ISSN 2186-7437

NII Shonan Meeting Report

No. 2019-149

Programming Languages for Distributed Systems

Philipp Haller
Guido Salvaneschi
Takuo Watanabe
Gul Agha

May 27–30, 2019

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Programming Languages for Distributed Systems

Organizers:
Philipp Haller (KTH Royal Institute of Technology, Sweden)
Guido Salvaneschi (Technical University of Darmstadt, Germany)
Takuo Watanabe (Tokyo Institute of Technology, Japan)
Gul Agha, (University of Illinois at Urbana-Champaign, USA)

May 27–30, 2019

Programming languages for distributed systems have been flourishing in the past, providing ideas for the development of complex systems that have been deeply influential. However, over the last few years, researchers focusing on this area are scattered across different communities with limited interaction. The goal of this Meeting is to build a community of researchers interested in programming languages for distributed systems, share current research results and set up a common research agenda.

Developing distributed systems is a well-known, decades-old problem in Computer Science. Despite significant research effort has been dedicated to this area, developing distributed systems remains challenging. To complicate things, over the last years, we observed the rise of a complex scenario, with heterogeneous platforms, interconnected systems and decentralized functionalities. This model, referred to as edge computing, is driven – among others – by forces like the need of reducing latency in geodistributed services via multiple data centers, the advent of purpose- driven microservices, and the spreading of in-field decision intelligence in the Internet of Things.

Programming languages are a fundamental tool to face the complexity of such a scenario. In comparison, however, these have seen modest improvements. Arguably, many ideas for supporting distribution adopted in production languages date back to the early '90s with CORBA/RMI, and even earlier with the Actor model and concurrent objects. In contrast, language abstractions have been mostly a key asset for determining the popularity of dedicated systems. For example, MapReduce takes advantage of purity to parallelise task processing, complex event processing adopts declarative programming to express sophisticated event correlations and Spark leverages functional programming for efficient fault recovery via lineage – to provide some examples.

There have been, notable advances in research on programming languages for distributed systems, such as cloud types for eventual consistency, conflict-free replicated data types (CRDT), language support for safe distribution of computations and fault tolerance, as well as programming frameworks for mixed IoT/Cloud development, such as Ericsson's Calvin – just to mention a few. However, these efforts have seen limited adoption and the researchers that have

been carrying out these efforts are scattered across different communities that include verification and formal methods in general, programming language design, database theory, distributed systems, systems programming, data-centric programming, and web application development.

In contrast, the field of programming languages for distributed systems has been a flourishing research area in the past, with influential contributions like Argus, Emerald, and Distributed OZ.

Overview of Talks

Programming Languages for Distributed Systems: Open Issues and Prospects

Gul Agha, University of Illinois, USA

The assumptions in the Actor model make it natural programming model for distributed systems. These assumptions include concurrency, asynchronous operation, autonomy, encapsulation (isolation) of state. Actors interact by sending each other messages. A major advantage of actors is that they enable a "big step" semantics: the behavior of individual actors in response to a message is atomic and thus can be considered as a macro-step, thus maintaining a separaration of what (interface) is done from how (representation) it is done. Moroeover, a consequence of the macro-steps is that much inconsequential nondeterminism need not be considered. Not surprisingly, Actor programming languages have been used to develop numerous large-scale commercial systems such as LinkedIn, Facebook Chat, the British National Health Service Portal, and fintech applications.

While it is useful to think in terms of actors, programming distributed systems remains complicated. A simple way to describe the challenge is that high-level actor programming needs abstractions to represent variants of "who, when, where, how long, and with what certainty?"

– Consider "who?" In the Actor model, each actor has a unique address which must be known before a message can be sent to that actor (a security property called locality which simplifies reasoning). However, in some applications, it is useful to send messages to actors by specifying a type (or property) of actors rather than individual addresses which may not be known to the sender– something captured in programming models such as ActorSpaces and Linda.

– Consider "when?" In sequential systems, programmers are responsible for specifying a total order of events in a system. This results in overly constraining when events may occur. In contrast, concurrent systems allow nondeterministic interleaving of messages processed by autonomous actors. However, without additional constraints on the order of events at participating actors, an interleaving may lead to incorrect operations–for example, one that results in a deadlock. Programming constructs such as Transactions, Synchronizers and (Multiparty) Session Types have been proposed to address this issue.

– While the Actor model assumes location transparency, depending on the runtime and underlying network, the question of where an actor is executed may have implications for execution efficiency, load balancing, and security. For example, the Charm++ system has to address this issue for efficiently processing scientific code on parallel computers.

– For modeling real-time computations, questions about communication delays and computation time ("how long") must be resolved.

– Scalable systems exhibit stochastic behaviors, for example, due to message delays, which may need to be constrained in order to guarantee emergent properties such as stability. Languages such as pMaude attempt to do this.

The challenge simplifying distributed programming remains very much open: while the above proposals represent progress, they are piecemeal and incomplete. The coming decade should provide exciting developments in the are of

Programming Languages for Distributed Systems, with new abstractions and implementations building on the basics of the Actor model in novel directions.

High Availability under Eventual Consistency

Carlos Baquero, Universidade do Minho, Portugal

When nodes are spread across large geographic distances, being available for local users, and providing short response times is often at odds with keeping strong consistency across the whole system. Several systems, that target large scale geo-replication, support multi-master operation and transient data divergence, allowing each site to update replicas with no immediate coordination. From the user application perspective, the system cannot be seen anymore as a single sequential copy, since now operations can be processed concurrently at different locations. Conflict-free Replicated Data Types (CRDTs) can take away a lot of the complexity when migrating from a sequential to a concurrent setting.

Here we explore a bit of the path in this transition, cover what can be expected, and present a few guiding principles, such as: (1) permutation equivalence; (2) preserving sequential semantics; and (3) non-sequential outcomes. Evolution from sequential to concurrent behaviour was explored for common data types, such as counters, registers and sets.

Durable Functions for Serverless Computing

Sebastian Burckhardt, Microsoft Research, USA

Serverless computing, often called Functions-as-a-Service, is a fast-growing paradigm that enables high productivity for cloud application developers. Functions can execute in response to external events (e.g. web requests) and can in turn call stateful services (e.g. cloud storage). In combination with a runtime that handles deployment and scale automatically, stateless functions open an appealingly simple and rapid way to cloud application development. However, for some workloads, stateless functions are inefficient and/or difficult to use. In particular, (1) excessive data movement can impact latency and throughput, (2) the possibility of failures (e.g. partial execution of functions) or any need for synchronization (e.g. to handle conflicts and races between functions) creates significant design challenges and complexity.

To address this, Durable Functions re-introduces durable state and synchronization into the serverless programming model, by extending stateless functions with the concepts of Orchestrations, Entities, and Critical Sections. Orchestrations solve the partial execution problem by tracking the execution of a multi-step function durably. Entities provide addressable stateful abstract data types, preventing excessive data-transfer and allowing sequencing of single-object operations. Finally, critical sections allow mutual exclusion between functions, providing a straightforward, familiar means for multi-object synchronization.

Towards a Distributed Actor-Reactor Programming Model

Wolfgang De Meuter, Vrije Universiteit Brussel, Belgium

Reactive programming and Distributed reactive programming is becoming a popular mainstream programming paradigm. However, even though there are many industrial strength incarnations of the idea, solid foundations are lacking. Questions such as what exactly constitutes reactive programming, how do reactive programs compose with each other and how reactive programs interact with non-reactive programs in a predictable manner are largely unaddressed in the scientific literature. In our research group, we are investigating a bipolar programming language design in which actors and reactors coexist with one another. Actors constitute the imperative parts of the system, i.e. they contain long lasting loops and they encapsulate the mutable state. Reactors host the parts of the system that are always(!) ready to react to events generated by third parties. Hence, reactors are always in O(1). The goal of the research is to define the distributed composition operators that allow actors and reactors to coexist in a predictable manner even in the face of partial failure.

Should Programming Languages Support Distributed Systems or Vice-versa?

Patrick Eugster, University of Lugano, Switzerland

There are several ways in which distributed systems can benefit from support from programming languages and vice versa. In this talk we outline 3 ongoing experiences: (1) Protocol types are a variant of session types targeted at verification of distributed systems software in the "middleware stack". The challenge here consists in reconciling tolerance to partial failures with coordinated (re)actions, while steering clear from impossibilities. (2) AEON is an actor-based distributed programming language that supports reasoning and programming across multiple actors in an atomic fashion, and adds a second layer of programming for capturing elasticity behavior. AEON reconciles high performance and "transaction-like" semantics by streamlining execution along a directed acyclic graph induced by an original ownership-based referencing discipline. Finally, (3) SASSY is a datacenter network architecture that supports communication with very tightly bounded delays, thus enabling the implementation of coordination protocols for synchronous systems which are much less costly, complex, and error-prone than their asynchronous counterparts. SASSY achieves this without overconstraining the entire network by logically splitting it into (a) an synchronous slice for time-sensitive coordination and (b) an asynchronous slice for regular best-effort traffic.

Programming Languages for Distributed Systems: From Languages to Services

Pascal Felber, University of Neuchâtel, Switzerland

Development of applications for distributed systems is notoriously complex as one has to deal with various problems such as partial failures, concurrency,

remote interactions, synchronization, etc. To capture this complexity and simplify the task of the programmer, it is tempting to develop new languages and tools tailored for distributed systems. But do we really need new languages? Our experience in designing such a high-level framework revealed that, while one can indeed hide much of the complexity of distributed systems, the programmer is still exposed to it for critical performance optimizations or in some corner cases. Furthermore, given the wide diversity of distributed programs ranging from simple client-server applications to large-scale peer-to-peer networks or stream-oriented systems, it is illusory to believe that a single dedicated language can satisfactorily target all domains. Instead, a more attractive alternative is to allow multiple languages to seamlessly co-exist in a distrusted system. Each component can be developed with the language most appropriate to the task, and software components can be packaged in micro-services, typically deployed within containers and interacting with one another through standard REST interface. As a consequence, one can provide efficient implementations by developing each component with the best language. The key remaining issue is to properly deploy and "orchestrate" the interactions between these components.

Building Correct-by-Design Highly Available Cloud Applications

Carla Ferreira, Universidade Nova de Lisboa, Portugal

Building trustworthy and highly available cloud applications is inherently complex and error-prone. Current industrial solutions are either ad-hoc solutions, relying heavily on the developer's expertise, or use protocols that blindly restrict concurrency reducing performance and availability. We have proposed sound techniques that either reduce or avoid coordinating the execution of operations, while enforcing the correctness of the application. To minimise coordination based on application specific invariants, an analysis technique was developed that identifies which operations would be unsafe under concurrent execution. The developer can then devise and formally check an optimized coordination protocol that guarantees the application invariants. Alternatively, coordination can be eschew altogether by extending operations with repair actions that ensure application invariants are always maintained. The modified operations preserve the original semantics when no conflicting updates occur. If a concurrent conflict occurs, we leverage conflict resolution policies to ensure a deterministic result that preserves the application invariants.

The Links Programming Language

Simon Fowler, University of Edinburgh, United Kingdom

The Links programming language pioneered the tierless/multi-tier/cross-tier paradigm of distributed (web) programming, allowing client, server, and database code to be written in a single uniform language.

Since the first paper about Links in 2006, much effort has gone into the development of the language, leading to many new features and changes. In this talk, I will give an overview of "modern Links", and describe the active projects currently extending the language.

As a concrete example, I will discuss recent work bringing distributed session-typed channels to the web setting, and how session-typed channels can be used to reimplement code which previously used RPC.

Language and Tool Support for (Available) Distributed Systems

Elisa Gonzalez Boix, Vrije Universiteit Brussel, Belgium

Even though distributed systems are wide-spread, building them is still hard due to their inherent concurrency combined with the fact that components can fail independently, i.e. failures are partial. In order to support the systematic construction of distributed systems, our research tries to aid developers with both novel programming constructs and tool support.

In this talk, I present our research on language support for replicated data. Replicating data in a distributed system can improve efficiency and availability but complicates software development because concurrent updates can lead to conflicts. In recent work (De Porre et al. 2019), we have introduced strong eventually consistent replicated objects (SECROs), a general-purpose data type for building available data structures that guarantee strong eventual consistency without requiring operations to commute. By specifying state validators arbitrary data types can be turned into highly available replicated data types. This means that replicated data types can be implemented similarly to their sequential local counterpart, with the addition of preconditions and postconditions to define concurrent semantics. We are currently working on improving the computation of valid executions to reduce runtime overhead.

To support the development of distributed applications, we study tool support in the form of debuggers. In recent work (Torres Lopez et al. 2019) presented in the demo session, we introduced multiverse debugging, a novel approach for debugging concurrent non-deterministic programs that allows developers to observe all possible execution paths of a program and debug it interactively in a fashion similar to breakpointed-based debuggers while being probe-effect free. This is meant to simplify the reproduction and inspection of concurrency bugs, because it removes chance and probability from the equation of hitting the problematic interleaving. Instead, an execution path that can lead to a bug can be explored interactively and a developer can see the state in all possible universes. Since exploring the multiverses of larger programs can become unwieldy, ongoing and future research is needed to guide the exploration of the state graph, e.g. novel stepping semantics that work at the level of universes, and to develop efficient runtime techniques that make multiverse debugging practical for distributed systems.

Processing Data in its Own Habit

Torsten Grust, Universität Tübingen, Germany

Relational database systems (RDBMSs) are widely used to persistently store and maintain data items. “Serious computation,” however, predominantly occurs outside of the database kernel: the persistent data is extracted and placed on some conventional programming language’s heap (provided that the heap is

able to hold the items at all), then processed, before the results are injected back into the database backend. This established approach exhibits a plethora of problems, among these (1) the possibly exorbitant price for data movement and (2) the sad fact that the RDBMS's carefully engineered abilities to process high-volume data simply go unused.

Here, instead, we describe methods to declaratively specify complex computations using recursive SQL user-defined functions. We admit a distinctively truly functional style of function formulation — foregoing non-intuitive fixpoint semantics as well as off-putting syntactic oddities — and compile such functions into efficient pure SQL that may be executed by the RDBMS itself, right next to the data and its supporting indexes. The compiler automatically discovers sharing and memoization opportunities during evaluation, and knows how to exploit linear as well as tail recursion .“Move you computations close to the data,” says Stonebraker and we heartily agree: RDBMSs make for capable, scalable, and declaratively programmable runtime systems. Use them as such!

Programming Models and Types for Reliability and Availability

Philipp Haller, KTH Royal Institute of Technology, Sweden

Distributed systems are often required to provide high scalability, reliability, and availability. We believe that programming languages play an important role in the development of distributed systems. Distributed systems are inherently concurrent, and are thus affected by a superset of hazards of concurrent software systems. Our research tries to address various hazards, including partial failure, data inconsistency, and race conditions by means of both dynamic and static approaches. In order to ensure fault tolerance, widely-used systems employ fault recovery mechanisms based on so-called lineages. A lineage maintains all information required to recover from the failure of a replica. In recent work (Haller et al. 2018), we present foundations for lineage-based distributed computation based on a programming model and type system. We prove that well-typed programs have two important properties: first, the mobility of lineages is preserved by reduction; second, materialization of remote, lineage-based data is finite. In ongoing and future work we study (a) problems related to interaction, including language constructs and type systems, and (b) problems related to latency and time. In all of these efforts, we consider modularity, scalability and availability to be essential aspects.

Session Type Implementations in Functional Programming Languages

Keigo Imai, Gifu University, Japan

We propose session-ocaml, a library for session-typed concurrent/distributed programming in OCaml. Our technique solely relies on parametric polymorphism, which can encode core session type structures with strong static guarantees. Our key ideas are: (1) polarised session types, which give an alternative formulation of duality enabling OCaml to automatically infer an appropriate

session type in a session with a reasonable notational overhead; and (2) a parameterised monad with a data structure called slots manipulated with lenses, which can statically enforce session linearity including delegations.

Separating Concerns in Concurrent Systems

Nadeem Jamali, University of Saskatchewan, Canada

Actors simplify distributed programming by abstracting over low-level networking and synchronization concerns, enabling greater concurrency in programs. However, programming of concurrent systems remains difficult, among other reasons because programmers using Actors try to build ever more complex systems. Our work attempts to address this challenge by separating a variety of concerns in Actor systems to improve modularity and reusability of code.

In two loosely related thrusts, we have worked on separating (1) location and resource concerns and (2) communication concerns.

The CyberOrgs model creates resource encapsulations for distributed Actor computations which can buy and sell resources. The encapsulations (called cyberorgs) then become spaces of resource certainty for computations. Cyberorgs negotiate contracts among themselves to secure resources in the future. Resources are modeled as ticks created at the hardware-level, and then steered by processor / network / caching schedulers toward their current owners. Cyberorgs can spawn new cyberorgs — offering autonomy to parts of their hosted computations — and can merge into other cyberorgs — giving up autonomy — by using isolate and assimilate primitives.

The interActors model puts communication at the center of an application, and actors carrying out computations connect to the communication at outlets and inlets. A communication itself is made up of special-purpose actors: in addition to the outlets and inlets, there are handler actors which orchestrate complex multi-stage communications. Although communications are intended to be programmed directly in a high-level language, they are compositionally defined in the sense that they can be constructed from primitive channels using three simple composition primitives. This compositional definition also allows for simpler compositions to be composed to create more complex ones.

My main question to the meeting participants is: "Actors allow highly complex systems to be built, with large numbers of diverse actors and complex communications. However, it appears that this full power is not being utilized. Large actor systems today often have just a few types of actors interacting in simple ways. What will it take to change this?"

Managing Dynamic Data

Alessandro Margara, Politecnico di Milano, Italy

Many modern distributed software applications involve processing, analyzing, and reacting to potentially large volumes of streaming data. Examples of such applications include monitoring systems, decision support systems, financial analysis tools and traffic control systems.

Designing and implementing these applications is difficult. Indeed, the streaming nature of the data demands for efficient algorithms, techniques, and

infrastructure to analyze the incoming information on the fly and extract relevant knowledge from it.

During this talk, I presented my experience in this area. I first focused on the design of a language explicitly conceived to identify situations of interest from large streams of low level data. Next, I discussed the implementation of efficient processing algorithms for streaming data that can exploit modern many-core architectures, such as multi-core CPUs and programmable GPUs, to reduce the processing latency and increase the overall throughput of the system. Finally, I presented the open issues and challenges in the field of stream processing, with emphasis on the integration within complex software architectures and the tradeoff between data consistency and data processing and management costs in such architectures.

Virtual Machine Support for Debugging Concurrency Bugs: Deterministic Replay, Asynchronous Snapshots, and Bug Detection for Event Loops

Stefan Marr, University of Kent, United Kingdom

Event-loop-based applications have a surprisingly high number of concurrency bugs. While they are rising in popularity with JavaScript, Akka, or other systems, debugging tools that support developers have not kept up. As a consequence, concurrency issues in event loop systems are very time consuming to track down issues. One reason is the inherent non-determinism. Another is that we rarely debug at the level of the chosen concurrency model, but at the level of threads and method calls.

In our work, we developed efficient deterministic record & replay and snapshotting approaches that allows us to analyze and debug a problem as often as we need to, to be able to fix a bug. Combined with our Kompos debugger, as presented in the demo session, these approaches give developers better tools to explore and understand complex concurrent interactions. Instead of debugging the implementation level, in Kompos we can reason about messages, event-loop turns, and promises, which reduces the semantic gap between programs and debugger.

We focused our work on the high-level support for concurrency models such as the actor model, but the underlying techniques are agnostic. With our work on performance, we hope that they are also practical enough to make their way into industrial systems.

A Parallel Object-Oriented Language for GPGPU

Hidehiko Masuhara, Tokyo Institute of Technology, Japan

This talk introduces our project Ikra, a data-parallel extension to Ruby for GPGPUs, and discuss its underlying implementations in relation to distributed programming. The goal of the project is to enable easy-to-write programming experiences into GPGPU, whose state-of-the-art programming style represent all data in the problem domain by using arrays of primitive data in C. Among the design and implementation issues in Ikra, this talk highlighted its support

for object-oriented programming, in particular its low-level memory layout optimized for efficient GPGPU computation and dynamic object allocation, which was not virtually possible before. The talk also presented a catalog of applications, including direct and Barns-Hut n-body simulations, an agent-based traffic simulation, and population dynamic simulations. They demonstrate the expressiveness of object-oriented support as well as the performance improvement based on the low-level optimization techniques.

Multitier Programming with ScalaLoci

Guido Salvaneschi, Technische Universität Darmstadt, Germany

Distributed applications are traditionally developed as separate modules, often in different languages, which react to events, like user input, and in turn produce new events for the other modules. Separation into components requires time-consuming integration. Manual implementation of communication forces programmers to deal with low-level details. The combination of the two results in obscure distributed data flows scattered among multiple modules, hindering reasoning about the system as a whole. The ScalaLoci distributed programming language addresses these issues with a coherent model based on placement types that enables reasoning about distributed data flows, supporting multiple software architectures via dedicated language features and abstracting over low-level communication details and data conversions. As we show, ScalaLoci simplifies developing distributed systems, reduces error-prone communication code and favors early detection of bugs.

Actors for Analysis

Marjan Sirjani, Malardalen University, Sweden

In the era of Cyber-Physical systems and Internet of Things, software system developers have to deal with increasing complexity of huge and heterogeneous systems. Building distributed, asynchronous, and event-based systems is a complicated task. Moreover, because of the dynamic and evolving nature of autonomous systems, a key challenge is providing runtime quality assurance techniques - in form of verification and performance analysis that can react to changes in a timely manner. Software needs to react to the uncertainty and possible changes in the system and environment. A family of actor-based languages are introduced to enable model driven development and provide a natural and usable model for building distributed, asynchronous, and event-based systems with least effort. To provide dependability in the context of a model-driven approach, model checking and simulation tools are built based on the formal semantics of the language. I will show how these models can be used in safety assurance and performance evaluation of different systems, like self-driving cars, Network on Chip architectures, sensor network applications, train scheduling, and flow management.

Towards Proving Runtime Properties of Data-Driven Systems Using Safety Envelopes

Carlos A. Varela, Rensselaer Polytechnic Institute, USA

Dynamic data-driven application systems (DDDAS) allow for unprecedented self-healing and self-diagnostic behavior across a broad swathe of domains. The usefulness of these systems is offset against their inherent complexity, and therefore fragility to specification or implementation error. Further, DDDAS techniques are often applied in safety-critical domains, where correctness is paramount. Formal methods facilitate the development of correctness proofs about software systems, which provide stronger behavioral guarantees than non-exhaustive unit tests. While unit testing can validate that a system behaves correctly in some finite number of configurations, formal methods enable us to prove correctness in an infinite subset of the configuration space, which is often needed in cyber-physical systems involving continuous mechanics. Although the efficacy of formal methods is traditionally offset by significantly greater development cost, we propose new development techniques that can mitigate this concern.

We explore novel techniques for assuring the correctness of data-driven systems based on certified programming and software verification. In particular, we focus on the use of interactive theorem-proving systems to prove foundational properties about data-driven systems, possibly reliant upon physics-based assumptions and models. We introduce the concept of the formal safety envelope, analogous to the concept of an aircrafts performance envelope, which organizes system properties in a way that makes it clear which properties hold under which assumptions. Beyond maintaining modularity in proof development, this technique furthermore enables the derivation of runtime monitors to detect potentially unsafe system state changes, allowing the user to know precisely which properties have been verified to hold for the current system state. Using this method, we demonstrate the partial verification of an archetypal data-driven system from avionics, where wing sensor data is used to determine whether or not an airplane is likely to be in a stall state.

FreeST: Context-free Session Types in a Functional Language

Vasco T. Vasconcelos, University of Lisbon, Portugal

FreeST is an experimental concurrent programming language. Based on a core linear functional programming language, it features primitives to fork new threads, and for channel creation and communication. A powerful type system of context-free session types governs the interaction on channels. The compiler builds on a novel algorithm for deciding type equivalence of context-free session types. This talk provides a gentle introduction to the language and discusses the validation process and runtime system. [Joint work with Andreia Mordido, Bernardo Almeida, Peter Thiemann]

A Distributed FRP Language with an Actor-based Execution Model

Takuo Watanabe, Tokyo Institute of Technology, Japan

Functional reactive programming (FRP) is a programming paradigm where a system is described using declarative abstractions of the change propagation of discrete events and continuous signals. Distributed-XFRP is a pure FRP language that facilitates a uniform description of distributed coordination and per-node computation. The runtime system of the language based on the Actor model guarantees that the propagation of values is single-source glitch-free. Namely, it ensures the temporal consistency for each source value. Also, two fault-tolerant mechanisms let a system work appropriately on unreliable networks where the out-of-order delivery or message loss may occur. Some case studies, including a sensor-actor network, show that the language is suitable for programming coordination behaviors.

Distributed Algorithms for Robot Systems

Défago Xavier, Tokyo Institute of Technology, Japan

Mobile robot systems constitute a form of distributed systems, where reaching agreement reliably is the cornerstone of cooperation. Our research focuses on these aspects, from theoretical mobile robots to swarm and multi-agent robotics. After a brief overview of our research activities, the presentation focuses on education of distributed algorithms.

Teaching distributed algorithms can be challenging and even frustratingly difficult when most students initially fail to connect the abstract models and definitions with reality. Making this link is particularly difficult because the abstract models in which distributed algorithms are expressed in textbooks are somewhat far from how such algorithms are typically implemented. In order to alleviate this problem, we have developed a simulation framework and a domain-specific language in Scala with which distributed algorithms can be expressed in a way that remains close to typical textbook pseudocode, but can also be executed in a simulated environment. The framework, called ScalaNeko/Ocelot, inherits from the project Neko developed in Java at EPFL in 2000. ScalaNeko/Ocelot supports both active and reactive protocols, as well as their flexible composition. The network simulation allows to define arbitrary network topologies. This allows to implement, in usually less than fifty lines of code, classical distributed algorithms as varied as: detection of cut vertices, distributed mutual exclusion in arbitrary graphs, randomized leader election in anonymous networks, construction of a spanning tree, etc.

HOPE: A New Parallel Execution Model Based on Hierarchical Omission

Masahiro Yasugi, Kyushu Institute of Technology, Japan

I present a new approach to fault-tolerant language systems without a single point of failure for irregular (divide-and-conquer) parallel applications. I

propose a novel "work omission" paradigm and its more detailed concept as a "hierarchical omission"-based parallel execution model called HOPE. HOPE programmers' task is to specify which regions in imperative code can be executed in sequential but arbitrary order and how their partial results can be accessed. HOPE workers spawn no tasks/threads at all; rather, every worker has the entire work of the sequential program with its own planned execution order, and then the workers and the underlying message mediation systems automatically exchange partial results to omit hierarchical subcomputations. Even with fault tolerance, the HOPE framework provides parallel speedups. As runtime systems, the message mediation systems are distributed and federated; the implementation should provide deadlock avoidance, fairness, and fault tolerance. Unfortunately, our current implementation is in C with POSIX threads and MPI; I hope for a nice programming language to implement the complex runtime systems.

Behavioural Type-based Static Verification Framework for Go

Nobuko Yoshida, Imperial College London, United Kingdom

Go is a production-level statically typed programming language whose design features explicit message-passing primitives and lightweight threads, enabling (and encouraging) programmers to develop concurrent systems where components interact through communication more so than by lock-based shared memory concurrency. Go can detect global deadlocks at runtime, but does not provide any compile-time protection against all too common communication mismatches and partial deadlocks.

In this work we present a static verification framework for liveness and safety in Go programs, able to detect communication errors and deadlocks by model checking. Our toolchain infers from a Go program a faithful representation of its communication patterns as behavioural types, where the types are model checked for liveness and safety.

Programs with Time in a Discrete Runtime Environment

Shoji Yuen, Nagoya University, Japan

Yampa is a domain-specific language of the Haskell programming language for hybrid systems, where its behaviour is continuous and discrete. A Yampa program executes continuous behaviour in a discrete runtime environment by sampling. If sampling should miss a critical point in switching continuous behaviour, the Yampa program may result in an unexpected error, although the program is correct in the continuous semantics. Since samplings are uncertain in the runtime, correctness of a Yampa program depends on the runtime. We describe the discrete behaviour of Yampa programs explicitly triggered by the sampling signal. We translate a subclass of Yampa programs to transition systems composed with automata generating sampling triggers with jitter. Then, we check the behavioural properties of Yampa programs under explicit sampling by the Uppaal model checker.

Verification of Message Passing Programs

Damien Zufferey, Max Planck Institute for Software Systems, Germany

Verification of message-passing systems is a challenging task. How we structure communication can have a great influence on the difficulty of verifying message-passing programs. With a perfect network, the problem is undecidable. While considering fault makes the verification problem simpler in theory (lossy channel systems), the problem is still intractable. These results uses the communication channels to encode a large memory but hardly any real program do so. Instead, writing a program using communication-closed rounds can greatly simplify the verification. The key insight is the use of communication-closure (logical boundaries in a program that messages should not cross) to structure the code. Communication-closure gives a syntactic scope to the communication, provides some form of logical time, and give the illusion of synchrony. Communication-closed rounds are particularly suitable to implement fault-tolerant distributed algorithms. Unfortunately, communication-closed rounds have their own limitations and I will discuss our current work on lifting these limitations by borrowing ideas from multiparty session types. Session-types are more flexible and can express richer communication patterns. They also simplify the verification by integrating a top-down decomposition from the global specification to the individual components and the verification is performed at the level of individual components. On the other hand, the system models used in session types does not consider faults.

List of Participants

- Gul Agha, agha@illinois.edu
- Carlos Baquero, cbm@di.uminho.pt
- Sebastian Burckhardt, sburckha@microsoft.com
- Wolfgang De Meuter, wdmeuter@vub.ac.be
- Patrick Eugster, peugster@dsp.tu-darmstadt.de
- Pascal Felber, pascal.felber@unine.ch
- Carla Ferreira, carla.ferreira@fct.unl.pt
- Simon Fowler, simon.fowler@ed.ac.uk
- Elisa Gonzalez Boix, egonzale@vub.ac.be
- Torsten Grust, torsten.grust@uni-tuebingen.de
- Philipp Haller, phaller@kth.se
- Keigo Imai, keigoi@gifu-u.ac.jp
- Nadeem Jamali, jamali@cs.usask.ca
- Alessandro Margara, alessandro.margara@polimi.it
- Stefan Marr, s.marr@kent.ac.uk
- Hidehiko Masuhara, masuhara@is.titech.ac.jp
- Heather Miller, heather.miller@cs.cmu.edu
- Guido Salvaneschi, salvaneschi@cs.tu-darmstadt.de
- Marjan Sirjani, marjan@ru.is
- Carlos A. Varela, cvarela@cs.rpi.edu
- Vasco T. Vasconcelos, vmvasconcelos@ciencias.ulisboa.pt
- Takuo Watanabe, takuo@c.titech.ac.jp
- Défago Xavier, defago@c.titech.ac.jp
- Masahiro Yasugi, yasugi@ai.kyutech.ac.jp
- Nobuko Yoshida, n.yoshida@imperial.ac.uk
- Shoji Yuen, yuen@is.nagoya-u.ac.jp
- Damien Zufferey, zufferey@mpi-sws.org

Meeting Schedule

Check-in Day: May 26 (Sun)

- Welcome Banquet

Day1: May 27 (Mon)

- Talks and Discussions

Day2: May 28 (Tue)

- Talks and Discussions
- Group Photo Shooting
- Group Summaries

Day3: May 29 (Wed)

- Reports & Discussions
- Excursion

Day4: May 30 (Thu)

- Demos
- Reports & Discussions
- Wrap up