INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team mimosa

Migration et Mobilité: Sémantique et Applications

Sophia Antipolis

THEME COM

Activity Report

2006

Table of contents

1. Team

MIMOSA *is a joint project of INRIA and the Centre for Applied Mathematics (CMA) of the Ecole des Mines de Paris.*

Head of project team
Gérard Boudol [Research Director, Inria, HdR]

Vice-head of project team
Ilaria Castellani [Research Scientist, Inria]

Administrative assistant
Sophie Honnorat [Inria]

Staff members Inria
Manuel Serrano [Research Director, Inria, HdR]

Staff members CMA
Frédéric Boussinot [Research Director, CMA, HdR]

Reserch scientists (external)
Roberto Amadio [Professor, University of Paris 7, HdR]

Visting scientist
Erick Gallesio [Visiting Scientist, University of Nice Sophia-Antipolis]
Maria-Grazia Vigliotti [Visiting Scientist, till August 31]
Steffen Van Bakel [Visiting Scientist, till August 31]

Ph. D. students
Marija Kolundzja [University of Torino, from March 1]
Damien Ciabrini [MENRT, till October 1]
Frederic Dabrowski [MENRT]
Daoudou Maoulida [Inria, till July 31]
Stéphane Epardaud [MENRT]
Florian Loitsch [MENRT]
Ana Matos [Portuguese Gov., till February 28]

2. Overall Objectives

2.1. Overall Objectives

The MIMOSA project is a joint project with the Centre for Applied Mathematics of the *École Nationale Supérieure des Mines de Paris*. The overall objective of the project is to design and study models of concurrent, distributed and mobile programming, to derive programming primitives from these models, and to develop methods and techniques for formal reasoning and verification, focusing on issues raised by the mobile code. More specifically, we develop a reactive approach, where concurrent components of a system react to broadcast events. We have implemented this approach in various programming languages, and we have integrated migration primitives in this reactive approach. In the past we also intensively studied models of mobility, like the π-calculus and its distributed variants, and the calculus of Mobile Ambients. Our main research areas are the following:

- Security. We investigate security issues like confidentiality and resource consumption, using static analysis methods (for the verification of non-interference of programs with respect to given security policies, and of computational complexity), with an emphasis on the issues related to concurrent and mobile code.

- Models and languages for reactive programming. We develop several implementations of the reactive approach, in various languages. We have designed, and still develop, an alternative to standard thread systems, called FAIRTHREADS. We study the integration of constructs for mobile code in the model of reactive programming.
- Functional languages. We develop several implementations of functional languages, mainly based on the SCHEME programming language. Our studies focus on designing and implementing a platform for a *distributed environment*. All our developments on the web rest on these implementations.
- Web programming. We design and implement a programming environment for the web 2.0. It relies on a new distributed programming architecture where a program executes simultaneously on a server and a client. We aim at providing a realistic implementation that we constantly validate by developing and using end-users web applications.

3. Scientific Foundations

3.1. Semantics of mobility and security

Mobility has become an important feature of computing systems and networks, and particularly of distributed systems. Our project is more specifically concerned with the notion of a mobile code, a logical rather than physical notion of mobility. An important task in this area has been to understand the various constructs that have been proposed to support this style of programming, and to design a corresponding programming model with a precise (that is, formal) semantics.

The models that we have investigated in the past are mainly the π-calculus of Milner and the Mobile Ambients calculus of Cardelli and Gordon. The first one is similar to the λ-calculus, which is recognized as a canonical model for sequential and functional computations. The π-calculus is a model for concurrent activity, and also, to some extent, a model of mobility: π-calculus processes exchange names of communication channels, thus allowing the communication topology to evolve dynamically. The π-calculus contains, up to continuation passing style transforms, the λ-calculus, and this fact establishes its universal computing power. The Mobile Ambient model focusses on the migration concept. It is based on a very general notion of a domain – an Ambient –, in which computations take place. Domains are hierarchically organized, but the nesting of domains inside each other evolves dynamically. Indeed, the computational primitives consist in moving domains inside or outside other domains, and in dissolving domain boundaries. Although this model may look, from a computational point of view, quite simple and limited, it has been shown to be Turing complete. In the past we have studied type systems and reasoning techniques for these models. We have, in particular, used models derived from the π-calculus for the formalization and verification of cryptographic protocols.

We are now studying how to integrate the model of reactive programming, described below, into a "global computing" perspective. This model looks indeed appropriate for a global computing context, since it provides a notion of time-out and reaction, allowing a program to deal with the various kinds of failures (delays, disconnections, etc.) that arise in a global network. We have started the design and implementation of a core programming language that integrates reactive programming and mobile code, in the context of classical functional and imperative programming.

3.2. Security of concurrent and mobile programs

We are studying security issues, especially those related to concurrent and mobile programming. In the past we have developed methods and tools for the verification of cryptographic protocols. We also work on secure information flow. This is motivated by the observation that access control is not enough to ensure confidentiality, since access control does not prevent authorized users to disclose confidential information. We use the language-based approach, developing static analyses, and especially type systems, to ensure that programs do not implement illegal flow of information. We work particularly on specific confidentiality issues arising with concurrent and mobile programming, but we also work on more general questions, like how to allow some pieces of code to declassify some information, while still ensuring some confidentiality policy.

We also use static analysis techniques, namely polynomial quasi-interpretations, to ensure that programs do not use computational resources beyond fixed limits. Again, a special effort is put here in finding methods that apply to reactive and/or mobile programs. This could also have applications to embedded code.

3.3. Reactive and functional programming

Reactive programming deals with systems of concurrent processes sharing a notion of time, or more precisely a notion of instant. At a given instant, the components of a reactive system have a consistent view of the events that have been, or have not been emitted at this instant. Reactive programming, which evolves from synchronous programming à la ESTEREL, provides means to react – for instance by launching or aborting some computation – to the presence or absence of events. This style of programming has a mathematical semantics, which provides a guide-line for the implementation, and allows one to clearly understand and reason about programs.

We have developed several implementations of reactive programming, integrating it into various programming languages. The first instance of these implementations was Reactive-C, which was the basis for several developments (networks of reactive processes, reactive objects), described in the book [5]. Then we developed the SUGARCUBES, which allow one to program with a reactive style in JAVA, see [4]. Reactive programming offers an alternative to standard thread programming, as (partly) offered by JAVA, for instance. Classical thread programming suffers from many drawbacks, which are largely due to a complicated semantics, which is most often implementation-dependent. We have designed, following the reactive approach, an alternative style for thread programming, called FAIRTHREADS, which relies on a cooperative semantics. Again, FAIRTHREADS has been integrated in various languages, and most notably into SCHEME via the BIGLOO compiler that we develop. One of our major objectives is to integrate the reactive programming style in functional languages, and more specifically SCHEME, and to further extend the resulting language to support migration primitives. This is a natural choice, since functional languages have a mathematical semantics, which is well suited to support formal technical developments (static analysis, type systems, formal reasoning).

We also designed a tool to graphically program in the reactive style, called ICOBJS. Programming in this case means to graphically combine predefined behaviours, represented by icons and to implement reactive code.

4. Application Domains

4.1. Simulation

Simulation of physical entities is used in many distinct areas, ranging from surgery training to games. The standard approach consists in discretization of time, followed by the integration using a stepwise method (e.g. Runge-Kutta algorithms). The use of threads to simulate separate and independent objects of the real world appears quite natural when the focus is put on object behaviours and interactions between them. However, using threads in this context is not so easy: for example, complex interactions between objects may demand complex thread synchronizations, and the number of components to simulate may exceed the number of available threads. Our approach based on FAIRTHREADS, or on the use of reactive instructions, can be helpful in several aspects:

- Simulation of large numbers of components is possible using automata. Automata do not need thread stacks, and the consumption of memory can thus stay low.
- Interactions are expressed by means of broadcast events, and can thus be dealt with in a highly modular way.
- Instants provide a common discrete time that can be used by the simulation.
- Interacting components can be naturally grouped into synchronized areas. This can be exploited in a multiprocessing context.

4.2. Embedded systems

Embedded systems with limited resources are a domain in which reactive programming can be useful. Indeed, reactive programming makes concurrent programming available in this context, even in the absence of a library of threads (as for example the `pthreads`). One objective is to build embedded systems from basic software components implementing the minimal functionalities of an operating system. In such an approach, the processor and the scheduler are considered as special resources. An essential component is a new specialized scheduler that should provide reactive engines with the functionalities they need.
This approach is useful for mobile telecom infrastructures. It could also be used in more applicative domains, as the one of gaming consoles. PDAs are also a target in which the proposed approach could be used. In this context, graphical approaches as ICOBJS could be considered to allow end-users to build some part of their applications.

4.3. Scripting

Because functional languages offer a high level of abstraction, they generally enable compact implementations. So, they enable fast prototyping and fast implementing. In consequence, they are generally convenient when used as scripting languages. For this reason, many famous end-user applications (such as Emacs, Gimp, Auto-Cad, ...) embed interpreters of functional languages. The compilation of functional languages, at least for the representatives that use strict evaluation order, is now well understood. Hence, programming in a functional language does not forbid to produce fast applications that do not clutter the computers they run on. With some of the modern implementations, it is possible to blend compiled code, for fast execution, and interpreted code, for scripting. The combination of both execution modes brings expressiveness *and* efficiency. Few other languages offer this capability. Exploiting this specificity we have conceived an email synchronizer that is implemented in Scheme and that also uses this language for supporting user scripting.

4.4. Web programming

Along with games, multimedia applications, and email, the web has popularized computers in everybody's life. The revolution is engaged and we may be at the dawn of a new era of computing where the web is a central element.
Many of the computer programs we write, for professional purposes or for our own needs, are likely to extensively use the web. The web is a database. The web is an API. The web is a novel architecture. Therefore, it needs novel programming languages and novel programming environments.
In addition to allowing reactive and graphically pleasing interfaces, web applications are de facto distributed. Implementing an application with a web interface makes it instantly open to the world and accessible from much more than one computer. The web also partially solves the problem of platform compatibility because it physically separates the rendering engine from the computation engine. Therefore, the client does not have to make assumption on the server hardware configuration, and vice versa. Lastly, HTML is highly durable. While traditional graphical toolkits evolve continuously, making existing interfaces obsolete and breaking backward compatibility, modern web browsers that render on the edge web pages are still able to correctly display the web pages of the early 1990's.
For these reasons, the web is arguably ready to escape the beaten track of n-tiers applications, CGI scripting and interaction based on HTML forms. However, we think that it still lacks programming abstractions that minimize the overwhelming amount of technologies that need to be mastered when web programming is involved. Our experience on reactive and functional programming is used for bridging this gap.

5. Software

5.1. Mimosa softwares

Most MIMOSA softwares, even the older stable ones that are not described in the following sections (such as the SugarCubes and Rejo-Ros) are freely available on the Web. In particular, some are available directly from the INRIA Web site:

http://www.inria.fr/valorisation/logiciels/langages.fr.html
Most other softwares can be downloaded from the MIMOSA Web site:
http://www-sop.inria.fr/mimosa

5.2. Reactive programming

Participants: Frédéric Boussinot, Stéphane Epardaud.

5.2.1. Reactive-C

The basic idea of Reactive-C is to propose a programming style close to C, in which program behaviours are defined in terms of reactions to activations. Reactive-C programs can react differently when activated for the first time, for the second time, and so on. Thus a new dimension appears for the programmer: the logical time induced by the sequence of activations, each pair activation/reaction defining one instant. Actually, Reactive-C rapidly turned out to be a kind of *reactive assembly language* that could be used to implement higher level formalisms based on the notion of instant.

5.2.2. FairThreads in Java and C

FAIRTHREADS has first be implemented in JAVA. It is usable through an API. The implementation is based on standard JAVA threads, but it is independent of the actual JVM and OS, and is thus fully portable. There exists a way to embed non-cooperative code in FAIRTHREADS through the notion of a fair process. FAIRTHREADS in C introduces the notion of unlinked threads, which are executed in a preemptive way by the OS. The implementation in C is based on the pthreads library. Several fair schedulers, executed by distinct pthreads, can be used simultaneously in the same program. Using several schedulers and unlinked threads, programmers can take advantage of multiprocessor machines (basically, SMP architectures).

5.2.3. LURC

LURC is a Reactive threading library in C. It is based on the reactive model of ULM (*Un langage pour la mobilité*) and the desynchronization feature of FAIRTHREADS in C. It provides several types of thread models, each with different performance trade-offs at run-time, under a single deterministic semantics. Its main features as taken from ULM are preemption, suspension, cooperation and signal emission and waiting. On top of that, threads can switch from asynchronous to synchronous at will. Event-loop programming has been integrated in a reactive style under the form of a Reactive Event Loop. The main difference with the syntax of LOFT, another threads library developed in the team, is that LURC is a pure C library, on top of which a pseudo-language layer can be added in the form of C macros in order to make reactive primitives look and behave like language primitives. LURC is available on the INRIA website at the following URL: http://www-sop.inria.fr/mimosa/Stephane.Epardaud/lurc.

5.3. Functional programming

Participants: Damien Ciabrini, Stéphane Epardaud, Erick Gallesio, Bernard Serpette [Project Oasis], Manuel Serrano.

5.3.1. The Bigloo compiler

The programming environment for the Bigloo compiler [9] is available on the INRIA Web site at the following URL: http://www-sop.inria.fr/mimosa/fp/Bigloo. The distribution contains an optimizing compiler that delivers native code, JVM bytecode, and .NET CLR bytecode. It contains a debugger, a profiler, and various Bigloo development tools. The distribution also contains several user libraries that enable the implementation of realistic applications.
BIGLOO was initially designed for implementing compact stand-alone applications under Unix. Nowadays, it runs harmoniously under Linux and MacOSX. The effort initiated in 2002 for porting to Microsoft Windows is pursued by external contributors. In addition to the native back-ends, the BIGLOO JVM back-end has enabled a new set of applications: Web services, Web browser plug-ins, cross platform development, etc. The new BIGLOO .NET CLR back-end that is fully operational since release 2.6e enables a smooth integration of Bigloo programs under the Microsoft .NET environment.

5.3.2. Bugloo

Every programmer is frequently faced with the problem of debugging programs. Paradoxically, debuggers are hardly used in practice and have not evolved that much in the last decades. We believe these tools can be made more attractive by following some rules. Debuggers must be easily accessible from the programming environment. When using them, the performance slowdown must keep reasonable. At last, the debuggers have to match the specificities of the language of debugged programs.

These ideas have driven the design and implementation of BUGLOO, a source level debugger for Scheme programs compiled into JVM bytecode. It focuses on providing debugging support for the Scheme language specificities, such as the automatic memory management, high order functions, multi-threading, or the runtime code interpreter. The JVM is an appealing platform because it provides facilities to make debuggers, and helps us to meet the requirements previously exposed.

5.3.3. ULM

ULM is a new language for the mobility that is developed in the team. The ULM Scheme implementation is an embedding of the ULM primitives in the Scheme language. The bytecode compiler is available on PCs only but there are two ULM Virtual Machines: one for PCs and one for embedded devices supporting Java 2 Mobile Edition (J2ME) such as most mobile phones. The current version has preliminary support for a mixin object model, mobility over TCP/IP or Bluetooth Serial Line, reactive event loops, and native procedure calls with virtual machine reentry. The current version is available at http://www-sop.inria.fr/mimosa/Stephane.Epardaud/ulm.

5.4. Web programming

Participants: Erick Gallesio, Florian Loitsch, Manuel Serrano.

5.4.1. The Hop web programming environment

Hop is a new higher-order language designed for programming interactive web applications such as web agendas, web galleries, music players, etc. It exposes a programming model based on two computation levels. The first one is in charge of executing the logic of an application while the second one is in charge of executing the graphical user interface. Hop separates the logic and the graphical user interface but it packages them together and it supports strong collaboration between the two engines. The two execution flows communicate through function calls and event loops. Both ends can initiate communications.

The Hop programming environment consists in a web *broker* that intuitively combines in a single architecture a web server and a web proxy. That broker embeds a Hop interpreter for executing server-side code and a Hop client-side compiler for generating the code that will get executed by the client.

An important effort is devoted to providing Hop with a realistic and efficient implementation. The Hop implementation is *validated* against web applications that are used on daily-basis. In particular, we have developed Hop applications for authoring and projecting slides, editing calendars, reading RSS stream, or managing blogs.

Hop is available at http://hop.inria.fr

5.5. Old softwares

5.5.1. Skribe

SKRIBE is a functional programming language designed for authoring documents, such as Web pages or technical reports. It is built on top of the SCHEME programming language. Its concrete syntax is simple and looks familiar to anyone used to markup languages. Authoring a document with SKRIBE is as simple as with HTML or LaTeX. It is even possible to use it without noticing that it is a programming language because of the conciseness of its original syntax: the ratio *markup/text* is smaller than with the other markup systems we have tested.

Executing a SKRIBE program with a SKRIBE evaluator produces a target document. It can be HTML files for Web browsers, a LaTeX file for high-quality printed documents, or a set of *info* pages for on-line documentation.

5.5.2. Icobjs

ICOBJS programming is a simple and fully graphical programming method, using powerful means to combine behaviours. This style of programming is based on the notion of an *icobj* which has a behavioural aspect (object part), and a graphical aspect (icon part), and which can be animated on the screen. ICOBJS programming evolves from the reactive approach and provides parallelism, broadcast event communication and migration through the network. The Java version of ICOBJS unifies icobjs and workspaces in which icobjs are created, and uses a specialized reactive engine. Simulations in physics and the mobile Ambient calculus have been ported to this new system.

5.5.3. TypI

TypI is a type inference interpreter for the intersection types discipline. It implements, in CAML, the algorithm designed and proved correct by Boudol and Zimmer. A reference manual (in french) can be found on the web page of the project, and is a chapter of Zimmer's thesis .

5.5.4. MLObj

MlObj is an interpreter for a prototype language composed of a functional core, objects, mixins and degree types, written in CAML. It implements Boudol's theory of objects as recursive records. A reference manual (in french) can be found on the web page of the project, and is a chapter of Zimmer's thesis.

5.5.5. Trust

The TRUST tool, designed for the verification of cryptographic protocols, is an optimized OCAML implementation of the algorithm designed and proved by Amadio, Lugiez and Vanackère. It is available via the url http://www.cmi.univ-mrs.fr/~vvanacke/trust.html.

6. New Results

6.1. Security

Participants: Roberto Amadio, Gérard Boudol, Ilaria Castellani, Frédéric Dabrowski, Ana Matos.

6.1.1. Controlling information flow

Non-interference is a property of programs asserting that a piece of code does not implement a flow of information from classified or secret data to public results. In the past we have followed Volpano and Smith's approach, using type systems, to statically check this property for concurrent programs. The motivation is that one should find formal techniques that could be applied to mobile code, in order to ensure that migrating agents do not corrupt protected data, and that the behaviour of such agents does not actually depend on the value of secret information.

Ana Matos has defended her PhD Thesis this year [10]. The main contributions are: a new approach to the problem of declassifying information, and a study of new kinds of leaks arising in a mobile code scenario. Both have been reported on in last year's activity report. A revised version of the work by Matos, Boudol and Castellani on "Typing non-interference for reactive programs" has been accepted for publication in the Journal of Logic and Algebraic Programming [22].

The non-interference property is very often questioned, for various reasons (an important one is that is rules out declassification). In particular, a practical problem is that there is an important discrepancy between the notion of non-interfering program, and the methods that are used to reject unsecure programs, and especially type systems. Indeed, many programs are rejected by a type system for secure information flow, that are secure from the non-interference point of view, and it may be difficult to find explanations (hence, error messages) for this fact. Some researchers have taken the way of finding new program analysis methods (mainly logical ones) in order to better, and sometimes completely, capture the notion of non-interfering program (which is non-computable, in an expressive programming language). In our work [26] we have taken the opposite way: we think that static analysis by means of type systems is a very valuable tool for the programmers, which, in the case of secure information flow, points out pieces of code which clearly are programming errors, from the security point of view, even if they are "secure" from the non-interference point of view. A typical example is conditional branching on a secret information, where the branches illegitimately perform low assignments, which happen to be the same, whatever the value of the predicate is. We then proposed in [26] an alternative notion of secure information flow, more intensional than non-interference, namely: a program is secure if it never stores in the low part of the memory a value that has been elaborated using confidential information. We formalize this idea regarding a higher-order imperative language à la ML, using a standard labelled semantics where values are decorated with confidentiality levels, and we show that a standard type system for secure information flow in this language indeed ensures this property, which we conjecture to be (strictly) stronger than non-interference. An advantage of this approach is that this security property is a safety property, and in particular, the proof of type safety is much easier than non-interference proofs (using bisimulations for instance, or Pottier-Simonet's technique). Moreover, this safety property appears to be quite intuitive, and one may hope to design error messages for the type-checking phase that really point out programming errors, from the confidentiality viewpoint. Last but not least, we show that declassification is handled in a very easy and natural way in this approach.

In a work in progress, Boudol and Kolundzija are investigating a formal framework where two facets of confidentiality issues, namely access control and secure information flow, are taken into account, according to a given, common security policy (that is, a lattice of security levels, with a flow relation). This is done for a high level programming language, extending the one we used for dealing with declassification, that is, a higher-order imperative language à la ML, enriched with contructs for dynamically managing the use of security policy, namely a mechanism for locally extending the flow relation (as in the work of Boudol and Matos on declassification), and constructs for extending, restricting and testing the access rights assigned to a piece of code. The purpose of this work is to find an appropriate notion of "non-interference", and a static analysis technique to ensure it, for a language where the confidentiality dimension of security is taken into account in an integrated way.

Another current line of work, mainly pursued by Ilaria Castellani, concerns the relation between notions of noninterference on programming languages and similar notions on process calculi. On the language side, we focus on a standard parallel imperative language, for which several noninterference properties and related type systems have been studied, following the approach proposed by Volpano, Smith and Irvine. We select two of these properties, together with the associated type systems. On the process calculus side, we consider Milner's calculus CCS equipped with the notion of "Persistent Bisimulation-based Non Deducibility on Compositions" (PBNDC) proposed by Focardi and Rossi. We have studied a type system to ensure the PBNDC-property, which is essentially a restriction to CCS of security type systems previously proposed for the π-calculus. The aim is to devise a translation from the imperative language into CCS, based on Milner's well-known translation, which preserves both the security notions and the security types.

6.1.2. Controlling the complexity of code

The objective of this research activity is to design, study and implement static analysis techniques by which one can ensure that the computational complexity of a piece of code is restricted to some known classes. The motivation is primarily in the mobile code, to ensure that migrating agents are not using local resources beyond some fixed limits, but this could also apply to embedded systems, where a program can only use limited resources.

The work by Amadio and Dal Zilio on "Resource control for synchronous cooperative threads", which we reported upon in the activity report of 2004, has now appeared in the Journal of Theoretical Computer Science [14]. The work by Amadio and Dabrowski on "Feasible reactivity for synchronous cooperative threads" (see our 2005 report) has appeared in the Electronic Notes in Theoretical Computer Science [13].
In the synchronous reactive paradigm, at each instant, the components of the system react to the presence or the absence of events, produce events and synchronize on the end of the instant. More precisely, the end of the instant occurs when every single component has completed its computation for the current instant. A global computation is then performed to determine the state of the system for the next instant. Intuitively, an essential property is that the instants always terminate (in reasonable time and space) so that each component has the opportunity to carry on its future computations. In the the π-calculus (a process algebra inspired by the SL language) events carry values of general data types (lists, trees,...), recursive functions definitions and name mobility are possible and the system receives inputs from its environment at the beginning of each instant. We call *feasibly reactive* a program such that at each instant: (1) the size of the system is polynomial in the size of the initial program and the maximal size of the inputs, and (2) the instant terminates in time polynomial in the size of the initial program and the maximal size of the inputs. We have developed static analysis [23] tools to guarantee that a program meets these requirements. In our approach, the behavior of a component can be described at any moment as a function of the size of the initial program and of the sizes of a finite number of values read on the events. Following this, we rely on a stratification of events to restrict the communication flows in such a way that we are able to deduce an effective bound on the size of the system at the beginning of any instant. Our approach generalizes and improves techniques used in the field of First Order Programs/Term Rewriting Systems to control the complexity of code. More precisely, we combine termination proofs techniques (to guarantee the termination of instants) and the notion of *quasi-interpretation* (to bound the sizes of the values computed by a program). With our static analysis, each thread can be analyzed separately. The complexity of the analysis grows linearly in the number of threads and an incremental analysis of a dynamically changing system of threads is possible. This work improves on our previous work by considering a more general model and by introducing more powerful analysis techniques. In particular, we are now able to consider programs denoting behaviors spawning an arbitrary number of instants while in previous work we only considered programs that, up to constants (denoting for instance the state of the system), were reset at the beginning of each instant.

6.2. Reactive programming

Participants: Roberto Amadio, Gérard Boudol, Frédéric Boussinot, Ilaria Castelani, Frédéric Dabrowski, Stéphane Epardaud.

6.2.1. The reactive model

The paper by Amadio, Boudol, Boussinot and Castellani on "Reactive concurrent programming revisited" has been published [12]. The work by Amadio on "The SL: synchronous language, revisited" (see our activity report of last year) has been accepted for publication in the Journal of Logic and Algebraic Programming. The SL synchronous programming model is a relaxation of the Esterel synchronous model where the reaction to the absence of a signal within an instant can only happen at the next instant. In the above mentioned work, we have revisited the SL synchronous programming model. In particular, we have discussed an alternative design of the model including thread spawning and recursive definitions, introduced a CPS translation to a tail recursive form, and proposed a notion of bisimulation equivalence. In [20], we extend the tail recursive model with first-order data types, obtaining a non-deterministic synchronous model whose complexity is comparable to the one of the π-calculus. We show that our approach to bisimulation equivalence can cope with this extension and in particular that labelled bisimulation can be characterised as a contextual bisimulation.

One of the main aims of our project is to integrate the constructs of reactive programming into expressive programming languages, like SCHEME or ML. However, there is in principle a difficulty in this integration, which is that in the reactive style of programming, and more generally in any multi-threading context where the scheduling is cooperative, any program must be cooperative, or fair, that is any program must release the control after a finite amount of time of running. Thread code may be written so as to be fair, using a "yield" instruction for instance, but it is not clear how to formally check the fairness property (which is undecidable in an expressive programming language). Moreover, we would like to reuse sequential code (that does not "yield"), such as library functions, in a cooperative, multi-threading context, without rewriting it. We cannot require that this code always terminates, because we wish to be able to program inherently non-terminating applications, like any server for instance. As a simple way out of this dilemma, we have proposed in [24] to introduce a touch of preemptive scheduling in the cooperative model. The idea is very simple: it is to consider that, in a higher-order imperative language à la ML, every recursive call to a function yields the scheduler for a while. We think that this is a natural and intuitive idea, since recursion appears to be the source of non-termination. However, there is a technical difficulty, which is that, in a language such as SCHEME for instance, non-termination may occur even without explicitly resorting to recursion. There are two facets to this phenomenon: a first one is that one can encode fixpoint combinators in the pure λ-calculus. The cure in this case is well-known: it is to use a type system, as in ML, to prevent paradoxical programs. Second, as shown long ago by Landin, recursion can be implemented by means of circular references – and this is indeed the way it is implemented in languages like SCHEME or ML. Our main contribution in [24] is to show that circularities in the memory (other than the ones explicitly introduced by recursion) can be ruled out using a type and effect system, and that this is a way to ensure that the (typed) programs are "fair", in our mixed preemptive and cooperative model. The fairness property is shown, as a type safety result, using the classical realizability technique.

6.2.2. Safe concurrent programming

We are designing a new secure language based on Fair Threads [15]. As in Fair Threads, a thread may either be linked to one of the schedulers, or be unlinked. A thread communicates with other threads using both broadcast events and shared memory. Each event is associated with a unique scheduler which defines global instants for the threads linked to it and for its associated events. In our proposal, with each scheduler and each thread is associated a distinct portion of the memory (different portions of the memory are disjoint) such that:

- The memory of a scheduler can only be accessed by the threads which are linked to it.
- The memory of a thread can only be accessed by it. In our proposal, disjointness of such distinct portions of the memory is statically checked.

We also require that the entrance of a new thread within a synchronization space (a scheduler), or the delivery of a signal emitted by a thread not linked to it, can only happen at the beginning of an instant. While under the control of a given scheduler, threads can directly communicate through the part of the memory associated with that scheduler. Within the instant, as the scheduling strategy is cooperative, no time-dependent error leading to unpredictable behaviour can occur. Indeed, the effect of the cooperative strategy is to render atomic the sequence of instructions executed between two cooperation points. The basic property that concurrent programs only access disjoint portions of the memory holds in that case.
A very concise formal semantics exists for fair threads which takes into account the cooperative aspects as well as the preemptive ones. The semantics restricted to a unique scheduler is presented in [17]; the complete semantics, when there are several schedulers, is described in the forthcoming thesis of F. Dabrowsky. Autonomous unlinked fair threads and schedulers co-existing in the same application can be run by distinct kernel processing units (kernel threads) on distinct processors (on multi-processors or multi-core architectures).
We are currently developing an implementation prototype of this new proposal [27]. This implementation serves as a laboratory for the static analysis of both memory disjointness and reactivity. Indeed, we are currently experimenting means to insure the termination of instants in finite time. For this purpose, we first check termination of functions calls (which are first-order functions terminating within an instant and in which only terminal recursion is allowed). This is done by using techniques borrowed from Term Rewriting Systems.

Second, for the purpose of bounding the size of the computed values, stratification-based restrictions are made on the use of shared memory and signals. Third, we check that the number of threads is asymptotically bounded. This static analysis is to be presented in forthcoming work.
The prototype is used to implement various examples of cellular automata made of several thousands of cells, each being a thread. These examples show that systems with large numbers of concurrent units can be efficiently implemented in our language.

6.2.3. Reactive Programming: LURC

LURC, a lightweight reactive library based on ULM, has seen several enhancements this year. LURC is a full featured reactive multi-threading C library with the addition of various thread implementations, all obeying a single semantics, a Reactive Event Loop and language-level features for reactive programming. This year, LURC has gained mutex support, an IO library for fair Input/Output, and support for "upwards" intra-instant continuation (a feature already present in the Scheme ULM embedding).
LURC has been used in the STklos Scheme Virtual Machine as one of two threading backends, which permits the addition of threads in the underlying Scheme VM. This is a joint work with Erick Gallesio (author of STklos), and has proven very beneficial for testing LURC in a real-world scenario. The integration of the LURC thread backend has proven to be as straightforward as the PThread backend.
The use of LURC in a real-world example has outlined several areas where some work is needed in order to bring consistency between the two categories of threads in LURC (synchronous and asynchronous). We believe we have found a semantics for signals and most scheduling features which would work similarly whether called from a synchronous or asynchronous thread, thus allowing the programmer to use the exact same set of features in both cases. This new semantics would also allow to have several synchronous schedulers (possible asynchronous with respect to each other) and allow synchronous threads to migrate from one to another (on the same machine). This could be very beneficial on embedded systems with several processors, such as most game machines – a target for reactive programming.

6.2.4. Reactive and mobile programming: ULM

This year has seen a new implementation of the ULM Virtual Machine, targeted at mobile phones and generally small embedded devices where Java 2 Mobile Edition (J2ME) is available. A lot of work as been put into making the J2ME ULM VM as small as possible (101kb) to fit the device's constraints. This also required a new binary format for representing modules and agents in a compact way, as well as a new communication means for transfering agents: Bluetooth. Now ULM agents can move back and forth between PCs and mobile devices.

6.3. Functional programming

Participants: Gérard Boudol, Damien Ciabrini, Erick Gallesio, Florian Loitsch, Bernard Serpette [Oasis project], Manuel Serrano.

6.3.1. Intersection types

We have collected our work of the last years about type inference for the intersection type discipline of the λ-calculus, and about the proof of strong normalization in this type system, into an article [25] that should appear as a contribution to the Festschrift for the inventors of this type dsicipline, Mario Coppo, Mariangiola Dezani and Simona Ronchi. On a similar topic, van Bakel gives in [21] a new proof for the approximation theorem and the characterisation of normalisability using intersection types. The technique applied is to define reduction on derivations and to show a strong normalisation result for this reduction. From this result, the characterisation of strong normalisation and the approximation result will follow easily; the latter, in its turn, will lead to the characterisation of (head-)normalisability.

6.3.2. Bigloo

The main effort of 2006 have been to had many new libraries required for implementing web applications. In particular, we have added support for SSL, memory mapped areas, and SQL bindings. We have also added various parsers and generators for popular formats TAR, GZIP, ICAL. In addition, we have also implemented new compiler optimizations. In particular, we have added an important optimization that efficiently compiles variable arity functions. It make memory allocation useless when invoking such functions.
We have distributed two major releases of Bigloo during 2006, the version 2.8a and 2.9a. The activity on the Bigloo mailing list has been steady with approximatively new 800 messages posted.

6.3.2.1. Bugloo

The paper submitted last year on stack frame filtering has been accepted to appear in the international journal "Software: Practice and Experience" [16].
Damien Ciabrini has finished his work on Bugloo and has written its PhD thesis. This thesis is devoted to improving symbolic debuggers so that they can deal with the specifics of high-level languages, in particular their complex compilation to general-purpose execution platforms.
Two novel mechanisms are introduced in order to mask artifacts that show up in the stack due to the compilation of high-level languages. The first one is an algorithm that uses special rules designed by language implementors to locate unwanted frames in the stack and to reconstruct a logical view of this stack where the compilation details have been filtered out. With this technique, stacks that contain both native and interpreted code can be visualized seamlessly. The second mechanism provides a means to control single stepping, in order to disallow execution to stop in intermediate functions that were generated by the compiler.
Various memory debugging tools are presented, including a memory profiler designed to support high-level languages and their complex compilation. This tool uses logical stack representation techniques in order to generate statistics that only contain user-level functions. Moreover, these statistics correctly incorporate the allocations that occurred within intermediate functions generated by the compiler.
During this work, a complete debugging support has been developed for Bigloo, a dialect of the functional language Scheme. Other experiments have been conducted on the high-level languages ECMAScript and Python. Results show that our virtualization techniques can be efficiently applied whatever the compilation scheme is. Moreover, they remain effective when programs are composed of several high-level languages.

6.4. Web programming

Participants: Erick Gallesio, Florian Loitsch, Manuel Serrano.

6.4.1. Hop

Hop is a programming language and an execution environment for web applications. It relies on a web *broker* that combines the functionalities of a web server and a web proxy. The broker embeds a native code generator, an interpreter for evaluating server-side code and a compiler for compiling client-side code.
The Hop programming language enforces a programming model where the graphical user interface and the logic of an application were executed on two different engines. In theory, the execution happens as if the two engines are located on different computers even if they are actually frequently hosted by a single computer. In practice, executing a Hop application requires:

- A Hop broker which is the execution engine of the application. All computations that involve resources of the local computer (CPU resource, storage devices, various multi-media devices, ...) are executed on the broker. The broker it also in charge of communicating with other Hop brokers or regular web servers in order to gather the information needed by the application.
- A web browser that plays the role of the engine in charge of the graphical user interface. It is the terminal of the application. It establishes communications with the Hop broker.

The Hop programming language provides primitives for managing the distributed computing involved in a whole application. In particular, at the heart of this language, we find the with-hop form. Its syntax is:

```
   (with-hop (service a0 ..) callback)
```

Informally, its evaluation consists in invoking a remote service, i.e., a function hosted by a remote Hop broker, and, on completion, locally invoking the callback. The form with-hop can be used by engines executing graphical user interfaces in order to spawn computations on the engine in charge of the logic of the application. It can also be used from that engine in order to spawn computations on other remote computation engines.
The first version of Hop has been released in May 2006. The statistics of the site http://hop.inria.fr/ have shown that during the first week, the main Hop web page has been visited more than 10.000 times, which have required to serve more than a million files.

6.4.2. Scm2Js

The Hop programming language is based on the Scheme programming language. In order to be executed Hop client-side codes have to be compiled to JavaScript which is the language of web browsers. JavaScript and Scheme are similar languages and share many features. Despite JavaScript's high level of abstraction it is hence possible to compile Scheme code to efficient JavaScript code. Our compiler SCM2JS generally produces JavaScript code that is as efficient as equivalent handwritten code.
SCM2JS has not been designed to implement the full R5RS specification, but it features proper tail recursion and continuations. Our tail recursion technique is reasonably efficient (our benchmarks ran at most 2.1 slower with this technique enabled) and has the merits of being compatible with existing JavaScript code. That is the call conventions are not changed and the generated code can still use JavaScript libraries and conversely JavaScript code can use generated functions and data.
Our `call/cc` implementation is still a work in progress, but we expect it to be sufficiently fast to be enabled by default in Hop.

7. Contracts and Grants with Industry

7.1. CRE France-Télécom R&D

A CRE (contract for external research) started this year, funded by France-Télécom R&D on "Analysis of security properties for global programming frameworks". The duration of the project is 3 years (may be extended to 4), and the total funding is 120 kEuros. The purpose of the project is to support the research done in MIMOSA on security issues and mobile code, and to study the applicability of our methods and results to concrete problems investigated at France-Télécom R&D.

8. Other Grants and Activities

8.1. National initiatives

8.1.1. APP registration for Hop

Hop has been registered to the APP on June 1fst, 2006 under the reference : `IDDN.FR.001.260002.000.S.P.2006.000.10400`.

8.1.2. ACI Sécurité Informatique ALIDECS

Frédéric Boussinot is participating to the ACI Sécurité Informatique ALIDECS whose coordinator is Marc Pouzet. The ACI started in october 2004. Paricipants are Lip6 (Paris), Verimag (Grenoble), Pop-Art (Inria Rhône-Alpes), Mimosa (Inria Sophia) and CMOS (LaMI Évry). The objective is to study an integrated development environment for the construction and use of safe embedded components.

8.1.3. ACI Sécurité Informatique CRISS

This action started in July 2003. The participants are, besides MIMOSA and the *Laboratoire d'Informatique Fondamentale* of Marseilles, the LIPN from Paris (Villetaneuse) and the INRIA project CALLIGRAMME from LORIA in Nancy. Roberto Amadio is the coordinator of the CRISS action. Its main aim is to study security issues raised by mobile code. The project ends this year, with a final report delivered in September and presented at PARISTIC in Nancy, November 22-24.

8.1.4. ACI Nouvelles Interfaces des Mathématiques GEOCAL

Roberto Amadio and Gérard Boudol are participating in this action. The other teams are the LMD Marseilles Luminy (coordinator), PPS Paris, LCR Paris Nord, LSV Cachan, LIP Lyon (PLUME team), INRIA Futurs, IMM Montpellier, and LORIA Nancy (CALLIGRAMME project). This action ends this year.

9. Dissemination

9.1. Seminars and conferences

Gérard Boudol participated in February in a workshop of the CRISS Project in Marseilles. In July, he participated in a seminar at France-Télécom R&D in Grenoble, and presented the work of MIMOSA on security and mobile programming. In September, he participated in the last workshop of the CRISS project in Paris, and gave a talk on the work on the reactive model of programming, with a focus on [24]. He was an invited speaker at the FST-TCS Conference in Kolkata, December 13-15, where he gave a talk on [24].

Ilaria Castellani participated to the final meeting of the ACI project CRISS (PPS laboratory, Paris, September 14, 2006), where she gave a review talk on the project's work on type systems for information flow. She attended the LIX Colloquium on Emerging Trends in Concurrency Theory (École Polytechnique, Paris, November 13-15, 2006).

Frédéric Dabrowski presented the paper [17] at the TV'06 Workshop, and attended to the CAV conference. He participated to the GEOCAL 2006 workshop where he presented some preliminary results about ongoing work on feasible reactivity for synchronous languages. In February, he participated in a CRISS workshop in Marseille. In September, he participated in a CRISS workshop in Paris where he gave a talk on [23].

Marija Kolundzija participated in the Bertinoro International Spring School in March, in the International School on Rewriting in Nancy in July, and the International Chambéry-Torino Summer School in Torino in August.

Manuel Serrano gave an invited talk on *the Hop programming environment* [19] at the 7th Scheme and Functional Programming Workshop that took place in Portland, Oregon. He gave an invited seminar at the Brown University in Providence, Rhodes Island, USA. He gave a presentation on the *Hop programming language* at the first symposium on Dynamic languages [18]. He participated to the Dagstuhl seminar on *Dynamic Languages*.

9.2. Animation

Gérard Boudol was examiner in the defense of the PhD Thesis of Fransisco Martins (University of Lisbon). He was a member of the program committee of the ESOP'07 Conference.

Frédéric Boussinot was examiner in the defense of the PhD Thesis of Louis Mandel (University Paris 6). F. Boussinot is examiner of the PhD thesis submitted by A. Teitelbaum entitled "Arts'codes: a New Methodology for the Development of Real-Time Embedded Applications for Control Systems", RMIT University, Melbourne, Autralia.

Ilaria Castellani was a member of the programme committee of the conference MFCS'06 (31st International Symposium on Mathematical Foundations of Computer Science, Star_ Lesn_, Slovakia, August 28 - September 1, 2006).

Manuel Serrano was a referee for the PhD thesis of Jean Privat (University of Montpellier II).

10. Bibliography

Major publications by the team in recent years

[1] R. AMADIO, P.-L. CURIEN. *Domains and Lambda-Calculi*, Cambridge University Press, 1998.

[2] G. BERRY, G. BOUDOL. *The chemical abstract machine*, in "Theoretical Computer Science", vol. 96, 1992.

[3] G. BOUDOL. *The π-calculus in direct style*, in "Higher-Order and Symbolic Computation", vol. 11, 1998.

[4] F. BOUSSINOT. *Objets réactifs en Java*, Collection Scientifique et Technique des Telecommunications, PPUR, 2000.

[5] F. BOUSSINOT. *La programmation réactive*, Masson, 1996.

[6] F. BOUSSINOT, J.-F. SUSINI. *Java threads and SugarCubes*, in "Software Practice & Experience", vol. 30, 2000.

[7] I. CASTELLANI. *Process Algebras with Localities*, in "Handbook of Process Algebra, Amsterdam", J. BERGSTRA, A. PONSE, S. SMOLKA (editors). , North-Holland, 2001, p. 945-1045.

[8] D. SANGIORGI, D. WALKER. *The π-Calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.

[9] M. SERRANO. *Bee: an Integrated Development Environment for the Scheme Programming Language*, in "Journal of Functional Programming", vol. 10, n^{o} 2, May 2000, p. 1–43.

Year Publications

Doctoral dissertations and Habilitation theses

[10] A. ALMEIDA MATOS. *Typing Secure Information Flow: Declassification and Mobility*, Ph. D. Thesis, École National Supérieure des Mines de Paris, January 2006.

[11] D. CIABRINI. *Débogage symbolique multi-langages pour les plates-formes d'exécution généralistes*, Ph. D. Thesis, Université de Nice Sophia Antipolis, Oct 2006.

Articles in refereed journals and book chapters

[12] R. AMADIO, G. BOUDOL, F. BOUSSSINOT, I. CASTELLANI. *Reactive concurrent programming revisited*, in "Electronic Notes in Theoretical Computer Science", vol. 162, 2006, p. 49-60.

[13] R. AMADIO, F. DABROWSKI. *Feasible reactivity for synchronous cooperative threads*, in "Electronic Notes in Theoretical Computer Science", vol. 154-3, 2006.

[14] R. AMADIO, S. DAL ZILIO. *Resource control for synchronous cooperative threads*, in "Theoretical Computer Science", vol. 358, 2006, p. 229-254.

[15] F. BOUSSINOT. *FairThreads: mixing cooperative and preemptive threads in C*, in "Concurrency and Computation: Practice and Experience", vol. 18, n^{o} DOI: 10.1002/cppe.919, 2006, p. 445-469.

[16] D. CIABRINI. *Stack Filtering for Source Level Debugging*, in "To appear in Software: Practice and Experience", 2006.

Publications in Conferences and Workshops

[17] F. DABROWSKI, F. BOUSSINOT. *Cooperative Threads and Preemptive Computations*, in "Proceedings of TV'06, Multithreading in Hardware and Software: Formal Approaches to Design and Verification, Seattle", 2006.

[18] M. SERRANO, E. GALLESIO, F. LOITSCH. *HOP, a language for programming the Web 2.0*, in "Proceedings of the First Dynamic Languages Symposium, Portland, Oregon, USA", Oct 2006.

[19] M. SERRANO. *The HOP Development Kit*, in "Invited paper of the Seventh Acm sigplan Workshop on Scheme and Functional Programming, Portland, Oregon, USA", Sep 2006.

Internal Reports

[20] R. AMADIO. *A synchronous pi-calculus*, Technical report, Université Paris 7, Laboratoire PPS, 2006.

[21] S. VAN BAKEL. *The heart of intersection type assignment*, Technical report, n^{o} RR-5984, INRIA, 2006, https://hal.inria.fr/inria-00096419.

Miscellaneous

[22] A. ALMEIDA MATOS, G. BOUDOL, I. CASTELLANI. *Typing Noninterference for Reactive Programs*, to appear in Journal of Logic and Algebraic Programming, 2006.

[23] R. M. AMADIO, F. DABROWSKI. *Feasible Reactivity in a synchronous pi-calculus, Draft*, 2006.

[24] G. BOUDOL. *Cooperative vs preemptive scheduling: a trade-off, or: typing termination in a higher-order imperative language*, 2006, draft, submitted.

[25] G. BOUDOL. *On strong normalization and type inference in the intersection type discipline*, 2006, invited paper for a special issue of TCS in honor of Mario Coppo, Mariangiola Dezani and Simona Ronchi on the occasion of their 60th birthday.

[26] G. BOUDOL. *Secure information flow as a safety property*, 2006, unpublished draft.

[27] F. BOUSSINOT, F. DABROWSKI. *Secure Multicore Programming*, 2006.