ISSN 2186-7437

NII Shonan Meeting Report

No. 2019-147

Self-supporting, Extensible Programming Languages and Environments for Exploratory, Live Software Development

Luke Church
Richard P. Gabriel
Robert Hirschfeld
Hidehiko Masuhara

February 25–28, 2019

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

Self-supporting, Extensible Programming Languages and Environments for Exploratory, Live Software Development

Organizers:
Luke Church (Lark Systems and University of Cambridge, UK)
Richard P. Gabriel (Dream Songs and HPI, USA)
Robert Hirschfeld (HPI, University of Potsdam, Germany)
Hidehiko Masuhara (Tokyo Institute of Technology, Japan)

February 25–28, 2019

A new paradigm for programming is like a new genre in an art, akin to the "nonfiction novel" introduced by Truman Capote, postmodern poetry, and impressionism, cubism, and abstract expressionism in painting. The problem domain constrains what we want to talk about, and the paradigm domain constrains how we can say it.

We are talking about paradigms in the large, not small-scale paradigms like object-oriented, logic, or functional programming. We are talking about alternatives to the two governing metaphors of mainstream programming: program as text and software development as problem solving. Key concerns within these metaphors are correctness, productivity, (business) value-creation, and performance.

Programming accomplishes many different things—products, apps, science and engineering, learning, solving puzzles, uncovering mysteries, discovering and understanding, and supporting business. This has always been the case; but today we face a set of new issues and approaches for designing and developing software—some of these approaches are actually old but are now beginning to influence how we design and develop. Examples include the availability of big data; hardware systems that rely on software for reliability; AI and AI components; the need for autonomous systems; "app" and micro-service thinking; and the need to embrace legacy systems.

Whereas in the past we faced primarily the problem of requirements emerging from the design and code as they unfold or as slowly dawning insights into the problem and solution domains present themselves—a formidable problem that is not going away—now we also have unfamiliar realities adding to the fog of system making. This is the time for programming languages and development environments to step in to help.

The question of interest to us is how to frame this emerging paradigm to help us with problem setting, and which more narrowly focused areas in the new paradigm are ready for deeper exploration. Unlike mainstream programming, we do not have strongly governing metaphors, which is why we use terms like

"self-sustaining," "live programming," "exploratory programming," and "extensible programming languages and environments" to point to the questions that interest us. Also, we are not sure whether we are talking about one paradigm or several.

Purely textual approaches focus on the abstract and thematic. Type correctness ensures only that thematic bugs are absent and that the form of the program is abstractly correct, but not that the program in fact does what is intended or needed. Proof of correctness ensures only that one view of the specifications agrees with another view. These guarantees are important but can disappoint when the notion of correctness is in flux or is not relevant. For example, is Google's Go-playing program correct? It beats a lot of human players. Is a recommendation system correct? Is a software system that self-repairs correct? More troubling and surprising questions are beginning to arise: does a particular system have a "moral compass" (self-driving cars, for example), does it embody and thereby promote racism (various classifiers)?

Live systems give the programmer a window into the concrete operation of the program. A program fragment that has been proven abstractly to be OK can be checked concretely. The chronology of the program can be observed, its intentions checked, and the problem domain (what the program is addressing) along with the paradigm details can be checked both by direct human observation and by ad hoc, in situ testing. One way to look at it is that the program and human programmer are a symbiotic pair.

We observe that these two approaches—textual and live—are steps along a path that the new paradigms might extend in directions hard to envision at present.

Overview of Talks

The Matriona Module System for Squeak

Matthias Springer and Hidehiko Masuhara (Tokyo Institute of Technology, Japan)

Our work is concerned with Smalltalk programming environments. Such environments contain multiple applications/libraries in an image. Sometimes multiple versions of a library are simultaneously needed due to dependencies of applications.

Hosting multiple applications/libraries in one environment is problematic because there can be naming conflicts between classes. It is also not clear how multiple versions of the same application/library can be hosted in one environment. Such conflicts are typically resolved on the file system level: Every library version has a different file name. Newspeak is an example for a Smalltalk system that resolves such problems with a platform object which contains all required dependencies.

We prototyped the Matriona module system in Squeak, a Smalltalk environment.

In contrast to Newspeak, Matriona has a global, hierarchical namespace, which is similar to the file system of an operating system. Furthermore, nested classes are not possible on a per-instance basis; similar functionality can be achieved via class parameterization. We believe that, through a simpler design, our system will make it easier for programmers to use versioning, dependency management and potentially conflicting libraries in one Squeak image.

We evaluated Matriona by porting two applications, that were developed by Bachelor students, to Matriona.

In the spirit of Smalltalk, we believe that our work will bring us closer to a programming system that also fulfills tasks that are typically solved on the operating system level.

Delivering Benefits of Programming to Everyone

Jun Kato (AIST, Japan)

There has been much research work on HCI (toolkit), PL (novel ideas on programming), CSCW (collaborations between people with diverse technical backgrounds) but was done almost independently.

Creativity support tools (e.g., Microsoft Office suite, Adobe CC, ...) benefit from programming. Relevant research includes end-user programming and programming education, but they focus on turning non- programmers into a sort of programmers. There is only a handful of work done on the collaborations between professional and novice or non- programmers.

Rethink programming environments so that they can be used as foundations for communication between people with diverse technical backgrounds. They can be "environments" not only for programming but for the entire life cycle of the programs being developed. For instance, in such environments, how can we help programmers to expose the degree of freedom in customizing the program behavior to end-users? How can we improve the process of issue reporting?

Graphical representation of program specification and runtime state are particularly useful. For instance, constant values can be represented as photos [Picode, CHI '13]. A variable declaration can be represented as a slider, checkbox, color palette, etc., and its state can reflect the runtime state [User-Generated Variables, PX '17].

See TextAlive or f3.js websites that realize the proposed approach.

Computer programs have become ubiquitous in our life and work, and programmers are becoming responsible for too much stuff [IEEE Computer 49(7), pp.34-42]. We need to separate the responsibility into multiple new roles. "Data scientist" can be considered as one such successful case.

Liveness in Problem Solving Tools

Steven Tanimoto (University of Washington, USA)

In the context of software tools to support general problem solving within mixed-initiative, human-machine scenarios based on the classical theory of state-space search from artificial intelligence, an overarching issue is how best to help the users think about the problems they are trying to solve and move towards meaningful problem formulations and workable solutions. One specific category of features in this vein is providing immediate feedback when users make constructions or modifications to their formulations and partial solutions. Liveness, now well known in the programming environments context, can be applied in the problem-solving context.

A demonstration will be given of one form of "live solving" in which the tool offers the human user a visualization of a problem space, and the user draws a trajectory through the space to indicate a solution path, and during the drawing process, the tool makes a live interpretation of the path as a solution process.

Several questions can be asked that are associated with this demonstration: what sorts of problems seem to most naturally lend themselves to such solution? Where to the problem space visualizations come from? How important is the particular problem-space visualization? What if the problem space is infinite? Can programming problems be addressed in this manner? Does this live solving approach facilitate particular forms of artificial intelligence in the process of solving problems?

While solving problems by drawing solution paths is arguably a Turing-complete means of programming (when the programming problem is formulated in an particular way), this implies an unusual programming experience for human programmers. An open question is whether it changes the experience in ways that are a net improvement. The question is analogous to one answered in the visual programming community: can blocks-based programming environments offer advantages over textual languages and tools? The answer to that question is yes in the context of novice programming, and some domain-specific programming, and this is evidenced by the wide uptake of Scratch programming activities by children and the commercial success of LabView with engineers, to name just two examples.

Live solving also has possible applications in the arts. Like live coding of music, it can be considered a performance art in its own right. Live solving may be improvisational, but rather than producing music, it might produce a succession of candidate solutions to a problem.

Change Provenance for Data Science

Roly Perera (Alan Turing Institute, UK)

Interactive notebooks such as Jupyter and Observable interleave data, computation and visualisations with human-authored narrative content. Related developments include tools for data-driven journalism like The Gamma, new forms of digital dissemination like the online journal Distill, and "explorable explanations", computational documents which allow end users to manipulate the parameters of an underlying model.

These new tools and proofs-of-concept reflect an emerging new role for computation: as a literate medium for expressing and communicating technical ideas, experimental results, methods, arguments and scenarios.

I will briefly present some of the recent efforts in this area, discuss some limitations, and then try to motivate my current project. My research hypothesis is that for computations to fulfil their potential as human-readable explanations, we should make their fine-grained dependency structure explicit, allowing users to explore how parts of a data set or model relate to parts of the outcome – or more generally, how changes to a data set or model relate to changes to the outcome. The guiding principle is captured by the slogan "the explanation of a change is a change in an explanation".

Polyglot Programming Experience

Fabio Niephaus and Robert Hirschfeld (Hasso Plattner Institute, Germany)

When building applications, software developers usually have to decide on a programming language to use. This decision is often influenced by the libraries and frameworks available in a language and once the decision is made, it is usually hard to reuse software artifacts from other programming languages.

Polyglot runtime environments give software developers a much broader choice in terms of libraries and frameworks they can use and therefore foster code reusability and productivity. However, writing applications using multiple languages can also be a cognitive challenge because developers may have to switch between different language designs and concepts.

We have built GraalSqueak, a Squeak/Smalltalk-based IDE platform for polyglot programming. It runs on top of GraalVM, a high-performance polyglot virtual machine, and allows us to explore the domain of polyglot programming with the live tools of a Smalltalk environment.

We want to gather the needs of "polyglot developers" by building polyglot applications ourselves and by observing others building them. Based on our findings, we plan to use GraalSqueak to prototype novel software development tools for polyglot programming.

As a first prototype, we have implemented a Jupyter-like notebook systems which allows data engineers to use multiple languages at once. After prototyping further tools for polyglot programming, we want to conduct a user study to evaluate how they help developers using multiple languages in a software project.

Polyglot programming still is a rather unexplored programming style. Polyglot runtime environments such as GraalVM have demonstrated that direct and

fast language interoperability is possible. The programming experience of polyglot programming is just as important and must be worked on. Only then polyglot programming could become a serious choice for software developers.

Integrating Intelligence: Making Doubt an Advantage

Luke Church (Lark Systems and University of Cambridge, UK)

Computer systems amplify one mode of thought at a time. You can either click and drag things, or you can write programs, or you do complicated statistics. Once you've picked your interaction modality, changing it might take a few people-centuries of effort.

I discuss ongoing work in designing loci where different intelligences can come together, allowing people to operate with different modalities. Along the way we discover that, in some cases, being uncertain about how you represent the structure of the information in a computer becomes an advantage.

I conclude with a discussion of how it is, practically, that we might bring such a system into being.

On Bidirectionality Between Media

Mariana Mărășoiu (University of Cambridge, UK)

Most typical programs are unidirectional: they usually take an input, change it in some way, and then produce an output. Even the language we use suggests this: we don't expect to be able to use the output to affect the input!

In this talk, I invite you to consider, when should we be able to?

One place is in graphics design. Imagine if a user writes some code to add a header of 12 equally spaced blue rectangles to an A4 poster. They then use their mouse to select the the third rectangle in the line and move it downwards slightly, separating it from the others. The code that generated the visuals is updated with an additional statement encoding the move. The user then returns to the program and adjusts the colour of the rectangles to dark blue. The graphics update accordingly.

Luke Church and I have been working on tools that support or reveal the need for this kind of interaction, in the domains of architectural modelling and data visualisation. Other researchers have also prodded at this problem in various domains (e.g. text formatting, GUI design, interactive diagrams) and through different means of achieving this (e.g. programming by demonstration, program synthesis, bidirectional transformations).

I will briefly review several tools that support various degrees of bidirectionality, focusing in on what kinds of user experiences these tools enable. I will then discuss the place of bidirectionality in the context of live programming systems and highlight bidirectionality as a necessity of breaking the direct manipulation vs programming dichotomy.

Semantic Code Models for Better Modularity and Program Comprehension

Toni Mattis and Robert Hirschfeld (Hasso Plattner Institute, Germany)

Names can concisely represent concepts and their meaning in program code. Programmers choose analogies and metaphors to describe and communicate programmatic concepts. In source code, such analogies manifest themselves as a particular choice of names. The apparent imprecision that comes with names drawn from natural language is greatly outweighed by their role in program comprehension: reading such names is a starting point to understand roles in the domain, build testable hypotheses, and verify them by exploring the behavior of live instances.

Understanding programs by looking for names that suggest particular concepts can be a time-consuming process, especially if the modularity of the system is in poor condition. At the same time, a lack of awareness how concepts are distributed and interacting can reinforce modularity issues, such as redundancy, tangling and scattering, or longer feedback cycles due to entwined setup routines and data structures.

Recent work on topic modeling, software clustering, and recommender systems has given rise to statistical models that learn and correlate names and high-level concepts from program artifacts, edit history, and run-time behavior. The use of these tools for program comprehension and modularity feedback usually constitutes a separate task, which contrasts with live programming environments where comprehension, editing, and feedback is tightly interwoven.

If programming environments were aware of the meaning of names and the concepts underlying each module, tools could not only help programmers understand a code base in terms of how high-level concepts are distributed and related but offer a wide range of concept-aware assistance during programming activities.

Therefore, we aim at improving existing statistical models to better support live programming workflows, extending tools to navigate and explore concepts, providing live modularity-oriented feedback, e.g., by displaying how well identifiers fit into their conceptual context, and improving the relevance of results in recommender systems like code completion, search, or refactoring suggestions.

We hope that the use of machine learning models that support programming activities can help programmers make sense of larger and larger software systems and ultimately assist with software quality analogously to pair-programming peers.

Induction Via Recursion

Youyou Cong (Tokyo Institute of Technology, Japan)

Mathematical induction is one of the hardest topics in high school math. For many students, induction is simply "steps to follow", and as previous work shows, they sometimes write correct proofs without being able to justify them.

Our study aims to help students' learning of induction by teaching recursion, a programming-counterpart of induction. The approach is based on our conjecture that recursion is more accessible than induction: since recursive programs

"work", they are more visually appealing than proofs, and when programming in certain beginner-friendly environments, students can observe how a recursive computation goes step by step.

We have conducted several experiments in order to answer the following research questions:

1. What specifically do students find hard when learning induction?
2. How well do computer science students identify the correspondence between induction and recursion?

To answer question 1, we designed a paper test that asks students to explain how a proof by induction works in their own words. We gave this test to 80 high school students and 43 CS-major undergraduate students. The result was unexpectedly bad; only 8 students got full marks. By observing students' answers, we identified three challenges in learning induction: (i) why we need a base case; (ii) what role an induction hypothesis plays; and (iii) how mathematical induction differs from everyday induction.

To answer question 2, we designed a table-filling test, which shows four components of a proof by induction (base case, inductive case, induction hypothesis, and conclusion) and asks students to write their recursion-counterparts. As the results show, most students identify the base and inductive cases, but many of them do not view a recursive call as an induction hypothesis, and very few of them understand what it means to have a complete definition of a recursive function. What this suggests is that, if we wish to teach induction via recursion, we must put considerable effort into explaining their correspondence.

This study is still ongoing. As future work, we intend to design a series of introductory programming lessons for understanding induction. This would make learning of induction more active and fun, and as a side effect, it might also motivate teachers to integrate programming into school curriculums.

Incorporating Examples Into Live Programming

Patrick Rein, Jens Lincke, and Robert Hirschfeld (Hasso Plattner Institute, Germany)

When working on a program, developers traditionally have to simulate the behavior of the abstract code in their heads until they can execute the application. Live programming aims to support the development and comprehension of programs by providing more immediate feedback on program behavior, but the divide between code and behavior often remains. The goal of example-based live programming is to remove this gap by allowing programmers to explore the actual behavior of their code during development as ubiquitously as the static source code can be explored today. This is achieved by defining live examples for parts of the application.

The idea of live examples has been already addressed in other tools and environments. However, most of those solutions are limited to specific domains and are suitable only for small programs. Thus, we aim to extend the application of example-based live programming to more complex programs potentially spanning multiple modules.

We investigate existing solutions to derive a set of requirements for an integration of live examples into source code. Based on these requirements we propose a new approach to live examples and present a prototype in its support.

Using the resulting prototype with scenarios from related work, we illustrate how example-based live programming can provide more insights into the runtime behavior of parameterized code for non-trivial programs. In presenting this more general approach to example-based live programming, we hope to make practical solutions available.

Beyond this prototype, we want to investigate the general effects of live programming with examples. The advantages of incorporating examples into source code have often been stated in live programming research, but empirical evidence on the effects on programmer behavior are still inconclusive. Thus, we outline a set of future controlled experiments to investigate the impact of live examples. In particular, the moderation effects of the task complexity has not be investigated yet. Together with existing evidence on live programming, our results will provide a more complete model of the settings in which liveness is beneficial. This in turn can guide future designs decisions in programming environment design.

Lively4: An Exploratory Web Programming Environment

Jens Lincke, Stefan Ramson, and Robert Hirschfeld (Hasso Plattner Institute, Germany)

Exploratory programming workflows are often only applicable to content residing inside dedicated environments, requiring special workflows or languages.

While working on programs and objects that are not created in such a special way one cannot make use of exploratory workflows. Further even when creating new content inside a special exploratory environment, it is hard make use of content and programs created outside of that environment.

To overcome the gap between explorable content created in special environments and outside content as HTML content and JavaScript programs, we create Lively4, a new environment that embraces standard HTML and JavaScript. HTML used to be only a generation target UI for frameworks other systems. In Lively4, we use HTML/JavaScript to build a collaborative, self-supporting exploratory development environment for all HTML/JavaScript content.

With this approach, we are pushing the boundaries of exploratory programming environments. Given the restrictions of HTML and JavaScript, we can explore how working with documents, names, explicit references compares to working with pure object graph of special environments.

We build Lively4 environment in a self-supporting way, which allowed us to develop, use and evolve tools and workflows from within it.

Exploratory workflows can enrich HTML/JavaScript development experience. Tools and environment can be easier to create if external contributions are easier to integrate and use.

Active Expressions as a Basic Building Block for Reactive Programming Concepts

Stefan Ramson and Robert Hirschfeld (Hasso Plattner Institute, Germany)

State dependencies, constraints, as well as action-triggering events are part of many relevant problem domains. Describing such behavior using only means of object-oriented programming often results in high architectural overhead. One main reason for this overhead is the semantic mismatch between the problem domain to be described and the available programming language concepts. Using a more fitting programming model may bridge the gap between a problem domain and the available language elements to describe it. In particular, reactive programming is dedicated to express cause-action relationships and data dependencies. Therefore, reactive programming is particularly suited to describe the aforementioned behavior.

While reactive programming is an active field of research, the implementation of reactive concepts remains challenging. Especially in object-oriented environments, change detection represents a hard but inevitable necessity when implementing reactive concepts. Typically, change detection mechanisms are not intended for reuse, instead, they are tightly coupled to the particular change resolution mechanism. As a result, developers often have to re-implement similar abstractions. Conclusively, this fact hints the existence of a reusable reactive primitive for change detection.

To find a suitable primitive, we identified commonalities in existing reactive concepts. We discovered a class of reactive concepts, state-based reactive concepts. All state-based reactive concepts share a common change detection mechanism: they detect changes in the evaluation result of an expression.

Based on the identified common change detection mechanism, we propose active expressions as a reusable reactive primitive. By abstracting the tedious implementation details of change detection, active expressions can ease the implementation of reactive programming concepts.

To evaluate the design of active expressions, we re-implemented a number of existing state-based reactive concepts using active expressions. The resulting active expression-based implementations are simpler in terms of code complexity compared to respective plain implementations.

With active expressions, system developers may separate essential from non-essential parts when reasoning about reactive programming concepts. By using active expressions as a primitive for change detection, developers of reactive language constructs and runtime support can now focus on the design of how application programmers should be able to react to change.

Programming Experiences With a Live Programming Environment for Data Structures

Hidehiko Masuhara (Tokyo Institute of Technology, Japan)

This demonstration presents a live programming environment for data structures called Kanon. The goal of the environment is to support professional programmers when they develop new data structures and operations thereof. The

key feature for data structures is its automatic visualization of objects that are created in a program execution.

We first overviews the environment through a coding session of a small data structure. The environment assumes a program is written in a test-driven style, and displays objects as a node-link diagram. We introduce the notion of visualization contexts, which the program state stopped at the cursor position, and used for navigating the visualization.

We then discusses the features that support liveness and data-structures. The jump-to-construction mechanism navigates the programmer from a visual element to a code fragment that made the element. Our proposed graph layout algorithm automatically analyzes the object structure and places nodes so as to help the programmers to recognize the relations between objects. An on-the-fly test case generation mechanism enables top-down programming by letting the programmers handcraft an expected result upon a call to an unimplemented function.

Lastly, we discuss several issues for future research. Programming complicated data structures require customized visualization so that the programmer can see the objects at different levels of abstraction. In order not to distract the programmers' focus, it is crucial to handle errors appropriately especially in an environment where program code can be erroneous in the middle of editing. Our user experiment discovered some programming mistakes that can only be observed with the live programming environment, which might be an effect of live programming on the way of programming.

Live Programming With Dyanmo

Elayabharath Elango (Singapore)

arcos: A Virtual 3D System

Yoshiki Ohshima (CEO Vision, USA)

A virtual 3D system called arcos is presented. The system allows the user not only to create and edit new 3D objects but also to modify the system itself by using the programming tools in the system. The system is written in JavaScript and requires zero-install to run on laptops and mobile phones. In addition, a massively-parallel particle simulation domain-specific language called Shadama, running in the arcos system is also presented. Shadama's syntax is designed for end-users and easy to use, but it can unleash the power of GPU; the code is dynamically compiled to the GPU code via Open GL Shading Language and allows the user to create simulations with millions of particles in the live programming style.

Mini-maru Self-bootstrap

Ian Piumarta (Ritsumeikan University, Japan)

How small, flexible, general, and complete can we make a programming language? Maru is an experiment in such minimalism. It begins as a tiny, interpreted dialect of Lisp (which we might call "hajimaru"). As with McCarthy's

original Lisp 1.5, the language is just powerful enough to write a meta-circular interpreter for itself. Unlike Lisp 1.5, Maru also implements a compiler for itself that generates Intel machine code.. Feeding the interpreter (reader, allocator, evaluator, garbage collector) through the compiler produces a self-hosted Maru interpreter, decoupled from any third-party run-time library dependencies. The entire system is approximately 1,700 lines of relatively simple code and the performance of compiled code approaches 30% of optimised C (for programs that do not allocate memory). This takes care of "small" and, in one sense of the word, "complete".

Reasonable definitions of "flexible" and "general" might include control over front-end syntax and extensible semantics. Lisp code is really just an abstract syntax tree, and an alternative read-eval-print loop is one line of code. New syntactic shells are easy to write (a parsing expression grammar framework is a few tens of lines of code) taking care of syntactic freedom.

An old computer scientist's adage claims that "any programming problem can be solved by adding another level of indirection (excepting the problem of too many levels of indirection)". The kernel of a Lisp evaluator typically implements two built-in semantic functions: "eval" tells us how to produce a value for any value type of the language, and when that type happens to represent a function call another function "apply" takes over. These two functions are written in terms of each other, with a few primitive operations thrown in to ground everything. (The original Lisp 1.5 had five such "magic primitives".)

Freedom from fixed semantics is provided by dispatching both eval and apply on the type of the value involved using look-up tables — a kind of "poor man's object orientation" with exactly two methods. For example, "eval" on integers and strings is dispatched to the identity function. A pair (representing the first value and link of a list) is dispatched to the built-in "apply" function, which evaluates the head of the list and dispatches to an applicator based on the resulting value type. Only three built-in primitive behaviours are required to reproduce Lisp's semantics, and the built-in behaviour of either eval or apply can be changed at run-time for any value type, including all of the built-in types. The result is a system where robust, user-defined generic functions can be implemented in less than ten lines of code. Thanks to the two-stage function application (functional value in eval, argument evaluation and invocation in apply) adding fixed expressions (the underlying semantics for any kind of special form or macro system) is a trivial exercise for the end user. This takes care of extensible semantics. The freedom to add new syntax and semantics together take care of a relatively liberal interpretation of "flexible" and "general" (and many remaining senses of "complete" that might otherwise have been in question).

For deeper modifications (a real-time, generational garbage collector, for example) the user can always appeal to the meta-circular nature of the language and recompile the system with whatever new run-time features they desire.

This talk will take the audience on a stroll through the Maru source, bootstrap process, and then demonstrate some of the use-defined semantic mechanisms alluded to above.

Meeting Schedule

February 24, 2019 (Sunday)

- Welcome Reception

February 25, 2019 (Monday)

- Session 1
  - Introduction
  - Patrick Rein
  - Robert Hirschfeld
- Session 2
  - Steve Tanimoto
  - Stefan Ramson
  - Roly Perera
- Session 3
  - Mariana Mărășoiu
  - Luke Church
  - Youyou Cong
- Session 4
  - Marcel Taeumel
  - Ian Piumarta
  - Topic Discovery and Discussion

February 26, 2019 (Tuesday)

- Session 5
  - Jun Kato
  - Elayabharath Elango
  - Jens Lincke
- Session 6
  - Toni Mattis
  - Yoshiki Ohshima
  - Hidehiko Masuhara
- Session 7
  - Matthias Springer
  - Fabio Niephaus
- Evening Session: Demonstrations

February 27, 2019 (Wednesday)

- Session 8: Group Discussion on Selected Topics
- Session 9: Group Discussion on Selected Topics
- Excursion and Banquet

February 28, 2019 (Thursday)

- Discussion: Next Steps and Workshop Report

List of Participants

- Luke Church, University of Cambridge, UK
- Youyou Cong, Ochanomizu University, Japan
- Elayabharath Elango, Singapore
- Robert Hirschfeld, HPI, University of Potsdam, Germany
- Jun Kato, AIST, Japan
- Jens Lincke, HPI, University of Potsdam, Germany
- Mariana Mărășoiu, University of Cambridge, UK
- Hidehiko Masuhara, Tokyo Institute of Technology, Japan
- Toni Mattis, HPI, University of Potsdam, Germany
- Fabio Niephaus, HPI, University of Potsdam, Germany
- Yoshiki Ohshima, CEO Vision, USA
- Roly Perera, Alan Turing Institute, UK
- Ian Piumarta, Ritsumeikan University, Japan
- Stefan Ramson, HPI, University of Potsdam, Germany
- Patrick Rein, HPI, University of Potsdam, Germany
- Matthias Springer, Tokyo Institute of Technology, Japan
- Marcel Taeumel, HPI, University of Potsdam, Germany
- Steve Tanimoto, University of Washington, USA