Formal Methods for Reversible Concurrent Calculi

Comprehensive Exam written component

Submitted to the Faculty of The Graduate School of Augusta University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy in Computer and Cyber Sciences

Gabriele Cecilia

Document defended on May 11, 2026

Advisory Committee

Dr. Clément Aubert, Major Advisor
Dr. Yuyan Bao
Dr. Harley Eades
Dr. Peter Robinson
Dr. Davide Sangiorgi

1 Introduction

Reversible computation is a model of computing where every performed computational step can be undone. Program executions therefore include both forward computation steps, in which some action is performed, and backward computation steps, in which a previously executed action is reversed. Reversible computation is of interest in several areas, including debugging, database design, simulation, robotics, quantum computing and biochemical modeling [17]. Its study provides a deeper understanding of fundamental aspects of computation, such as identifying causal dependencies between actions and controlling how much information is discarded during execution. Additionally, in many applications, its appeal is justified by its connection with low-power computing: according to Landauer's principle [18], the irreversible erasure of information is necessarily accompanied by heat dissipation and an increase in entropy. This suggests that computations which avoid information loss may, in principle, reduce energy consumption.

Reversing computation is significantly more complex in concurrent systems than in sequential ones. **Concurrency** refers to the ability to parallely execute multiple operations or computations at the same time. In a concurrent system where multiple threads interact, a thread that has previously interacted with others is not allowed to freely backtrack independently, otherwise the system could reach computation states which were previously inaccessible [15]. Instead, a sound way to reverse computation follows the principle of *causal-consistent reversibility*: an action can be undone provided that its consequences, if any, are undone beforehand.

At a foundational level, one of the main formalisms to model and reason about (reversible) concurrent computation is **(reversible) concurrent calculi**, also known as process algebras. These models describe computation as the interaction between units called processes. The *syntax* of a calculus, typically given by a grammar, defines the set of processes and how they can be constructed. The behavior of processes is then specified by the *semantics*, often presented as a Labelled Transition

System (LTS), where transitions represent processes' executions. Finally, **bisimulations** are used to identify processes that have the same observable behavior, even when syntactically distinct, and to abstract away from irrelevant details of the systems of interest.

Process algebras belong to the broader family of *formal methods*, a collection of techniques that use mathematics and logical principles to specify, develop and verify computer systems. Given the prevalence of programming errors (it is estimated an average of 15–50 errors per 1000 lines of code [25]), formal methods provide one of the strongest guarantees of software reliability. Alongside process algebra, another discipline which extends the scope of formal methods beyond the verification of software is **interactive theorem proving**. Interactive theorem proving consists in the development of machine-checked proofs using software systems known as *proof assistants*, where the proof writing process involves a continuous interaction between the user and the machine. One specific domain of application of proof assistants is the mechanization of languages, including concurrent calculi, and their metatheory.

2 Paper Summaries

2.1 Reversible Communicating Systems [15]

Introduction and Motivation

This paper, published in 2004, is regarded as one of the foundational works in the theory of reversible concurrent calculi. It introduces RCCS (Reversible CCS), a reversible extension of the concurrent calculus CCS (Calculus of Communicating Systems [21]). The proposed idea to implement backtracking in CCS consists in equipping threads with a memory stack; the memory stack is updated at every forward computation step and contains precisely the information needed to backtrack. In addition to introducing the syntax and semantics of RCCS, the authors prove properties that characterize the degree of flexibility required for the calculus to support backtracking. Furthermore, they show how to incorporate irreversible actions into the calculus.

Summary

The syntax of RCCS consists of *monitored processes*. Each monitored process can be obtained from units of the form $m \triangleright P$, where P is a CCS process and m is a memory. Memories are stacks containing unique identifiers for threads and information regarding the previously executed actions. Processes are considered up to structural congruence, and a notion of coherence excludes ill-formed processes. The semantics of the calculus is given in terms of a *Labelled Transition System* (LTS) which includes forward and backward computation rules. In an LTS, a transition $R \xrightarrow{l} S$ denotes a computation that leads a process R (the source of the transition) to a process S (the target) through the execution of an action labelled as l. Each forward transition updates the memory of the involved threads with the information needed to backtrack; vice versa, each backward transition uses the information contained in the memory to undo a previously executed action.

The authors then justify the soundness of their implementation of backtracking by defining and characterizing the notion of *causally equivalent traces*. A trace is a sequence of transitions $t_1 \cdots t_n$ that can be executed one after the other (i.e., the target of t_i coincides with the source of t_{i+1} for all i in $1, \ldots, n-1$). The auxiliary notion of concurrent transitions, together with the fundamental reversibility properties known as the loop and the square lemmas, is required to define causal equivalence as an equivalence relation on traces. Informally, causal equivalence captures the following principles: executing two concurrent transitions in either order yields causally equivalent traces; executing a transition and then undoing it is equivalent to performing no transition at all.

The main theorem of the paper characterizes causally equivalent traces as those who begin from the same process and end with the same process. In other words, given any two processes, all the computation traces leading one to the other are causally equivalent. One important consequence is that one is not allowed to undo an action unless it leads to a causally equivalent past: this idea will later be reformulated as *causal-consistent reversibility*, i.e., the principle that an action cannot be undone unless all of its consequences have already been undone. The proof of the theorem relies, among other ingredients, on the parabolic lemma, another key result in the theory of reversible concurrent calculi.

Discussion

This paper is the first to introduce a reversible concurrent calculus in which any forward computation can potentially backtrack. The notions reported, e.g., the definitions related to transitions and traces, as well as the loop or the parabolic lemmas, have since become standard in the field of reversible concurrent calculi. Moreover, the notion of causal-consistent reversibility that they introduce is now widely regarded as the main requirement for ensuring sound backtracking in such systems. Taken together, these factors make this paper one of the most influential in the field.

The paradigm proposed for making CCS reversible is general and can be applied to other concurrent calculi to support backtracking. However, equipping processes with an external memory is not the only approach to make a concurrent calculus reversible: another method, introduced by Phillips and Ulidowski [26], consists in enriching the syntax of processes and the transition labels with communication keys.

2.2 An Axiomatic Theory for Reversible Computation [19]

Introduction and Motivation

The two decades following the publication of *"Reversible Communicating Systems"* [15] have seen the study and development of a large number of different formalisms for reversible computation. Most of these works consider specific systems and provide similar but unrelated proofs of the same results. This paper provides a general and abstract framework for reversible computation and introduces a basic set of axioms for reversibility, from which all the other relevant results follow. Importantly, in order to prove properties of specific reversible formalisms, it will be enough to instantiate the proposed general model and show that the basic axioms hold. The authors additionally define two new properties, *causal safety* (CS) and *causal liveness* (CL), which capture and refine the concept of causal-consistent reversibility. Finally, they analyze the applicability of their method to case studies and distinguish between two different structured notions of independence.

Summary

The abstract framework used to study reversibility in a setting as general as possible is Labelled Transition Systems with Independence (LTSIs) with reverse transitions. LTSIs are labelled transition systems over a generic set of processes and labels, with the addition of an irreflexive symmetric binary relation on transitions called *independence*. To introduce reversibility, the system is enriched with a reverse transition $Q \overset{a}{\rightsquigarrow} P$ for each forward transition $P \overset{a}{\rightarrow} Q$.

The first introduced axioms are SP (square property), BTI (backward transitions are independent), WF (well-foundedness) and PCI (propagation of coinitial independence). They express fundamental properties held by any sound reversible model, such as the fact that two independent transitions can be performed in either order; a system satisfying those axioms is called *pre-reversible*. The authors prove that a pre-reversible LTSI satisfies properties such as the parabolic lemma, causal consistency, backward label determinism or polychotomy of events. Some of the proved properties

involve the notion of events, i.e., equivalence classes of transitions generated by equating transitions on the opposite sides of commuting squares. Several examples show that the pre-reversibility axioms are independent with each other.

Next, the paper proposes the two novel concepts of *causal safety* (CS) and *causal liveness* (CL), which refine the idea of causal-consistent reversibility expressed in [15]. CS states that an action cannot be reversed until any actions caused by it have been reversed; CL states that actions should be allowed to reverse in any order compatible with CS. Causal safety and causal liveness are expressed in three different fashions, respectively based on independence of transitions, independence of events and ordering of events. To investigate their mutual relationship, the authors introduce new axioms such as IRE (independence respects events) and IEC (independence of events is coinitial) and identify the minimal sets of axioms from which each property follows.

The proposed approach is general and can be instantiated to heterogeneous reversible formalisms. The authors illustrate a variety of case studies, spanning from reversible concurrent calculi like RCCS to reversible programming languages or Petri nets, in which they examine which of the proposed axioms are satisfied.

Discussion

One of the main contributions of this paper is to provide a unifying axiomatic framework for reversible computation: rather than proving results such as the parabolic lemma independently for each reversible model, one can instead verify the axioms once for a given model and obtain such results for free. This is analogous to how, in mathematics, abstraction eliminates redundant proof work: for instance, category theory allows results like the first isomorphism theorem to be proved in a single abstract setting rather than separately for groups, rings, and vector spaces.

The axioms and their properties are discussed with care and rigor, with the goal of identifying the weakest hypotheses under which each result holds; concrete examples are provided to show that no set of axioms implies another. Given the foundational role of this work, a mechanized verification of its results in a proof assistant would be valuable.

2.3 Locality and Interleaving Semantics in Calculi for Mobile Processes [28]

Introduction and Motivation

Bisimulations are one of the most widely used notions of behavioral equivalence in process algebra. However, what is considered as equivalent behavior depends on the chosen perspective: this gives rise to a variety of bisimulation notions that capture different observational distinctions. In particular, *non-interleaving* equivalences distinguish processes with a different causal or concurrent behavior, while *interleaving* equivalences reduce concurrency to sequentiality plus nondeterminism.

This paper focuses on *location bisimulation*, a non-interleaving equivalence design to capture spatial dependencies on processes, in the setting of the π-calculus [22], one of the most studied non-reversible process calculi. It defines location bisimulation for the π-calculus and proves that it can be characterized in terms of *observation equivalence* — a classical interleaving-based behavioral equivalence — via a fully abstract encoding.

Summary

The π-calculus is a process algebra in which processes interact along channels by exchanging channel names themselves; this feature allows it to model distributed systems whose communication structure may vary through time. The paper focuses on the *polyadic* π-calculus, a variant in which channels are assigned a sort: a channel of sort $n \in \mathbb{N}$ can send or receive n channels at a time. Processes of the polyadic π-calculus, which include the operators of input and output prefixing,

nondeterministic choice, parallel composition, restriction, match and constants, are called standard. Semantics is given by an *early* LTS, in which the instantiation of names received by input processes is performed as soon as possible, i.e., directly in the transition rule for inputs. Communication between two processes, denoted as $\xrightarrow{\tau}$, is meant to be internal and not visible from an external perspective: weak transitions $\stackrel{\mu}{\Longrightarrow}$, where μ ranges over inputs, outputs, and τ actions, represent transitions $\xrightarrow{\mu}$ up to any number of internal communications.

Observation equivalence, also called weak bisimilarity, identifies processes that can match each other's observable transitions, allowing for any number of internal transitions before or after each visible step. Formally, two standard processes P and Q are observation equivalent ($P \approx Q$) if there exists a weak bisimulation $\mathcal{R}$ such that $P\mathcal{R}Q$; a weak bisimulation $\mathcal{R}$ is a symmetric binary relation on standard processes such that, whenever $P\mathcal{R}Q$ and $P \stackrel{\mu}{\Longrightarrow} P'$, there exists Q' such that $Q \stackrel{\hat{\mu}}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$ (where $\stackrel{\hat{\mu}}{\Longrightarrow}$ coincides with $\stackrel{\mu}{\Longrightarrow}$ except for $\mu = \tau$, in which case it may also denote the empty sequence of transitions).

Before introducing location bisimulation, the paper presents auxiliary notions and techniques for the rest of the development. These include the expansion relation, an asymmetric variant of observation equivalence which allows to count the number of τ-actions performed by processes; and the "up-to" techniques, useful to reduce the size of the relations needed to show bisimilarity of processes.

Location bisimulation ($\approx_l$) is defined by adapting the approach of Boudol et al. [11] to the π-calculus. The syntax of standard processes is extended to consider *located processes*: specifically, a location u, i.e., a word over an infinite alphabet of atomic locations At, can be prefixed to a process P, yielding a located process of the form $u :: P$. The previously defined LTS is modified to include transitions of the form $\xrightarrow[u]{\mu}$, where u is a location. Intuitively, the location u in a transition $A \xrightarrow[u]{\mu} A'$ identifies the position of the subprocess of A that is being modified: for example, actions happening at opposite sides of a parallel composition occur at different, independent locations. Location bisimulation $\approx_l$ is then defined as a weak bisimulation on located processes in which the locations appearing in transitions must match. For this reason, location bisimulation distinguishes processes that have different degrees of parallelism.

Next, the author defines the *encoding* $\mathcal{S}$ of standard processes into the π-calculus, which establishes the correspondence between $\approx$ and $\approx_l$; to prove its correctness, $\mathcal{S}$ needs to be extended to an encoding $\mathcal{L}$ on located processes. The main idea behind the definition of $\mathcal{L}$ is to associate each location u in a located process A to a special name in $\mathcal{L}[\![A]\!]$; thanks to nested restrictions in $\mathcal{L}[\![A]\!]$, the way one can access these special names (called locating names) mirrors the spatial dependencies of their associated locations in A. Since a standard process P has no locations, its encoding $\mathcal{S}[\![P]\!]\langle y\rangle$ takes a single locating name y as parameter, representing the empty location path — that is, the absence of any spatial nesting. The main theorem of the paper, granting the semantic correctness of $\mathcal{S}$, states the following: for any standard processes P_1 and P_2 and locating name y, it holds that $P_1 \approx_l P_2$ iff $\mathcal{S}[\![P_1]\!]\langle y\rangle \approx \mathcal{S}[\![P_2]\!]\langle y\rangle$.

Discussion

Comparing different behavioral equivalences is an important problem and is often far from trivial: the machinery required to build the encoding of located processes and to prove its full abstraction is considerable. The effort is repayed by the fact that the simpler theory of observational equivalence can be used to reason about a non-interleaving bisimulation.

According to the author, two distinct subgroups of non-interleaving equivalences can be identified: those that capture the spatial-sensitive equivalences of processes, and those that capture their

causal-sensitive equivalences. A similar study to the one proposed in this paper has also been carried out for causal bisimilarity [10], which belongs to the second group.

The study of non-interleaving semantics and of causality and concurrency is central in the setting of reversible concurrent systems. A relevant example is CCSK^{P} [4], which employs a proved transition system in the style of Degano and Priami [16] to distinguish between dependent and concurrent transitions.

2.4 Bisimulations and Reversibility [3]

Introduction and Motivation

While *"Locality and Interleaving Semantics in Calculi for Mobile Processes"* [28] focuses on bisimulations for a non-reversible concurrent calculus, this paper provides an overview of different behavioral equivalences in the reversible setting. The reversible concurrent models taken into consideration are CCSK (CCS with Keys [26]), a reversible extension of CCS, and KCS (keyed configuration structures), a variant of identified configuration structures [5]. For these models, the authors study two kinds of bisimulations that capture true-concurrency behaviors, namely history-preserving (HP) and hereditary history-preserving (HHP) bisimulation; both are characterized in terms of several other bisimulations that make use of forward and backward transitions.

Summary

Recall that RCCS implements reversibility by equipping CCS processes with a memory. CCSK is another extension of CCS that, instead, implements reversibility by marking forward transitions with *communication keys*; keys are stored in the syntax of processes and denote past executions. Backward transitions undo previously executed actions by removing their associated key from a process.

Configuration structures (CS) are an alternative model of concurrency that describes possible executions in terms of events, rather than processes. A configuration structure is given by a set of events E, a labeling function mapping events to labels, and a set of subsets of E called configurations, satisfying some sanity axioms. Configurations are consistent sets of events that represent possible executions. A *keyed configuration structure* (KCS) is a configuration structure together with a key mapping, an injective function that associates a key to the events of a configuration.

Just as the operators of (keyed) prefixing, choice, parallel composition and restriction are used to construct CCSK processes, the authors show that new configuration structures can be built from the same operators. This correspondence can be used to recursively define an encoding of CCSK processes into keyed configuration structures. Keyed configuration structures can then be given an operational semantics, which is proved equivalent to the operational semantics of CCSK processes: given a labelled transition between two processes, there exists a transition between their encodings with the same label, and vice versa. The authors additionally define a relation $<_x$ on the events of a configuration x capturing causal dependencies of events.

Next, the paper presents a variety of bisimulations, some of which novel, together with theorems proving their mutual inclusion. The list of bisimulations includes hereditary history-preserving (HHP), history-preserving (HP), forward-reverse (FR), back and forth (BF), and forward-backward (FB) bisimulation. Such bisimulations are discriminated according to different factors:

- the underlying model (KCS or CCSK);
- the presence of a bijection preserving label and causal dependencies (HHP, HP) or its absence (FR, BF, FB);

- the invariance under key renaming (BF) or not (FR, FB);
- the inclusion of backward moves in the bisimulation definition (HHP, FR, BF) or not (HP, FB).

The bisimulations that include backward moves in their definition are proved to be equivalent in the setting of KCS and standard CCSK processes (a process is standard if it does not contain communication keys); analogously for the set of bisimulations not including backward moves. For keyed CCSK processes, it is shown that FR, BF and FB bisimulations are not equivalent.

Discussion

This paper provides a comprehensive analysis of the mutual relationship of the bisimulations over CCSK and KCS that are distinguished based on different criteria: the presence of a bijection preserving label and order, the invariance under key renaming, the inclusion of backward moves. However, as partly stated in the conclusions, these are not the only possible criteria to take into account when defining bisimulations: for instance, one could also consider the power of observation on processes (reduction bisimilarity, barbed bisimilarity) or the abstraction from internal transitions (weak bisimilarity, strong bisimilarity). Other possible directions include the study of a suitable notion of context to define congruence in the reversible setting, or the use of other underlying models of concurrency (CCSK$^{\mathrm{P}}$, higher-order calculi).

Analogously to *"Locality and Interleaving Semantics in Calculi for Mobile Processes"* [28], the authors identify two distinct ways to adapt bisimulation to capture true-concurrency differences in behavior: generalizing the notion of "performing an action" (like in step bisimulation), or requiring that the elements matched by bisimulation are in a causal order-preserving bijection (like in HHP or HP). This categorization appears different than the one by Sangiorgi, which instead considers the spatial or causal dependencies in processes. The bisimulations HHP and HP introduced in this paper likely fall into the subgroup of bisimulations capturing causal dependencies on processes; conversely, the locality bisimulation studied by Sangiorgi may be seen as an example of bisimulation where the notion of action is generalized, since action labels are enriched with locality information.

2.5 A Certified Study of a Reversible Programming Language [24]

Introduction and Motivation

The formalization of CCSK$^{\mathrm{P}}$ in [13] is currently the only example of mechanization of a reversible concurrent calculus. However, if we extend our scope to sequential models of reversible computation, the literature provides a handful of examples of formalizations of reversible models; in particular, the following paper presents a formalization of the reversible programming language Janus in the proof assistant Matita [2].

The paper describes two different semantics for Janus: a big-step operational semantics and a denotational semantics. The main proved result is the full abstraction between the two semantics. The syntax and semantics of Janus and the full abstraction theorem are formalized in Matita.

Summary

The authors first provide the syntax of Janus, given by three different grammars for expressions, programs and statements. Expressions consist in usual arithmetic expressions (sums, products, etc.) and relations (equalities and inequalities). Programs are lists of procedure declarations, which contain an identifier and a statement. Statements include reversible assignments, sequences, if-then-else statements, reversible loops, skip statements, procedure calls and uncalls. A state is a function from the variables of a programs into $\mathbb{N}$.

Next, the authors describe a *big-step operational semantics* for Janus in terms of three relations $\Downarrow_e$, $\Downarrow_p$ and $\Downarrow_p^{®}$. The first expresses the evaluation of an expression in a certain state, returning a natural number; $\Downarrow_p$ expresses the forward evaluation of a statement in a certain state, returning the next state; $\Downarrow_p^{®}$ expresses the backward execution of a statement in a certain state. The three relations are defined by a set of inference rules. The first contribution of the authors consists in a new rule for procedure uncalls; thanks to this modification, they are able to prove that the relations $\Downarrow_p$ and $\Downarrow_p^{®}$ annihilate each other (a version of the loop lemma) and that they are functional and injective.

The *denotational semantics* for Janus maps statements to partial injective functions. Before its definition, the authors present the properties of partial injective functions in a category-theoretic setting: such properties are required to prove the correctness of the denotational semantics.

Denoting the set of all states as Σ, the interpretation $[\![e]\!]$ of an expression e is a relation on $\Sigma \times \mathbb{N}$, with the property that $(\sigma, n) \in [\![e]\!]$ iff the evaluation of e in a state σ yields n. The interpretation of programs is more complex. Recall that a program P is a list of k procedure declarations, each containing a statement and an identifier; because procedures can be mutually recursive, one cannot interpret the body s_i of a procedure before knowing what the body s_j of another procedure denotes. For this reason, the interpretation of a statement $[\![s]\!]\varphi$, defined by structural induction on s, is parametrized by a tuple $\varphi = (\varphi_1, \ldots, \varphi_k)$ called functional environment; each φ_i is a partial injective function representing the intended denotation of s_i. The interpretation of a program $[\![P]\!]$ is then defined, as a fixed point, as the functional environment φ satisfying $\varphi_i = [\![s_i]\!]\varphi$ for all $i = 1, \ldots, k$. The main result of the paper is the correspondence between operational and denotational semantics: given a program P and a state function σ, the evaluation of a statement s in the state σ returns the state σ' iff $(\sigma, \sigma') \in [\![s]\!]([\![P]\!])$.

All the definitions and results are formalized in Matita, an interactive theorem prover developed at the Computer Science Department at University of Bologna. It is based on the Calculus of (Co)Inductive Constructions (CIC), the same dependent type theory underlining the general-purpose proof assistant Coq [9] (now Rocq). In the formalization, variables are implemented as an inductive type with one constructor storing a natural number, and the programming language does not appear to have binding constructs. Interestingly, the encoding of the evaluation predicates $\Downarrow_p$ and $\Downarrow_p^{®}$ takes a natural number as an additional argument, used as an upper bound on the number of steps needed to reach a final state.

Discussion

An interesting aspect of this paper is that it is not limited to the areas of reversible programming languages and interactive theorem proving, but a substantial part is dedicated to non-trivial properties in the category theory field. The versatility of the covered topic is also reflected in the versatility of the Matita theorem prover, that has to deal with definitions of both the programming languages and category theory areas. Given the absence of object-level binders, the difficulty of the formalization does not lie in the encoding of syntax and operational semantics, but rather in the definition of the interpretation.

2.6 The Concurrent Calculi Formalisation Benchmark [12]

Introduction and Motivation

Beginning with the formalizations of Nesi [23] and Melham [20], the past three decades have witnessed the extensive use of interactive theorem provers to mechanize the theory of concurrent calculi. This paper proposes three benchmark challenges that address common issues encountered

in such formalizations. It aims to assess the current state of the art in the mechanizations of models of concurrency, to highlight open research problems in the area, and to support the identification of best practices and techniques, as well as the improvement of mechanization tools.

Summary

The idea of benchmark problems for the formalization of models of concurrency is motivated by the importance and success of earlier challenges, such as POPLMark [6] or POPLMark Reloaded [1], which focused on interactive theorem proving in areas such as programming languages and proofs by logical relations. The proposed challenges isolate and address three main issues that frequently arise when formalizing concurrency theory: linearity, scope extrusion and coinductive reasoning. Each challenge problem targets a different fragment of the π-calculus, specifically tailored to highlight the issue under consideration.

Linearity is a property satisfied when each channel is used exactly once by a process: more generally, this notion is useful to keep track of the amount of resources used by a program. For the π-calculus, this is achieved by assigning types to processes, obtaining an instance of the so-called session types. One of the challenging aspects of formalizing session types is deciding how to split the typing context of parallel compositions. To focus on linearity only, the proposed calculus for this challenge is a version of the π-calculus without name passing and constructs like recursion or replication; its semantics is given by a reduction relation. The objective of the challenge is to prove subject reduction: well-typed processes can only transition to well-typed processes in the same context.

Scope extrusion refers to the fact that a name which is restricted (i.e., a name whose usage is limited to a certain subprocess) can be sent to a process outside its scope, as long as this scope is expanded to include the receiving process. This feature of the π-calculus allows it to represent systems whose communication structure may vary through time. Reasoning about scope extrusion is challenging because it requires dealing with the properties of binders. For the π-calculus, there exist two ways of providing an operational semantics that address scope extrusion differently: while an LTS presents transition rules that deal with scope extrusion explicitly, a reduction system delegates it to structural congruence. The objective of the challenge is to prove that the two semantics are equivalent up to structural congruence, for an untyped version of the π-calculus with name passing and without constructs like replication or recursion.

Coinduction, the dual of induction, is a technique that allows defining and reasoning about potentially infinite objects. Bisimulations are an example of coinductive definition. In concurrent calculi, the syntax of processes typically includes operators like replication or recursion to allow processes to have infinite behaviors. Coinduction is in general less understood than induction, is not supported by all interactive theorem provers, and those who do often use different formalisms. The challenge addresses an untyped version of the π-calculus without name passing and with replication. The goal of this benchmark problem is to prove a characterization result for strong barbed congruence: making strong barbed bisimilarity sensitive to parallel composition and substitutions is enough to turn it into a congruence.

Discussion

This recent paper provides a set of three open problems in the field of the mechanization of concurrent calculi. By collecting solutions that employ different techniques and different proof assistants, the aim of the authors is to compare them and identify the best techniques to address the identified challenges. As criteria of comparison, the authors mention the mechanization overhead, the adequacy of the formal statements in the formalization, and the cost of entry for the employed

techniques. Solutions to the first and second challenge have already been proposed, while a solution to the third problem is currently under development.

This paper demonstrates that the problem of formalizing concurrent calculi is of interest for the community and presents its own intrinsic difficulties. Despite the large number of existing mechanizations, no particular technique or approach emerges as the clear best choice when dealing with linearity, scope extrusion or coinductive reasoning. Some of the addressed issues are also relevant to the mechanization of reversible concurrent calculi.

2.7 Engineering Formal Metatheory [7]

Introduction and Motivation

When mechanizing systems like concurrent calculi, or more in general programming languages, several design choices in the formulation of definitions and theorems need to be made; such choices heavily impact the amount of effort required to complete the formal development. One of the main points to be addressed is the approach to represent binders of the object language.

This paper presents a new approach to formalize binding constructs by combining a locally nameless representation of variables and cofinite quantification to introduce free names. In doing so, it offers an accurate overview of the binding encoding techniques at the time of the paper publication and gives insights on some of the technical difficulties of formalizing programming languages metatheory.

Summary

The POPLMark challenge [6], published just three years prior (2005), had succeeded in raising interest in machine-checked proofs of programming languages. As a result, a large number of different logics, proof assistants and binding representation approaches had emerged — yet the community remained fragmented between groups and this range of alternatives made it difficult for newcomers to navigate the field.

Focusing on binding encoding techniques, the paper first provides an overview of the different existing approaches, dividing them between concrete and higher-order. Concrete approaches represent variables with names (e.g., named syntax or nominal logic), natural numbers (e.g., De Bruijn indices), or use two distinct classes for free and bound variables (e.g., locally named or nameless). Higher-order approaches use the function space of the meta-logic to encode variables as arguments of functions (higher-order abstract syntax and its weak variant). Each technique presents its own advantages and drawbacks and needs to deal with α-equivalence and capture-avoiding substitutions.

The *locally nameless* representation is described by showing, as an example, an encoding of the simply-typed λ-calculus. Bound variables are encoded using De Bruijn indices, while free variables are encoded using a type of names. This distinction is supposed to offer the best of both worlds: statements and proofs mirror their mathematical version thanks to the use of names, while α-equivalent terms have a unique representation thanks to De Bruijn indices, and the separation of free and bound names avoids variable capture. The encoding of the λ-calculus includes definitions of (locally-closed) terms, term opening, typing, call-by-value evaluation, substitution and variable closing. The authors additionally recall the importance of the adequacy of the encoding, i.e., the informal proof that the mechanized definitions correctly represent their mathematical counterparts.

The *cofinite quantification* style is also explained by taking the simply-typed λ-calculus as an example. Instead of requiring the existence of a particular fresh name for a certain judgment to hold, the cofinite approach requires that the judgment holds for all variables not belonging to

a finite set L. This approach is justified by the fact that it gives rise to stronger inversion and induction principles compared to the "exists-fresh" style of quantification, with consequent increased ease of use. This method is then compared to the already existing locally nameless representation approaches, highlighting its benefits.

The new approach has been employed to develop several examples in the Coq proof assistant[1]: type soundness for the simply-typed λ-calculus, System $F_{<:}$ and ML, as well as subject reduction for the Calculus of Constructions. Since the proof of type soundness for System $F_{<:}$ is one of the challenges of the POPLMark benchmark, the authors' development has been compared to several other mechanizations and techniques. The metrics used include the number of lemmas and steps of reasonings, where the latter is defined as the number of non-trivial tactics invoked. Overall, the provided approach appears to require less infrastructure than most of the others and the proofs are generally close to their informal equivalents.

Discussion

Despite being published almost two decades ago, several of the points addressed in this paper remain accurate and valuable. First of all, the landscape of the formalization of programming language metatheory is still very similar: there is a wide variety of possible tools and approaches, that excel in some situations and are less appropriate in others. The problem of representing binder constructs is still central, and the overview of binding encoding techniques is still useful, although in part outdated (for example, there is an increased support of proof assistants based on HOAS representations [27]). Adequacy of the encoding is mentioned and discussed. Additionally, the discussion on the difficulties to find good metrics to evaluate a mechanization is interesting and on point: the authors acknowledge that there are no meaningful metrics across different proof assistants, and that counting the number of invoked tactics can also be misleading, since the amount of automation may differ between users.

Although no binding encoding technique has emerged as the best, we cannot say that this paper has not been successful: the proposed locally nameless with cofinite quantification approach has been rather widely employed, as witnessed by its high number of citations and the development of dedicated tools [8].

3 Open Problems

3.1 Formalizing Reversible Concurrent Calculi

Mechanizing systems and the proofs of their properties significantly raises the confidence in their correctness and provides a valuable means of detecting subtle defects or gaps in reasoning. In contrast to non-reversible programming languages or process algebra, formalizations of models of reversible computation are less common and focus on the sequential setting: the development of *"A Certified Study of a Reversible Programming Language"* [24] is one of the few notable examples. To the best of my knowledge, apart from the formalization of CCSK$^{\mathrm{P}}$ that I developed earlier in my Ph.D. [13], there are currently no other formalizations of reversible concurrent calculi.

In this context, mechanizing the results in *"An Axiomatic Theory for Reversible Computation"* [19] would be particularly valuable. In addition to verifying the correctness of the proposed approach, encoding the fundamental axioms of reversibility and their consequent properties would yield practical benefits: it would enable the development of a mechanized framework for checking the soundness of concrete reversible concurrent models.

[1] All the formalized case studies are available in the hyperlinked GitHub repository.

3.2 Reversible Notions of Bisimulation and Congruence

Different notions of bisimulations capture different behavioral equivalences of processes; this is especially true in the reversible setting. As shown in *"Locality and Interleaving Semantics in Calculi for Mobile Processes"* [28] and *"Bisimulations and Reversibility"* [3], the choice of which behavioral aspects should be observed varies along several orthogonal dimensions: the power of observation on processes, the degree of abstraction from internal transitions, invariance under key renaming, the inclusion of backward moves in the bisimulation definition, or the ability to distinguish true-concurrency behaviors. Several of these dimensions are yet to be explored in the reversible setting, and a unified, comprehensive view is missing. To address this problem, one challenge to overcome is the identification of an appropriate notion of context for defining congruences induced by bisimulations. Moreover, as highlighted in *"The Concurrent Calculi Formalisation Benchmark"* [12], mechanizing these coinductive notions and results in a proof assistant is not straightforward.

3.3 Benchmark Challenges for the Mechanization of Concurrent Models

The POPLMark challenge significantly increased the community's interest in the mechanization of programming languages metatheory and led to the development of new tools and techniques, such as the locally nameless approach introduced in *"Engineering Formal Metatheory"* [7]. Following in its footsteps, challenges focused on the mechanization of proofs involving logical relations [1] and models of concurrency (*"The Concurrent Calculi Formalisation Benchmark"* [12]) have emerged. For the latter, solutions to the linearity and scope extrusion challenges have already been collected (including [14] for the second challenge); however, the third challenge about coinductive reasoning remains open. Together with Lea Trogni and Dr. Alberto Momigliano, I am currently working on a solution to this problem in the proof assistant Beluga [27]. In the future, a potential research direction is the development of a set of benchmark problems specifically devoted to reversibility, with the goal of increasing the visibility of reversible computation within the interactive theorem proving community and encouraging new mechanizations of reversible models.

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark Reloaded: Mechanizing Proofs by Logical Relations*. *Journal of Functional Programming* 29, p. e19, doi:10.1017/S0956796819000170.

[2] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen & Enrico Tassi (2011): *The Matita Interactive Theorem Prover*. In Nikolaj Bjørner & Viorica Sofronie-Stokkermans, editors: *Automated Deduction – CADE-23*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 64–69, doi:10.1007/978-3-642-22438-6_7.

[3] Clément Aubert, Iain Phillips & Irek Ulidowski (2026): *Bisimulations and Reversibility*. In Claudio Antares Mezzina & Alan Schmitt, editors: *Components Operationally: Reversibility and System Engineering: Essays Dedicated to Jean-Bernard Stefani on the Occasion of His 65th Birthday*, Springer Nature Switzerland, Cham, pp. 46–67, doi:10.1007/978-3-031-99717-4_3.

[4] Clément Aubert (2024): *The Correctness of Concurrencies in (Reversible) Concurrent Calculi*. *Journal of Logical and Algebraic Methods in Programming* 136, p. 100924, doi:10.1016/j.jlamp.2023.100924.

[5] Clément Aubert & Ioana Cristescu (2020): *How Reversibility Can Solve Traditional Questions: The Example of Hereditary History-Preserving Bisimulation*. In Igor Konnov & Laura Kovács, editors: *CONCUR 2020*, *LIPIcs* 2017, Schloss Dagstuhl, pp. 13:1–13:24, doi:10.4230/LIPIcs.CONCUR.2020.7.

[6] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: the PoplMark Challenge*. In Joe Hurd & T. Melham, editors: *TPHOLs 2005*, LNCS, Springer, pp. 50–65, doi:10.1007/11541868_4.

[7] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack & Stephanie Weirich (2008): *Engineering Formal Metatheory*. In: *POPL '08*, Association for Computing Machinery, New York, NY, USA, p. 3–15, doi:10.1145/1328438.1328443.

[8] Brian E. Aydemir & Stephanie Weirich (2010): *LNgen: Tool Support for Locally Nameless Representations*. Available at `https://repository.upenn.edu/handle/20.500.14332/7902`.

[9] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-07964-5.

[10] Michele Boreale & Davide Sangiorgi (1998): *A Fully Abstract Semantics for Causality in the π-Calculus*. *Acta Inform.* 35(5), pp. 353–400, doi:10.1007/s002360050124.

[11] Gérard Boudol, Ilaria Castellani, Matthew Hennessy & Astrid Kiehn (1994): *A Theory of Processes with Localities*. *Form. Asp. Comput.* 6(2), p. 165–200, doi:10.1007/BF01221098.

[12] Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Tirore, Martin Vassor, Nobuko Yoshida & Daniel Zackon (2024): *The Concurrent Calculi Formalisation Benchmark*. In Ilaria Castellani & Francesco Tiezzi, editors: *COORDINATION 2024*, Springer Nature Switzerland, Cham, pp. 149–158, doi:10.1007/978-3-031-62697-5_9.

[13] Gabriele Cecilia (2025): *A Formalization of the Reversible Concurrent Calculus CCSKP in Beluga*. In Clément Aubert, Cinzia Di Giusto, Simon Fowler & Violet Ka I Pun, editors: *ICE 2025*, EPTCS, pp. 55–72, doi:10.4204/EPTCS.425.5.

[14] Gabriele Cecilia & Alberto Momigliano (2024): *A Beluga Formalization of the Harmony Lemma in the π-Calculus*. In Florian Rabe & Claudio Sacerdoti Coen, editors: *LFMTP 2024*, *Electronic Proceedings in Theoretical Computer Science* 404, Open Publishing Association, pp. 1–17, doi:10.4204/EPTCS.404.1.

[15] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR 2004*, *LNCS* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[16] Pierpaolo Degano & Corrado Priami (2001): *Enhanced Operational Semantics*. *ACM Comput. Surv.* 33(2), pp. 135–176, doi:10.1145/384192.384194.

[17] Robert Glück, Ivan Lanese, Claudio Antares Mezzina, Jarosław Adam Miszczak, Iain Phillips, Irek Ulidowski & Germán Vidal (2023): *Towards a Taxonomy for Reversible Computation Approaches*. In: *RC 2023*, Springer-Verlag, Berlin, Heidelberg, p. 24–39, doi:10.1007/978-3-031-38100-3_3.

[18] Rolf Landauer (1961): *Irreversibility and Heat Generated in the Computing Process. IBM Journal of Research and Development* 5, pp. 183–191, doi:10.1147/rd.53.0183.

[19] Ivan Lanese, Iain Phillips & Irek Ulidowski (2024): *An Axiomatic Theory for Reversible Computation. ACM Trans. Comput. Logic* 25(2), doi:10.1145/3648474.

[20] Tom F. Melham (1994): *A Mechanized Theory of the π-Calculus in HOL. Nordic J. of Computing* 1(1), p. 50–76, doi:10.48456/tr-244.

[21] Robin Milner (1980): *A Calculus of Communicating Systems.* LNCS, Springer-Verlag, doi:10.1007/3-540-10235-3.

[22] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I. Inf. Comput.* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[23] Monica Nesi (1992): *A Formalization of the Process Algebra CCS in High Order Logic.* Technical Report UCAM-CL-TR-278, University of Cambridge, Computer Laboratory, doi:10.48456/tr-278.

[24] Luca Paolini, Mauro Piccolo & Luca Roversi (2018): *A Certified Study of a Reversible Programming Language.* In Tarmo Uustalu, editor: *TYPES 2015*, *Leibniz International Proceedings in Informatics (LIPIcs)* 69, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 7:1–7:21, doi:10.4230/LIPIcs.TYPES.2015.7.

[25] Doron A. Peled (2019): *Formal Methods*, pp. 193–222. Springer International Publishing, Cham, doi:10.1007/978-3-030-00262-6_5.

[26] Iain Phillips & Irek Ulidowski (2007): *Reversing Algebraic Process Calculi. J. Log. Algebr. Program.* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[27] Brigitte Pientka & Jana Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description).* In Jürgen Giesl & Reiner Hähnle, editors: *IJCAR 2010*, *Lecture Notes in Computer Science* 6173, Springer, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[28] Davide Sangiorgi (1996): *Locality and Interleaving Semantics in Calculi for Mobile Processes. Theor. Comput. Sci.* 155(1), pp. 39–83, doi:10.1016/0304-3975(95)00020-8.