermediate composition.

When compositional verification is applied to labelled transition systems, as in [160] or [150, Section 7.3], the desired property is verified with respect to globally visible actions. For the purpose of verification, it may be necessary to make some internal actions visible and thus increase the size of the resulting composed transition systems. It is more efficient to decompose the automaton describing the property into something that can be directly synchronised with the components [25, Section 3.2].

One compositional technique that can be automated fairly easily is modular state space exploration, which is discussed in Section 5.4.

3.3 Reducing the Memory Usage

A major problem of state space enumeration is its potentially large memory usage. In typical workstations, mechanical storage (disk space) is some orders of magnitude bigger and several orders of magnitude slower to access than solid-state memory (register file, caches and main memory). Therefore, it is essential for performance to reduce the size of the data structures, so that they fit in the main memory, or to design algorithms that reduce access to external storage, as in [130].

Two data structures that dominate the memory usage of the state space enumeration algorithm in Appendix B.1: the set S of reachable states and the "work queue" Q of states that have not been explored. The queue ensures that all states will be included in the search, and the set prevents infinite loops.

Many memory reduction techniques aim to reduce the number of states inserted into S, taking the risk that parts of the state space may be explored several times. Valmari [150, Section 5.4] presents an extreme case—a recursive algorithm that does not construct a set of reachable states but enumerates the set in each recursion level. He concludes:

> [I]t is in theory possible to solve interesting verification tasks in relatively small memory. However, the known algorithms that do that consume unimaginable amounts of time.

Relaxing the Loop Check

In the tic-tac-toe game, pieces are inserted to the board until one of the players wins or the board becomes full. Thus, no board state can occur twice in a game, and the set S that prevents infinite loops in the state space enumeration algorithm (Appendix B.1) can be omitted to save memory space.

Unfortunately, models of parallel and distributed systems tend to produce cyclic state space graphs. Therefore, the loop check can only be omitted in special cases, or states that are known not to recur can "bypass" the set S.

State Space Caching. The idea of state space caching [60, Section 2] is to discard parts of the set S when the exploration algorithm is running out of memory. Loops can be prevented by organising the unexplored states Q as a stack and by replacing the test $s' \notin S$ with $s' \notin S \wedge s' \notin Q$. The drawback of this idea is that discarded states may be revisited if they can be reached via different execution paths. The situation can be ameliorated by applying partial order reductions and by adapting the cache deletion policy [60, Sections 4–5]. The compression of non-branching paths [**P3**, Section 4.1] bears some similarity with state space caching, but it also works in breadth-first search, producing short error traces.

The Sweep-Line Method. When the state space may contain cycles, some states must be stored in the set S, so that infinite loops can be avoided. However, S could be reduced by removing those states that the system cannot possibly enter any more. For instance, the number of pieces on a chess board is a *progress measure* that never increases on any path consisting of valid moves. In the sweep-line method [99, 100], once all states with a given progress measure have been explored, they are discarded. The ability of MARIA to distinguish "visible" (transient) and "hidden" (persistent) states [**P5**, Section 4.3.3], called "predicate-bounded depth first search" in a later publication [50], is a special case of the generalised sweep-line method [100], which is compatible with non-monotonic progress measures. A practical problem with the sweep-line method is the definition of suitable progress measures.

Compression Techniques

Various compression techniques can be used for removing redundant information from the data structures S and Q used by the state space enumeration algorithm (Appendix B.1). For instance, if the states in S can be addressed individually, Q can contain references to S instead of copies of states. In SPIN [73, Chapter 13] and LoLA [136, Section 5.2], Q keeps track on the performed transitions, so that backtracking becomes a matter of "undoing" transitions.

Utilising Constraints. MARIA encodes model states to bit strings before storing them [**P2**, Section 3.4]. In the bit strings, a data item that has been constrained to n possible values is represented with a fixed amount of $\lceil \log_2 n \rceil$ bits. A variable-length representation is used for unconstrained multi-sets (Petri net places without capacity constraints). Places for which an invariant expression (a special case of the place invariants of algebraic system nets [98]) is given are omitted from the bit vector, since their markings can be recomputed.

Representing Sets with Graphs. If the reachable states follow some pattern, it can be efficient to represent them with a graph structure. The performance of Boolean Decision Diagrams [18] (BDD) greatly depends on the way how the system state is translated into Boolean variables and in which order these variables occur in the diagrams. Also, a BDD can consume more space when state space reductions are applied, since reductions remove regularity [156]. Graph Encoded Tuple Sets [62] (GETS) seem to yield good results even when partial order reduction is applied. However, this kind of dynamically growing and shrinking graph data structures tend to create very random access patterns with potentially large transformation steps. Thus, it is hard to implement them efficiently for other than uniprocessor systems working on central memory.

Hierarchal Representations. Comparable performance to GETS can be reached with the much simpler "collapse" option [62, Section 5.8] of SPIN, which divides the state into process-specific components. The state is represented as a sequence of index numbers to component state sets. In modular state spaces, further savings may be achieved by representing the nodes of the synchronisation graph with indexes to strongly connected components of the module state space graphs [29, Section 4.1].

Distributed Storage. There is a practical limit on the amount of main memory that can be easily installed in commonly available computing workstations. One way of gaining access to more storage space is utilising the networking capabilities of modern workstations. Schmidt [137] describes a tree data structure for representing the set of reachable states, and an algorithm that allows the migration of subtrees to other computing nodes. When a computing node is about to run out of memory, it can delegate work to other nodes. This should work better in dynamic environments than Parallel Murφ [146], which distributes states among the workstations according to predefined hash signatures.

Approximating the Set of Reachable States

The memory usage reduction methods discussed so far have one thing in common: when the state space exploration algorithm terminates without encountering any errors, the model is guaranteed to be error-free.

To save memory space, the set S of reachable states can be approximated with a one-way hash data structure. If the structure maps two states $s \neq s'$ identically, the states reachable from s or s' may remain unexplored. In other words, the algorithm may produce *false positive* answers—claiming that the system is free of errors when it is not.

MARIA implements two methods of this kind: bit-state hashing [74, Section 3.4.2] and hash compaction [145, Section 2.3]. They can be used for estimating the lower bound of the state space graph size for models whose full state space graph would exceed the available memory capacity. These methods can reach nearly full coverage in a fraction of the storage capacity that would be needed for a full search.

3.4 Reducing the Execution Time

For a smooth workflow, it is important that the verification tool finds errors quickly. Many tools check properties "on-the-fly" while exploring the reachable states. In this way, errors that are reachable from the initial state with a small number of steps can be found quickly, without generating the full state space graph of the system.

Also simulation can be characterised with the terms "off-line" and "on-the-fly." For instance, the **probe** tool of PROD [155] explores previously generated state space graphs, while the graph browser of MARIA [**P2**] can compute successor states on demand. Simple modelling errors can be found by browsing the state space for a few steps from the initial state. An "on-the-fly" tool allows this without exploring the full state space.

Once the simple mistakes have been corrected, the model checking runs may need to visit a substantial part of the reachable states of the system. In explicit state space enumeration, this may take several hours on a contemporary workstation if there are tens or hundreds of millions of reachable states or enabled transition instances. One way of reducing the processing time is parallel processing, which is the topic of Section 5.3.

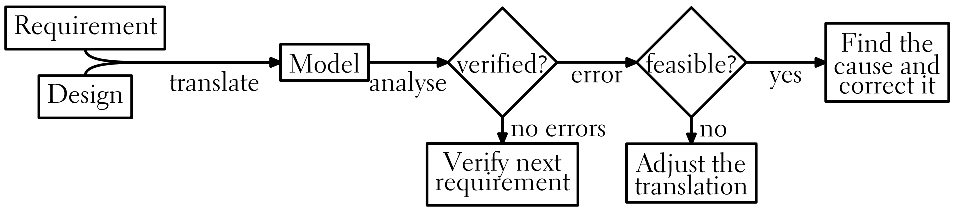


Figure 6: The verification procedure.

4 DEVELOPING VERIFIED SOFTWARE SYSTEMS

The main obstacle that prevents the widespread application of verification in the development of distributed software systems is the problem of automating the verification procedure, which is illustrated in Figure 6. The design can be anything between an abstract specification and a concrete implementation. The translation of a design into a model can produce a refinement of an abstract specification or an abstraction of an implementation. If the analysis of the model reveals an error, its feasibility is checked. Should the problem not manifest itself in the original design, the translation is adjusted, so that the model would better reflect the design. Otherwise the cause of the error is located, and the design or its requirements are revised.

As reasoned in Section 3, it is relatively easy to implement state space enumeration on almost any computing system. In other words, the core verification procedure can be automated even if the specialised methods reviewed in Section 2.3 cannot be applied. However, the huge number of reachable states that practically every system of interest is bound to have, calls for abstract or reduced models, as discussed in Section 3.2. Because not all errors of the abstract model can happen in the design, they must be tested for feasibility. Sometimes, this test and the adjustment of the translation rules can be automated, as in [70] and [158, Chapter 7].

Section 2 discusses formalisms and tools for describing systems and their intended behaviour. In *top-down* design methodologies, the system is gradually refined from an abstract description. In the *bottom-up* approach, the system is composed from readily implemented components.

The top-down approach permits early "debugging" of designs before anything is implemented, which can greatly reduce the cost of correcting errors. However, errors can still be introduced when the design is implemented [157, Section 2]. Thus, in order to assert that also the implementation meets the requirements, verification models must be extracted from the implementation. This is also how those bottom-up designs that lack formal descriptions can be verified.

The rest of this section discusses the verification of designs and implementations, emphasising the degree of automation that can be reached.

4.1 Verifying Abstract Designs

In top-down design methodologies, the distributed system is first described at an abstract level. Ideally, this description is a formal model that abstracts from implementation details, such as the communication architecture, as in Unity [24]. If the specification is not executable [67], the under-specified parts must be refined in order to verify the design [54, Section 1]. Once the abstract design has been verified, it is gradually refined until an implementation is obtained, possibly in a restricted high-level programming language that is translated into executable code.

The process algebraic method of verifying concurrent systems appears to work well on relatively abstract specifications. Many techniques, such as visual verification [152] and compositional verification [150, Section 7.3], seem to require similar insight as the derivation of mathematic proofs, something that is difficult to teach or automate.

If a theorem prover tool is used for verifying the design, and it is unable to complete a proof, a logic expert will be needed, either to guide the tool or to explain the designers or programmers what is going wrong.

If the modelling language lacks a construct that is necessary for verifying a property, the construct must be simulated. For instance, data arrays can be simulated by instantiating a variable for each array element and translating array operations into if-then-else blocks that branch according to the index value. The result can be a model that is harder to understand and much larger than the piece of software being modelled.

When Avrunin et al. [7] compared finite-state verification techniques, they noticed that the SMV [117] version of one of their models is about three times the size of the Spin [74] version. For manually constructed models, it is good to use a formalism that has direct counterparts for the most frequently used constructs of the specification language. For instance, Maria [**P2**] was selected for modelling a complex protocol because of its support for structured data types [149, Section 4.4].

Model checking tools demonstrate violations of requirements with execution traces that can be an invaluable diagnostic aid to the designer. The design can be described in any language whose execution semantics have been implemented in software. Ideally, the process of verifying the requirements can be fully automated, and expertise in formal methods is only needed when the requirements are formulated.

4.2 Verifying Implementations

In the bottom-up approach, the system is composed from readily implemented subsystems or modules. It is unrealistic to assume that there would exist formal specifications for these components, or any detailed requirements. However, there are some implicit requirements, such as the absence of deadlocks and resource leakages. More specific requirements can be embedded in the program code as runtime assertions. All these properties can in principle be verified by state space enumeration.

Verifying implementations would be easier if languages like Gypsy [61] and Occam [79] were in wider use. These languages isolate the programmer from the details of the runtime system and lack constructs that are problematic in verification, such as pointer data structures. However, such omissions make the language inflexible and inefficient for certain system-level applications.

The PathStar development project at Bell Labs [76] showed that a general-purpose programming language can be treated as a formal model, provided that the source code is annotated in such a way that an automated translator is able to make suitable abstractions. In that project, verification experts translated requirement specifications from English prose into temporal logic formulae and maintained the abstraction rules of the translator, so that it was possible to model check the software under development on a daily basis.

The Spin model checker [74], which was used in the PathStar project, has been designed for analysing computer protocols. It is based on a process-oriented language Promela [73, Chapter 5]. The application-oriented approach of Spin has turned out to suit extremely well to modelling communication protocols. There are translations to Promela from many languages, including Java [38] and TNSDL [148], a programming language of Nokia digital exchanges. Third-party modifications to Spin include dSpin [42], with improved support for modelling object-oriented software, and EASN [139], which supports ASN.1 data types that are used in some protocol standards.

Although the syntax of Promela resembles some programming languages, Promela is not a programming language. The second generation of Java PathFinder [157] does not translate Java to Promela, because Spin does not support some features of the source language, such as floating point arithmetics. Furthermore, some parts of the implementation could have been implemented in a different language, as in [78, Section 4.1], or the source code can be unavailable. Java PathFinder addresses these problems with a custom implementation of explicit state space enumeration that can handle programs of up to 10,000 source code lines by interpreting Java bytecode.

Model checking bigger programs requires some simplifications. The designers of Java PathFinder mention *abstract interpretation* and techniques based on *static analysis* and *runtime analysis* [157, Section 3.3].

Predicate abstraction [41] reduces the domains of data variables. For instance, an integer variable x can in some cases be represented with a Boolean variable b that indicates whether x is positive. The state space of the program can be reduced by replacing all occurrences of x with b and rewriting the operations so that b is updated in a consistent way. Java PathFinder and Bandera [38] rely on user-defined predicates, while SLAM [8], BLAST [70] and BOOP [158], which check non-distributed C programs by translating them into automata with Boolean variables, derive the predicates automatically with a theorem proving tool by using weakest preconditions. It should be noted that an automated abstraction refinement loop might never terminate [158, Section 7.4]. MAGIC [22] checks C programs based on simulation relations between labelled transition systems. It employs both theorem provers and propositional satisfiability solvers.

Slicing [147] can eliminate those statements that have no effect on the predicates that are needed for answering the verification question at hand. Shape analysis [37] can detect data structures that may be shared between concurrently running threads. This is essential for detecting the component structure of the system, which can be utilised by the methods discussed in Section 3.2. Corbett's shape analysis method may have problems with systems that make intensive use of procedures, as all procedure calls must be inlined [37, Section 5.1].

Runtime analysis focuses on a single execution trace at a time, by capturing information from a running system. The algorithm suggested by Dinning and Schonberg [47] is based on Lamport's "happened before" relation [104] and partial ordering of events. The Eraser algorithm [135] checks that all shared memory accesses follow a consistent locking discipline. Due to the nondeterministic nature of distributed systems, the information is less accurate than could be obtained from exhaustive analysis. However, the information can be used for focusing the state space enumeration efforts on potentially erroneous parts of the system. Java PathFinder can better check locking problems in this way [157, Section 3.3.3].

Spin and Java PathFinder are based on specialised modelling formalisms. A more generic formalism, such as algebraic system nets [98], could be applied to the analysis of a broader range of distributed systems. However, readability can be lost in a translation of software into Petri nets. Dividing the net into components, as in [56], provides some help, but dynamic structures in the source language easily lead into duplicated net structure, as in [78, Section 3.2]. Our proposed tool for verifying data-centric enterprise applications [**P5**] translates only a small amount of high-level application code and composes it with compact models of the surrounding entities of the code.

5 SUMMARY

The five subsections of this section summarise the publications [**P1**]–[**P5**], present related work and list the contributions of each publication.

5.1 A Unification Algorithm for Computing Successors

This section is a summary of the publication [**P1**], which describes an algorithm for finding the enabled bindings of transitions in a high-level Petri net. A simplified version of the algorithm is presented in Appendix B.3.

Background

Compared to many other high-level modelling formalisms of parallel and distributed systems, high-level Petri nets (Section 2.2) have two fundamental differences. First, the state transformation rules of high-level nets, called *transitions*, do not directly define equations or assignments on the state variables of the model, called *places*. Instead, transitions are connected to places via *arcs* that are labelled with expressions on *transition variables*. Second, each place can contain multiple values (*tokens*) at a time, and a transition is enabled if the places connected to its input arcs contain all values that the input arc expressions expand to.

The strength of high-level nets lies in the decoupling of places and variables. For instance, the protocol for leader election in an unidirectional ring [48] can be modelled with transitions that atomically receive a message, make decisions and send a message if needed. In a PROMELA [73, Chapter 5] model of the protocol, each Petri net transition corresponds to several transitions. This explains why the Petri net model generates much smaller state spaces than the PROMELA model (Table 1).

Unfortunately, the strength is also a shortcoming. High-level Petri nets are a somewhat difficult language for modelling distributed software systems. On one hand, they lack the structuring capabilities of programming languages and the notion of control flow. On the other hand, programming environments are rarely based on multi-set data structures or the ability to perform several assignments simultaneously. Thus, the mechanical translation of a software system into a high-level Petri net is unlikely to fully utilise the potential of Petri nets.

From the analysis point of view, high-level Petri nets require a more

Table 1: Unreduced state space sizes for the leader election protocol [48] for different numbers of processes, obtained with SPIN 4.0.4 and MARIA 1.3.3.

	PROMELA		High-level net	
n	states	events	states	events
3	399	873	69	126
4	2,400	7,073	240	588
5	15,779	58,181	870	2,693
6	106,435	470,271	3,213	12,013
7	723,053	3,725,165	11,949	52,310

Table 2: Time needed to solve the "n queens" problem on a 700 MHz Intel Pentium III system.

Problem Size		Design/CPN	PROD	MARIA	FC-CBJ
n	solutions	4.0.5	April 2, 2003	1.3.4	
4	2	0 s	0.0 s	0.0 s	0.0 s
5	10	0 s	0.0 s	0.0 s	0.0 s
6	4	0 s	0.0 s	0.0 s	0.0 s
7	40	1 s	0.0 s	0.0 s	0.0 s
8	92	33 s	0.0 s	0.0 s	0.0 s
9	352	631 s	0.2 s	0.3 s	0.0 s
10	724	20,100 s	2.1 s	3.4 s	0.0 s
11	2,680		23.5 s	39.0 s	0.2 s
12	14,200		304.4 s	506.0 s	0.9 s
13	73,712		4,018.1 s	7,042.8 s	5.1 s

complicated algorithm than extended finite state automata, the formal model behind PROMELA [73, Chapter 5], Estelle [86] and SDL [89]. While extended automata operate directly on state variables, high-level nets call for a unification algorithm that finds all valuations of transition variables under which the arc expressions are compatible with place markings.

Related Work

Unification as a Constraint Satisfaction Problem. Sanders has mapped the computation of enabled transition bindings in high-level nets to a constraint satisfaction problem. Originally [133], he supported a class of nets where the place markings are restricted to power-sets rather than multi-sets. Later [134], he presented a variation of the FC-CBJ algorithm (Forward Checking with Conflict-Directed Backjumping) that supports multi-set markings and constant-multiplicity arc expressions. It should be noted that this algorithm cannot be used for finding the bindings of the transition in Figure 3(b) because of the non-constant multiplicity $\frac{x}{y}$.

Sanders presents a high-level net model [133, Figure 1] of the problem of placing n queens on an $n \times n$ chess board in such a way that they do not threaten each other. The net consists of one place and one transition, whose n variables are bound to the tokens $1, \ldots, n$ that the place is initially marked with. The diagonal movement rules of the queens are formulated in the transition guard.

The state space of the high-level net consists of two states and a number of events. For each solution to the problem, there is one event from the initial to the final state. We explored the net with some Petri net tools, and we also solved the problem with the CSPLIB [154] implementation of the FC-CBJ algorithm. For the Petri net tools, the execution times reported in Table 2 exclude the time needed by code generators.

The main reason why FC-CBJ outperforms the Petri net tools in this example is that it allocates look-up tables of infeasible solutions [134, Section 6]. When the search backtracks, it updates the conflict relation, so that related combinations will not be explored again. Such tables can be impractical if there are many variables with very large domains.

Since the implementations of Sanders do not appear to be available, we were unable to measure their performance on models of high-level software systems, which are unlikely to feature this kind of combinatorial problems within a single transition. Sanders admits [133, Section 6]:

> The provision of advanced constraint satisfaction algorithms for all transitions in a model introduces a problem on its own: some transitions in a model will have limited bindings and be very easy to solve. It is important not to spend excessive time attempting to reduce such problems when they can be easily solved by brute force.

Unification in Generic High-Level Nets. To our knowledge, the only state space exploration tools for high-level nets with variable-multiplicity arc inscriptions that work without unfolding are Design/CPN [119] and its successor CPN/Tools [10], Prod [155], and Maria [**P2**].

The algorithm of Design/CPN has been described in [140, pages 10–13] and in more detail by Haagh and Hansen [65, Chapter 4]. Haagh and Hansen lay emphasis on randomised simulation, which can be sped up by various look-ahead techniques [65, Chapter 2] and by finding one enabled transition instance without enumerating all of them. Their improvements to the original Design/CPN algorithm have been implemented in a separately available "new simulator" package. Unfortunately, we were unable to measure the performance of this package, because it does not support exhaustive state space enumeration, only random simulation.

In Table 2, the proportional speed difference between Design/CPN and other Petri net tools seems to grow with the size of the problem. Thus, it cannot be solely explained by the choice of implementation language. Without access to the internals of Design/CPN, we cannot offer a definite explanation. One possible cause for the poor performance of Design/CPN could be that the order in which it binds variables to input arcs is determined dynamically, while Prod and Maria apply a static scheduling, which implies less overhead. Another possibility is that Prod and Maria evaluate the transition guard earlier in the search tree and thus enumerate fewer infeasible solutions than Design/CPN.

The reason why Prod performs slightly better than Maria in Table 2 is that its inscription language corresponds more closely to the target language of the code generator. Furthermore, unlike Maria, Prod does not trap evaluation errors or report domain violations of output places. The two tools use a very similar unification algorithm. The main differences are in the data type system and expression evaluation.

A Description of the Algorithm

Appendix B.3 lists the algorithm in a form that can be used as a subroutine of the state space enumeration procedure, which is listed in Appendix B.1 and discussed in Section 3. The algorithm is presented as a recursive procedure analyse_arcs, which is invoked by successors for each transition of the net. The recursive loop processes each input arc of the transition, augmenting the valuation of transition variables during the search. Once all input arcs have been processed and the valuation has been completed to a full transition instance, the procedure analyse_arcs computes and reports the resulting successor state.

The algorithm distinguishes two kinds of input arc expressions: ones that contain unbound variables, and ones whose value can be evaluated under the valuation collected so far. The former kind is processed by the subroutines analyse_variable and bind_variables. The latter kind of expressions is managed by the procedure analyse_constant, which evaluates the expression and subtracts it from the marking before proceeding to the next input arc via the recursive invocation of analyse_arcs. In case the expression cannot be subtracted from the marking, the search backtracks.

Arc expressions that contain unbound variables can be associated with any available token in the corresponding input place. This association is recorded in the array *t.unified* in the procedures analyse_variable and bind_variables. First, analyse_variable evaluates the multiplicity of the arc expression. If it turns out to be zero, the expression will evaluate to the empty multi-set, regardless of the values of the unbound variables, and the search can proceed to the next input arc. Only if the multiplicity is positive will the valuation be augmented by bind_variables.

The procedure bind_variables iterates over every available sub-marking $c‘m$ of the place corresponding to the input arc, and associates the input arc expression with one of them at a time. It extracts a value for every variable that is to be bound from the input arc expression, and augments the assignment. Provided that the transition guard is enabled and that the markings associated with previously processed input arc expressions are possible under the augmented valuation, the algorithm advances to the following input arc by invoking analyse_arcs.

Details of finding assignment candidates and checking the compatibility of previously unified expressions can be found in the publication [**P1**].

Contributions

Publication [**P1**] presents a search algorithm for determining the assignments under which transitions are enabled in a given marking of a high-level Petri net. A similar algorithm has been implemented earlier in PROD [155], but it was never formally documented.

Unlike PROD, the presented unification algorithm [**P1**, Section 3] traps evaluation errors. Also, while PROD processes the input arcs in the order of appearance, a preprocessing step in MARIA tries to improve the ordering [**P1**, Section 4.2]. Both tools skip the unification loop if an input place is empty (PROD) or does not contain enough tokens (MARIA).

The algorithm is normally used for computing possible successor states in high-level Petri nets. In MARIA, it is also the core of an algorithm [**P1**, Section 3.3] that may drastically reduce the size of place/transition nets translated from models containing large data domains.

5.2 Modular Reachability Analyser

This section is a summary of the publication [**P2**], which presents an extensible state space enumeration tool for high-level Petri nets.

Background
Verifying industrial-size designs with minimal manual effort is a challenge. Publication [**P2**] lists the following tool requirements for automated checking of distributed software systems:

High-level formalism. The formalism should provide enough expressive power, so that high-level system descriptions can be modelled in a straightforward way.

Ease of use via translators. If there are automated model translators for the tool, end users do not need to be familiar with the underlying formalism; they can work in the domain they are used to.

Efficient utilisation of computing resources. The tools should work in a variety of computer systems, ranging from personal computers to multiprocessor supercomputers, and memory management should be optimised to accommodate large state spaces.

Section 5.1 discusses the merits and weaknesses of high-level Petri nets. We believe that implementation-level designs are best verified using translations into low-level formalisms, such as extended automata or transition systems. The tools and methods for this are discussed in Section 4.2. Table 1 in Section 5.1 supports our belief that high-level Petri nets are an efficient formalism for modelling and verifying designs of parallel and distributed systems. The rest of this section compares MARIA to other verification tools for Petri nets.

Related Work
Place/Transition Systems. PEP [13] and LoLA [136] operate on ordinary place/transition systems. They can analyse simple high-level net models via unfolding. Also MARIA can unfold models into PEP or LoLA format.

PEP models can consist of components, which can be nets or simple high-level languages, such as the Basic Petri Net Programming Notation [13, Section 3.1]. For analysis purposes, PEP unfolds the system into a place/transition system. This can be impractical when the data domains are large and only a small fraction of the potential states will be reachable. With some work, PEP models can be translated into MARIA format in order to verify them, as in [20, Section 5.1].

PEP implements McMillan's algorithm [118] for unfolding nets into complete finite prefixes. LoLA supports symmetry reduction [144] and partial order reduction methods [128].

High-Level Nets. To our knowledge, the only state space exploration tools for high-level nets that support conditionals in arc expressions [15, Section 6.5] are Design/CPN [119] and its successor CPN/Tools [10], PROD [155], and MARIA [**P2**].

Design/CPN [119], a system for modelling and analysing with coloured nets [91], has evolved around a graphical editor. It supports the modelling of timed systems, and it offers symmetry reduction [77] of state spaces. The analysis algorithms of Design/CPN have been implemented in Standard ML, which makes it easy to experiment with different analysis methods, such as the sweep-line method [99, 100]. The choice of implementation language could be one reason why Table 2 suggests that Design/CPN consumes more resources than other high-level net tools.

Prod [155] is one of the best-known model checker tools for high-level nets. It has long served as a benchmark for competing tools. Prod is based on predicate/transition nets [59], but its state space reduction methods operate internally on place/transition nets.

Contributions

Many state space enumeration tools translate the model into a high-level programming language and thus delegate the computation of the successor relation to the runtime system of that language. Such tools include Spin [74], Murφ [46], Prod [155] and Design/CPN [119]. In Maria, the executable code generator is optional [**P2**, Section 3.2]. The interpreter-based operation of Maria is useful in interactive simulations and debugging, as it avoids the overhead of invoking a compiler.[3] The interpreter is also good for analysing automatically generated models of SDL specifications [1] or Java code [**P5**], which tend to become very large.

The built-in preprocessor of Maria supports aggregation operations over bounded domains. For instance, the Maria model of the "n queens" problem (Section 5.1) can be varied for different values of n by simply redefining the "queen" data type. For Prod and Design/CPN, the transition guard $\forall q_1, q_2 \neq q_1 : |x_{q_1} - x_{q_2}| \neq |q_1 - q_2|$ has to be expanded differently for each value of n. For the experiment reported in Table 2, the expansions were obtained from the Maria syntax tree dump facility.

Maria represents the states of its models in two different ways. The expanded representation is used when determining successor states and performing computations. Long-term storage of explored states in disk files or main memory is based on a condensed representation, a compact bit string whose encoding has been documented in [116]. Due to this compact encoding, Maria usually needs less storage space than Prod or Design/CPN. In the experiment reported in [**P2**, Section 4.3], Maria was also significantly faster than Prod.

5.3 Parallelised State Space Enumeration

This section is a summary of the publication [**P3**], which describes a distributed algorithm for enumerating state spaces. The algorithm is presented in Appendix B.2. It is a simple modification of the sequential algorithm (Appendix B.1), which is discussed in Section 5.1. There is a central server that stores the set of reachable states S locally and distributes the work queue Q of unprocessed states among clients, which send back the successor states for each unprocessed state.

[3]One of the most complex Maria models written so far, about 500 kilobytes of text, is a model of a radio link control protocol [149]. Compiling the Maria-generated C code of the model takes tens of minutes. In that time, the interpreter would generate hundreds of thousands of states of that model.

Background

Parallel processing can speed up the execution of algorithms that contain loops which can be iterated in arbitrary order. Ideally, if a single processor completes n iterations of a loop in t units of time, k processors will complete the task in $\frac{t}{k}$ time units if $1 \leq k \leq n$. In practice, the execution time of a single iteration may vary, and inter-processor communication may slow down the execution. Furthermore, the number of iterations may be determined at run time, as in the state space enumeration algorithm.

There are two types of memory in parallel computing systems: distributed and shared. Distributed memory can be implemented by building a networked cluster of inexpensive commonly available processing units, but shared memory requires special hardware. The nice thing about state space enumeration is that the algorithms can be implemented efficiently on distributed memory systems. The specialised methods discussed in Section 2.3 require random access to a large pointer-linked data structure that must be shared among the processors. Parallel implementations of these methods may be inefficient even on shared memory systems. Heljanko, Khomenko and Koutny [69, Section 4] offer insufficient memory bandwidth as an explanation for the poor performance they observed on a four-processor shared memory workstation.

Many algorithms contain multiple levels of loops. Inner loops are iterated more frequently than outer loops. On one hand, the computations in inner loops could be distributed to more processors than the computations in outer loops. On the other hand, the processors need to communicate when entering or leaving a parallelised loop.

If the inner loops are parallelised, the processors may spend more time in communications than in computations. Lorentsen and Kristensen [115] distribute a subroutine of the function that determines the successor states of given state. In terms of the algorithms in Appendices B.1 and B.3, their parallelised loop is nested inside one or two sequentially iterated loops. Several synchronisations are needed to process one state, while our algorithm [**P3**, Section 3] can exchange several states in one synchronisation. This could explain why the algorithm of Lorentsen and Kristensen could not efficiently utilise more than 4 or 5 slave processors [115, Table 2].

Parallelising the outermost loop could leave many processors "unemployed" if there are not enough iterations, but the communications overhead will be low and the iteration steps can be existing programs. The TrailBlazer system [76] does not speed up a single model checker run, but it can check several properties in parallel by invoking SPIN [74] on multiple computers. The rest of this section discusses parallel processing within a single model checker run.

Related Work

Various distributed algorithms and strategies have been suggested for discrete event simulation [26, 57]. However, these simulation algorithms do not systematically explore the reachable states of the model—instead, they perform a random walk in the state space graph and collect statistics on the performance of the modelled system. In state space exploration, the enabled actions are not executed at random, but all reachable states are covered. The look-ahead techniques of distributed simulation cannot be utilised because of the large number of possible successor states.

Schmidt [137] lists two objectives for using clusters of workstations in state space verification. First, larger state spaces can be explored without running out of memory. Second, the parallel processing may speed up computation. Parallel Murφ [146] tries to combine both aspects by distributing states to processors according to a hash function, but Schmidt argues that its memory usage might not be evenly distributed if there were hundreds rather than tens of workstations. Furthermore, Nicol and Ciardo [124], who present a distributed algorithm for exploring integer manipulation systems, point out that automatic parallelisation cannot depend on a user-defined hash function for every model. Černá and Pelánek [21], who present a distributed algorithm for finding fair counterexamples for liveness properties, do not provide any statistics on memory usage distribution.

Inggs and Barringer [82] present a parallel state space enumeration algorithm for shared memory supercomputers. To improve load balancing, each process has *two* queues of unprocessed states: a private one and a shared one, from which other processes can "steal" work when their own queues become empty. The private queue can be accessed without any inter-process communication overhead.

Schmidt's distributed algorithm [137] is an extreme case that tries to fully utilise the distributed memory capacity of the cluster at the expense of processing time. The other extreme would centralise the memory usage and distribute the computations only. Both extremes may seem unreasonable at first. However, Schmidt's evenly distributed memory scheme provides orders of magnitude faster random access times than magnetic storage. Also, while it makes sense to utilise all the available memory in a dedicated computing cluster, there could be networked devices that have spare processing power but relatively little memory.

Contributions

Unlike many distributed model checking algorithms [21, 69, 76, 82, 115, 124, 146], our distributed state space exploration algorithm [**P3**] has been optimised for utilising the available processing power of regular office computers, whose processors are idle or underutilised most of the time. A computer user is unlikely to notice a background process, as long as it does not consume much memory or cause disk activity. In our algorithm, the memory usage is centralised on one dedicated computer. The server distributes unprocessed states to clients, which send back the successor states for each unprocessed state. The clients can join and leave the computation at any time. Our implementation of the server distributes the work queue of a disconnected client among the remaining clients.

The implementation of this distributed algorithm in Maria is based on connection-oriented communications [80, Section 2.10] provided by all commonly used operating systems. In our experiments, a heterogeneous network of office computers performed well compared to a supercomputer. In the algorithm, the average number of new successor states generated from each state is a critical factor. If the server runs out of unprocessed states that it could distribute to clients, some clients may become idle. Thus, the number of clients the algorithm can utilise depends on the structure of the state space being explored.

The processor utilisation of our algorithm can be improved by applying state space reductions in the clients [**P3**, Section 4.1]. Partial order reduction could improve the scalability even further, for two reasons: First, it reduces the number of explored events and thus the number of old states the clients report to the server. Second, each client will spend more time per reported state and thus preserve the communication bandwidth of the server, allowing more clients to participate.

5.4 Modular State Space Enumeration

This section is a summary of the publication [**P4**], which presents a modular state space enumeration algorithm for nested hierarchies of high-level Petri nets that synchronise via shared transitions. A simplified version of the algorithm is presented in Appendix B.4.

Background

Modular state spaces were originally presented on transition systems. Later, the idea has been lifted to higher-level formalisms, such as communicating state machines [94] and high-level Petri nets with flat [28] and nested synchronisation hierarchies [**P4**].

Let us consider the system of [**P4**, Figure 1], comprising three deterministic processes that synchronise on two actions. The *synchronisation graph* of this system, depicted at the right of Figure 7, shows all observable synchronisations of the system. The full state space in Figure 8 is dominated by interleaved executions of concurrently enabled internal actions of the processes. Clearly, for this kind of a system, the time and space needed for constructing the modular state space is linearly proportional to the number of internal steps, while conventional state space enumeration requires an exponential amount of time.

Model checking works in a slightly different way in modular state spaces than in a flat state space. A property of a single module can be checked by synchronising the automaton of the property with some

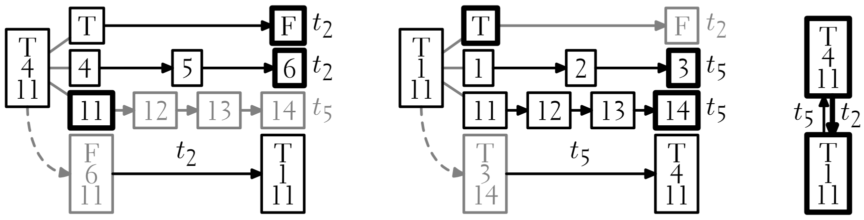


Figure 7: The construction of a synchronisation graph. There are three modules and two synchronising actions, t_2 and t_5. In the initial state, only t_2 is enabled, as all modules that synchronise on it can internally reach a state where it is enabled. Similarly, only t_5 is enabled in the successor state. In a synchronisation, the states of non-participating modules do not change.

actions of the module [**P4**, Section 4.2]. Properties over multiple modules can be verified by adding synchronisations between the modules or by changing some internal actions visible, so that interesting changes in the states of the modules become observable in the synchronisation graph.

Related Work

Modular analysis of high-level nets was introduced by Christensen and Petrucci [28, 29], who did not present an algorithm. Instead, they showed how certain Petri net properties can be proved on modular state spaces. Christensen and Petrucci suggest that the algorithm should compute strongly connected components of module state space graphs. Our algorithm for checking safety properties [**P4**, Section 4] does not do so. As a result, it requires less memory but typically revisits module state spaces several times. The running time of our algorithm can be reduced by the use of caches [**P4**, Section 5.3].

In order to roughly compare the performance of the algorithm suggested by Christensen and Petrucci with our algorithm, we obtained from Laure Petrucci a prototype implementation for computing the modular synchronous product of finite state machines, and some models in this formalism. We implemented our algorithm for the same formalism and made some experiments. As expected, our implementation uses much less memory than Petrucci's. To our surprise, for this set of models, our implementation turned out to explore the synchronisation graph an order of magnitude faster than Petrucci's, even though it revisits the module state spaces thousands of times. The situation would be different if the strongly connected components were needed in further analysis, or if it were expensive to explore the successor states in the modules.

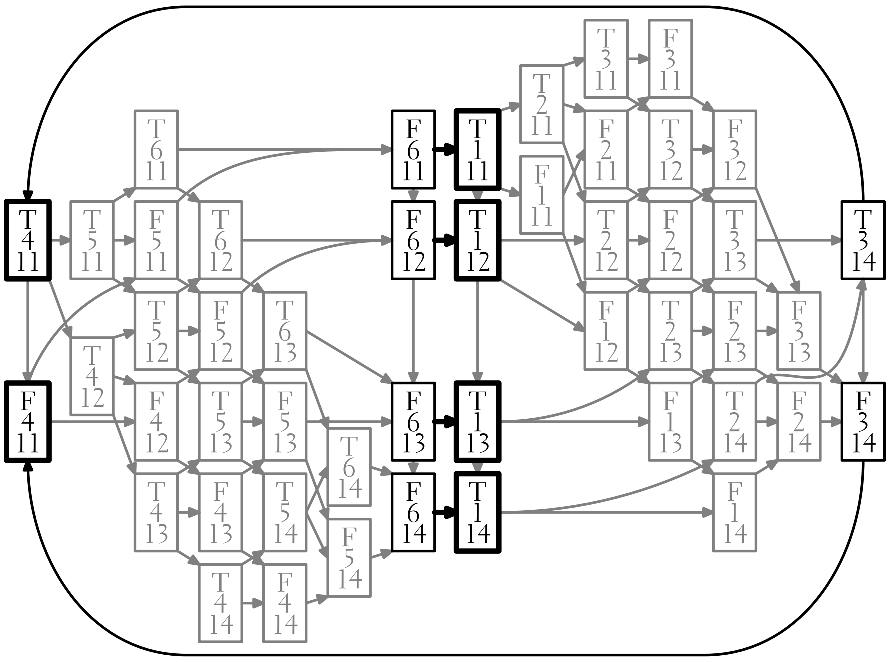


Figure 8: The flat state space graph of the system. The initial state is the node at the top left. The states where a synchronisation occurs are highlighted with a black border. There are four occurrences of t_2 and two occurrences of t_5.

Partial order reduction, which postpones the occurrences of some enabled transitions that are independent of others, could achieve similar results. For performance reasons, the dependence relation is often approximated, as in [74, Section 3.3]. Although modular state space enumeration treats the processes as "black boxes" with known interfaces, it can outperform partial order reduction, as in [131, Section 3.4.3].

Karaçalı and Tai [94] have improved partial order reduction by making it aware of modules. Their algorithm also needs transition dependency information, which may be difficult to derive for complex components.

Under thread-modular model checking [55], each thread is checked separately, abstracting the behaviour of interleaved steps of other threads. The algorithm works by incrementally computing two relations. It does not handle dynamic thread creation or message passing operations.

For high-level Petri nets, modular analysis has been generalised to incremental state space construction of net refinements [111], which seems to work well on object-oriented systems refined by inheritance. However, it is unclear how the substitutability criteria that incremental analysis depends on could be defined for other than object-oriented models.

Usually, the system is divided into modules by the designer. Buchholz and Kemper [19] present criteria for dividing place/transition nets into modules. A similar division can be obtained for software systems by shape analysis [37]. It would be nice to know how effective this kind of algorithms are compared to a human designer: If a modular specification were flattened and then remodularised using an automatic tool, could the desired properties be verified more efficiently in the transformed model? Unfortunately, we were unable to obtain a copy of APNN Toolbox, which should implement the algorithm of Buchholz and Kemper.

A Description of the Algorithm

Compared to basic state space enumeration, the modular algorithm needs one extra bit of information is of the system: it must be possible to determine which synchronisations are possible in a given state of a module.

The modules are arranged as a hierarchy tree. Synchronisations are possible between sibling modules (which have a common parent). The parent module encapsulates the child modules; the state of a module is a component of its parent's state. Transitions synchronising on a given label may only occur as a concurrent step comprising all transitions synchronising on the label.

The algorithm listed in Appendix B.4 comprises several procedures. The entry procedure is explore, and the states reachable via local transitions are enumerated by transitions. The procedure modules enumerates the possible synchronisation points of child modules by recursively invoking explore on them. It also invokes sync in order to evaluate the effect of synchronisations in the state space of the parent module.

The algorithm is invoked as explore on a given state of the root module. The parameter $\mathcal{S}$ is an initially empty set that will record the possible synchronisation points in the modules. It will be extended by transitions and examined by sync. For the root module, which does not have any transitions that synchronise on a label, the set $\mathcal{S}$ remains empty.

Figure 7 can help to understand the algorithm. To construct the synchronisation state space of the root module, the algorithm would be invoked on its initial state as $\mathsf{explore}(m, \langle \mathrm{T}, 4, 11\rangle, \emptyset)$. The invocation of transitions has no effect on the root module of this system, since it does not contain any transitions. The call to modules will split the state to three components, $s_1 = \mathrm{T}, s_2 = 4, s_3 = 11$, corresponding to the three child modules of the root module m, $C(m) = \{1, 2, 3\}$. The procedure modules will recursively invoke explore on each of these modules, to construct the mapping $\mathcal{S} = \{(1, t_2, \mathrm{F}), (2, t_2, 6), (3, t_5, 14)\}$ of possible synchronisation points. It will pass this mapping to sync, which will enumerate all its subsets that contain all transitions synchronising on a given label. In this case, there is only one such subset, $\{(1, t_2, \mathrm{F}), (2, t_2, 6)\}$. The procedure sync makes a copy of the state of the parent module, $s = \langle \mathrm{T}, 4, 11\rangle$, and overwrites the components corresponding to the child module states where the synchronisation is possible, $s^* = \langle \mathrm{F}, 6, 11\rangle$. Finally, in lines 6 and 7 of sync, the state $s' = \langle \mathrm{T}, 1, 11\rangle$ resulting from the concurrent step of the synchronising transitions will be added to the state space of the parent module. The synchronisation on t_5 will be explored in a similar way in the second iteration of explore, with $s = \langle \mathrm{T}, 1, 11\rangle$.

The algorithm is described in more detail in [**P4**, Section 4.1].

Contributions

The article presents a slightly more general version of modular high-level nets than Christensen and Petrucci [28] and an algorithm for checking safety properties in these nets by exhaustive enumeration of modular state spaces. To our knowledge, no concrete algorithm for exploring modular high-level nets has been presented before.

The ability of modular state space enumeration to treat the modules as “black boxes” is important, since it avoids the need to construct detailed formal models. Our algorithm supports modules within modules, which may be useful when verifying layered designs, such as distributed applications on computer networks [**P4**, Figure 3].

We believe that in practice, the verification of distributed software systems can benefit from modular analysis. By focusing on synchronisations, modular state space enumeration constructs a more abstract or optimised model “on-the-fly.” Thus, some of the tedious work of optimising models for verification [150, Section 7.1] is avoided and shifted to the exploration algorithm. Unoptimised models are likely to be easier to maintain and reuse than optimised ones.

5.5 Managing Component-Oriented Enterprise Applications

This section is a summary of the publication [**P5**], which presents a method of developing component-based enterprise applications in a way that enables the use of model checking tools.

Background

Enterprise application systems aim to integrate business processes within companies. The current trend is to expand integration across organisations, which requires that the software systems of different organisations be integrated together. It is likely that the pieces of the system originate from many different software vendors.

Software components have been suggested as a way of managing the situation where nobody has a full view of the entire system. Components are packaged software artifacts that provide functionality through a set of well defined interfaces. These interfaces isolate component development from the rest of the system and allow components to be replaced individually. The core system contains only minimal functionality, and the necessary tailoring is done by writing some application code that composes software components within the system framework.

Enterprise applications used to be deployed on a mainframe computer that was accessed through dumb terminals. In the early 1990's, client/ server architectures migrated the application logic to personal computers. In the late 1990's, web-based architectures introduced application servers that reside between clients and database servers.

Sneed and Göschl [143] distinguish *unit testing*, *integration testing* and *system testing*. Unit testing can reveal trivial programming mistakes within a single software component. Integration testing focuses on interactions between components, and system testing covers the entire system and its external interfaces. Holzmann [75] shows that model checking can reach very good coverage with a harness that is a fraction of the size of the software being tested.

Related Work

VeriWeb [11] is an automated tool for testing web applications. It combines the VeriSoft tool [23] with a web browser. The authors give some examples of generating input for forms and implementing checks against errors in the server responses. VeriWeb could be a useful regression testing tool for checking applications after they have been updated and installed.

The large number of core hardware and software components in web-based architectures makes it expensive to set up environments for integration and system testing. A testing environment that works on abstract models could save a lot of money and effort. The application code could be explored by Spin [74] or Java PathFinder [157]. However, it can be difficult to model relational data with these tools.

Lie et al. [112] present a method for automatically extracting models from low level software implementations. The extracted model is combined with a model of the hardware implementation. MAGIC [22] checks C programs with respect to models specified with labelled transition systems. These approaches are similar to the one suggested in Publication [**P5**], except that in Publication [**P5**], models extracted from program code are combined with high-level Petri net models of software components and the environment of the application.

Contributions

Publication [**P5**] proposes a tool that allows software maintainers to verify the correctness of enterprise applications before system level testing.

Since the tool is independent of the deployment environment, modifications the applications can be tested more frequently, without setting up a copy of the production environment. Not all components need to be available in order to test the application; high-level models of them will suffice.

It is difficult to avoid the state space explosion when exploring software systems. In Publication [**P5**], the model is constructed in such a way that there are *persistent* and *transient* states. In a persistent state, all local data is discarded. Also, similar to the sweep-line method [100], sequences of transient states are discarded when the search enters a persistent state. Finally, the user is given control on the behaviour of the environment and the domain sizes of parameters.

6 CONCLUSION

Our research goal has been to develop efficient computerised tools and methods for verifying safety properties of parallel and distributed high-level software systems. The main challenges are the state explosion problem and the amount of human effort needed to construct models suitable for verification. Specialised analysis methods can manage very large state spaces, but they usually require a tedious translation of the software system into a formal model. State space enumeration is a generic technique that can be implemented easily on almost any computing system. The techniques presented in this work allow state space enumeration to work efficiently on a broader range of systems.

All algorithms presented in this work have been implemented in a freely distributable state space enumeration tool and model checker for high-level Petri nets. Its modelling language facilitates compact description of systems that make use of structured data. The inherent concurrency in Petri nets makes it straightforward to model hybrid systems containing nondeterministic components.

This work describes a selection of automated techniques for locating errors in distributed software systems. The interpreter-based state space enumeration described in Publication [**P2**] is most useful in early stages of system development, when the design is likely to contain a large number of simple mistakes. Parallel search algorithms [**P3**] and modular state spaces [**P4**] become important when more detail is added to the models and their state spaces grow bigger.

There is evidence that model checking can outperform traditional testing by an order of magnitude. Widespread adoption of verification techniques among software engineers requires integration of formal methods in the software development process. Initial steps have been taken in the development of telephone switches [76] and embedded controllers [157]. Developers of enterprise applications [**P5**] could be the next beneficiaries of automated formal methods. It took two decades to develop efficient verification tools, and it will take some time to change the attitudes.

6.1 Topics for Further Research

Our distributed implementation of state space enumeration [**P3**] traverses the state spaces in roughly breadth-first order, which produces short error traces for violations of safety properties. An algorithm for verifying liveness properties has to traverse the state space in depth-first order, in order to detect loops. It would be interesting to see whether the work queues of unprocessed states could be permuted in such a way that the parallel traversal proceeds in nearly depth-first order.

Our implementation of the algorithm has not been tested on very large networks of workstations. Its scalability could be improved by distributing the server process on multiple physical computers, perhaps as outlined in [127].

Java PathFinder [157] can verify Java programs on the object code level, by implementing the algorithms in a special virtual machine. The freely available Ladybug tool [35] is a runtime analyser that works by instrumenting Java bytecode. This kind of tools are helpful when not all source code is available. Developers of device drivers or other system-level software could benefit from similar tools that operate on machine code rather than on the bytecode of a virtual machine. Such a tool could be constructed from a virtual machine like Bochs [108] or a binary-to-binary translator like that of the Valgrind [138] memory debugger. Valgrind already includes an implementation of the Eraser algorithm [135] for checking the locking discipline of shared memory accesses, but it does not implement any state space exploration or model checking algorithms.

The processes or modules of a system and the relations between them are not always clearly reflected by models or program code. With the help of shape analysis [37], our modular state space enumeration algorithm [**P4**] could also be applied to such "flat" descriptions. In order to verify reactive systems with the algorithm, it should be extended to support the verification of liveness properties.

Combinations of different types of formal methods can be more powerful than a single method. Tools based on automated predicate abstraction [8, 22, 70, 158] combine propositional satisfiability solving, theorem proving, symbolic model checking, and techniques on transition systems. The currently available tools of this kind can manage software written for a single processor or for multiple processors that communicate via shared memory, but not by passing messages. This kind of methods are challenged by undecidability results from computational complexity theory.

References

[1] Annikka Aalto, Nisse Husberg, and Kimmo Varpaaniemi. Automatic formal model generation and analysis of SDL. In Rick Reed and Jeanne Reed, editors, *SDL 2003: System Design, 11th International SDL Forum*, volume 2708 of *Lecture Notes in Computer Science*, pages 285–299, Stuttgart, Germany, July 2003. SDL Forum Society, Springer-Verlag, Berlin, Germany.

[2] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy, FIFO channels. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification 1998, 10th International Conference*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318, Vancouver, BC, Canada, June 1998. Springer-Verlag, Berlin, Germany.

[3] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, UK, 1996.

[4] Nina Amla, Robert Kurshan, Kenneth L. McMillan, and Ricardo Medel. Experimental analysis of different techniques for bounded model checking. In Garavel and Hatcliff [58], pages 34–48.

[5] Rob D. Arthan. On formal specification of a proof tool. In Søren Prehn and Hans Toetenel, editors, *4th International Symposium of VDM Europe*, volume 551 of *Lecture Notes in Computer Science*, pages 356–370, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, Berlin, Germany.

[6] Edward A. Ashcroft and Zohar Manna. Formalization of properties of parallel programs. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the Sixth Annual Machine Intelligence Workshop, Edinburgh, 1970*, volume 6 of *Machine Intelligence*, pages 17–41. Edinburgh University Press, UK, 1971.

[7] George S. Avrunin, James C. Corbett, Matthew B. Dwyer, Corina S. Păsăreanu, and Stephen F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts at Amherst, Amherst, MA, USA, November 1999.

[8] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In Alex Aiken and Greg Morrisett, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–105, New Orleans, LA, USA, January 2003. ACM Press, New York, NY, USA.

[9] Thomas Ball and Sriram K. Rajamani, editors. *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, Portland, OR, USA, May 2003. Springer-Verlag, Berlin, Germany.

[10] Michel Beaudouin-Lafon, Wendy Mackay, Peter Andersen, Paul Janecek, Mads Jensen, Michael Lassen, Kasper Lund, Kjeld Mortensen, Stephanie Munck, Anne Ratzer, Katherine Ravn, Søren Christensen, and Kurt Jensen. CPN/Tools: A post-WIMP interface for editing and simulating coloured Petri nets. In Colom and Koutny [36], pages 71–80.

[11] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic web sites. In *The Eleventh International World Wide Web Conference*, pages 654–668, Honolulu, HI, USA, May 2002. International World Wide Web Conference Committee.

[12] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Natarajan Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *Lfm2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Williamsburg, VA, USA, June 2000. National Aeronautics and Space Administration.

[13] Eike Best. Partial order verification with PEP. In Peled et al. [128], pages 305–328.

[14] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. Thèse no 2919, Université de Genève, Genève, Switzerland, July 1997.

[15] Jonathan Billington et al. High-level Petri nets—concepts, definitions and graphical notation, version 4.7.3. Final Draft International Standard ISO/IEC 15909, ISO/IEC JTC1/SC7, Genève, Switzerland, May 2002.

[16] Per Bjesse. *Gate Level Description of Synchronous Hardware and Automatic Verification Based on Theorem Proving*. PhD thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, May 2001.

[17] Eric Bruneton and Jean-François Pradat-Peyre. Automatic verification of Concurrent Ada programs. In Michael González Harbour and Juan A. de la Puente, editors, *Ada-Europe International Conference on Reliable Software Technologies*, volume 1622 of *Lecture Notes in Computer Science*, pages 146–157, Santander, Spain, June 1999. Springer-Verlag, Berlin, Germany.

[18] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[19] Peter Buchholz and Peter Kemper. Hierarchical reachability graph generation for Petri nets. *Formal Methods in System Design*, 21(3):281–315, November 2002.

[20] Cécile Bui Thanh, Hanna Klaudel, and Franck Pommereau. Petri nets with causal time for system verification. *Electronic Notes in Theoretical Computer Science*, 68(5), May 2003. CONCUR 2002 Satellite Workshops—Models for Time-Critical Systems.

[21] Ivana Černá and Radek Pelánek. Distributed explicit fair cycle detection (set based approach). In Ball and Rajamani [9], pages 49–73.

[22] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In Lori A. Clarke, Laura K. Dillon, and Walter Tichy, editors, *Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, Portland, OR, USA, May 2003. IEEE Computer Society Press, Los Alamitos, CA, USA.

[23] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: An industrial case study. In Will Tracz, Jeff Magee, and Michal Young, editors, *Proceedings of the 24th International Conference on Software Engineering*, pages 431–441, Orlando, FL, USA, May 2002. ACM Press, New York, NY, USA.

[24] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[25] Shing Chi Cheung and Jeff Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, January 1999.

[26] Giovanni Chiola and Alois Ferscha. Distributed simulation of timed Petri nets: Exploiting the net structure to obtain efficiency. In Marco Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993, 14th International Conference*, volume 691 of *Lecture Notes in Computer Science*, pages 146–165, Chicago, IL, USA, June 1993. Springer-Verlag, Berlin, Germany.

[27] Giovanni Chiola, Giuliana Franceschinis, Rossano Gaeta, and Marina Ribaudo. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1&2):47–68, November 1995. Special Issue on Performance Modeling Tools.

[28] Søren Christensen and Laure Petrucci. Modular state space analysis of coloured Petri nets. In Giorgio De Michelis and Michel Diaz, editors, *Application and Theory of Petri Nets 1995, 16th International Conference*, volume 935 of *Lecture Notes in Computer Science*, pages 201–217, Turin, Italy, June 1995. Springer-Verlag, Berlin, Germany.

[29] Søren Christensen and Laure Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.

[30] Tatyana Gennadievna Churina, Michail Yu. Mashukov, and Valery Alexandrovitch Nepomniaschy. Towards verification of SDL specified distributed systems: coloured Petri nets approach. In Ludwik Czaja, editor, *Proceedings of the CS&P 2001 Workshop*, pages 37–48, Warsaw, Poland, October 2001. Warsaw University.

[31] Gianfranco Ciardo, Robert L. Jones III, Andrew S. Miner, and Radu I. Siminiceanu. SMART: Stochastic model analyzer for reliability and timing. In Peter Kemper, editor, *Tools of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 29–34, September 2001.

[32] Duncan Clarke, Insup Lee, and Hong-Liang Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3(2):189–215, April 1995.

[33] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[34] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 185(2):187–243, August 2002.

[35] Jr. Clovis Seragiotto. Ladybug: a tool for dynamic detection of race conditions in Java programs. `http://www.par.univie.ac.at/~clovis/ladybug/`, July 2003.

[36] José-Manuel Colom and Maciej Koutny, editors. *Application and Theory of Petri Nets 2001, 22nd International Conference*, volume 2075 of *Lecture Notes in Computer Science*, Newcastle upon Tyne, UK, June 2001. Springer-Verlag, Berlin, Germany.

[37] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, January 2000.

[38] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM Press, New York, NY, USA.

[39] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for Petri net analysis. In Esparza and Lakos [51], pages 101–120.

[40] Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors. *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops, Trento, Italy, July 1999, Toulouse, France, September 1999. Proceedings*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1999.

[41] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171, Trento, Italy, July 1999. Springer-Verlag, Berlin, Germany.

[42] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In Dams et al. [40], pages 261–276.

[43] Jörg Desel. Model validation—a theoretical issue? In Esparza and Lakos [51], pages 23–43.

[44] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, NY, USA, 1968.

[45] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, USA, 1976.

[46] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, Cambridge, MA, USA, October 1992. IEEE Computer Society Press, Los Alamitos, CA, USA.

[47] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In Barton P. Miller and Charles E. McDowell, editors, *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, New York, NY, USA, May 1991.

[48] Danny Dolev, Maria Klawe, and Michael Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, September 1982.

[49] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, Los Angeles, CA, USA, May 1999. IEEE Computer Society Press, Los Alamitos, CA, USA.

[50] Matthew B. Dwyer, Robby, Xianghua Deng, and John Hatcliff. Space reductions for model checking quasi-cyclic systems. In Rajeev Alur and Insup Lee, editors, *EMSOFT 2003, 3rd International*

Conference on Embedded Software, volume 2855 of *Lecture Notes in Computer Science*, pages 173–189, Philadelphia, PA, USA, October 2003. Springer-Verlag, Berlin, Germany.

[51] Javier Esparza and Charles Lakos, editors. *Application and Theory of Petri Nets 2002, 23rd International Conference*, volume 2360 of *Lecture Notes in Computer Science*, Adelaide, Australia, June 2002. Springer-Verlag, Berlin, Germany.

[52] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A toolbox for the verification of LOTOS programs. In Tony Montgomery, editor, *Proceedings of the 14th International Conference on Software Engineering*, pages 246–259, Melbourne, Australia, May 1992. ACM Press, New York, NY, USA.

[53] Clemens Fischer. CSP-OZ: A combination of Object-Z and CSP. Technical Report TRCF-97-2, Universität Oldenburg, Fachbereich Informatik, Abteilung Semantik, Oldenburg, Germany, April 1997.

[54] Clemens Fischer and Heike Wehrheim. Model-checking CSP-OZ specifications with FDR. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 315–334, York, UK, June 1999. Springer-Verlag, Berlin, Germany.

[55] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In Ball and Rajamani [9], pages 213–224.

[56] Hans Fleischhack and Bernd Grahlmann. A compositional Petri net semantics for SDL. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998, 19th International Conference*, volume 1420 of *Lecture Notes in Computer Science*, pages 144–164, Lisbon, Portugal, June 1998. Springer-Verlag, Berlin, Germany.

[57] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990. Special Issue on Simulation.

[58] Hubert Garavel and John Hatcliff, editors. *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, volume 2619 of *Lecture Notes in Computer Science*, Warsaw, Poland, April 2003. Springer-Verlag, Berlin, Germany.

[59] Hartmann J. Genrich. Predicate/transition nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and their Properties—Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, Berlin, Germany, 1987. Bad Honnef, Germany, September 1986.

[60] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. *Formal Methods and System Design*, 7(3):1–15, November 1995.

[61] Donald I. Good, Richard M. Cohen, Charles G. Hoch, Lawrence W. Hunter, and Dwight F. Hare. Report on the language Gypsy: Version 2.0. Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, Institute for Computing Science and Computer Applications, The University of Texas at Austin, Austin, TX, USA, September 1978.

[62] Jean-Charles Grégoire. State space compression in SPIN with GETSs. In Grégoire et al. [63], pages 90–108.

[63] Jean-Charles Grégoire, Gerard J. Holzmann, and Doron Peled, editors. *Second SPIN Workshop*, Rutgers University, NJ, USA, August 1996.

[64] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[65] Torben Bisgaard Haagh and Tommy Rudmose Hansen. Optimising a coloured Petri net simulator. Master's thesis, University of Århus, Denmark, December 1994. `http://www.daimi.au.dk/CPnets/publ/thesis/HanHaa1994.pdf`.

[66] Klaus Havelund and Jens Ulrik Skakkebæk. Applying model checking in Java verification. In Dams et al. [40], pages 216–231.

[67] Ian J. Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, November 1989.

[68] Ian J. Hayes, Cliff B. Jones, and John E. Nicholls. Understanding the differences between VDM and Z. Technical Report UMCS-93-8-1, University of Manchester, Department of Computer Science, Manchester, UK, 1993.

[69] Keijo Heljanko, Viktor Khomenko, and Maciej Koutny. Parallelisation of the Petri net unfolding algorithm. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 371–385, Grenoble, France, April 2002. Springer-Verlag, Berlin, Germany.

[70] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In John Launchbury and John C. Mitchell, editors, *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, Portland, OR, USA, January 2002. ACM Press, New York, NY, USA.

[71] Charles Anthony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[72] Charles Anthony Richard Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, NJ, USA, 1985.

[73] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[74] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[75] Gerard J. Holzmann. From code to models. In Valmari and Yakovlev [153], pages 3–10.

[76] Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, April 2002.

[77] Peter Huber, Arne M. Jensen, Leif O. Jepsen, and Kurt Jensen. Reachability trees for high-level Petri nets. *Theoretical Computer Science*, 45(3):261–292, 1986.

[78] Nisse Husberg and Tapio Manner. Emma: developing an industrial reachability analyser for SDL. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 642–661, Toulouse, France, September 1999. Springer-Verlag, Berlin, Germany.

[79] Daniel C. Hyde. Introduction to the programming language Occam. `http://www.eg.bucknell.edu/~cs366/occam.pdf`, March 1995. Course material for CSCI 366 Parallel Computation, Bucknell University, Lewisburg, PA, USA.

[80] Standard for information technology—portable operating system interface (POSIX®). IEEE Std 1003.1-2001, Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, December 2001.

[81] Transmission control protocol. STD 7, Internet Engineering Task Force, September 1981.

[82] Cornelia P. Inggs and Howard Barringer. Effective state exploration for model checking on a shared memory architecture. *Electronic Notes in Theoretical Computer Science*, 68(4), October 2002. CONCUR 2002 Satellite Workshops—Parallel and Distributed Model Checking.

[83] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.

[84] Information technology—open systems interconnection—basic reference model: the basic model. ISO/IEC 7498-1:1994, International Organization for Standardization, Genève, Switzerland, 1994.

[85] Information technology—open systems interconnection—LOTOS —a formal description technique based on the temporal ordering of observational behaviour. ISO 8807:1989, International Organization for Standardization, Genève, Switzerland, 1989.

[86] Information technology—open systems interconnection—Estelle: a formal description technique based on an extended state transition model. ISO 9074:1997, International Organization for Standardization, Genève, Switzerland, 1997. Withdrawn May 1999.

[87] Information technology—Z formal specification notation—syntax, type system and semantics. ISO/IEC 13568:2002, International Organization for Standardization, Genève, Switzerland, 2002.

[88] Information technology—programming languages, their environments and system software interfaces—Vienna development method—specification language—Part 1: base language. ISO/IEC 13817-1:1996, International Organization for Standardization, Genève, Switzerland, 1996.

[89] Specification and description language (SDL). Recommendation Z.100 (08/02), International Telecommunication Union, Genève, Switzerland, September 2002.

[90] Hannu-Matti Järvinen and Reino Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.

[91] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 2$^{\text{nd}}$ corrected edition, 1997.

[92] Tommi Junttila. Symmetry reduction algorithms for data symmetries. Research Report A-72, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, Espoo, Finland, May 2002.

[93] Roope Kaivola. Equivalences, preorders and compositional verification for linear time temporal logic and concurrent systems. Report A-1996-1, University of Helsinki, Department of Computer Science, Helsinki, Finland, March 1996. PhD Thesis.

[94] Bengi Karaçalı and Kuo-Chung Tai. Model checking based on simultaneous reachability analysis. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 34–53, Stanford, CA, USA, August 2000. Springer-Verlag, Berlin, Germany.

[95] Shmuel Katz and Hillel Miller. Saving space by fully exploiting invisible transitions. *Formal Methods in System Design*, 14(3):311–332, May 1999.

[96] Pertti Kellomäki. Composing distributed systems from reusable aspects of behavior. In *Distributed Computing Systems Workshops, 22nd International Conference*, pages 481–486, Vienna, Austria, July 2002. IEEE Computer Society Press, Los Alamitos, CA, USA.

[97] Victor Khomenko and Maciej Koutny. Branching processes of high-level Petri nets. In Garavel and Hatcliff [58], pages 458–472.

[98] Ekkart Kindler and Hagen Völzer. Algebraic nets with flexible arcs. *Theoretical Computer Science*, 262(1–2):285–310, July 2001.

[99] Lars M. Kristensen and Thomas Mailund. A compositional sweep-line state space exploration method. In Peled and Vardi [129], pages 327–343.

[100] Lars M. Kristensen and Thomas Mailund. A generalised sweep-line method for safety properties. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right. International Symposium of Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 549–567, Copenhagen, Denmark, July 2002. Springer-Verlag, Berlin, Germany.

[101] Olaf Kummer. Simulating synchronous channels and net instances. In Jörg Desel, Peter Kemper, Ekkart Kindler, and Andreas Oberweis, editors, *5. Workshop Algorithmen und Werkzeuge für Petrinetze*, Forschungsbericht 694, pages 73–78, Dortmund, Germany, October 1998. Universität Dortmund, Fachbereich Informatik.

[102] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes—The Automata-Theoretic Approach.* Princeton University Press, Princeton, NJ, USA, 1994.

[103] Charles Lakos. Object oriented modelling with object Petri nets. In Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 1–37. Springer-Verlag, Berlin, Germany, 2001.

[104] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[105] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[106] Timo Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In Colom and Koutny [36], pages 242–262.

[107] Timo Latvala and Keijo Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.

[108] Kevin Lawton et al. The open source IA-32 emulation project. `http://bochs.sourceforge.net`, 1994–2002.

[109] Ranko Lazić and David Nowak. A unifying approach to data-independence. In Catuscia Palamidessi, editor, *CONCUR 2000—Concurrency Theory, 11th International Conference*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–596, University Park, PA, USA, August 2000. Springer-Verlag, Berlin, Germany.

[110] Sari Leppänen and Matti Luukkainen. Compositional verification of a third generation mobile communication protocol. In Ten-Hwang Lai, editor, *Proceedings of the 2000 ICDCS Workshops*, pages E118–E125, Taipei, Taiwan, April 2000.

[111] Glenn Lewis and Charles Lakos. Incremental state space construction for coloured Petri nets. In Colom and Koutny [36], pages 263–282.

[112] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001*, pages 192–203, Göteborg, Sweden, July 2001. IEEE Computer Society Press, Los Alamitos, CA, USA.

[113] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[114] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Sadek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, January 1995.

[115] Louise Lorentsen and Lars Michael Kristensen. Exploiting stabilizers and parallelism in state space generation with the symmetry method. In Valmari and Yakovlev [153], pages 211–220.

[116] Marko Mäkelä. Condensed storage of multi-set sequences. In Kurt Jensen, editor, *Practical Use of High-Level Petri Nets*, DAIMI report PB-547, pages 111–125. University of Århus, Denmark, June 2000.

[117] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[118] Kenneth L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, January 1995.

[119] Meta Software Corporation, Cambridge, MA, USA. *Design/CPN Reference Manual for X-Windows, Version 2.0*, 1993.

[120] Robin Milner. Synthesis of communicating behaviour. In Józef Winkowski, editor, *Mathematical Foundations of Computer Science, Proceedings, 7th Symposium*, volume 64 of *Lecture Notes in Computer Science*, pages 71–83, Zakopane, Poland, September 1978. Springer-Verlag, Berlin, Germany.

[121] Robin Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, NJ, USA, 1989.

[122] Daniel Moldt and Frank Wienberg. Multi-agent-systems based on coloured Petri nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997, 18th International Conference*, volume 1248 of *Lecture Notes in Computer Science*, pages 82–101, Toulouse, France, June 1997. Springer-Verlag, Berlin, Germany.

[123] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[124] David M. Nicol and Gianfranco Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47(2):153–167, December 1997.

[125] OMG Unified Modeling Language. Specification v1.4, formal/01-09-67, Object Management Group, Needham, MA, USA, September 2001.

[126] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[127] Andrew Page, Thomas Keane, Richard Allen, Thomas J. Naughton, and John Waldron. Multi-tiered distributed computing platform. In *Proceedings of the Second International Conference on the Principles and Practice of Programming in Java*, pages 191–194, Kilkenny City, Ireland, June 2003. Department of Computer Science, National University of Ireland.

[128] Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors. *Partial Order Methods in Verification: DIMACS Workshop, July 24–26, 1996*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Princeton University, NJ, USA, 1997. American Mathematical Society, Providence, RI, USA.

[129] Doron A. Peled and Moshe Y. Vardi, editors. *Formal Techniques for Networked and Distributed Systems—FORTE 2002, 22nd IFIP WG 6.1 International Conference*, volume 2529 of *Lecture Notes in Computer Science*, Houston, TX, USA, November 2002. Springer-Verlag, Berlin, Germany.

[130] Giuseppe Della Penna, Benedetto Intrigila, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in the disk based Murφ verifier. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design, 4th International Conference*, volume 2517 of *Lecture Notes in Computer Science*, pages 202–219, Portland, OR, USA, November 2002. Springer-Verlag, Berlin, Germany.

[131] Laure Petrucci. *Modélisation, vérification et applications.* Mémoire d'habilitation à diriger des recherches, Université d'Évry, Évry, France, December 2002.

[132] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets.* Springer-Verlag, Berlin, Germany, 1998.

[133] Michael J. Sanders. Constraint programming with object-oriented Petri nets. In *1998 IEEE International Conference on Systems, Man, and Cybernetics*, pages 289–294, San Diego, CA, USA, October 1998.

[134] Michael J. Sanders. Efficient computation of enabled transition bindings in high-level Petri nets. In *2000 IEEE International Conference on Systems, Man, and Cybernetics*, pages 3153–3158, Nashville, TN, USA, October 2000.

[135] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[136] Karsten Schmidt. LoLA: A low level analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000, 21st International Conference*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474, Århus, Denmark, June 2000. Springer-Verlag, Berlin, Germany.

[137] Karsten Schmidt. Distributed verification with LoLA. *Fundamenta Informaticae*, 54(2–3):253–262, February 2003.

[138] Julian Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. `http://developer.kde.org/~sewardj/`, May 2003.

[139] Vivek K. Shanbhag, K. Gopinath, Markku Turunen, Ari Ahtiainen, and Matti Luukkainen. EASN: Integrating ASN.1 and model checking. In Gérard Berry, Hubert Comon, and Alain Finkel, editors,

Computer Aided Verification, 13th International Conference, CAV 2001, volume 2102 of *Lecture Notes in Computer Science*, pages 382–386, Paris, France, July 2001. Springer-Verlag, Berlin, Germany.

[140] Robert M. Shapiro, Jawahar Malhotra, Kurt Jensen, Søren Christensen, and Peter Huber. Computer-aided generation of programs modelling complex systems using colored Petri nets. United States Patent 5,257,363, Meta Software Corporation, Cambridge, MA, USA, October 1993. Filed April 9, 1990.

[141] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, January 2000.

[142] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):133–166, April 2000.

[143] Harry Sneed and Siegfried Göschl. Testing software for Internet applications. *Software Focus*, 1(1):15–22, September 2000.

[144] Peter H. Starke. Reachability analysis of Petri nets using symmetries. *Systems Analysis Modelling Simulation*, 8(4/5):293–303, 1991.

[145] Ulrich Stern and David L. Dill. A new scheme for memory-efficient probabilistic verification. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI*, volume 69 of *IFIP Conference Proceedings*, pages 333–348, Kaiserslautern, Germany, October 1996. Chapman & Hall, London, UK.

[146] Ulrich Stern and David L. Dill. Parallelizing the Murφ verifier. *Formal Methods in System Design*, 18(2):117–129, March 2001.

[147] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[148] Heikki Tuominen. Embedding a dialect of SDL in Promela. In Dams et al. [40], pages 245–260.

[149] Teemu Tynjälä, Sari Leppänen, and Vesa Luukkala. Verifying reliable data transmission over UMTS radio interface with high level Petri nets. In Peled and Vardi [129], pages 178–193.

[150] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, Berlin, Germany, 1998.

[151] Antti Valmari et al. Tampere Verification Tool. `http://www.cs.tut.fi/ohj/VARG/TVT/`, 2000–2003.

[152] Antti Valmari and Manu Setälä. Visual verification of safety and liveness. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*, pages 228–247, Oxford, UK, March 1996. Springer-Verlag, Berlin, Germany.

[153] Antti Valmari and Alex Yakovlev, editors. *2nd International Conference on Application of Concurrency to System Design*, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society Press, Los Alamitos, CA, USA.

[154] Peter van Beek. An extensive library of routines for experimenting with different backtracking methods for solving binary CSPs. `http://ai.uwaterloo.ca/~vanbeek/software/csplib.tar.gz`, April 1994.

[155] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. PROD reference manual. Technical Report B-13, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, August 1995. See also `http://www.tcs.hut.fi/Software/prod/`.

[156] Willem Visser. Memory efficient storage in SPIN. In Grégoire et al. [63], pages 21–35.

[157] Willem Visser, Klaus Havelund, Guillaume Brat, and Seungjoon Park. Model checking programs. In Perry Alexander and Pierre Flener, editors, *International Conference on Automated Software Engineering*, pages 3–11. IEEE Computer Society Press, Los Alamitos, CA, USA, September 2000.

[158] Georg Weißenbacher. An abstraction/refinement scheme for model checking C programs. Diplomarbeit in Telematik, IST—Institut für Softwaretechnologie der Technischen Universität Graz, Graz, Austria, March 2003.

[159] Colin H. West. Applications and limitations of automated protocol validation. In Carl A. Sunshine, editor, *IFIP WG6.1 Second International Workshop on Protocol Testing, Specification and Verification*, pages 361–371, Idyllwild, CA, USA, May 1982. North-Holland Publishing Company, Amsterdam, The Netherlands.

[160] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification*, pages 49–59, Victoria, British Columbia, October 1991. ACM Press, New York, NY, USA.

A CORRECTIONS TO PUBLICATIONS

[P1]:
- In Figure 3, the test $k = n$ should be $k > n$ or $k = n + 1$.
- In Section 4.2, the summand in the definition of the secondary cost function $c_2(S_k)$ should be $[X_j \neq \emptyset]$ instead of $[X_k \neq \emptyset]$.

[P3]: In Algorithms 1 and 3, S should be initialised to $S = \{s_0\}$ instead of $S = \emptyset$.

B ALGORITHM LISTINGS

B.1 Sequential State Space Enumeration

```
/* Enumerate states reachable from $s_0$ */
void verify($s_0$) {
    Set $S = \{s_0\}, Q = \{s_0\}$;
    /* Report a reachable state (call-back procedure) */
    void reportState($s, t$) {
        if (error($s$))
            reportError($s$, “erroneous state”);
        else if ($s \notin S$)
            $S = S \cup \{s\}, Q = Q \cup \{s\}$;
    }
    while ($Q \neq \emptyset$) {
        let $s \in Q; Q = Q \setminus \{s\}$; successors($s$, reportState);
    }
}
```

B.2 Parallel State Space Enumeration

```
/* Client: compute successor states */
void client($server$) {
    Set $S$;
    /* Report a reachable state (call-back procedure) */
    void reportState($s, t$) {
        if (error($s$)) reportError($s$, “erroneous state”);
        else $S$.add($s$);
    }
    State $s$;
    while ($null \neq (s = server$.getState())) {
        $S = \emptyset$; successors($s$, reportState);
        $server$.putStates($S$);
    }
}
/* Server: enumerate states reachable from $s_0$ */
void server() {
    Set $S = \{s_0\}, Q = \{s_0\}$;
    while (not all clients block in getState)
        serve remote procedure invocations;
}
/* Server: assign an unprocessed state to a client */
remote procedure State getState() {
    wait until $Q \neq \emptyset$; let $s \in Q, Q = Q \setminus \{s\}$; return $s$;
} /* returns null if all clients are waiting here */
/* Server: receive successor states from a client */
remote procedure void putStates($S'$) {
    $Q = Q \cup (S' \setminus S); S = S \cup S'$;
}
```

B.3 Generating Successor States in a High-Level Petri Net

```
/* Report all successors of the given marking */
void successors($M, rep$) {
    for each transition $t$ {
        analyse_arcs($M, t, EmptyValuation, 0, rep$);
    }
}
/* Analyse the remaining input arcs */
void analyse_arcs($M, t, x, k, rep$) {
    if ($k \geq t.numInputs$)
        $rep(M + \mathsf{eval}(t.outputs, x), t)$;
    else if ($t.variables[k] \neq \emptyset$)
        analyse_variable($M, t, x, k, rep$);
    else
        analyse_constant($M, t, x, k, rep$);
}
/* Evaluate an input arc expression */
void analyse_constant($M, t, x, k, rep$) {
    Marking $m$ = eval($t.inputs[k], x$);
    if (an error occurred in eval)
        reportError($M$, “undefined input arc”, $t, x$);
    else if ($m \leq M$)
        analyse_arcs($M - m, t, x, k + 1, rep$);
}
/* Process an input arc with bindable variables */
void analyse_variable($M, t, x, k, rep$) {
    int $c$ = eval_multiplicity($t.inputs[k], x$);
    if ($c > 0$)
        bind_variables($M, t, x, k, rep, c$);
    else if ($c \neq 0$)
        reportError($M$, “undefined multiplicity”, $t, x$);
    else {
        $t.unified[k] = \emptyset$;
        analyse_arcs($M, t, x, k + 1$);
    }
}
/* Bind variables from an input arc */
void bind_variables($M, t, x, k, rep, c$) {
    for each $m$ such that $c`m \leq M[t.inplace[k]]$) {
        Valuation $x' = x$;
        for each variable $v \in t.variables[k]$
            $x'[v]$ = candidate($m, t.inputs[k], v$);
        $t.unified[k] = c`m$;
        if (enabled($t.guard, x'$))
            if ($\bigwedge_{j=0}^{k}$ compatible($t.unified[j], t.inputs[j], x'$))
                analyse_arcs($M - t.unified[k], t, x', k + 1, rep$);
    }
}
```

B.4 Modular State Space Enumeration

```
/* Enumerate states reachable from $s_0$ in module */
/*1*/  void explore($module, s_0, \mathcal{S}$) {
/*2*/     Set $S = \{s_0\}, Q = \{s_0\}$;
/*3*/     while ($Q \neq \emptyset$) {
/*4*/        let $s \in Q, Q = Q \setminus \{s\}$;
/*5*/        transitions($module, s, S, Q, \mathcal{S}$);
/*6*/        modules($module, s, S, Q$);
/*7*/     }
/*8*/  }
/* Enumerate locally reachable states from s in module */
 /*1*/  void transitions($module, s, S, Q, \mathcal{S}$) {
 /*2*/     /* Report a reachable state (call-back procedure) */
 /*3*/     void reportState($s', t$) {
 /*4*/        for each sync label $tf$ of $t$
 /*5*/           $\mathcal{S} = \mathcal{S} \cup \{(module, tf, s)\}$;
 /*6*/        if $t$ has no sync label
 /*7*/           report($s', S, Q$);
 /*8*/     }
 /*9*/     $module$.successors($s$, reportState);
/*10*/  }
/* Explore the modules of m from s */
/*1*/  void modules($m, s, S, Q$) {
/*2*/     Set $\mathcal{S} = \emptyset$;
/*3*/     for each $module \in C(m)$
/*4*/        explore($module, s_{module}, \mathcal{S}$);
/*5*/     for each sync transition $tf$ of $m$
/*6*/        sync($m, s, S, Q, \mathcal{S}, tf$);
/*7*/  }
/* Compute the synchronisations on tf */
/*1*/  void sync($m, s, S, Q, \mathcal{S}, tf$) {
/*2*/     for each $\bigcup_{\substack{module \in C(m) \\ \text{synchronising on } tf}} \{(module, tf, s^{module})\} \subseteq \mathcal{S}$ {
/*3*/        State $s^* = s$;
/*4*/        for each $module \in C(m)$ synchronising on $tf$
/*5*/           $s^*_{module} = s^{module}$;
/*6*/        for each $s'$ such that $s^* \xrightarrow{tf} s'$
/*7*/           report($s', S, Q$);
/*8*/     }
/*9*/  }
/* Report a reachable state */
/*1*/  void report($s, S, Q$) {
/*2*/     if (error($s$))
/*3*/        reportError($s$, “erroneous state”);
/*4*/     else if ($s \notin S$)
/*5*/        $S = S \cup \{s\}, Q = Q \cup \{s\}$;
/*6*/  }
```

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A68	Javier Esparza, Keijo Heljanko Implementing LTL Model Checking with Net Unfoldings. March 2001.
HUT-TCS-A69	Marko Mäkelä A Reachability Analyser for Algebraic System Nets. June 2001.
HUT-TCS-A70	Petteri Kaski Isomorph-Free Exhaustive Generation of Combinatorial Designs. December 2001.
HUT-TCS-A71	Keijo Heljanko Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets. February 2002.
HUT-TCS-A72	Tommi Junttila Symmetry Reduction Algorithms for Data Symmetries. May 2002.
HUT-TCS-A73	Toni Jussila Bounded Model Checking for Verifying Concurrent Programs. August 2002.
HUT-TCS-A74	Sam Sandqvist Aspects of Modelling and Simulation of Genetic Algorithms: A Formal Approach. September 2002.
HUT-TCS-A75	Tommi Junttila New Canonical Representative Marking Algorithms for Place/Transition-Nets. October 2002.
HUT-TCS-A76	Timo Latvala On Model Checking Safety Properties. December 2002.
HUT-TCS-A77	Satu Virtanen Properties of Nonuniform Random Graph Models. May 2003.
HUT-TCS-A78	Petteri Kaski A Census of Steiner Triple Systems and Some Related Combinatorial Objects. June 2003.
HUT-TCS-A79	Heikki Tauriainen Nested Emptiness Search for Generalized Büchi Automata. July 2003.
HUT-TCS-A80	Tommi Junttila On the Symmetry Reduction Method for Petri Nets and Similar Formalisms. September 2003.
HUT-TCS-A81	Marko Mäkelä Efficient Computer-Aided Verification of Parallel and Distributed Software Systems. November 2003.

ISBN 951-22-6791-8
ISBN 951-22-6792-6 (PDF)
ISSN 1457-7615