COST Action IC1405
Reversible Computation
Extending Horizons of Computing

Working Group 2 - Software and Systems

Year-End Report

Editors:
Claudio Antares Mezzina and Rudolf Schlatte

Contributors:
Paola Giannini, James Hoey, Ivan Lanese,
Claudio Antares Mezzina, Jaroslaw Miszczak,
Rumyana Neykova, Jorge A. Pérez, Ulrik Schultz,
Harun Siljak, Irek Ulidowski, German Vidal

May 2019

1 Introduction

In this document we report the research activity that the Working Group 2 (WG2) on Software and System has actively carried out during the last 3 years of the Action. According to the Action's Memorandum of Understanding [35] the WG2 mission is to provide linguistic abstractions, languages and tools to develop safer and more reliable applications. To do so, one of the main objective is to exploit reversibility to define actual reliability constructs and frameworks for programming reliable applications. This implies on one hand to integrate reversibility with mainstream programming languages and well know paradigms and on the other hand to understand what is the connection between reversibility and established techniques to achieve reliability such as transactions and checkpointing. Lastly, WG2 topics also include reversibility in robotics and control theory.

Progress of the WG. This document builds on the year two [25] and year three [26] reports, by recalling their content and structure. The idea is to create a whole document which witness the activity of the WG2 in the period 2016-2019. For an exhaustive analysis on the state of the art on the main topics of the Action we refer to [27]. Here we report the improvements of the state of the art with respect of the following topics:

- type systems, with a special focus on behavioural types (e.g., session types)
- mainstream languages (e.g., Erlang [1])
- modularity aspects such as classes, objects and components
- recovery techniques in massage passing concurrency model
- shared memory systems
- quantum computing
- control theory

2 Session Types

The interest in session types stems from the fact that working with a system whose behaviour (in terms of communications) is strongly disciplined by a type theory is easier.

Reversibility and monitored semantics for *binary session types* has been recently studied by Mezzina and Pérez [30, 28]. In their works, they propose a *monitor as memory* mechanism in which information about the monitor of a process can be used to enable its reversibility. Moreover, by adding *modalities* information at the level of session types, reversibility can be controlled.

In the context of multiparty session types, *global types* describe the message-passing behaviour of a set of participants in a system from a global point of view. A global type can be projected onto each participant so as to obtain local types, which describe individual contributions to the global protocol. The work [29] extends global and local types to keep track of the stage of the protocol that

has been already executed; this enables reversible steps in an elegant way. The authors develop a rigorous process framework for multiparty communication, which improves over prior works by featuring asynchrony, decoupled rollbacks and process passing. In this framework, concurrent processes are untyped but their forward and backward steps is governed by monitors. The main technical result is that the developed multiparty reversible semantics is causally-consistent. Finally, [7] proposes a Haskell implementation of the asynchronous reversible operational semantics for multiparty session types proposes in [29]. The implementation exploits algebraic data types to faithfully represent three core ingredients: a process calculus, multiparty session types, and forward and backward reduction semantics. This implementation bears witness to the convenience of pure functional programming for implementing reversible languages.

In a series of works [8, 5] *multiparty session types* (aka global types) have been enriched with checkpoint labels on choices that mark points of the protocol where the computations may roll back. In [8] a simple model in which rollback could be done any time after a participant had crossed the checkpointed choice. In [5] a more refined model is presented, in which the programmer can define points where the computation may revert to a checkpointed label, and rollback has to be triggered by the participant that made the decision.

Behavioural contracts are abstract descriptions of expected communication patterns followed by either clients or servers during their interaction. Behavioural contracts come naturally equipped with a notion of *compliance*: when a client and a server follow compliant contracts, their interaction is guaranteed to progress or successfully complete. In [3] two extensions of behavioural contracts, *retractable contracts* dealing with *backtracking* and *speculative contracts* dealing with *speculative execution* are studied. The two extensions give rise to *the same notion of compliance*. As a consequence, they also give rise to the same *subcontract relation*, which determines when one server can be replaced by another preserving compliance. Moreover, compliance and subcontract relation are both decidable in quadratic time. Finally, the relationship between retractable contracts and calculi for reversible computing is studied.

3 Erlang

A formal reversible causal semantics for (core) Erlang has been presented in [23]. Then, on top of it a rollback operator is built, which can be used to undo the actions of a process up to a given checkpoint. By exploiting this causal semantics and the rollback operator, the first reversible debugger for Erlang is presented in [22]. The debugger may help programmers to detect and fix various kinds of bugs, including message order violations and livelocks.

Debugging of concurrent systems is a tedious and error-prone activity. A main issue is that there is no guarantee that a bug that appears in the original computation is replayed inside the debugger. This problem is usually tackled by so-called replay debugging, which allows the user to record a program execution and replay it inside the debugger. In [24] a novel technique for replay debugging called *controlled causal-consistent replay* is presented. Controlled causal-consistent replay allows the user to record a program execution and, in contrast to traditional replay debuggers, to reproduce a visible misbehaviour inside the debugger including all *and only* its causes. In this way, the user is

not distracted by the actions of other, unrelated processes.

4 Modularity Aspects

Initial ideas for the design and implementation of reversible object-oriented language have been presented, based on extending Janus with object-oriented concepts such as classes that encapsulate behavior and state, inheritance, virtual dispatching, as well as constructors. Schultz and Axelsen showed that virtual dispatching can be seen as reversible decision mechanism that is easily translatable to a standard reversible programming model such as Janus, and argued that reversible management of state can be accomplished using reversible constructors [33]. These concepts were informally described and implemented by source-to-source translation from the reversible object-oriented language Joule to Janus. A similar design was adopted for the ROOP reversible object-oriented language, but fully formalized and implemented by compiling to the Pendulum reversible ISA [17, 18]. ROOP demonstrates how to generate low-level code implementing reversible vtable-based dispatching, and investigates the restrictions that must be imposed on reversible object-oriented programs to avoid run-time aliasing checks as found in Joule. The original Joule prototype relied on static and stack allocation of objects, which does not permit garbage-free OO programming: common patterns such as factories are for example not possible [33]. The initial presentation of the ROOPL language relied exclusively on stack allocation [18], but has subsequently been extended with a heap-based memory manager [6] — this is however a significantly more complex system that may not be entirely garbage-free [2].

In [11] reversibility in the field of component-based language is studied.

5 Recovery

Distributed programs are hard to get right because they are required to be open, scalable, long-running, and tolerant to faults. In particular, the recent approaches to distributed software based on (micro-)services where different services are developed independently by disparate teams exacerbate the problem. In fact, services are meant to be composed together and run in open context where unpredictable behaviours can emerge. This makes it necessary to adopt suitable strategies for monitoring the execution and incorporate recovery and adaptation mechanisms so to make distributed programs more flexible and robust. The typical approach that is currently adopted is to embed such mechanisms in the program logic, which makes it hard to extract, compare and debug. An approach that employs formal abstractions for specifying failure recovery and adaptation strategies has been proposed in [4]. Although implementation agnostic, these abstractions would be amenable to algorithmic synthesis of code, monitoring and tests. Message-passing programs (à la Erlang, Go, or MPI) are considered, since they are gaining momentum both in academia and industry. In [9] an instance of the framework proposed in [4] is given. More precisely, this approach imbues the communication behaviour of multi-party protocols with minimal decorations specifying the conditions triggering monitor adaptations. It is then shown that, from these extended global descriptions, one can (i) syn-

thesise actors implementing the normal local behaviour of the system prescribed by the global graph, but also (ii) synthesise monitors that are able to coordinate a distributed rollback when certain conditions (denoting abnormal behaviour) are met. The synthesis algorithm produces Erlang code. More precisely, for each role in the global description are generated two Erlang actors: one implementing the normal (forward) behaviour of the system and a second one (the monitor) in charge of implementing the reversible behaviour of the role. When certain condition are met at runtime, then the monitors will coordinate each other in order to bring back, if possible, the system. One interesting property of such approach is that the two semantics are highly decoupled, meaning that the system is always able to normally execute (e.g., going forward) even in case of some monitor crash.

A static analysis based on multiparty session types that can efficiently compute a safe global state from which a system of interacting processes should be recovered, has been integrated with the Erlang recovery mechanism [31]. From a global description of the program communication flow, given in multiparty protocol specification, causal dependencies between processes are extracted. This information is then used at runtime by a recovery mechanism, integrated in Erlang, to determine which process has to be terminated and which one has to be restarted upon a node failure. Experimental results indicate that the proposed framework outperforms a built-in static recovery strategy in Erlang when a part of the protocol can be safely recovered.

In [10] a rollback operator, based on the notion of causal-consistent reversibility, is defined for a language with shared memory. A rollback is defined as the minimal causal-consistent sequence of backward steps able to undo a given action. In [36] the relationship between a distributed checkpoint/rollback scheme based on causal logging, called Manetho, and a reversible concurrent model of computation, based on the π -calculus with imperative rollback called `roll`-π. A rather tight relationship between rollback based on causal logging as performed in Manetho and the rollback algorithm underlying `roll`-π. The main result is that the latter can faithfully simulate Manetho, where the notion of simulation we use is that of weak barbed simulation, and that the converse only holds if possible rollbacks in are restricted.

6 Shared memory

The interplay between reversibility and imperative programming language with shared memory is studied in [20, 21]. More in details, it is shown how to reverse a while language extended with blocks, local variables, procedures and the interleaving parallel composition. Annotation is defined along with a set of operational semantics capable of storing necessary reversal information, and identifiers are introduced to capture the interleaving order of an execution. Inversion is defined with a set of operational semantics that use saved information to undo an execution. It is then shown that annotation does not alter the behaviour of the original program, and that inversion correctly restores the initial program state.

In [19] a state-saving approach to reversible execution of imperative programs containing parallel composition is presented. Given an original program, we produce an annotated version of the program that both performs forwards

execution and all necessary state-saving of required reversal information. Further, the authors produce an inverted version of our program, capable of using this saved information to reverse the effects of each step of the forwards execution. Then, they show that this process implements correct and garbage-free inversion. Finally, the performance and overheads associated with state-saving and inversion is evaluated.

7 Quantum Computing

In [16] it is described QSWalk.jl package for Julia programming language [14], developed to simulate the evolution of open quantum systems. The package enables the study of reversible quantum procedures developed using stochastic quantum walks on arbitrary directed graphs. A detailed description of the implemented functions is provided along with some usage examples. The package has been used for the purpose of determining the differences between limiting properties in various models of quantum stochastic walks [15].

The idea of using Julia as a host language for the simulator of quantum computing was motivated by the solid support for numerical procedures available in this language. Moreover, the strong typing capabilities of Julia has been used for developing type hierarchy for various models of quantum walks. The implementation of this hierarchy is available as a library of functions [13]. This library has been used for proposing a framework suitable for analysing the efficiency of attacks on quantum search algorithms [12].

8 Control Theory

Petri nets are a formalism for modelling and reasoning about the behaviour of distributed systems. Recently, a reversible approach to Petri nets, Reversing Petri Nets (RPN), has been proposed, allowing transitions to be reversed spontaneously in or out of causal order. In [32, 34] proposes an approach for controlling the reversal of actions of an RPN, by associating transitions with conditions whose satisfaction/violation allows the execution of transitions in the forward/reversed direction, respectively. We illustrate the framework with a model of a novel, distributed algorithm for antenna selection in distributed antenna arrays. contributed to interfacing reversible computing optimisation and control systems, represented by Reversing Petri Nets, with the irreversible environment, solving the problem of antenna selection in radio communications.

References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in Erlang (2nd edition)*. Prentice Hall, 1996.

[2] Holger Bock Axelsen and Robert Glück. Reversible representation and manipulation of constructor terms in the heap. In *RC2013*, pages 96–109. Springer, 2013.

[3] Franco Barbanera, Ivan Lanese, and Ugo de'Liguoro. A theory of retractable and speculative contracts. *Sci. Comput. Program.*, 167:25–50, 2018.

[4] Ian Cassar, Adrian Francalanza, Claudio Antares Mezzina, and Emilio Tuosto. Reliability and fault-tolerance by choreographic design. In Adrian Francalanza and Gordon J. Pace, editors, *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017.*, volume 254 of *EPTCS*, pages 69–80, 2017.

[5] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Concurrent reversible sessions. 2017.

[6] Martin Holm Cservenka. Design and implementation of dynamic memory management in a reversible object-oriented programming language. Master's thesis, DIKU, University of Copenhagen, 2018.

[7] Folkert de Vries and Jorge A. Pérez. Reversible session-based concurrency in haskell. In Michal H. Palka and Magnus O. Myreen, editors, *Trends in Functional Programming - 19th International Symposium, TFP Revised Selected Papers*, volume 11457 of *Lecture Notes in Computer Science*, pages 20–45. Springer, 2018.

[8] Mariangiola Dezani-Ciancaglini and Paola Giannini. Reversible multiparty sessions with checkpoints. In *EXPRESS/SOS'16*, volume 222 of *EPTCS*, pages 60–74, 2016.

[9] Adrian Francalanza, Claudio Antares, and Emilio Tuosto. Reversible choreographies via monitoring in erlang. In Silvia Bonomi and Etienne Rivière, editors, *Distributed Applications and Interoperable Systems - 18th IFIP WG 6.1 International Conference, DAIS 2018*, Lecture Notes in Computer Science. Springer, 2018. to appear.

[10] Elena Giachino, Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent rollback in a tuple-based language. *J. Log. Algebr. Meth. Program.*, 88:99–120, 2017.

[11] Vaidas Giedrimas. Reversibility in component-based programming language. In *The IEEE 12th International Conference on Application of Information and Communication Technologies*. IEEE, 2018.

[12] A. Glos and J.A. Miszczak. Impact of the malicious input data modification on the efficiency of quantum algorithms. *arXiv:1802.10041*, 2018.

[13] A. Glos and J.A. Miszczak. QuantumWalk.jl: Package for building algorithms based on quantum walks, 2018. https://github.com/QuantumWalks/QuantumWalk.jl.

[14] A. Glos, J.A Miszczak, and M Ostaszewski. QSWalk.jl: simulating the evolution of open quantum systems on graphs, 2017. https://github.com/QuantumWalks/QSWalk.jl.

[15] A. Glos, J.A. Miszczak, and M. Ostaszewski. Limit properties of global interaction stochastic quantum walks on directed graphs. *J. Phys. A: Math. Theor*, 51:035304, 2018. arXiv:1703.01792.

[16] A. Glos, J.A. Miszczak, and M. Ostaszewski. Qswalk. jl: Julia package for quantum stochastic walks analysis. *arXiv:1801.01294*, 2018.

[17] T. Haulund. Design and implementation of a reversible object-oriented programming language. Master's thesis, University of Copenhagen, DIKU, 2016.

[18] Tue Haulund, Torben Ægidius Mogensen, and Robert Glück. Implementing reversible object-oriented language features on reversible machines. In Iain Phillips and Hafizur Rahaman, editors, *Reversible Computation*, pages 66–73, Cham, 2017. Springer International Publishing.

[19] J. Hoey and I. Ulidowski. Reversible imperative parallel programs and debugging. In *Reversible Computation - 11th International Conference, RC 2019*, Lecture Notes in Computer Science. Springer, 2019. To appear.

[20] James Hoey, Irek Ulidowski, and Shoji Yuen. Reversing imperative parallel programs. In Kirstin Peters and Simone Tini, editors, *Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS*, volume 255 of *EPTCS*, pages 51–66, 2017.

[21] James Hoey, Irek Ulidowski, and Shoji Yuen. Reversing parallel programs with blocks and procedures. In *Combined Proceedings of EXPRESS/SOS 2018.*, volume 276 of *EPTCS*, pages 69–86, 2018.

[22] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. Cauder: A causal-consistent reversible debugger for erlang. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*, volume 10818 of *Lecture Notes in Computer Science*, pages 247–263. Springer, 2018.

[23] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for erlang. *J. Log. Algebr. Meth. Program.*, 100:71–97, 2018.

[24] Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay debugging for message passing programs. In *FORTE*, LNCS. Springer, 2019. to appear.

[25] Claudio Antares Mezzina and Rudolf Schlatte (eds). Working Group 2, Software and Systems, Report of the Second Year. COST Action IC1405, Reversible Computation, 2016. `http://topps.diku.dk/ic1405/wg2_yearendreport2017.pdf`.

[26] Claudio Antares Mezzina and Rudolf Schlatte (eds). Working Group 2, Software and Systems, Report of the Third Year. COST Action IC1405, Reversible Computation, 2016. `http://topps.diku.dk/ic1405/wg2_yearendreport2018.pdf`.

[27] Claudio Antares Mezzina and Rudolf Schlatte (eds). State of the art report, Working Group 2, Software and Systems. COST Action IC1405, Reversible Computation, 2017. http://topps.diku.dk/ic1405/WG2yearendreport2017.pdf.

[28] Claudio Antares Mezzina and Jorge A. Pérez. Reversible sessions using monitors. In *Proceedings of the Ninth workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016, Eindhoven, The Netherlands, 8th April 2016.*, volume 211 of *EPTCS*, pages 56–64, 2016.

[29] Claudio Antares Mezzina and Jorge A. Pérez. Causally consistent reversible choreographies: a monitors-as-memories approach. In Wim Vanhoof and Brigitte Pientka, editors, *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, pages 127–138. ACM, 2017.

[30] Claudio Antares Mezzina and Jorge Andrés Pérez. Reversible semantics in session-based concurrency. In *Proceedings of the 17th Italian Conference on Theoretical Computer Science, Lecce, Italy, September 7-9, 2016.*, volume 1720 of *CEUR Workshop Proceedings*, pages 221–226. CEUR-WS.org, 2016.

[31] Rumyana Neykova and Nobuko Yoshida. Let It Recover: Multiparty Protocol-Induced Recovery. In *26th International Conference on Compiler Construction*, pages 98–108. ACM, 2017.

[32] Anna Philippou, Kyriaki Psara, and Harun Siljak. Controlling reversibility in reversing Petri nets with application to wireless communications. In *Proceedings of RC 2019*. Springer, 2019.

[33] Ulrik Pagh Schultz and Holger Bock Axelsen. Elements of a reversible object-oriented language. In Simon Devitt and Ivan Lanese, editors, *Reversible Computation*, pages 153–159, Cham, 2016. Springer International Publishing.

[34] Harun Siljak, Kyriaki Psara, and Anna Philippou. Distributed antenna selection for massive mimo using reversing petri nets. *IEEE Wireless Communications Letters (under review)*, 2019.

[35] Irek Ulidowski. IC1405 - Reversible Computation: extending horizons of computing - Memorandum of Understanding. https://e-services.cost.eu/files/domain_files/ICT/Action_IC1405/mou/IC1405-e.pdf.

[36] Martin Vassor and Jean-Bernard Stefani. Checkpoint/rollback vs causally-consistent reversibility. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation*, volume 11106 of *Lecture Notes in Computer Science*, pages 286–303. Springer, 2018.