# 1

# Introduction

In the context of hardware and software systems, formal verification is the act of proving or disproving a property of a system with respect to a formal specification, using methods rooted in mathematics, such as logic and graph theory. A formal specification of a system can help to obtain not only a better (more modular) description, but also a better understanding and a more abstract view of the system. Formal verification, supported with (semi-)automated tools, can detect errors in the design that are not easily found using testing, and can be used to establish the correctness of the design. Formal verification has, for instance, been applied to communication and cryptographic protocols, distributed algorithms, combinatorial circuits, and software expressed as source code. A comprehensive overview of the field of formal verification can be found in [86].

Process algebra focuses on the specification and manipulation of process terms as induced by a collection of operator symbols. Such a process term constitutes a formal specification of a system. Typically, process algebras contain action names, to express atomic events, and the two basic operators alternative and sequential composition to build finite processes. Recursion allows one to capture infinite behaviour.

Verifying the correctness of distributed systems is a challenge, due to their inherent parallelism. In order to study the behaviour of distributed systems in detail, it is imperative that they are dissected into their concurrent components. Fundamental to process algebra is therefore a parallel operator, to break down distributed systems into their concurrent components, at the same time expressing the communication of corresponding send and receive events at different components. An encapsulation operator takes care that such corresponding send and receive events can only occur in synchronisation. Finally, a hiding operator allows one to abstract away from the resulting communication events, and from the internal events at a component.

In process algebras, each operator in the language is given meaning through a characterising set of equations, called axioms. If two process terms (built from the aforementioned operators) can be equated by means of the axioms, then they represent equivalent system behaviours. Thus the axioms form an elementary basis for equational reasoning about processes. Process algebras such as CCS [22, 81], CSP [64, 93] and ACP [10, 6, 41] offer an excellent framework for the description of distributed systems, and they are well equipped for the study of their behavioural properties. Temporal logics can be used to formally express such properties.

System behaviour generally consists of a mix of processes and data. Processes are the control mechanisms for the manipulation of data. While processes are dynamic and active, data are static and passive. In algebraic specification [72], each data type is defined by declaring a collection of function symbols, from which one can build data terms, together with a set of axioms, saying which data terms are equal. Algebraic specification allows one to give relatively simple and precise definitions of abstract data types. A major advantage of this approach is that it is easily explained and formally defined, and that it constitutes a uniform framework for defining general data types. Moreover, all properties of a data type must be denoted explicitly, and henceforth it is clear which assumptions can be used when proving properties about data or processes. Term rewriting [99] provides a straightforward method for implementing algebraic specifications of abstract data types. Concluding, as long as one is interested in clear and precise specifications, and not in optimised implementations, algebraic specification is the best available method. However, one should be aware that it does not allow one to conveniently use high-level constructs for compact specification of complex data types, nor optimisations supporting fast computation (such as decimal representations of natural numbers).

Process algebras tend to lack the ability to handle data. In case data become part of a process theory, one often has to resort to infinite sets of axioms where variables are indexed with data values. In order to make data a first class citizen in the formal specification of systems, the language $\mu$CRL [54] has been developed. Basically, $\mu$CRL is based on the process algebra ACP, extended with the algebraic specification of abstract data types. In order to intertwine processes with data, the action names and recursion variables that are used to express process behaviour can be parametrised with data types. Moreover, a conditional (if-then-else) construct can be used to let data elements influence the course of a process, and the alternative composition operator is allowed to range over possibly infinite data domains. Despite its lack of 'advanced' features, $\mu$CRL has been shown to be remarkably apt for the description of real-life distributed systems.

A proof theory for $\mu$CRL has been developed [53], based in part on the axiomatic semantics of the process algebra ACP and on some basic abstract data types. This proof theory, in combination with proof methods that were developed in e.g. [14, 58], has enabled the verification of distributed systems in a precise and logical way, which is slowly turning into a routine. Theorem provers such as PVS [83], Isabelle/HOL [82] and Coq [12] are being used to help in finding and checking derivations in $\mu$CRL. A considerable number of distributed systems from the literature and from industry have been verified in $\mu$CRL, e.g. [26, 49, 84, 94, 100], often with the help of a theorem prover, e.g. [5, 13, 50]. Typically, these verifications lead to the detection of a number of mistakes in the specification of the system under scrutiny, and the support of theorem provers helps to detect flaws in the correctness proof, or even in the statement of correctness.

To each $\mu$CRL specification there belongs a directed graph, called the state space, in which the states are process terms, and the edges are labelled with actions. In this state space, an edge $p \overset{a(d)}{\to} p'$ means that process term $p$ can perform action $a$, parametrised with datum $d$, to evolve into process term $p'$. If the state space belonging to a $\mu$CRL specification is finite, then the $\mu$CRL toolset [17], in combination with the CADP toolset [43], can generate and visualise this state space. Model checking [32] provides a framework to efficiently prove interesting properties of large state spaces, formulated in some temporal logic. While the process algebraic proofs that were discussed earlier can cope with an open environment, such as an unspecified data type or network topology, the generation of a state space belonging to a distributed system requires that the environment is given in full detail. This means that for instance each unspecified data type (typically, the set of objects that can be received by the distributed system from the ‘outside world’) has to be instantiated with an ad hoc finite collection of elements, and that a particular configuration of the network topology has to be chosen.

A severe complication in the generation of state spaces is that, in real life, a distributed system typically contains in the order of $2^{100}$ states or more. In that sense a $\mu$CRL specification is like Pandora’s Box; as soon as it is opened, the state space may explode. This means that generating, storing and analysing a state space becomes problematic, to say the least. Several methods are being developed to tackle large state spaces. Distributed state space generation and verification algorithms make it possible to store a state space on a number of processors, and analyse it in a distributed fashion [16]. On-the-fly analysis [65] allows one to generate only part of a state space. Structural symmetries in the description of a system can often be exploited to reduce the resulting state space [31]. Scenario-based verification [36] takes as its starting point a certain scenario of inputs from the outside world, to restrict the behavioural possibilities of a distributed system. A $\mu$CRL specification may be manipulated in such a way that the resulting state space becomes significantly smaller [47]. And the ATerm library [21] allows one to store state spaces in an efficient way by maximal sharing, meaning that if two states (i.e., two process terms) contain the same subterm, then this subterm is shared in the memory space.

This text is set up as follows. Chapter 2 gives an introduction into the algebraic specification of abstract data types. Chapter 3 provides an overview of process algebra, and presents the basics of the specification language $\mu$CRL. In Chap. 4 it is explained how one can abstract away from the internal and communication events of a process. Chapter 5 contains a number of $\mu$CRL specifications of network protocols from the literature, together with extensive explanations to guide the reader through these specifications. In Chap. 6 it is explained how a $\mu$CRL specification can be reduced to a linear form, from which a state space can be generated. Also some process algebraic techniques are described that can be applied to such linear forms. Chapter 7 describes verification algorithms on state spaces. In Chap. 8, techniques are presented to analyse $\mu$CRL specifications on a symbolic level. Also a symbolic verification of the tree identify protocol is presented. Finally, Appendix A contains a brief explanation on how to use the $\mu$CRL and CADP toolsets.