Digital Comprehensive Summaries of Uppsala Dissertations
from the Faculty of Science and Technology 2605

Types and Time

Languages, Tools, and Methods for Reliable Systems Engineering

NIKOLAUS HUBER

ACTA UNIVERSITATIS
UPSALIENSIS
2025

ISSN 1651-6214
ISBN 978-91-513-2653-5
urn:nbn:se:uu:diva-570474

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in Ångströmlaboratoriet, 101121, Sonja Lyttkens, Lägerhyddsvägen 1, Uppsala, Monday, 15 December 2025 at 09:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Timothy Bourke (École polytechnique).

Abstract

Huber, N. 2025. Types and Time: Languages, Tools, and Methods for Reliable Systems Engineering. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 2605. 57 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-2653-5.

Software plays a vital role across many areas of modern society. The personal gadgets we carry with us every day, the medical devices many people rely on, and the massive city-spanning systems that keep our critical infrastructure up and running are all powered by software, which is at the heart of every modern computing device. Naturally, we are therefore interested in ensuring that its execution is as error-free as possible, as errors might lead to significant financial loss or even endanger human life. To that end, this dissertation presents work aimed at helping design, implement, and verify reliable software systems.

The first part of this dissertation presents a design for a programming language for the development of embedded systems, called Mimosa. Mimosa builds upon MIMOS, a novel model of computation that allows for describing an overall software system as a graph of computation nodes. As this model enjoys both functional and timing determinism, Mimosa is particularly suited for the development of software for real-time systems. The presented work covers formal semantics, simulation, and compilation to a set of tasks running on a real-time operating system.

The second part of this dissertation presents broader work in the realm of reliable software engineering, targeting, in particular, software systems developed in the programming language OCaml. First, a tool for dynamic verification of stateful OCaml programs is presented. The primary focus of the tool lies on the automatic extraction of a functional model from formal contracts provided as part of a module's interface. Second, a method for encoding interaction nets, a particular form of graph rewriting, in OCaml is presented. Particular properties of interaction nets enable automatic scaling of the rewriting process across multiple processing cores. Compared to previous work, the presented encoding focuses on correctness guarantees provided by the interaction net type system, which is encoded in the type system of OCaml. It is shown that certain algorithms can exploit the inherent parallelism of the rewriting process, and therefore automatically utilise OCaml's recently added native support for parallel evaluation.

Keywords: embedded systems, real-time systems, programming language design, testing, dynamic verification, OCaml, graph rewriting, interaction nets

Nikolaus Huber, Department of Information Technology, Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-2653-5
URN urn:nbn:se:uu:diva-570474 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-570474)

To my family

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **Huber, N.**, Graf, S., Rümmer, P., Yi, W. (2025). Mimosa: A Language for Asynchronous Implementation of Embedded Systems Software. In: *Di Giusto, C., Ravara, A. (eds) Coordination Models and Languages*. COORDINATION 2025. Lecture Notes in Computer Science, vol 15731. Springer, Cham.
https://doi.org/10.1007/978-3-031-95589-1_5

II **Huber, N.**, Graf, S., Rümmer, P., Yi, W. (2025). Compiling the Mimosa Programming Language to RTOS Tasks.
Technical Report
https://doi.org/10.48550/arXiv.2510.20547

III **Huber, N.**, Spargo, N., Osborne, N., Hym, S., Midtgaard, J. (2025). Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM.
In: *Gurfinkel, A., Heule, M. (eds) Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2025. Lecture Notes in Computer Science, vol 15698. Springer, Cham.
ETAPS Distinguished Paper & ETAPS Best Tool Paper 2025
https://doi.org/10.1007/978-3-031-90660-2_1

IV **Huber, N.**, Yi, W. (2024). An Encoding of Interaction Nets in OCaml. In: *Endrullis, J., Grzelak, D., Heindel, T., Kosiol, J. (eds) Proceedings of the Fourteenth and Fifteenth International Workshop on Graph Computation Models*. (GCM 2023 and 2024). Electronic Proceedings in Theoretical Computer Science 417
https://dx.doi.org/10.4204/EPTCS.417.1

Reprints were made with permission from the publishers.

Summary of Contributions

The Roman numerals correspond to the numbers in the list of papers.

I Sole contributor to the Mimosa language design, including its formal semantics. Sole contributor to the paper idea, theoretical design, and artifact implementation. Significant contributions to the writing and conclusions.

II Sole contributor to the paper idea, theoretical design, and implementation. Main contributor to the paper writing and conclusions.

III Contributor to ORTAC/QCheck-STM as part of an internship placement at Tarides, France. Responsible for the design and implementation of various extensions to the tool. Design of the evaluation, including the implementation of the GOSPEL specifications for most of the tested modules. Responsible for creating the accompanying software artifact. Significant contributions to the writing and conclusions.

IV Sole contributor to the paper idea, theoretical design, implementation, paper writing, and conclusions.

Related Work

In addition to the papers included in this thesis, the author has also contributed to the following works:

Associated Acts (peer-reviewed)

R1 Graf, S., Jonsson, B., Khodabandeloo, B., Huang, C., **Huber, N.**, Rümmer, P., Yi, W. (2025). Timing is All You Need.
In: *Hinchey, M., Steffen, B. (eds) The Combined Power of Research, Education, and Dissemination*. Lecture Notes in Computer Science, vol 15240. Springer, Cham.
https://doi.org/10.1007/978-3-031-73887-6_18

Software Artifacts (peer-reviewed)

C1 **Huber, N.** (2025). The Mimosa Language - Software Artifact (v0.1). Zenodo. (Paper I accompanying artifact)
https://doi.org/10.5281/zenodo.14963241

C2 **Huber, N.**, Osborne, N., Hym, S., Midtgaard, J. (2024). Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM - Software Artifact (v0.1). Zenodo. (Paper III accompanying artifact)
https://doi.org/10.5281/zenodo.13988146

Software Artifacts (non peer-reviewed)

C3 **Huber, N.** (2024). An Encoding of Interaction Nets in OCaml - Software Artifact (v1.0.0). Zenodo. (Paper IV accompanying artifact)
https://doi.org/10.5281/zenodo.12633477

Contents

1. Introduction

> *The most important property of a program is whether it accomplishes the intention of its user.*
>
> — C.A.R. Hoare [49]

It is an undeniable fact that we find ourselves in a world where technology contributes to almost every aspect of our modern lives. This trend is visible in many different areas:

Our workplaces are shaped by the computerised machines we use, whether it be the personal computers we utilise in our offices, the robots we employ to help us perform manual labour, or even the use of artificial intelligence to eliminate tedious work tasks completely.

Technology has had and continues to have a significant impact on modern healthcare. Personal health monitors, such as smartwatches, which we carry with us almost constantly, provide more profound insights into a person's health-related habits and enable the detection of various pathological patterns early on. The data collected through these devices is also increasingly used by doctors when assessing a person's health. For example, the U.S. Food and Drug Administration (FDA) has cleared several of these devices to detect different forms of cardiac problems, including atrial fibrillation. Fall and accident detection, coupled with automated emergency alerts, can provide peace of mind to the elderly and may allow them to live outside of care-taking facilities for longer. Medical devices, such as pacemakers, blood glucose measurement and management devices, and even artificial organs and limbs, have a direct impact on the quality of life for people who rely on these technologies. Healthcare providers nowadays have access to patients' connected health-related data, which they can utilise for more holistic diagnoses and an overall better treatment plan. Finally, the tools and methods used by medical professionals are significantly influenced by modern technology. For example, advancements in micro-robotics have found applications in surgical equipment, enabling less invasive procedures.

Transportation and public infrastructure have undergone significant changes through technological advancements as well. Our cars are now frequently equipped with advanced computing devices, offering road safety monitoring, driving assistance, and increasingly even self-driving capabilities. Aircraft have shifted to fly-by-wire technology, which replaces conventional manual flight controls, resulting in significant weight savings. The cities we live in

now often incorporate various forms of advanced technologies, such as smart power grids and fully connected, automated waste management systems.

These are just a few examples of how technology shapes our modern lives. In all of these usage scenarios, software naturally plays an important role. Given the importance of software, errors observed during its execution can lead to catastrophic consequences, and sources for errors are plentiful.

The possibility of such consequences is, unfortunately, documented in various incidents. The Therac-25 radiation therapy machine was responsible for at least five deaths in the 1980s by delivering lethal radiation dosages due to a software bug [71]. Similarly, an error in a radiation therapy planning program, resulting in occasionally doubled dosages, killed at least eight patients, while causing significant health problems in at least another 20 [57]. These catastrophic incidents, among others, have led to the creation of an international standard for the development of medical device software [58]. In 1994, a Royal Air Force Chinook helicopter crashed on the Mull of Kintyre, Scotland, killing all 29 people on board. While the incident was initially attributed to pilot error, later investigations revealed that a software error in the aircraft's engine control computer may have been responsible [50]. Both the Boeing 787 Dreamliner and the Airbus A350 occasionally experienced an integer overflow, which caused both planes to shut down essential equipment if they were not powered down regularly [1, 38]. These are just a few examples of various documented incidents where a software error led to the loss of, or at least endangered, human life.

Another specific class of errors can occur when software interacts with the physical world. Aircraft must manoeuvre their control surfaces at the right time to achieve the desired attitude, and an implanted cardiac pacemaker must deliver electrical signals to the heart at the proper intervals to regulate the flow of blood through the body. For these kinds of devices, the output of a computation must not only be numerically correct, but it must also be available at the right time. Therefore, they are often referred to as *real-time systems*.

Naturally, our dependence on software for critical applications has led to research efforts from various research communities, all trying to answer the question of how this software should be developed to minimise the chances of errors. Research in real-time systems focuses on temporal behaviours, while *software verification* concentrates on the verification of functional aspects of programs. The kind of systems we are interested in within this dissertation needs both.

As software engineering is a complex process, errors can occur at various phases of the development life cycle. Therefore, the contributions of this dissertation target different parts of the development process. The *V-model* [76] is an established process model for developing critical software systems, highlighting, in particular, the relationship between efforts performed before and after implementation. The model shown in Figure 1.1 is a simplification of the full V-model, and shows which phases are targeted by each paper.

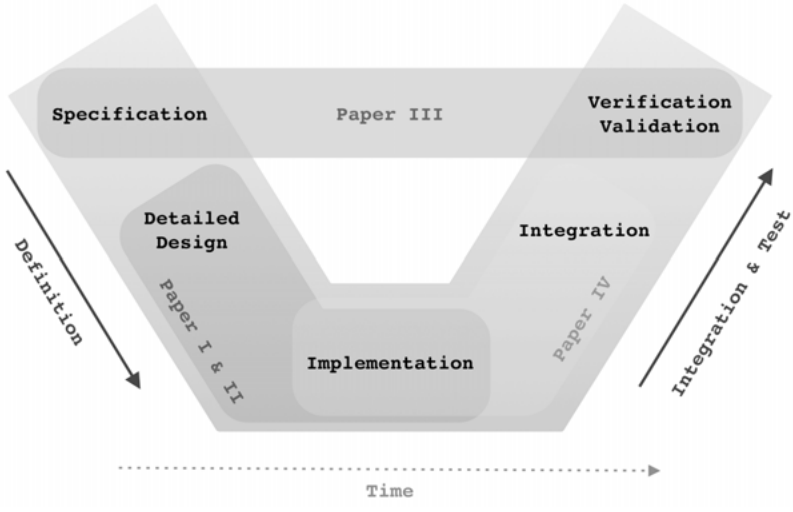


Figure 1.1. Paper contributions in different phases of the simplified V-model

Papers I and II introduce the Mimosa language, a new programming language specifically designed for the development of real-time systems. It builds upon a novel model of computation, called MIMOS [101]. Paper I introduces the syntax and semantics of the language, while Paper II describes a compilation routine to Real-Time Operating System (RTOS) tasks. MIMOS encourages the development of complex software systems as a network of computation nodes, and can therefore be used in the detailed design of a system.

Paper III showcases our work on ORTAC/QCheck-STM, a tool for dynamic verification of stateful OCaml programs. It works by generating verification conditions from specifications on model interfaces. It therefore concerns both the specification and verification phases of the software development life cycle, as shown in Figure 1.1.

Finally, our work on interaction nets in Paper IV provides a method for implementing parallel algorithms that allows for automatic scaling across a set of processing units. While it primarily concerns the implementation phase, it also has an impact on the integration of software components, as the unique properties of interaction nets prevent certain errors from occurring during parallel evaluation.

The outline of this dissertation is as follows. Following this introduction, Chapter 2 introduces some of the technical background that is relevant to this work. Chapter 3 presents the contributions of this dissertation. Chapter 4 illustrates future research directions. Finally, conclusions and perspectives of this work are discussed in Chapter 5.

2. Background

If I have seen further, it is by standing on the shoulders of giants.

— Isaac Newton

This chapter introduces some of the necessary background relevant to this dissertation. Section 2.1 introduces the kind of systems we are primarily targeting, while Section 2.2 gives an overview of one of the prevalent approaches for implementing these systems. Section 2.3 illustrates the MIMOS model of computation, which builds the basis for our work in Papers I and II. Section 2.4 introduces *dynamic verification*, and in particular *testing*, as the foundation for our work on verifying stateful OCaml programs in Paper III. A brief introduction into *graph rewriting* and *interaction nets* is given in Sections 2.5 and 2.6. Finally, Section 2.7 introduces *generalised algebraic data-types*, which builds the foundation for our work in Paper IV.

2.1 Real-Time Embedded Systems

There are various terms in use today for the kinds of systems we are targeting in this dissertation, which usually highlight particular characteristics that distinguish them from other computing systems. In this section, we introduce some of these terms.

Harel and Pnueli [45] introduced the term *reactive system*, which describes a computing system that interacts continuously with its environment. The authors use the term *reactive* as a distinction from *transformational systems*, which only react to particular user inputs (e.g., compilers). Halbwachs [44] later refines these definitions, distinguishing between *interactive* and *reactive* systems. In his nomenclature, an interactive system can synchronise with its environment, i.e., make it wait for a response. Examples of such systems include operating system schedulers, databases, and web servers. In contrast, the environment of a reactive system cannot wait for a response, so typical reactive systems are those that interact with physical processes (e.g., aircraft control systems). Particular features expected of reactive systems have been described in the literature [10, 12, 43], and typically include concurrency, security, safety, functional and temporal determinism, and increased tolerance to faults.

Cyber-Physical Systems (CPSs) highlight the tight integration between their control algorithms and their physical components. The term is more commonly used for networked devices, including the *Internet of Things* (IoT). Distinguishing qualities of a CPS often entail a distributed, networked architecture, adaptability, autonomy, and various dimensions of efficiency (e.g., power, latency, and memory). CPSs also span a broad range of sizes, from small handheld devices to systems spanning entire cities or even countries, such as *smart grids* [20].

The term *embedded system* is usually used more broadly to describe specialised computing devices with a dedicated function. This differentiates them from general-purpose computing systems, which can be easily reprogrammed for different tasks. Furthermore, the term *embedded* commonly implies that these systems are part of a larger device that also incorporates additional electrical and mechanical components. Similar to CPSs, embedded systems come in all kinds of sizes, from the tiniest smart sensors (e.g., *smart dust* [97]) to subsystems of various types of industrial machinery and vehicles. Particular qualities of embedded systems typically include a tailored combination of hardware and software, a focus on reliability and safety, and consideration of various design parameters such as size, weight, and power consumption.

Many of the systems mentioned above must adhere to timing constraints during execution, leading to their classification as *real-time systems*. According to Shin and Ramanathan [92], three central components characterise such systems: First, there is an obvious focus on time as a resource, where the correctness of a computation does not only depend on its logical or functional correctness, but also on the time at which the result is available. Second, most real-time systems have high reliability requirements, in which an error during execution may lead to significant financial losses or, in the worst case, loss of human life. Therefore, these systems are often referred to as *safety-critical systems*. Third, real-time systems are usually not independent of their execution environments. For example, in a fly-by-wire system, the onboard control computer cannot be considered independent of the aircraft and its operating environment, as those are the drivers of the system's timing constraints.

Here, the connection between these different system definitions becomes already apparent. The focus on timing aligns with the concept of reactive computing, while the reliability requirements are present in all the systems defined previously, and the inclusion of the surrounding execution environment is indicative of the design of embedded systems.

Given their importance in industry, considerable research efforts have been devoted to the development of design and implementation methods for real-time embedded systems. This includes the development of *task models* (see [95] for an overview), the creation and use of various forms of *timed logic* (see [37] for an overview), and the invention of dedicated programming languages.

2.2 The Synchronous Approach

As indicated in the previous chapter, the design, implementation, and verification of real-time embedded software are complex and error-prone tasks. Therefore, the real-time systems community has dedicated significant research efforts to developing models, methods, and tools to address this complexity. One direction of research, which also lays the foundation for some of our work, focuses on the *synchronous approach*, originating from electronic circuit design methodology.

Modern electronic circuits are built from two different circuit design patterns. *Combinational circuits* are those which implement functions that do not have any form of memory, so that their output at any point in time is purely a function over their current inputs. They can be described using combinational logic. Examples include the fundamental Boolean logic gates that implement the **`and`**, **`or`**, and **`not`** operations. By the laws of combinational logic, any circuit built from only combinational sub-circuits is itself combinational.

On the other hand, *sequential circuits* are those which possess any form of *state*. This means that their outputs are not only a function of their inputs, but also depend on the history of values that have appeared on their input ports. To talk about the history of values, one needs to define the time points at which these values are sampled, which divides sequential circuits into *synchronous* and *asynchronous* types. Synchronous circuits change their state in response to a *clock signal*, which can be seen as an additional input, while asynchronous circuits react to changes in the input signals themselves. The majority of sequential circuits are implemented synchronously, which offers advantages such as an easier way to deal with and reason about delays.

Both combinational and sequential circuits have a *propagation delay*, which is the time it takes for the outputs of a circuit to stabilise as a reaction to changes in its inputs. For an overall circuit, this delay increases with complexity, as the *critical path* (i.e., the slowest path in the circuit) becomes longer. This determines the *maximum clock frequency*, with which the circuit can be driven, as all signals along all paths must have enough time to settle on stable levels. Modern circuit design tools automatically calculate these propagation delays for a given circuit and provide the designer with the maximum possible clock frequency at which the circuit can be driven.

This affords a nice abstraction for the design of *synchronous sequential circuits*. Suppose we treat time as discrete and use clock pulses as time ticks. In that case, the outputs of our circuits change *instantaneously* with their inputs (i.e., within the same tick), assuming the clock frequency respects the propagation delays. This is known as the *synchrony hypothesis* [13], which builds the foundation for the timing semantics of the family of *synchronous programming languages*.

Interestingly, early developments in synchronous programming arose almost exclusively within French academia. The three languages Esterel [13],

Signal [8], and Lustre [42] were developed independently by different French research groups at the beginning of the 1980s. A cooperation was later established, with a focus on compilation methods and increasing industrial adoption of the synchronous programming approach.

While synchrony underlies all three of these languages, they differ significantly in how they express computation. Esterel is primarily an *imperative language*. Signal and Lustre, on the other hand, were built on the *data-flow* model, where computation is expressed as a graph with nodes modelling operations, and edges encoding data dependencies. This approach to modelling computation is beneficial for control engineers, as it is already similar to the *block diagram* formalisms commonly used in their field. Since our work draws heavily on Lustre, we will focus our presentation on this particular language. An overview and comparison of all three languages is given in [43].

Gilles Kahn, in his seminal paper [62], provided a remarkably concise model of computation for parallel data-flow programming. A *Kahn process network* (KPN) is a graph, where the nodes are processes, and the edges are FIFO queues. If a process tries to read from any of its inputs, it must block until data is available, while writing to a queue is always non-blocking. Kahn showed that the output of such a network is independent of the particular scheduling order of the set of processes, as long as the only communication between them is through the queues (i.e., there are no global memory cells shared between processes), and that the schedule ensures that each process that can run will eventually also be selected for execution. Therefore, this model can act as a basis for different deterministic implementations of concurrent or parallel computation. Whether a given KPN is deadlock-free and whether it can be executed within bounded memory (i.e., there exists an upper bound for each FIFO queue) is, in general, undecidable [21]. This has led to various restrictions of the model, which enable the creation of provably deadlock-free schedules and the efficient determination of queue bounds.

Lee and Messerschmitt [68] introduced such a restricted model called *Synchronous Data Flow* (SDF). In an SDF graph, the number of tokens consumed and produced by each input and output of a process is known ahead of time. They showed that for SDF, the question of deadlock-freedom and boundedness can be answered efficiently.

Lustre can be seen as a particular instantiation of the SDF formalism. Conceptually, Lustre works on a set of *flows*. A flow is a (possibly infinite) sequence of values together with a *clock*, which describes at which instances of a base block the values of a particular flow are present. Flows can be combined by operators (e.g., arithmetic or logic operators), which are extended to work point-wise on flows with the same clock. Slower clocks can be defined using Boolean-valued flows, which indicate the presence of a value at the timeticks when the flow is true.

In Lustre, flows can be given names via the definition of *equations*. Multiple equations can then be combined into a *node*, an abstraction similar to functions

in other programming languages. For example, a node for a **nand** operator can be implemented in the following way:

```
node nand (a, b : bool) returns (c : bool);
  var tmp : bool;
let
  tmp = a and b;
  c = not tmp;
tel
```

The node **nand** takes two Boolean-valued input flows `a` and `b`, and returns a Boolean flow `c`. It is defined through two equations. The first introduces an auxiliary flow `tmp`, which is the point-wise logical **and** of the two input flows. The second defines the output flow as the negation of that auxiliary flow.

Lustre also defines two *sequence operators*, which explicitly operate on flows directly. The **pre** operator refers to the previous value of its argument flow, which introduces an undefined value $\perp$ at the first position. The *initialisation operator* $\rightarrow$ defines the first (i.e., initial) value of a flow, and can therefore be used to remove the undefined value introduced by a **pre**. These sequence operators are illustrated in Table 2.1. It also shows the derived operator **fby** (followed-by), which is defined as a **fby** $b \equiv a \rightarrow$ **pre** b. All three, **pre**, $\rightarrow$, and **fby** result in flows with the same clock as their argument flows.

	1	2	3	$\cdots$
x	x_1	x_2	x_3	$\cdots$
y	y_1	y_2	y_3	$\cdots$
pre x	$\perp$	x_1	x_2	$\cdots$
$x \rightarrow y$	x_1	y_2	y_3	$\cdots$
x **fby** y	x_1	y_1	y_2	$\cdots$

Table 2.1. Lustre sequence operators

There are two further flow operators, which allow manipulating flow expressions on different clocks. The **when** operator allows sampling (or filtering) a flow according to a different Boolean flow, which then acts as the (slower) clock of the resulting flow.

The **current** operator can be used to *project* a slower flow onto a faster one by essentially holding the last value at which the clock of the slower flow was true. This is illustrated in Table 2.2.

	1	2	3	4	$\cdots$
b	false	true	true	false	$\cdots$
x	x_1	x_2	x_3	x_4	$\cdots$
$y = x$ **when** b		x_2	x_3		$\cdots$
current y	$\perp$	x_2	x_3	x_3	$\cdots$

Table 2.2. Lustre flow operators

There are a couple of interesting issues arising from Lustre's data-flow formalism that the compiler must handle. First, Lustre equations are allowed to be recursive, so that the equation

$$x = 0 \rightarrow \textbf{pre}\, x;$$

is valid, and defines an infinite sequence of zeros. However, the equation

$$x = 0 \rightarrow x;$$

is not valid, as the current value of x can only depend on previous values of itself. The Lustre compiler, therefore, performs a *causality analysis* [42], which ensures that the set of equations making up a program can be ordered automatically.

Another issue arises from the **pre** operator, which introduces undefined values at the start of flows. These undefined values can lead to non-deterministic outputs if they are not removed by $\rightarrow$ operators. This is checked during *initialisation analysis*, which has been formulated nicely as a type-checking problem by Colaço and Pouzet [30].

The final, and most distinct, problem regards the different clocks of flows in a program. Most operators (such as arithmetic and logic operators) can be used only on flows that have the same clock, i.e., they are present at the same time-points. As demonstrated above, the **when** and **current** operators can be used to manipulate the clock of a flow, so that the compiler has to prove the compatibility of clocks between flows that interact with each other. This has led to the development of the *clock calculus* [42], which serves as the basis for the analysis performed by the Lustre compiler. Caspi and Pouzet [24] showed that Lustre's clock calculus can be formalised as a classical *dependent type system*. Colaço and Pouzet [31] later introduced a simplified type system, informed by use cases observed in real-world applications. The calculus they proposed is based on classical Hindley-Milner type systems [80], extended with a restricted form of existential types as introduced by Läufer and Odersky [66]. This results in a type system that allows the description of higher-order process networks, an efficient implementation, and full clock-type inference, which is particularly important for Lustre's industrial implementation, SCADE [11].

The original Lustre paper [23] introduced a compilation scheme that translated Lustre's data-flow equations into sequential code, incorporating all three analyses mentioned above. It works by recursively inlining node definitions to create a single, large control loop, which is supposed to be called periodically by an external runtime system. This compilation scheme was later improved by Biernacki et al. [14] to allow for separate, modular code generation for individual Lustre nodes. However, the output of the described compilation scheme is still a single function which has to be executed periodically by an external runtime.

While it is possible to define multi-rate systems in Lustre using the **when** and **current** operators, this often results in programs that are hard to understand and reason about. This is because Lustre allows arbitrarily complex Boolean expressions to define the clocks of different flows. It also precludes the compiler from performing specific optimisations, as reasoning about these clocks becomes undecidable in the general case (e.g., when a clock depends on runtime values). This has led to the development of different approaches to dealing with multi-rate systems, often at the expense of restricting the expressions allowed as clocks.

In his PhD dissertation [41], Julien Forget introduced the Prelude language, an extension of Lustre with real-time primitives that allows the description of multi-periodic systems. It restricts clocks to be strictly periodic, i.e., each clock has a single period and an optional phase, which is the offset of its first instance. This affords two simplifications. First, evaluating clock compatibility of flows becomes easier, as clocks are fully described by their period and phase, both of which are known statically. Second, it allows programs to be compiled into a set of RTOS tasks rather than the single-loop code generated by the Lustre compiler. This works by translating a Prelude program into a dependent task set, which encodes the data dependencies between tasks as precedence constraints. This task set is then transformed into an equivalent independent task set by adapting the task parameters to account for the required dependencies [82].

Bourke et al. [16] presented an extension of Lustre for multi-rate systems, including constraints such as load balancing, resource limiting, and end-to-end latencies. Similar to Prelude, their language uses statically defined periods, while leaving the phases of clocks open to be determined by the compiler. This allows the compiler to automatically find static schedules that satisfy the given constraints by solving an integer linear program to find suitable phase offsets. They also incorporate a novel approach to handling cyclic dependencies between equations, allowing the compiler to automatically use the previous value of a flow instead of the current one if that resolves the direct dependency. This is similar to the **dc** (*don't care*) operator introduced by Wyss et al. [98] and the **last?** operator by Iooss et al. [59]. They already remark that in most industrial applications, it does not matter whether the most recent or last value is used in a particular instance. This is especially true for sensor values, which only change slowly, and therefore, the difference between consecutive values of the same flow has minimal contribution to internal feedback and output calculations. A similar line of reasoning will be used to motivate specific communication patterns in the next section.

2.3 MIMOS

As outlined in the previous section, the synchronous approach offers numerous advantages when designing and implementing software for real-time embedded systems.

The block-based formalism of Lustre/SCADE is already familiar to domain experts such as control engineers. This facilitates industry adoption and enables a visual programming style that is accessible even to non-experts.

The timing semantics of a language like Lustre is remarkably concise and sufficiently abstract to be usable in practice. This makes reasoning about the temporal behaviour of a program written in Lustre significantly easier than one implemented in a conventional programming language.

However, there are also various disadvantages connected to synchronous programming.

The synchrony hypothesis puts very stringent timing requirements on the underlying execution platform. This makes it challenging to execute Lustre programs on modern architectures, including multi-core and many-core processors and distributed systems.

Similar to the critical path in circuit design, the program's complexity affects the base clock at which a synchronous program can be executed. This makes the timing analysis brittle with respect to potential updates or changes to the system.

Particularly motivated by updatability, Yi et al. [101] introduced the MIMOS model as a framework for the design, implementation, and verification of real-time systems, which builds the foundation for our work performed in Papers I and II.

MIMOS is an extension of a KPN, where each process is activated according to a timed release pattern and has a deadline, within which its computation must finish. At the start of a release, a process checks all its inputs and, if enough data is available, computes its function and writes its outputs at the deadline. If insufficient data is available, the node remains idle until its next release. Various real-time task models may be used as release patterns, including periodic tasks [72], generalised-multiframe [5], DRT [94] or even timed-automata [39], as long as the releases are deterministic. For the remainder of this dissertation, we will focus solely on periodic releases. For simplicity, we will also assume implicit deadlines, i.e., each process must finish its computation before the next release.

As MIMOS is a particular instantiation of a KPN, and the added release patterns concern only the specific scheduling of processes, MIMOS automatically also offers value determinism. However, the added timing dimension enables communication patterns beyond FIFOs between processes, while maintaining the overall determinism of the computation in a network of MIMOS nodes.

Examples of such extensions are *registers* [101] and *up-to* ports [100]. Registers allow for modelling latest-value semantics, which is particularly helpful

when communicating with sensors that might have very different sampling frequencies than the control algorithm's period. Up-to ports essentially allow reading or writing a variable number of elements to or from a FIFO queue. A MIMOS model that includes such extended communication patterns remains deterministic if the reading and writing times are precisely defined, which they are through the release pattern and deadline of each process.

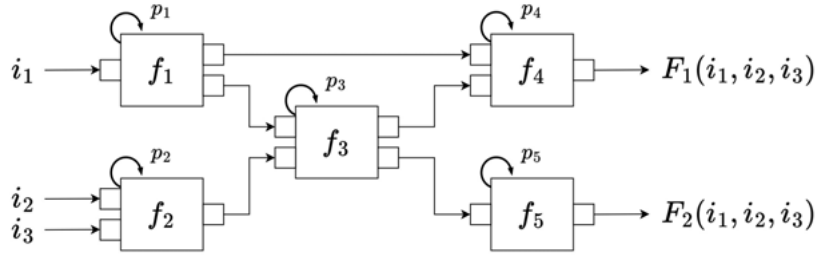


Figure 2.1. Graphical representation of a MIMOS graph

Figure 2.1 illustrates a graphical representation of a network of MIMOS nodes. It also depicts the FIFO queues connected to the *input* and *output ports* of each node, and its *period*. For simplicity, only FIFOs are used in the example.

The inputs to the processes f_1 and f_2 are open and are referred to as *system inputs*, which are intended to be provided by the execution environment, for example, by delivering sensor readings. Similarly, the outputs of f_4 and f_5 are not connected, and are meant to provide *system outputs*. Each system output stands for a particular *system-level function*, which we refer to as F_1 and F_2 in Figure 2.1. Here, both F_1 and F_2 are functions over (the histories of) all system inputs, while in practice, a system-level function may also only depend on a subset of the inputs. As we will see in Section 3.1, for Papers I and II we chose to restrict the model further, and require that all inputs and outputs must be connected, for example, to special sensor and actuator driver nodes.

A vast design space is available for implementing a particular MIMOS network. In practice, a specific implementation must adhere to the release patterns' timing constraints and allocate sufficient resources to a process to complete its computation before its deadline.

To facilitate different implementations and provide an efficient description of the overall system architecture, MIMOS follows a layered implementation approach [100], as depicted in Figure 2.2.

On the top layer, the *function layer*, we have the system-level functions. At this layer, we can perform functional and timing verification purely based on the given release patterns and deadlines, independent of any particular schedule for the set of processes.

Process nodes from the function layer get mapped onto *tasks* in the *task layer*. In practice, any real-time task model can serve as the underlying task

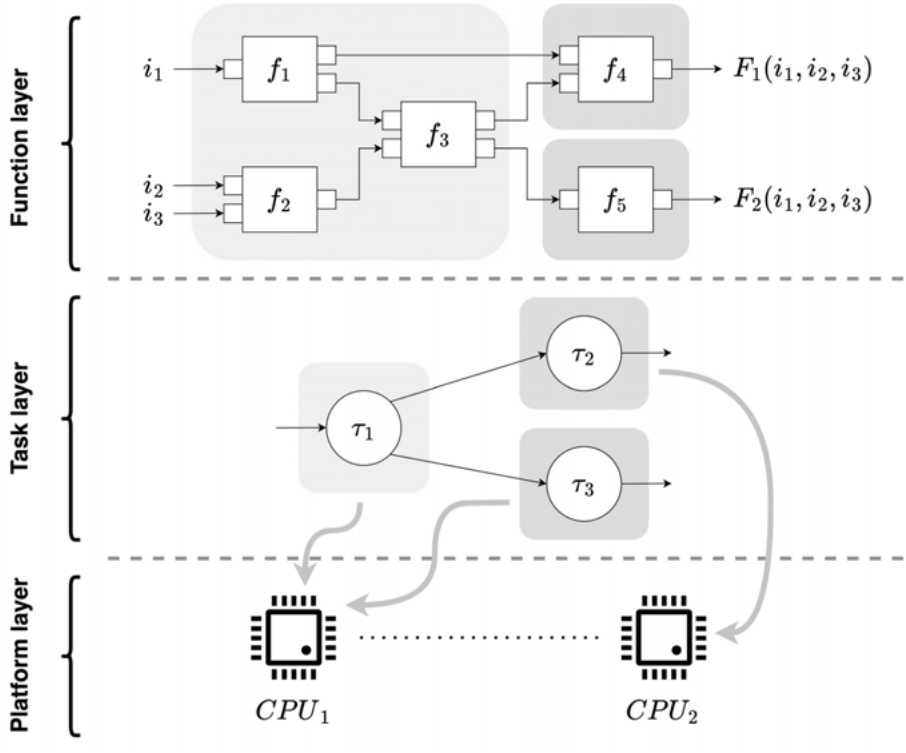


Figure 2.2. MIMOS layered implementation strategy

layer, provided it offers efficient implementation schemes and schedulability tests.

Finally, tasks get assigned to physical resources in the *platform layer*. In a distributed setting, other resources might also be allocated in this layer, such as memory blocks or dedicated and shared communication links between different platform components.

This layered approach has several advantages, particularly regarding potential updates of a system's functionality.

Functional and timing verification can be performed on the function layer using the abstractions provided by the MIMOS model. This includes verifying the system's functional behaviour and the required end-to-end delays for each system-level function. In the event of a potential update, verification can be performed again at the level of the functional model.

Each layer has its own design space, restricted only by the timing requirements imposed by the layer above it. The assignment of MIMOS nodes to tasks can be done in various ways, taking into account the particular topology of the respective MIMOS network. For example, a linear string of MIMOS nodes, where the output of each node is connected to the input of the next node in sequence, may be mapped to a single task with an obvious, internal, static

schedule. This allows reducing the total number of tasks a system scheduler has to manage, thereby lowering implementation complexity and improving schedulability.

Finally, modern embedded systems can incorporate a wide range of different computational resources beyond traditional processing units. These may include *Graphical Processing Units* (GPUs), *Digital Signal Processors* (DSPs), or even application-specific hardware circuits implemented as dedicated circuits or synthesised in *Field-Programmable Gate Arrays* (FPGAs). Keeping the platform layer separate from the task layer helps maintain the functional model's independence from the specific execution platform.

Various reference architectures also employ similar layered approaches for developing large industrial systems, such as AUTOSAR [4] and SAVOIR [2] for the automotive and European spaceflight industries, respectively.

2.4 Dynamic Verification

As outlined in the introduction of this dissertation, software plays an increasingly important role in our everyday lives, making its correct execution of great importance. Software verification, the process of checking that a particular application satisfies its expected requirements, encompasses a wide range of different techniques, methods, and tools. Broadly speaking, these techniques and procedures can be divided into two categories: *static* and *dynamic verification*.

Static verification includes a wide variety of techniques, ranging from relatively straightforward activities such as *code convention checking* or *type checking* to much more sophisticated methods, such as *formal verification*. A characteristic feature of static verification is that code is analysed directly without being executed. This approach provides several advantages, including the ability to use abstractions that reason about all possible program executions, rather than a subset. However, these benefits come at a cost: methods like formal verification (i.e., the process of establishing a mathematical proof of the correctness of a software system) are typically very demanding, both in terms of expertise and computational resources. Consequently, the significant overhead in development effort and time often discourages their widespread adoption in industry. It is reasonable to assume that this is at least part of the reason formal verification is not yet a ubiquitous practice outside of academia. Instead, it is usually reserved for domains where the consequences of failure are severe, such as systems where errors could lead to substantial financial losses (e.g. banking or trading systems) or where unintended behaviour may pose a threat to human safety (e.g. medical devices, autonomous vehicles, and aircraft control systems).

In contrast, *dynamic verification* refers to the process of analysing the behaviour of a software system during its actual execution. This category en-

compasses techniques such as *(dynamic) testing*, *runtime monitoring*, and *fuzzing* [79]. Unlike static approaches, dynamic verification observes and evaluates the software in action, which brings certain practical advantages. For instance, it allows for the analysis of components for which the original source code is not available, such as proprietary or legacy binaries. However, this strength is offset by a fundamental limitation. Because it examines only specific executions, dynamic verification cannot generally reason about all possible behaviours of a system.

Testing, one of the most widely used forms of dynamic verification, involves executing a program with a specific input (a *test case*) and examining the result of the computation to determine whether it conforms to expectations. Various testing methodologies exist, which usually align with different phases of a typical software development workflow, ranging from *unit* and *integration testing* to *system* and *acceptance testing*.

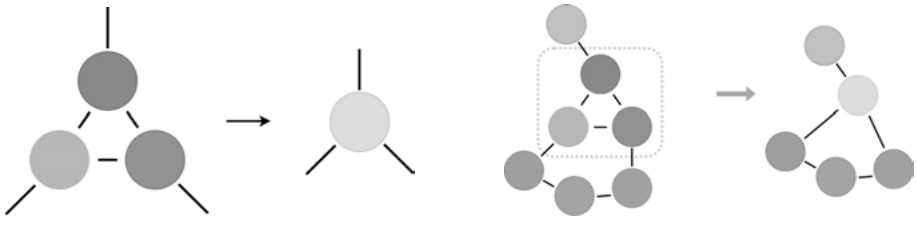
A particular and increasingly popular form of testing is *property-based testing* (PBT) [40]. In PBT, a testing framework automatically generates a large number of randomised or structured inputs and checks whether a program satisfies a set of user-defined executable specifications, called properties. When a violation of such a property is detected for a specific input, the testing framework reports the failure and often attempts to *shrink* the input to a simpler counterexample that is easier to understand and debug. The research literature enumerates various success stories of PBT, including in telecommunications software [3], file sharing services [56], key-value stores [15], e-commerce invoicing systems [55], election software [63], computational geometry algorithms [89], and optimising compilers [77, 83].

PBT was popularised by its implementation as the QuickCheck Haskell library [28], and similar libraries have since been created for most mainstream programming languages. The particular implementation most relevant to our work is the OCaml QCheck library [33].

While the original QuickCheck implementation was targeted toward purely functional programs, follow-up work [29] provided extensions to support testing monadic, effectful Haskell code, including the concept of model-based testing. Similarly, QCheck has been extended to support model-based state-machine testing with the QCheck-STM tool [78], which forms the basis for our work described in Paper III.

Even though QuickCheck was primarily created as a tool for testing, the underlying method has since been incorporated in various interactive theorem provers as well, including Isabelle [9, 22], Coq [84], and Agda [36], where they provide users with counterexamples during proof development. This highlights the idea that dynamic and static verification do not necessarily compete with each other but rather complement one another.

2.5 Graph Rewriting



(a) Rule *(b)* Rule application

Figure 2.3. Example of a graph rewriting step

Graph rewriting (also called *graph transformation* or *graph reduction*) is a generalisation of term rewriting and has played a vital role in the early implementation of functional programming languages (see, for example, the Clean language [19]).

Formally, a graph rewriting system consists of a set of *graph rewrite rules*, where each rule has the form $L \rightarrow R$. L is usually called the *mother* or *pattern graph*, while R is referred to as the *daughter* or *replacement graph* [86]. A visual example of a graph-rewriting rule is shown in Figure 2.3a.

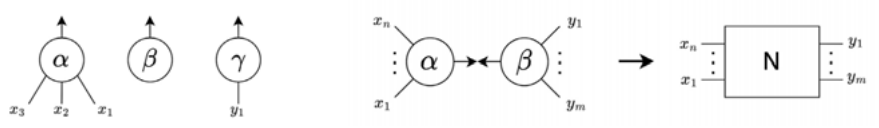
The actual rewrite process starts with a *host graph* H, which is searched for a sub-graph that is isomorphic to one of the pattern graphs of the given set of rewrite rules. That particular sub-graph is then cut out of H and exchanged with an instance of the replacement graph for that specific rewrite rule. This process is illustrated in Figure 2.3b.

Graph rewriting systems can be divided into different categories based on specific traits, such as the allowed structure of the pattern and replacement graphs, or whether different pattern graphs are allowed to overlap. This would mean that for a given host graph, there could be multiple applicable rules concerning the same nodes at a given point in the rewriting process, from which one needs to be chosen.

Similar to term rewriting systems, if for a particular graph H no further rewrite rule can be applied, H is said to be in *normal form*. We say that a graph rewriting system is *confluent* if any possible order of rule applications leads to the same normal form.

One of the difficulties in implementing a graph rewriting system is the efficient discovery of sub-graphs that are isomorphic to one of the pattern graphs. This problem is known as the *sub-graph isomorphism problem*, and is NP-complete in general [32]. As we will see in the next section, the particular graph rewriting system we used in Paper IV has an elegant solution to this problem.

2.6 Interaction Nets



(a) Example agents *(b)* Example rule

Figure 2.4. Graphical notations for agents and rules. Reprinted from [54]

Interaction nets, introduced by Lafont [65], are a special case of graph rewriting. As mentioned in the previous section, one of the difficulties in implementing graph rewriting is the efficient discovery of subgraphs isomorphic to one of the pattern graphs in a given set of rewrite rules. Interaction nets solve this issue by introducing the notion of a *principal port* and restricting rewrite rules to only use pattern graphs consisting of two nodes connected through their principal ports. The nodes in interaction nets are referred to as *agents*, each agent has a *label* $(\alpha, \beta, \gamma, \ldots)$ drawn from a finite set Σ of *symbols*. Each agent has exactly one principal port, and a (possibly empty) set of *auxiliary ports*. The number of auxiliary ports is fixed for each symbol, and usually described through an *arity function* $\mathbf{ar} : \Sigma \mapsto \mathbb{N}$. An example for the graphical notation of different agents is shown in Figure 2.4a.

A pair of agents that connects through their principal ports is called an *active pair* (which is related to the concept of a *redex* in term rewriting). Interaction rules describe how each active pair can be exchanged by a new subnet, which must connect all auxiliary ports of the particular active pair, and is not allowed to introduce any new edges that are unconnected (i.e., it must preserve the *interface* of the active pair). An example of an interaction rule is shown in Figure 2.4b.

A more concrete example is illustrated in Figure 2.5, which shows agents for natural numbers in unary encoding with addition. For this particular example $\Sigma = \{S, 0, +\}$ and $\mathbf{ar}(S) = 1$, $\mathbf{ar}(0) = 0$, and $\mathbf{ar}(+) = 2$. Figure 2.6 shows the two rewrite rules for addition. In the case of adding zero, the result (i.e., port *a* in Figure 2.6a) is directly connected to the second argument of the $+$-agent (i.e., port *b*). If the first operand is not zero, the addition is recursively applied to the remainder of the first operand (i.e., port *b* in Figure 2.6b), and the result of the addition is shifted by one (i.e., the *S*-agent is put before the result port *a*).

Figure 2.7 showcases an example of the rewrite process. On top, a network for the addition $1 + 0$ is shown. This network only has one active pair, which is marked in the rewrite application below. Applying the rule shown in Figure 2.6b connects the remainder of the left input (i.e., a 0-agent) to the $+$-agent, and puts an *S*-agent before the result. This creates another active pair, which is rewritten according to the rule shown in Figure 2.6a, leading to

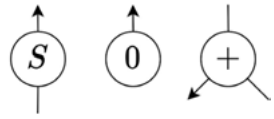
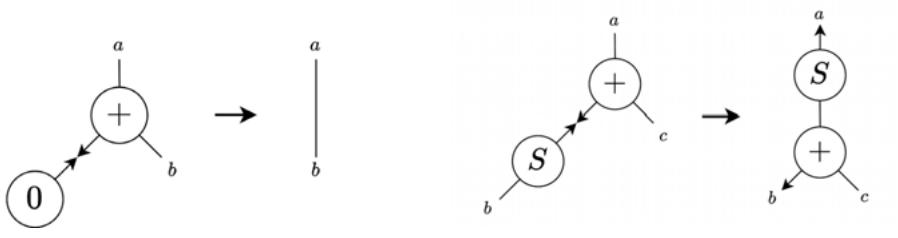


Figure 2.5. Agents for natural numbers with addition



(a) Addition with zero *(b)* Addition with non-zero

Figure 2.6. Rewrite rules for addition

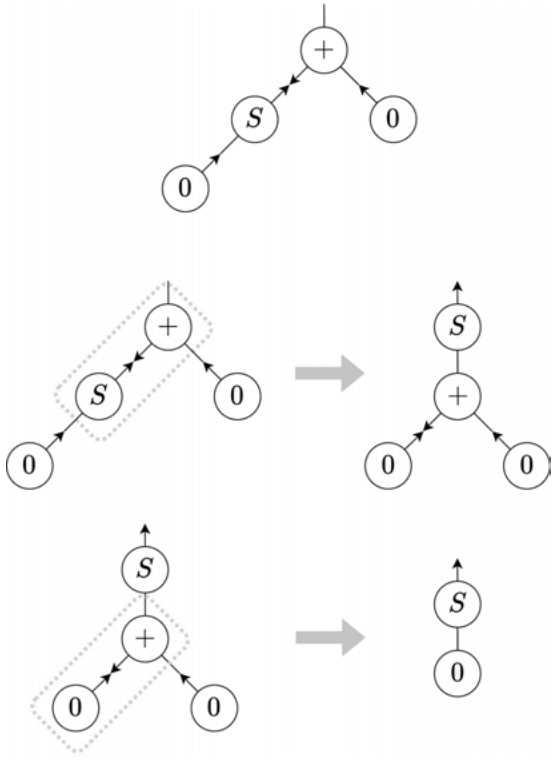


Figure 2.7. Example of a rewrite process

a net that cannot be further reduced (i.e., it is in normal form). This net, as expected, encodes the number 1.

Interaction nets exhibit several interesting properties, including locality of the rewrite process, as each agent has only one principal port and can therefore only be part of a single active pair. If the left-hand sides (i.e., the pattern graphs) of all interaction rules do not overlap, then the rewrite process can be shown to be *strongly confluent*, i.e., each possible rewrite sequence leads to the same normal form and has the same number of rewrite steps.

Not every possible active pair has a valid semantic interpretation. For example, if two natural numbers (e.g., two 0-agents) were connected together, no rule would apply for the resulting active pair, and the network would not constitute a valid natural number. This observation was already present in the original paper by Lafont [65], and he introduced a rudimentary type system to prevent the construction of such nonsensical interaction nets.

In this type system, each port gets assigned both a *value type* and a *polarity*. The value type encodes the particular sort of value represented by the port (e.g., integer, Boolean, float, ...), while the polarity encodes whether a port is used as an input or an output. It remains up to convention if positive polarity encodes inputs or outputs, as long as the assignment is consistent throughout the network. In our work, we have adopted the same convention as Lafont used in the original paper, assigning positive polarity to outputs and negative polarity to inputs. According to Lafont's type system, only ports with the same value type but opposite polarities can be connected.

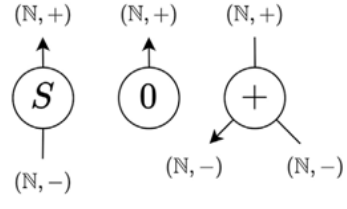


Figure 2.8. Example of typed agents

Figure 2.8 shows the same agents for natural numbers with addition from before, with added type information on each port. It is important to note that a principal port is not automatically assigned positive polarity, as shown by the $+$-agent.

Different evaluators and compilers have been developed for interaction nets, including AMINE [85], PIN [46], INET [47], amineLight [74], and inpla [88]. A comparison between the encoding approaches used by these tools is given in Shinya Sato's PhD thesis [87]. An embedding of interaction nets into Haskell has been done in [61], and evaluation on a GPU was demonstrated in [60].

2.7 Generalised Algebraic Data-Types

Generalised Algebraic Data-Types (GADTs) are a generalisation of *parametric algebraic data-types*, which represent an important feature of modern functional programming languages. They were introduced independently under various names, such as *first-class phantom types* [27], *guarded-recursive data-types* [99], and *equality-qualified types* [91]. At the time GADTs were introduced as a feature in functional programming, they had already played an important role within the dependent types community for over a decade as *inductive families of data types* [35].

GADTs are a helpful feature for various tasks in functional programming. They have, among other things, been used for generic programming, (typed) embeddings of programming languages, maintaining invariants in data structures, and modelling objects [27, 48, 90, 91, 99].

In essence, they allow the programmer to instantiate the type parameter in the signature of each constructor of a *polymorphic variant type*, which we will show on a simple example in the following paragraphs. One of the canonical use-cases of GADTs is the implementation of typed expression languages. The Abstract Syntax Tree (AST) of such a (simplified) language can be expressed in OCaml as a variant type:

```
type expr =
    | Lit of int
    | Eq of expr * expr
    | If of expr * expr * expr
```

An expression is either an integer literal (e.g., `1`), an equality comparison between sub-expressions (e.g., `1 = 2`), or a conditional (e.g., `if 1 = 2 then 3 else 4`). This way of expressing the AST is straightforward. However, it poses an issue when trying to evaluate an expression, as there are two different return types (e.g., `1` evaluates to an integer, while `1 = 2` evaluates to a Boolean). Moreover, the definition above allows for ill-typed expressions to be constructed (e.g., `if 1 then 2 else 3`). Both can be remedied by using a GADT instead of a simple variant type:

```
type _ expr =
    | Lit : int -> int expr
    | Eq : int expr * int expr -> bool expr
    | If : bool expr * 'a expr * 'a expr -> 'a expr
```

The variant type has been extended to be polymorphic. However, as the type variable in the definition is unused, we can utilise an *anonymous type* or *phantom type* `_` in the type signature. For each constructor, the instantiation of the type variable for each sub-expression and for each return type can be restricted independently. On the one hand, this allows us to refine the return type of each constructor, so that `Lit` is of type `int expr` while `Eq` is of type `bool expr`. On the other hand, we can also refine the types of sub-expression arguments for each constructor, so that `Eq` can only be used on integer expressions, and

that the condition in `If` must be of type `bool expr`. The last constructor also illustrates that the type signatures for each case are really type equations, so that we can express that both branches in `If` must be of the same type `'a expr`, and that the resulting value has the same type `'a expr`.

With this definition of the AST, an evaluator can be implemented in a straightforward way:

```
let rec eval : type a. a expr -> a = function
    | Lit n -> n
    | Eq (e1, e2) -> (eval e1) = (eval e2)
    | If (e1, e2, e3) ->
        if (eval e1) then (eval e2) else (eval e3)
```

The type signature for `eval` can be read as $\forall \alpha \,.\, \alpha$ `expr` $\rightarrow \alpha$. It must be provided, as the compiler is usually not able to infer principal types in the presence of GADTs and polymorphic recursion.

3. Contributions

> *We can only see a short distance ahead, but we can see plenty there that needs to be done.*
>
> — Alan Turing

The following chapter summarises the four papers constituting the contributions of this dissertation. Section 3.1 introduces Mimosa, a user-facing programming language for the MIMOS framework introduced in Section 2.3. Section 3.2 focuses on ORTAC, the OCaml Runtime Assertion Checker, which brings contract-based dynamic verification to the OCaml ecosystem. Finally, Section 3.3 introduces an embedding of interaction nets into the OCaml language.

3.1 Mimosa (Paper I and II)

A MIMOS model, as presented in Section 2.3, naturally divides into *coordination* and *computation*. The coordination layer is concerned with the topology of the processes and queues, while the computation layer concerns the functions performed by each process. The original MIMOS paper [101] focuses on the coordination layer, proving timing determinism, and showing possible extensions of the communication primitives between processes. It does not present a user-facing programming language and delegates the implementation of each node to conventional programming languages such as C or Java (e.g., for simulation). Different nodes need not even necessarily be implemented in the same programming language. This approach is similar to other coordination languages, such as Lingua Franca [73].

However, there are several advantages to having a dedicated programming language for implementing MIMOS nodes. Almost all programming languages in use today lack formally defined semantics, making formal reasoning about them challenging. While it might be interesting, or indeed useful, to implement different nodes of a system in different programming languages, this complicates communication between them, as mismatches in the encoding of data types and data structures must be handled, as well as multiple runtimes provided. Finally, since the systems MIMOS primarily targets are often safety-critical, it is advantageous to have a formally defined programming language available, on which various analysis tools and potentially even verified simulation and compilation routines (similar to CompCert [70], Vélus [17], and CakeML [64]) can be developed.

This motivation has led to the development of Mimosa, the MIMOS Application language. The syntax and formal semantics of the language are presented in Paper I, while a compilation scheme to RTOS tasks is described in Paper II.

Syntactically, Mimosa builds upon the Lustre [42] language, from which it retains the equation-based definition of programs. Figure 3.1 shows a graphical representation of a Mimosa program, consisting of three nodes and four FIFO queues, out of which two carry initial elements (i.e., elements that are present at the start of the program).

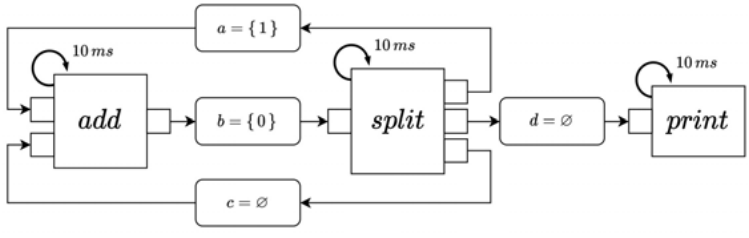


Figure 3.1. Fibonacci example (graphical). Adapted from [51]

```
1  step print_int (_ : int) --> ()
2  step add (x, y) --> z { z = x + y }
3  step split inp --> (o1, o2, o3)
4      { o1, o2, o3 = inp, inp, inp }
5
6  channel a : int = { 1 }
7  channel b : int = { 0 }
8  channel c : int
9  channel d : int
10
11 node add implements add (a, c) --> (b) every 10ms
12 node split implements split (b) --> (a, d, c) every 10ms
13 node print implements print_int (d) --> () every 10ms
```

Listing 1. Fibonacci example (Mimosa). Reprinted from [51]

The same program, expressed in Mimosa, is shown in Listing 1. A Mimosa program consists of a collection of top-level definitions including *steps*, *channels*, and *nodes*.

Steps are Mimosa's equivalent to function definitions in other programming languages. Each step has a name and a signature consisting of its inputs and outputs. The body of a step is a collection of equations which define the outputs in terms of the inputs and potentially additional auxiliary variables defined inside the step. A *step prototype* does not have a body, and must be provided externally (e.g., the `print_int` step). Mimosa uses a Hindley-Milner

style type system [80], so that only the types in the signatures of step prototypes must be given, and the compiler can infer all others.

Channels introduce FIFO queues. They fix the type of the elements the queue shall carry, and optionally list initial elements that shall be present at the start of the program. Each channel must be connected to exactly one reading and one writing node.

Nodes are instantiations of steps as periodically triggered processes. They list the steps they are implementing and how the ports of each node are connected to other nodes through the defined channels.

The example shown in Figure 3.1 is essentially synchronous, as all nodes execute with the same period. This, however, is not required. An obvious problem when attempting to connect two nodes with different periods is the availability of data. Suppose a faster executing node puts data into a queue to be consumed by a slower node. This inevitably leads to data accumulating in the queue and, in practice, causes an overflow.

On the other hand, a slower node feeding data to a faster one will, in practice, force the faster one to idle most of the time, and effectively synchronise the two periods. As mentioned in Section 2.3, MIMOS introduces new communication patterns between nodes, which can be used to mitigate this problem. The one we implemented in Mimosa is a particular instantiation of the up-to port [100], which we call *optional port*.

The idea is to utilise a polymorphic variant type, similar to the option type in OCaml (or equivalently the Maybe monad in Haskell):

```
type 'a option =
    | Some of 'a
    | None
```

When a node tries to read from an *optional input*, if data is present in the connected FIFO, it will take that element and wrap it within a **Some** constructor. Otherwise, it will use **None**. Similarly, if an *optional output* is written to a queue, a value of the form **Some** `t` is unwrapped and only the value `t` is written. In contrast, a **None** output indicates no output, and therefore nothing is written to the connected FIFO.

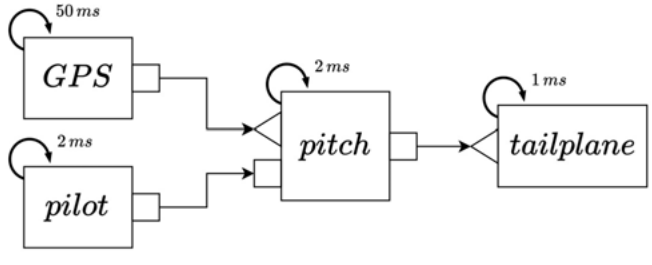


Figure 3.2. Aeroplane pitch control system. Adapted from [51]

An example of optional inputs can be seen in Figure 3.2, which illustrates a simplified pitch controller for an aeroplane. There are two inputs to the control algorithm, which is implemented within the `pitch` node. One input comes from the pilot, periodically sampled by the `pilot` node, and another comes from the onboard GPS peripheral. As GPS is a relatively slow system, the node implementing its driver has a larger period than the pitch control node. The GPS input to the `pitch` node is therefore optional, as indicated by a triangle port. The `tailplane` node implements the driver for the tailplane control surfaces of the plane. This node runs fastest because the driver must react to small changes in the aeroplane's mechanical parts. In practice, the pitch control algorithm may only output a new control command sporadically, whenever the desired pitch angle needs to be changed. In that case, the tailplane driver can operate with the last received command until a new one is provided.

One essential difference between Mimosa and Lustre is in the way expressions are evaluated. As outlined in Section 2.2, Lustre operates on conceptually infinite flows, which are combined point-wise by different operators to form new flows. Mimosa, on the other hand, operates on *sequences*, which can be finite or even empty, and not all operators combine sequences point-wise. For example, in Lustre the expression **if** e_c **then** e_t **else** e_e always evaluates both e_t and e_e and combines the flows according to the current value of e_c. In Mimosa, only one branch will be evaluated, depending on the value of the condition expression e_c. For example, in **if True then** e_t **else** e_e, the sequence generated by e_e is empty.

Expression evaluation in Mimosa is not assumed to be side-effect-free. For example, the `print_int` step in Listing 1 will most likely have an observable side-effect. By selectively evaluating certain sub-expressions, side effects can be guarded without relying on a more complex setup, such as utilising multiple clocks (as in Lustre).

The flow operators **pre**, **fby**, and $\rightarrow$, as known from Lustre, are extended to Mimosa with slightly different semantics caused by the selective evaluation of subexpressions. The expression **pre** e evaluates e in every cycle, and therefore any side effects of e are visible from the first cycle on. The same goes for $e_1 \rightarrow e_2$, which evaluates both e_1 and e_2 in the first cycle, and therefore would perform any side effects in the first cycle as well. The expression e_1 **fby** e_2 delays the evaluation of e_2 to the second cycle. Therefore, any side effects that occur during its execution are also delayed.

The semantics of step evaluation is described through a relation

$$\Gamma \vdash e \Rightarrow v, e'$$

which expresses, that under an environment Γ (which maps names to values), expression e evaluates to a value v and a next expression e'. This allows expressing the stateful evaluation of expressions as a form of rewrite calculus.

The coordination layer semantics is expressed similarly. However, to make it more accessible, Paper I presents the rewriting rules in terms of a graph

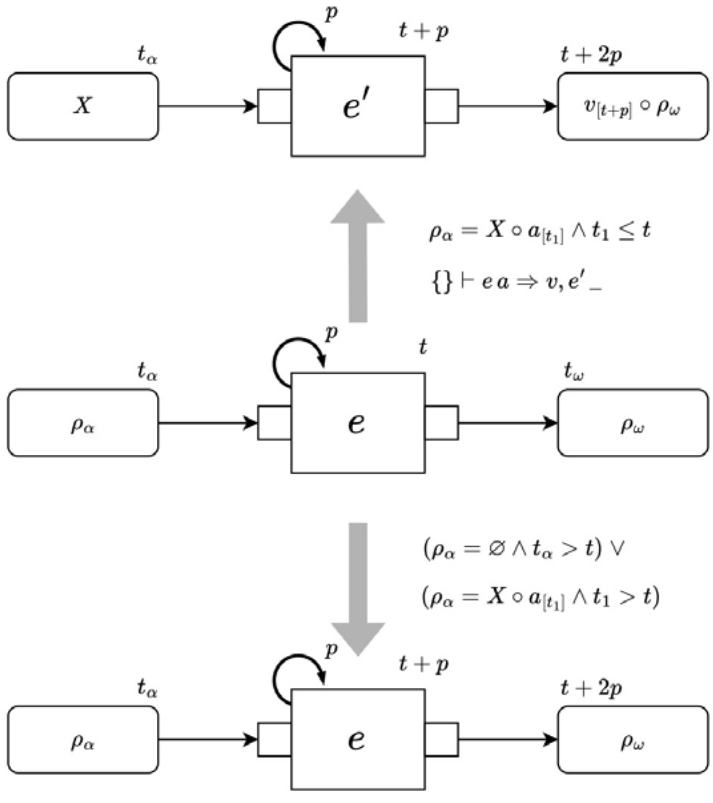


Figure 3.3. Rewriting rule for FIFO input. Adapted from [51]

rewriting system. Figure 3.3 shows the simplified rewriting rule for a node with a single FIFO input.

The idea is to annotate each element in a FIFO with a time stamp, which is the time it was written to the FIFO. Furthermore, each FIFO itself is equipped with a *validity time*, which is the earliest time at which an element might still be written to it. Each node keeps the time of its next activation. Two different rewritings are possible:

If the queue is empty and the validity time is greater than the current node activation time, or if the oldest element in the queue has a time stamp that is greater than the current node activation time, the node performs an idle step. This updates the next activation time of the node and the validity time of the output buffer. This scenario is depicted in the lower half of Figure 3.3.

Suppose the oldest element in the FIFO has a time-stamp that is smaller than or equal to the current node activation time. In that case, the expression inside

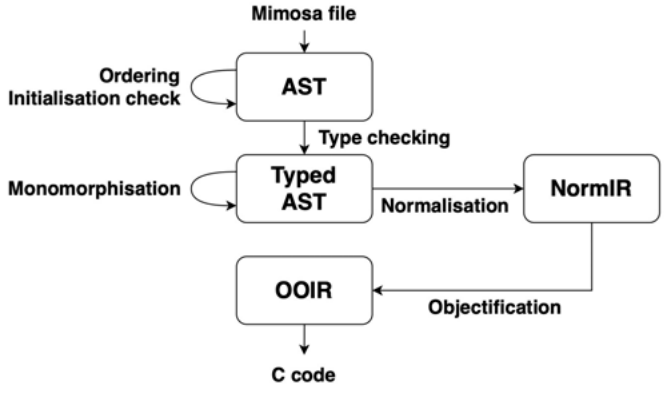


Figure 3.4. Mimosa compiler phases overview. Reprinted from [52]

the node can be evaluated with that element as an input, resulting in a value v and a subsequent expression e'. The value v is written to the output FIFO with a time stamp according to the node's deadline (i.e., the next activation time), and the next activation time of the node and the validity time of the output buffer can be updated. This corresponds to the upper half of Figure 3.3.

The rewriting rules can be extended to multiple inputs and outputs, as well as optional ports [51].

Paper I presents a simulator for the Mimosa language, which essentially consists of a deep-embedding of the language into OCaml and a discrete-event simulator, which constructs the full timeline for any desired duration. To execute Mimosa on real embedded systems hardware, Paper II introduces a compilation scheme towards RTOS tasks.

As mentioned in Section 2.3, the mapping from MIMOS nodes to RTOS tasks can be done in various ways. As an initial prototype, Paper II uses the most straightforward mapping, which translates each node into its own task. This simplifies the compilation of nodes and queues, so the focus of the paper is on translating the step functions themselves.

The overall compilation routine is inspired by the Lustre compiler [14]. Figure 3.4 gives an overview of the different phases of the compiler. First, the equations inside each node are brought into a causal order, and proper initialisation is checked with a typing approach similar to the one in Lustre [30]. As Mimosa allows for polymorphic types, the *monomorphisation phase* creates monomorphic copies of polymorphic steps according to the types of all call sites. During *normalisation*, the nested block-structure within each node is made explicit. The *objectification* phase makes state explicit. It uses an Object-Oriented Intermediate Representation (OOIR) because object-oriented languages already include concepts like object-local state. From OOIR, the translation to C (or potentially Java for simulation) is trivial.

3.2 Dynamic Verification of OCaml Programs (Paper III)

Even though OCaml is primarily a functional programming language, it also offers many imperative features, such as I/O, references, mutable arrays, and exceptions. This makes programs written in OCaml an interesting target for verification.

OCaml is an important implementation platform for various code analysis and verification tasks, including tools such as the interactive theorem prover Rocq/Coq [96] and the C code analysis tool Frama-C [34]. It is also a common choice for the implementation of research compilers (e.g, Lustre [42], Prelude [41], and Lucid Synchrone [25]). Given OCaml's prominence in program analysis and verification, there is a surprising lack of analysis and verification frameworks for OCaml programs themselves.

In an attempt to mitigate this void, the GOSPEL project [26] was created to equip OCaml with its own behavioural specification language, on which various analysis and verification tools can be built. To this end, the authors of the original GOSPEL paper [26] already remark that the language itself shall be tool-agnostic.

```
type 'a t
(*@ mutable model contents : 'a sequence *)

val push : 'a -> 'a t -> unit
(*@ push v t
    modifies t.contents
    ensures t.contents = Sequence.cons v (old t.contents) *)

val is_empty : 'a t -> bool
(*@ b = is_empty t
    ensures b = match Sequence.length t.contents with
        0 -> true | _ -> false *)
```

Listing 2. Example of GOSPEL specifications for the `Stack` module

An excerpt of the interface of the `Stack` module from the OCaml standard library, with added GOSPEL specifications, is shown in Listing 2. Specifications are added as structured comments in the tradition of other specification languages such as ACSL [6] or JML [67]. The stack type is annotated with a *model descriptor*. In the example, we model a stack as a sequence of values (i.e., a list). The functions are annotated with a *function contract*. Each contract lists which *model fields* are modified and describes how a call to the function updates these fields. The `old` operator allows access to the current value of the field (i.e., before calling the function). The specification of the `is_empty` function shows that GOSPEL also allows introducing Hoare-logic style *post-conditions* [49] for the return value. The syntax of GOSPEL con-

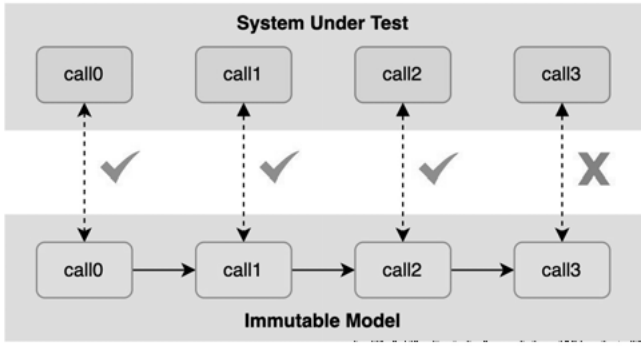


Figure 3.5. Overview of model-based state-machine testing. Reprinted from [53]

tracts is close to that of OCaml, therefore making it easier for programmers to develop specifications. A type can be described by a model with multiple fields, of which not all must necessarily be marked as *mutable*. For example, a fixed-size array may be modelled by a mutable contents field and an immutable size field. This simplifies the specification of certain invariants.

One of the tools developed, which can consume GOSPEL annotated OCaml module signatures, is ORTAC [53], the OCaml RunTime Assertion Checker, which is the focus of our work in Paper III. ORTAC is a tool for automatically testing stateful OCaml data structures. It builds upon various other tools, such as QCheck [33], which implements QuickCheck [28] style testing facilities in OCaml.

ORTAC offers a modular architecture that leverages GOSPEL annotations for various tasks. The focus of Paper III is on ORTAC/QCheck-STM, which uses the QCheck-STM [78] library for modelling state.

QCheck allows a programmer to express a property of an instance of a type as a function from that type to a Boolean value. The user must also define a generator for random values of that type, which QCheck will use to test the given property on. If a violation of the property is found, the particular value can be reported as a counterexample.

QCheck-STM extends this testing framework towards stateful types. The idea is that the user defines an immutable model of the type (e.g., lists for mutable arrays), and describes how different calls of the type's Application Programming Interface (API) act upon this model. It then uses QCheck to run random sequences of API calls on both an instance of the stateful type, which is referred to as the System-Under-Test (SUT), and the immutable model. If a difference between their behaviours is observed, the trace is shown as a counterexample to the user. This process is illustrated in Figure 3.5.

To work with QCheck-STM, the user must manually define the model and how it changes through API calls. ORTAC/QCheck-STM automatically extracts these two from the GOSPEL specifications provided in the module interface, which represents the stateful type.

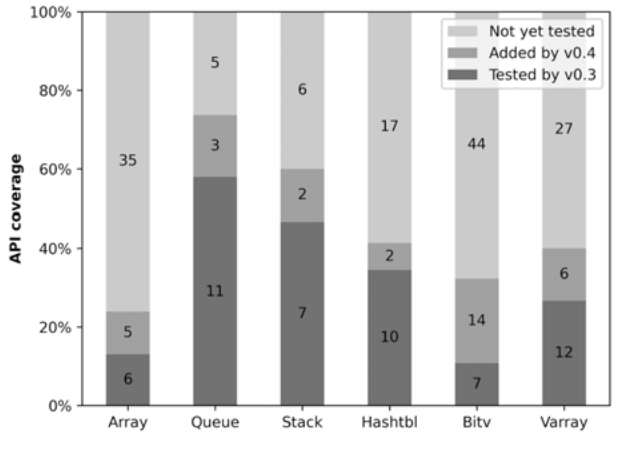


Figure 3.6. API coverage for different ORTAC/QCheck-STM versions. Reprinted from [53]

Using a single SUT allows testing many of the usual API functions for common stateful data types. However, certain API calls require multiple SUT arguments. For example, the function `val transfer : 'a t -> 'a t -> unit` from the `Queue` module of the OCaml standard library transfers all elements of the first queue argument to the second. Similarly, certain API functions may return a new SUT instance, such as initialisation or copy functions. Paper III reports on an extension of ORTAC/QCheck-STM to allow testing of these kinds of functions. It utilises a (potentially growing) stack of SUTs and a corresponding list of immutable models. For each API call, the required amount of SUT arguments is popped from the stack, the API call is issued, and the SUTs are pushed back onto the stack. Similarly, the required number of elements from the list of models is also altered.

Figure 3.6 illustrates the effect this extension has on the API coverage for different OCaml modules. For the evaluation part of Paper III, four modules from the OCaml standard library (`Array`, `Queue`, `Stack`, and `Hashtbl`) as well as two modules from the OCaml package repository (`Bitv` and `Varray`) have been selected. Version v0.4 introduced the described extension for multiple SUT arguments and for functions that return SUTs.

Among the tested modules, ORTAC/QCheck-STM was able to find five crashing bugs in the two modules from the package repository, and one documentation inconsistency in the standard library, which has since been fixed.

Figure 3.6 clearly shows that the described extensions increase API coverage, in some cases even significantly. However, it also indicates that a sizable portion of the API remains unchecked, offering opportunities for future work as outlined in Chapter 4.

3.3 Encoding Interaction Nets (Paper IV)

While previous work on interaction net encodings and evaluation focused on the efficiency of the rewriting process, our work in Paper IV focuses on embedding not only the interaction net primitives but also Lafont's type system [65]. By expressing the interaction net type system in terms of the type system of OCaml, we can prevent the construction of certain classes of nonsensical networks, both for the initial net (i.e., the starting point of the program) and the rewrite rules themselves.

A simple encoding of the three agents from Figure 2.5 can be done with a variant type in OCaml:

```
type agent =
    | Z
    | S of agent
    | Plus of agent * agent
```

A value of `type agent` represents the principal port of the respective agent. Each constructor takes a certain number of argument agents, according to the agent's arity. With this definition, the natural number 2 would, for example, be expressed as `S(S(Z))`.

To create a full network, we also need a way to connect two auxiliary ports together. This can be done by introducing *named ports*, which we will not cover here for brevity. To connect two agents through their principal ports, we introduce a custom infix operator:

```
let ( ><- ) a1 a2 = (* run the rewrite in parallel *)
```

With this encoding, the OCaml compiler would accept the nonsensical network `Z ><- Z` already mentioned in Section 2.6. Lafont's type system prevents this connection because the polarity of both ports is the same.

To embed both the interaction net primitives and Lafont's typing discipline, we present an encoding that relies on GADTs. We use two phantom types, one for encoding the value type and one for encoding the polarity:

```
type (_, _) agent =
    | Z : (int, pos) agent
    | S : (int, pos) agent -> (int, pos) agent
    | Plus : (int, neg) agent * (int, pos) agent -> (int, neg) agent
```

Note that the polarity of the argument agents is flipped with respect to the ones presented in Figure 2.8. This is because we need to refer to the type of the port that will ultimately be connected as the argument agent. With this definition in place, we can also restrict the construction of networks:

```
let ( ><- ) : type a. (a, pos) agent -> (a, neg) agent -> unit = (* ... *)
```

The OCaml compiler would now reject `Z ><- Z` as a valid net. The polarity types are used only as tags, and no values of these types are ever created. Therefore, we can define them as *empty types*:

```
type pos = |
type neg = |
```

Since types are erased during compilation in OCaml, these additional type annotations incur no runtime cost. In Paper IV, we also showcase how the additional typing information on each port further simplifies the implementation of the rewrite routine, which we will leave out here for brevity.

Due to the locality of the rewriting process for interaction nets, different active pairs can be rewritten in parallel without losing confluence. This raises the question of whether encoding interaction nets into a language like OCaml can serve as an implementation strategy for algorithms to automatically scale across multiple processing cores, thereby enabling automatic parallel program execution. This is of particular interest, as OCaml recently gained native support for parallel evaluation [93].

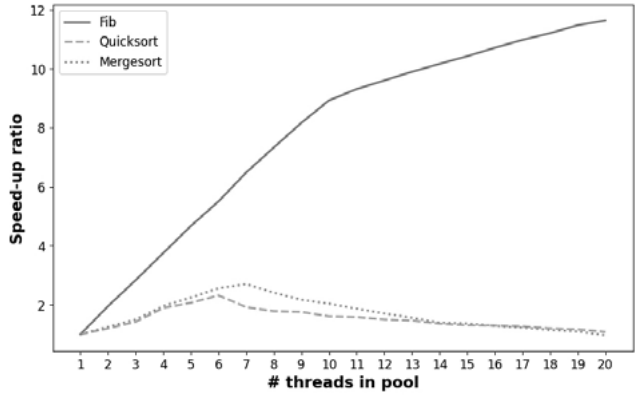


Figure 3.7. Evaluation of the relative speedup. Reprinted from [54]

We evaluated the presented encoding on three example interaction nets. Figure 3.7 shows the relative speedup achieved by using increasingly higher numbers of threads for parallel evaluation of the rewriting process. It shows that the Fibonacci number calculation scales well with increased thread count. In contrast, list sorting with mergesort or quicksort showed only an initial performance gain, then deteriorated as the thread pool size increased. This is most likely due to thread scheduling and CPU migration. However, further investigation is needed as outlined in Chapter 4.

For the Fibonacci example, we also compared the performance of our encoding with a hand-written sequential and parallel implementation. It turned out that the interaction net encoding was on par with the hand-written parallel one. This indicates that, at least for specific algorithms, our encoding is a competitive implementation strategy.

While the concrete examples in Paper IV are presented in OCaml, we note that our encoding method can be applied to any language that supports GADTs (such as Haskell or Scala).

4. Future Work

> *In the torment of the insufficiency of everything attainable, we come to understand that here, in this life, all symphonies remain unfinished.*
>
> — Karl Rahner

Engineering reliable software systems is a challenging endeavour. Despite receiving considerable attention from various research communities, it remains an active and ongoing area of research. The contributions of this dissertation have addressed multiple aspects of the engineering process for software systems that require high levels of reliability. For each, many different directions for future investigations remain. In this chapter, we want to illustrate some of these.

The Mimosa programming language introduced in Section 3.1 is currently only a language prototype, and various extensions are necessary to make it a viable candidate for implementing actual embedded system software. Registers and up-to ports, as mentioned in Section 2.3, need to be implemented to allow for greater flexibility when connecting nodes with different periods. The Mimosa type system should be extended to include additional basic data types (e.g., strings, records/structures, enumerations).

As mentioned before, the evaluation of expressions in Mimosa is not assumed to be side-effect-free. It would be advantageous to be able to track effects as part of the signature of steps, similar to other modern languages which offer an algebraic effect system. Examples of such languages include Koka [69], Flix [75], Effekt [18], and Eff [7]. The added effect types can be utilised for documentation and error finding, and may aid optimisation.

As mentioned in Section 3.1, one advantage of Mimosa is its formally defined semantics. This can be leveraged to implement formally verified tooling, such as simulators, compilers, and verification engines. An encoding of the presented semantics in the interactive theorem prover Isabelle/HOL [81] is ongoing work.

The compiler introduced in Paper II currently employs a fairly simplistic task model that maps each MIMOS node to a dedicated task. There is ongoing work [100] that focuses on using more sophisticated models, along with an automated procedure for mapping a MIMOS network to different task sets. This work also includes schedulability analysis for different execution platforms.

An interesting future extension to the compiler might be to allow part of a MIMOS network to be compiled for special hardware platforms, while auto-

matically generating the glue code to communicate with the rest of the network. Particular targets of interest could be GPUs and FPGAs.

Our work on the ORTAC/QCheck-STM tool has already demonstrated success in identifying bugs in real OCaml libraries and uncovered a documentation inconsistency in the OCaml standard library. As shown in the API coverage in Figure 3.6, a significant portion of the functions in these modules remains untested. The most critical remaining contributors to untested API calls are higher-order functions and casts to other data types (e.g., functions turning queues into sequences). Extending ORTAC/QCheck-STM also to support higher-order functions would significantly increase coverage. However, GOSPEL does currently not offer any specific support for specifying higher-order functions in general. An extension that relies solely on pure functions as arguments is ongoing work.

Our work on interaction net embedding has shown that specific algorithms can utilise the inherent potential for parallel evaluation of the rewritings. However, it also revealed that, for other algorithms, the introduction of additional evaluation threads actually degrades performance. Further investigation is needed, both to determine what constrains the rewriting for those negatively affected algorithms and to potentially find a more optimal scheduling of the rewriting process across the available cores.

One of the original motivations for studying interaction nets was their potential use as an implementation scheme for the computations performed by a MIMOS node. This could allow a single node to utilise unused cores automatically and even enable partial evaluation with already available inputs. This can only be done for nodes that do not have any visible side effects, which ties nicely with the proposed effect type system mentioned above.

5. Conclusion

Life is a journey, not a destination.
— Ralph Waldo Emerson

This dissertation presents a summary of our work performed within the overarching theme of reliable systems engineering. The different contributions thereby targeted different aspects and phases within this engineering process.

The first two papers introduced the Mimosa programming language, a new approach to creating embedded software built on the MIMOS model of computation [101]. The focus was on formal semantics, and a simple compilation scheme to a task set executing on any readily available RTOS.

Paper III presented ORTAC/Check-STM, a tool for property-based testing of stateful OCaml modules, with automatic model-extraction from function contracts provided in the GOSPEL specification language. While more work is needed to increase API coverage, the tool has already demonstrated success in finding undetected errors and documentation inconsistencies in both the standard library and user-provided packages.

Paper IV illustrated a method of encoding Lafont's interaction nets into the OCaml language, with a focus on embedding not only the interaction net primitives, but also Lafont's original type system. This provides greater confidence that the embedded interaction rules are correct and that certain types of runtime errors cannot occur. As the typing discipline is encoded entirely within OCaml's own type system, this incurs no additional runtime cost, since types are deleted during compilation.

Finally, to conclude the work performed in this dissertation, more work is needed, both for the specific contributions presented here and for reliable software systems engineering in general. It is safe to assume that with future technological developments, the role of software in our society will only increase, as will the requirements we put on it.

6. Sammanfattning på svenska

Mjukvara ligger i centrum för alla moderna datorsystem i vårt samhälle: från de personliga enheter vi bär med oss dagligen, till de medicinska apparater som många människor är beroende av och de omfattande systemen som säkerställer driften av vår kritiska infrastruktur. Mot denna bakgrund är det naturligt att vi strävar efter att utveckla mjukvara som är så felfri som möjligt. Denna avhandling redovisar arbete som utförts för att bidra till design, implementering och verifiering av pålitliga mjukvarusystem, med fokus på både teoretiska grunder och praktiska verktyg.

Artikel I introducerar ett nytt programmeringsspråk, Mimosa, som utvecklats specifikt för design och implementering av realtidssystem för inbyggda enheter. Denna typ av system särskiljs av att deras korrekta funktion inte endast beror på numerisk korrekthet utan även på tidsmässiga egenskaper. Exempel på detta inkluderar styrmjukvara i fordon och noggrant tidssynkroniserad signalgenerering i implanterbara hjärtstimulatorer.

Syntaktiskt bygger Mimosa på språket Lustre, vilket tillhör familjen av synkrona språk. Dessa språk är särskilt lämpade för utveckling av realtidssystem, då tidsaspekter har en entydig semantisk tolkning. De strikta tidskrav som synkrona språk ställer på exekveringsplattformen är dock svåra att uppfylla med modern hårdvara, som ofta består av flera processorkärnor eller till och med distribuerade arkitekturer. Detta har lett till olika anpassningar av det synkrona tillvägagångssättet, varav en är MIMOS-beräkningsmodellen [101], som utgör grunden för vårt arbete med Mimosa.

Vårt arbete med Mimosa omfattar formellt definierad semantik, vilket utgör ett viktigt steg mot utveckling av pålitliga verktyg, såsom verifikationssystem, samt potentiellt verifierad simulering och kompilering. Som ett kompletterande artefakt till Artikel I tillhandahåller vi även en prototypsimulator för Mimosa-program. Detta gör det möjligt att experimentera med språket i praktiken, samtidigt som den formella grunden lägger en bas för framtida utveckling av mer avancerade verktyg.

Även om simulering är ett värdefullt verktyg under utvecklingsfasen så krävs en formellt definierad kompileringsrutin för att köra Mimosa-program på inbyggd hårdvara. Artikel II presenterar därför en sådan rutin för realtidssystemuppgifter.

I Artikel III ligger fokus på dynamisk verifiering av tillståndsberoende program utvecklade i OCaml. Språket OCaml används ofta för implementering av forskningskompilatorer och utgör även grund för olika analys- och verifieringsverktyg, såsom Frama-C [34] och Rocq/Coq [96]. Man skulle därför

kunna förvänta sig ett omfattande utbud av verifieringsverktyg för OCaml-program, vilket dessvärre inte är fallet idag.

Vårt arbete med ORTAC/QCheck-STM syftar till att delvis fylla denna lucka genom att tillhandahålla ett verktyg för körtidskontroller, specifikt för OCaml-moduler med muterbara datatyper såsom arrayer, köer och stackar. Arbetet bygger på GOSPEL [26], ett beteendespecifikationsspråk som utvecklats särskilt för OCaml, och integreras i ekosystemet av andra verktyg som också använder GOSPEL-kontrakt. Utvärdering av verktyget över olika moduler, både från OCaml:s standardbibliotek och den officiella paketdatabasen, påvisade implementeringsfel som varit oupptäckta under många år. Vidare identifierades en dokumentationsinkonsekvens i standardbiblioteket, vilken därefter klargjordes i den officiella dokumentationen.

Slutligen illustrerar Artikel IV en inbäddning av interaktionsnät i OCaml. Fokus låg på att koda både de grundläggande interaktionsnät-primitiverna och det ursprungliga typ-systemet, vilket garanterar att specifika fel som kan uppstå under omskrivningsprocessen undviks. Tidigare kodningar av typ-systemet för interaktionsnät gjordes ofta dynamiskt, vilket medför en körtidskostnad. Genom användning av generaliserade algebraiska datatyper kan vår inbäddning representera typdisciplinen helt inom OCaml:s typ-system. Eftersom typer tas bort vid kompilering introduceras ingen extra kostnad.

Ett användningsområde för interaktionsnät är automatisk parallell utvärdering av omskrivningsprocessen, vilket möjliggör att algoritmer uttryckta genom interaktionsnäts omskrivningsregler automatiskt kan utnyttja fler processorkärnor. Eftersom OCaml nyligen fått inbyggt stöd för parallell utvärdering [93], undersöktes i denna artikel om uttryck av beräkningar i termer av interaktionsnäts omskrivning kan ge förbättrad prestanda i OCaml genom att utnyttja den nya parallella körtids-miljön. Utvärderingsresultaten visar att detta gäller för vissa algoritmer, medan andra istället uppvisar en försämrad prestanda.

7. Acknowledgements

It takes a village to raise a child.
— African proverb

As with most things that take years to accomplish, this work would not have been possible without the continuous support of many people.

First, I have to thank my supervisor, Wang Yi, for taking me on as a PhD student. Thank you for the discussions we had over the years and for providing me with many thoughts on research and academia in general.

To my co-supervisor, Pontus Ekberg, without whom this work would not have been possible. I cannot express how invaluable your input, both academically and personally, has been to me over the last six years. Even though I often worked on topics far from your area of expertise, you always encouraged me to develop my own research ideas and helped me follow them, for which I am deeply grateful.

To my second co-supervisor, Susanne Graf, with whom I have spent many hours discussing everything from research to life in France. Thank you for your support, your inputs during our work sessions, and for hosting me in Grenoble to meet some of your team there as well.

To those at the university who have helped me through these years of PhD studies. To Philipp Rümmer, for your support and guidance, and the many discussions we had throughout the years. To Mohamed Faouzi Atig, for providing help whenever I needed it. To Tjark Weber, from whom I have learnt a lot about the more formal aspects of computer science. To Justin Pearson, for organising the category theory reading group with me, and for all the stories you told me about the university and academic life in general. To Bengt Jonsson, for the many interesting conversations we had over the years.

To my senior-group supervisor, Christian Rohner. Thank you for all your support, for listening when I needed to complain about something, and for making sure I had the help I needed whenever I needed it.

To the people in the administrative section of the university: Ulrika Jaresund, Ulrika Andersson, Anna-Lena Forsberg, Therese Ekman, Sofia Stahre, and Marina Nordholm. Thank you for your assistance whenever I faced problems with the bureaucracy that comes with working at a university.

To those colleagues I worked closely with, especially the people of the MIMOS project: Gaoyang Dai, Behnam Khodabandeloo, Chengzi Huang, Duc Anh Nguyen, and Sanjit Kumar Roy. Thank you for all our discussions, both on the project and on all things academia and beyond.

A big thank you to Paul Fiterau Brostean, with whom I had the pleasure of teaching various courses. I believe we did well, and you were a big part of making sure it was an enjoyable experience both for me and for the students!

To my colleagues, of whom many I now count as my friends: Vera van Zoest, Riccardo De Masellis, Zafer Esen, Chencheng Liang, Xuezhi Niu, Bedour Alshaigy, Virginia Grande Castro, Stephan Spengler, Thom Kunkeler, Johan Snider, Samuel Grahn, Amanda Stjerna, Laura Feeney, Daniel Brown, Govind Rajanbabu, and Ali Semi Yenimol. Thank you for our time together at the university, and I hope we will stay in touch even now that I don't work in Uppsala anymore.

A special thank you to the people at Tarides for hosting me during my internship in France. I would like to particularly thank Jan Midtgaard, Nicolas Osborne, and Samuel Hym for both helping me during my time in Paris, as well as for agreeing to write a (I would claim fairly successful) paper together!

To my friends in Uppsala: Eva Breznik, Vera Wolf, and Elli Anastasiadi. Thank you for your friendship, and all the fun we had and hopefully continue to have, even though we don't live in the same city anymore.

To Tobias Mages, who was not only one of my first friends here, but with whom I also shared my first apartment in Uppsala. Thank you for your friendship, all our movie and cinema nights, and the many hours we spent at the university discussing both research and life together. And to Assel Sarsenbayeva, whom I was fortunate to meet through Tobias. Thank you for all the fun we had, both while living together and afterwards.

To Raphaela Heil, with whom I spent more time during my years in Uppsala than with anybody else. Thank you for your friendship, the many weekends, evenings, and holidays we spent working together, and the many hours we spent watching movies, playing Minecraft, and exploring Sweden (of which we still have much to do!). Here's to our next 50 years in academia.

To my friends I met during my studies in Kiruna: Cassidy Thompson, Kyriaki Blazaki, Georges Labrèche, Siiri Talvistu, Fabian Weber, and August Svensson. I am grateful that we managed to stay in contact, even though we all chose different paths after our time in the cold, dark north of Sweden.

To my friends from Austria, of whom many have now also left the country to work abroad: Ricarda Gansberger, Simon Liebentritt, Mathias Gotsmy, Anton Hößl, Carolina Koch, Florine Enengl, Eva Furtmüller, and Benedikt Jaros. Even though we are living far apart, we somehow still manage to keep up with each other's lives, for which I am deeply grateful.

Last, but not least, I must express my sincere gratitude to my family, especially to my parents Anita and Wolfgang, and to my brother Laurenz, for your unfailing and continuous support in all my endeavours. A special thank you also to my brother for creating the title image for this thesis. This would not have been possible without you. Thank you!

References

[1] Federal Aviation Administration. AD 2015-19-07. Technical report, U.S. Department of Transportation, Federal Aviation Administration, 2015. Docket No. FAA-2015-0936; Directorate Identifier 2015-NM-058-AD.

[2] European Space Agency. SAVOIR Onboard Software Reference Architecture. `https://savoir.estec.esa.int`.

[3] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, page 2–10, New York, NY, USA, 2006. Association for Computing Machinery.

[4] AUTOSAR. AUTomotive Open System ARchitecture. `https://www.autosar.org/`.

[5] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized Multiframe Tasks. *Real-Time Syst.*, 17(1):5–22, July 1999.

[6] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*.

[7] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

[8] A. Benveniste, P. Bournai, T. Gautier, M. Le Borgne, P. Le Guernic, and H. Marchand. The Signal declarative synchronous language: controller synthesis and systems/architecture design. In *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, volume 4, pages 3284–3289 vol.4, 2001.

[9] Stefan Berghofer and Tobias Nipkow. Random Testing in Isabelle/HOL. In *SEFM*, volume 4, pages 230–239, 2004.

[10] Gérard Berry. Real time programming : special purpose or general purpose languages. Research Report RR-1065, INRIA, 1989.

[11] Gérard Berry. SCADE: Synchronous Design and Validation of Embedded Control Software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33, Dordrecht, 2007. Springer Netherlands.

[12] Gérard Berry, Philippe Couronne, and Georges Gonthier. Synchronous programming of reactive systems: an introduction to ESTEREL. Research Report RR-0647, INRIA, 1987.

[13] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[14] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. *SIGPLAN Not.*, 43(7):121–130, June 2008.

[15] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Timothy Bourke, Vincent Bregeon, and Marc Pouzet. Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints. In Alessandro V. Papadopoulos, editor, *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, volume 262 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:22, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[17] Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A Formally Verified Compiler for Lustre. In *PLDI 2017 - 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelone, Spain, June 2017. ACM.

[18] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[19] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean - A language for functional graph rewriting. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 364–384, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[20] Bernd M. Buchholz and Zbigniew A. Styczynski. *Vision and Strategy for the Electricity Networks of the Future*, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2020.

[21] Joseph Tobin Buck and Edward A. Lee. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California, Berkeley, 1993. AAI9431898.

[22] Lukas Bulwahn. The New Quickcheck for Isabelle. In *Certified Programs and Proofs*, pages 92–108, 2012.

[23] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 178–188, New York, NY, USA, 1987. Association for Computing Machinery.

[24] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. *SIGPLAN Not.*, 31(6):226–238, June 1996.

[25] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000.

[26] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL - Providing OCaml with a Formal Specification Language. In *FM 2019 - 23rd International Symposium on Formal Methods*, 2019.

[27] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[28] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[29] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002*, pages 65–77, 2002.

[30] Jean-Louis Colaço and Marc Pouzet. Type-based initialization analysis of a synchronous dataflow language. *Int. J. Softw. Tools Technol. Transf.*, 6(3):245–255, August 2004.

[31] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, pages 134–155, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[32] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[33] Simon Cruanes. QCheck: QuickCheck inspired property-based testing for OCaml. `https://github.com/c-cube/qcheck`.

[34] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247, 2012.

[35] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In Gerard Huet and G.Editors Plotkin, editors, *Logical Frameworks*, page 280–306. Cambridge University Press, 1991.

[36] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining Testing and Proving in Dependent Type Theory. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, pages 188–203, 2003.

[37] E. Allen Emerson. CHAPTER 16 - Temporal and Modal Logic. In Jan van Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 995–1072. Elsevier, Amsterdam, 1990.

[38] European Union Aviation Safety Agency. Airworthiness Directive 2017-0129R1. Technical report, European Union Aviation Safety Agency, 2019.

[39] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[40] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, 1997.

[41] Julien Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. Theses, Institut Supérieur de l'Aéronautique et de l'Espace, November 2009.

[42] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
[43] Nicolas Halbwachs. Synchronous programming of reactive systems. *New York*, 1993.
[44] Nicolas Halbwachs. Synchronous programming of reactive systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 1–16, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
[45] D. Harel and A. Pnueli. On the Development of Reactive Systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
[46] Abubakar Hassan, Ian Mackie, and Shinya Sato. Interaction nets: programming language design and implementation. *ECEASST*, 10, 2008.
[47] Abubakar Hassan, Ian Mackie, and Shinya Sato. Compilation of Interaction Nets. *Electronic Notes in Theoretical Computer Science*, 253(4):73–90, 2009. Proceedings of the Fifth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2009).
[48] Ralf Hinze. Fun with phantom types. *The fun of programming*, pages 245–262, 2003.
[49] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.
[50] House of Lords Select Committee on Chinook ZD 576. Chinook ZD 576: Report of the Select Committee. Technical report, House of Lords, Parliament of the United Kingdom, London, UK, 2002.
[51] Nikolaus Huber, Susanne Graf, Philipp Rümmer, and Wang Yi. Mimosa: A Language for Asynchronous Implementation of Embedded Systems Software. In Cinzia Di Giusto and António Ravara, editors, *Coordination Models and Languages*, pages 90–113, Cham, 2025. Springer Nature Switzerland.
[52] Nikolaus Huber, Susanne Graf, Philipp Rümmer, and Wang Yi. Compiling the Mimosa Programming Language to RTOS tasks. Technical report, Uppsala University, 2025.
[53] Nikolaus Huber, Naomi Spargo, Nicolas Osborne, Samuel Hym, and Jan Midtgaard. Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–22, Cham, 2025. Springer Nature Switzerland.
[54] Nikolaus Huber and Wang Yi. An Encoding of Interaction Nets in OCaml. In Jörg Endrullis, Dominik Grzelak, Tobias Heindel, and Jens Kosiol, editors, Proceedings of the Fourteenth and Fifteenth International Workshop on *Graph Computation Models,* Leicester, UK and Enschede, the Netherlands, 18 July 2023 and 9 July 2024, volume 417 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16. Open Publishing Association, 2025.
[55] John Hughes. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 169–186. Springer International Publishing, Cham, 2016.

[56] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 135–145, 2016.

[57] International Atomic Energy Agency. Investigation of an Accidental Exposure of Radiotherapy Patients in Panama. Technical report, International Atomic Energy Agency, Vienna, Austria, 2001.

[58] International Electrotechnical Commission. IEC 62304:2006 — Medical Device Software — Software Life Cycle Processes. Technical Report IEC 62304:2006, International Electrotechnical Commission, Geneva, Switzerland, 2006. Amendment 1:2015 available.

[59] Guillaume Iooss, Albert Cohen, Dumitru Potop-Butucaru, Marc Pouzet, Vincent Bregeon, Jean Souyris, and Philippe Baufreton. Polyhedral Scheduling and Relaxation of Synchronous Reactive Systems. In *IMPACT 2022 - 12th International Workshop on Polyhedral Compilation Techniques*, pages 1–12, Budapest, Hungary, June 2022.

[60] Eugen Jiresch. Towards a GPU-based implementation of interaction nets. *Electronic Proceedings in Theoretical Computer Science*, 143:41–53, March 2014.

[61] Wolfram Kahl. A Simple Parallel Implementation of Interaction Nets in Haskell. *Electronic Proceedings in Theoretical Computer Science*, 179:33–47, April 2015.

[62] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 471–475. North-Holland, 1974.

[63] Pieter W. M. Koopman and Rinus Plasmeijer. Testing with Functional Reference Implementations. In *Trends in Functional Programming - 11th International Symposium, TFP 2010. Revised Selected Papers*, volume 6546 of *Lecture Notes in Computer Science*, pages 134–149, 2010.

[64] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014.

[65] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 95–108, New York, NY, USA, 1989. Association for Computing Machinery.

[66] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, September 1994.

[67] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[68] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[69] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science*, 153:100–126, June 2014.

[70] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[71] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[72] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973.

[73] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee. A Language for Deterministic Coordination Across Multiple Timelines. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8, 2020.

[74] Ian Mackie and Shinya Sato. A lightweight abstract machine for interaction nets. *ECEASST*, 29, 01 2010.

[75] Magnus Madsen. The Principles of the Flix Programming Language. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, page 112–127, New York, NY, USA, 2022. Association for Computing Machinery.

[76] Sonali Mathur and Shaily Malik. Advancements in the V-Model. *International Journal of Computer Applications*, 1(12):30–35, 2010.

[77] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. Effect-driven QuickChecking of Compilers. *Proc. ACM Program. Lang.*, 1(ICFP):15:1–15:23, August 2017.

[78] Jan Midtgaard, Olivier Nicole, and Nicolas Osborne. Multicoretests — Parallel testing libraries for OCaml 5.0. In *OCaml Users and Developers Workshop*, 2022. `https://github.com/ocaml-multicore/multicoretests`.

[79] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[80] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[81] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[82] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

[83] Michał H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST'11, pages 91–97, 2011.

[84] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational Property-Based Testing. In *ITP 2015 - 6th conference on Interactive Theorem Proving*, volume 9236 of *Lecture Notes in Computer Science*, 2015.

[85] Jorge Sousa Pinto. Sequential and Concurrent Abstract Machines for Interaction Nets. In Jerzy Tiuryn, editor, *Foundations of Software Science and Computation Structures*, pages 267–282, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[86] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., USA, 1997.

[87] Shinya Sato. *Design and implementation of a low-level language for interaction nets*. PhD thesis, University of Sussex, 5 2015.

[88] Shinya Sato. inpla. `https://github.com/inpla/inpla`, 2023. Version 0.11.0.

[89] Ilya Sergey. Experience report: growing and shrinking polygons for random testing of computational geometry algorithms. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*, pages 193–199, 2016.

[90] Tim Sheard. Languages of the future. *SIGPLAN Not.*, 39(12):119–132, December 2004.

[91] Tim Sheard and Emir Pasalic. Meta-programming With Built-in Type Equality. *Electronic Notes in Theoretical Computer Science*, 199:49–65, 2008. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

[92] K.G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, 1994.

[93] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020.

[94] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The Digraph Real-Time Task Model. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, page 71–80, USA, 2011. IEEE Computer Society.

[95] Yue Tang, Nan Guan, and Wang Yi. *Real-Time Task Models*, pages 469–487. Springer Nature Singapore, Singapore, 2022.

[96] The Coq development team. The Coq Proof Assistant, June 2024. `https://doi.org/10.5281/zenodo.11551307`.

[97] B. Warneke, M. Last, B. Liebowitz, and K.S.J. Pister. Smart Dust: communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[98] Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. A Synchronous Language with Partial Delay Specification for Real-Time Systems Programming. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, pages 223–238, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[99] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 224–235, New York, NY, USA, 2003. Association for Computing Machinery.

[100] Wang Yi et al. MIMOS in a nutshell. in preparation.

[101] Wang Yi, Morteza Mohaqeqi, and Susanne Graf. MIMOS: A Deterministic Model for the Design and Update of Real-Time Systems. In Maurice H. ter Beek and Marjan Sirjani, editors, *Coordination Models and Languages*, pages 17–34, Cham, 2022. Springer Nature Switzerland.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2605

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-570474

ACTA UNIVERSITATIS
UPSALIENSIS
2025