ANNEXE 1 : ANNEXE TECHNIQUE
ANR-06-SETIN-010

Sommaire

1 Acronyme et titre du projet

PARSEC : PARallelism and SECurity

2 Description courte du projet

2.1 Contexte et motivation du projet

Concurrent programming techniques have been elaborated long ago, with the earliest design of time-sharing operating systems in the late 60's. Simultaneously, security issues appeared with the advent of multi-users computing systems. The concept of a process, with its own address space, and techniques for access control to shared resources were developed, providing satisfactory solutions to the problems arising in such systems.

The last decade has seen concurrency entering programmer's everyday life, in particular because the popular JAVA language imposes programs to use threads, that share an entire address space. Indeed, the concept of a cooperating parallel component provides a very attractive idea for modular design. Moreover, concurrent programming techniques are needed to benefit from new computing architectures that are now widely spread, like multicore processors or multiprocessor machines, and also to program in an efficient way new applications that are inherently massively concurrent (but not necessarily distributed), and open to untrusted parties, like web servers, multi-player games or Internet-scale storage applications. In the past few years, it has been recognized that kernel threads are too restrictive and too inefficient to provide a convenient means for programming these applications, and that a user-level thread facility would provide better support [Anderson & al, Banga-Druschel-Mogul, von Behren & al, Engelschall]. Whether this should follow a preemptive discipline, or an event-driven model, or else a cooperative style is still a research matter [Adya & al, von Behren-Condit-Brewer, Ousterhout]. Similarly, the well-established technologies for security are inadequate in the case of applications where a number of concurrent activities should share an address space. It is clear for instance that a sand-boxing technique is far too restrictive in programming such applications. Moreover, access control has to be complemented with new security policies, ensuring for instance that a thread does not consume common resources in an immoderate way, or corrupt them, or disclose confidential information to other threads having low access rights.

Our PARSEC[1] project intends to study concurrent programming techniques for these new architectures and applications, focusing on the security issues that arise in multi-threaded systems. To this end, we have built a coherent group made of five teams, namely the CONCURRENCY group (R. Amadio) at the PPS Lab of Paris 7 University and CNRS, the EVEREST (G. Barthe) and MIMOSA (G. Boudol, coordinator) teams at INRIA Sophia Antipolis, the LANDE (T. Jensen) team at IRISA, and the MOSCOVA (J.-J. Levy) team at INRIA Rocquencourt, sharing their expertise in concurrency and/or security. As a basic research project, we do not aim at achieving a pre-competitive objective, and therefore our research activity will span a broad area - say from bisimulations for a synchronous (in the sense of synchronous languages à la ESTEREL) 7r-calculus, to access and resource control for multi-threaded JAVA programs in mobile phones -, and our "deliverables" will essentially consist in theoretical results published in articles. Our main objective is to understand what could be an *efficient* and *"security-minded" concurrent*

[1] for PARallelism and SECurity, where "parallelism" should here be understood as "concurrency". for administrative reasons, these two teams at the same laboratory are not counted as two distinct partners, and both appear under the "coordinator" heading in the project description.

programming model. Such a model should both be convenient for programming the kind of applications mentioned above, and support mechanisms to ensure high-level security policies. To achieve this objective we will have to study well-known models, like JAVA multi-threading, Posix threads or shared memory models, as well as less standard ones, like the cooperative model which is gaining some popularity in user-level thread systems, or the synchronous model. We will have to understand what is the right safety and security properties to guarantee regarding multi-threaded systems, like for instance the "non-interference" property, which is not so clear in this case, or the "availability" property, meaning in particular that a single thread should not monopolize any shared resource. We will also have to design and study means to ensure these properties, using and adapting standard tools like type systems, program logics (including Floyd-Hoare logic, temporal logics, or Reynolds' separation logic) or run-time verification mechanisms.

There are many difficulties in this task. First of all, the semantics of multi-threaded programming is not at all well understood. While it is very easy to provide a formal "interleaving" semantics for thread systems with shared memory, this semantics never coincides with what is actually implemented, for various reasons. One reason is that the grain of atomicity is usually not preserved by the implementation, and a program may be time-sliced at some point of its execution which makes no sense at the language level (see [Reynolds 04] for instance). Another reason is that compilers usually make optimizations that are valid for sequential programs, but break the interleaving semantics when they concern shared parts of the memory *(cf.* [Adve-Gharachorloo]). Similarly, modern hardware processors perform optimizations, rearranging the code and buffering data in ways that again break the high-level concurrent semantics (see again [Adve-Gharachorloo]). As a result, the memory model of the implementation usually does not correspond to the one of the semantics [Adve-Manson-Pugh, Boehm, Pugh]. For all these reasons, and in addition to the fact that synchronization is a delicate matter [Birrel], even at the language level, classical (i.e. preemptive) multi-threaded programming is extremely difficult, and error prone. It is thus necessary to design techniques to analyze concurrent programs in order to make them "thread safe", avoiding or detecting for instance race conditions [Abadi-Flanagan-Freund, Flanagan-Qadeer, Grossman], but also to improve the interaction with optimizing compilers and optimizing hardware (using "volatile" declarations for instance). Besides the study of this classical, preemptive model, one may also explore, as we plan to do, the use of cooperative threads at the language level (see [Serrano-Boussinot-Serpette]). This is by now advocated as a better model for applications like the ones mentioned above [Adya & al, von Behren & al], but this programming style is not free of defects. In particular, cooperative programs have to indeed cooperate, in order not to make the other threads starving for the processor, and there is by now no well established technique to guarantee cooperation for programs written in an expressive language. The same problem arises with the reactive [Hughes-Pareto-Sabry] or synchronous [Amadio-Dabrowski, Amadio-Dal Zilio] programming styles. Another question is how to take advantage of a multi-processor architecture in a cooperative style of programming [Boussinot 06]. Summarizing, there is still a lot of basic research to do in order to design a concurrent model in which programming would be reasonably safe.

Multi-threaded applications also raise some difficult questions regarding security. As we indicated above, in such applications access control has to be complemented with information flow control, and resource usage control. It is well-known that secure information flow is difficult to formalize in a concurrent setting, due to the non-deterministic character of concurrent systems (see [Ryan & al] for instance), and to the various kinds of leaks that can arise, like

termination leaks or timing leaks *(cf.* [Sabelfeld-Myers]). Moreover, even for weak notions of secure information flow, most of the well-known techniques (e.g. type systems [Smith-Volpano, Volpano-Smith-Irvine]) for enforcing such security properties are far too restrictive to be of practical use. We thus have to find means to improve this situation. We believe that the cooperative model for concurrency is better suited than the preemptive one for a language-based approach to information flow security, because in this model the flow of control is determined at the programming level, but this belief has still to be proved right by some formal results. As a matter of fact, some difficult questions regarding secure information flow already arise in a purely sequential setting. For instance, we are still lacking an adequate treatment of exception mechanisms à la JAVA; also, secure information flow (non-interference) should be made closer, in one direction or the other, to static analysis techniques (type systems) that are meant to enforce it. Similarly, although multi-threaded applications provide a strong motivation for studying resource usage control, there is still no well-established technique that would provide, in the sequential case, static means to enforce reasonably accurate bounds on resource usage. Even the standard issue of access control needs further investigation, to formally understand for instance what exactly is guaranteed by the run-time protection mechanisms in JAVA.

Our proposal is to make contributions to solving all the above mentioned issues, aiming at achieving, at the end of the project, a good understanding of what could be a *safe and secure concurrent programming model,* based on theoretical justifications. To this end, we shall follow in the PARSEC proposal a language-based approach to address in a formal way safety and security issues such as access control, secure information flow, and resource usage control, focusing in particular on concurrent programming constructs, and especially on cooperative concurrency. We will, as far as possible, experiment our ideas via prototype software, but we do not expect these to provide pre-competitive technology that could be directly exploited industrially.

3 Project description

We organize our project into three main tasks, without specific temporal or causal order.

- A first one deals with *concurrent programming models,*
- a second and most central one focuses on *security and concurrency,*
- a third is concerned with *machine-checking security properties.*

The second task is divided into three subtasks, dealing with *logics for reliable and secure concurrent programs, secure information flow,* and *resource and access control* respectively. For each of these (sub)tasks, we begin the description with an introduction to the area and then list the specific research objectives we have in the short and medium terms. In this description, we shall not detail the contributions of each team involved in the PARSEC proposal. Since many of our specific objectives are actually shared, sometimes with different approaches, by the various teams, we expect that active collaborations will naturally emerge and be developed for achieving these objectives. Let us just list at this point what are the areas of expertise of the various teams as concerns PARSEC'S research domain: *process calculi,* and more generally *semantics of concurrency* (MIMOSA, MOSCOVA, PPS), *logics of programs* (EVEREST, LANDE, MOSCOVA) and *static analysis for security* (EVEREST, LANDE, MIMOSA, PPS), *formal semantics and verification of* JAVA *programs* (EVEREST, LANDE).

3.1 Concurrent programming models

The major difficulty in concurrent programming with shared memory lies in the dependency of program execution on the scheduling. This difficulty arises in debugging, testing, analyzing, and porting concurrent applications. There are basically two ways in which the execution of concurrent programs can be managed. In the *preemptive* manner, a program, or more precisely its executable version, can be interrupted at any time during its execution by an external device, the scheduler, and the resources needed for execution are then given to another concurrent component for a while. This is the execution model that has been adopted in most multithreading and operating systems, as well as in multi-processor architecture. This model is also known as the *interleaving* model, which has been adopted in most process algebra models. In the *cooperative* scheduling discipline on the contrary, a program decides, by means of specific instructions, when to leave its turn to another concurrent component, and the scheduling is distributed among the components.

A specific difficulty for the programmers with scheduling is that their programs are executed via compilers and processors that invest a great time and effort optimizing the code and reordering it to ensure that it is run in the most efficient possible way [Adve-Gharachorloo]. At the same time, programmers make assumptions about the way the code is executed. In a single threaded program, it is fairly easy for a processor to ensure that program transformations do not interfere with possible results of the program, and programmers will generally not need to reason about potential optimizations when they write sequential programs. When there are multiple threads of instructions executing at the same time, and those threads are interacting, program transformations can result in bizarre side effects [Boehm]. Every hardware and software interface of a system that admits multithreaded access to shared memory requires the specification of how memory actions in a program will appear to execute to the programmer. This specification must strike a balance between ease-of-use for programmers and implementation flexibility for system designers. The model that is most commonly assumed and easiest to understand is *sequential consistency:* this model basically reflects an interleaving of the actions in each thread, at the programming language level. However, a sequentially consistent system does not have much freedom to transform memory statements within a thread, thus precluding many important optimizations. Since the 80's, processors implement relaxed memory models, which do not guarantee sequential consistency [Adve-Gharachorloo]. It is therefore important to understand in a formal way these memory models, in order to be able to define the semantics of multi-threaded systems, which is the basis for formal reasoning and analysis.

The cooperative programming model has been advocated as a better model than the preemptive one for programming some modern, massively concurrent applications [Adya & al, von Behren & al]. Moreover, since cooperative programs are usually compiled as sequential code, this model is also better than the preemptive one as regards the interactions with optimizing compilers and hardware. However, the cooperative concurrency model also has its drawbacks, the main one being that if the active program runs into an error, or raises an un-caught exception, or diverges, then the model is broken, in the sense that no other component will have a chance to execute. Then cooperative programming seems to be only practicable in a *safe* language, like ML, where errors are, in principle, detected during a static analysis phase. However, non-terminating computations cannot be forbidden. Any server for instance should conceptually have infinite life duration, and should not be programmed to stop after a while. Still, such a server should not enter into an infinite loop, it should rather be infinitely often waiting for a new request. In other words, in cooperative programming, programs should be cooperative, that is, they should be

guaranteed to either terminate or suspend themselves infinitely often.

Synchronous programming [Benveniste & al], as implemented with various languages such as LUSTRE and ESTEREL, can be seen as a particular instance of cooperative concurrency. LUSTRE follows a data-flow approach whereas in ESTEREL, program components release the control when waiting for a signal to be emitted, and resume when this signal is present. The SL (Synchronous Language) introduced in [Boussinot-de Simone], which can be regarded as a relaxation of the ESTEREL model where the reaction to the absence of a signal within an instant can only happen at the next instant, has gradually evolved into a general purpose programming language for concurrent applications and has been embedded in various programming environments such as C [Boussinot 91], JAVA [Boussinot-Susini], SCHEME [Serrano-Boussinot-Serpette], and CAML [Mandel-Pouzet]. However, the semantic theory of this family of synchronous languages remains largely underdeveloped. We will then investigate the semantical issues regarding the preemptive, cooperative and synchronous concurrent programming styles, and compare them, in order to get a better understanding of what could be a good model for programming modern applications. The main criterion in this study will be the ability for a particular style to support analysis and enforcement mechanisms for high-level security policies.

3.1.1 Objective 1. Formalization of the JAVA memory model

One of our objectives is to design practical verification methods for multi-threaded JAVA programs, and to analyze some security related features offered with this language. Then we have to precisely define the semantics of JAVA multi-threading, and this requires taking the JAVA Memory Model [Adve-Manson-Pugh] into account. We plan to formalize this model, using the COQ proof-assistant, with the objective of proving that in case a program is correctly synchronized (meaning that it does not contain race conditions [Abadi-Flanagan-Freund]) the set of its legal behaviors coincides with the behaviors described by an interleaving semantics (this property is claimed by the developers of the JAVA Memory Model - we intend to verify it formally). Restricting our attention to verifying correctly synchronized programs will allow us to assume an interleaving semantics to prove the correctness of the program logic and of the thread modular verification techniques (see below). Another line of work on this topic, namely the study of relaxed memory models using Reynold's separation logic, is described in the next section.

3.1.2 Objective 2. On the cooperative programming model

One of the aims of the project is to promote cooperative programming as an appropriate model for secure programming. Then a first step is to design means to guarantee that cooperative programs are indeed cooperative. To this end, we have to define suspension points in a program, and to design methods to ensure that a program either terminates or reaches a suspension point in finite time. We plan to do that for a higher-order imperative core programming language with cooperative threads, using type systems. We will have to solve a problem for which there is, to the best of our knowledge, no available solution in the literature, namely to find an analysis for guaranteeing termination for programs with a higher-order mutable store. In particular, the analysis will have to rule out circularities that can arise when storing functions in the memory, while allowing recursion.

3.1.3 Objective 3. Semantic foundations of a synchronous model

Regarding synchronous concurrency, the objective is to build a semantic theory comparable to

what has been achieved with process algebras for interleaving concurrency. In recent work [Amadio 05b], we have revisited the synchronous programming model, introducing an alternative design that includes thread spawning and recursive definitions. We have defined a CPS translation to a tail recursive form, and proposed a novel notion of bisimulation equivalence with good compositionality properties. The original SL language as well as the revised one assumes that signals are pure in the sense that they carry no value. Then computations are naturally deterministic and bisimulation equivalence collapses to trace equivalence. However, practical programming languages that have been developed on top of the model include data types beyond pure signals, and this extension makes the computation non-deterministic, unless significant restrictions are imposed. Therefore, an issue we plan to address is whether our preliminary theory is robust enough to support extension to a non-deterministic language with data types and signal name generation. A first step would be to consider an extension with first-order values, including signal names. This should lead us to the design of a synchronous calculus similar to the 7r-calculus. Regarding the semantic theory, we plan to use a notion of contextual bisimulation in the sense of [Honda-Yoshida 93], which has never been applied to a synchronous language. We expect that the resulting semantic theory for the SL model will have a positive fall-out on the development of various static analysis techniques to guarantee properties such as reactivity [Amadio-Dal Zilio], determinacy [Mandel-Pouzet], and non-interference [Almeida-Boudol-Castellani].

3.1.4 Objective 4. Determinacy

Regarding determinacy, one may observe that, to escape some of the difficulties caused by the scheduling, researchers have been interested in identifying composition and interaction mechanisms that, while allowing parallelism, preserve the determinacy of the observable behavior of a program (in this respect, two notable examples are Kahn networks and the languages of the ESTEREL family). In the synchronous model with value passing the non-determinism lies in the order in which the values emitted on a signal are received (whether during the instant or at the end of the instant), which depends on the choice of the next component to execute. Accordingly, one may consider two restrictions to make the computation deterministic: for signals that can be read during an instant, then at most one value should be emitted on that signal during each instant. For signal that is read at the end of the instant, one should process the emitted values in a way that is independent of the emission order. While this may work in certain cases, it is hard to conceive such a "commutative" processing when manipulating objects such as pointers. It seems that a general notion of deterministic program should be built upon a suitable notion of program equivalence. Our planned work here consists in designing checkable conditions that guarantee a deterministic behavior, and in experimenting the expressivity of these conditions in concrete situations. In another direction, we are also exploring the semantics of deterministic fragments of the asynchronous 7r-calculus. Specifically in [Varacca-Yoshida] we identify a linearly typed version of the 7r-calculus with "internal mobility," and we provide semantics for it in event structures that are conflict- and confusion-free. The fragment of the -zr-calculus considered is sufficiently general to encode the A-calculus. We plan to study in detail the derived event structure semantics and to compare it with a concurrent game semantics of the A-calculus.

3.2 Security and concurrency

3.2.1 Logics for reliable and secure concurrent programs

Program verification is the "holy grail" of software reliability: it allows one to formally prove that, for all possible runs of the program, a property holds. Floyd and Hoare pioneered the use of logics for program specification, by relating the logical descriptions of the states of a machine before and after the execution of a command. Despite its many advantages, program verification is rarely used in practice. One key reason for this lack of use is that current program verification techniques do not scale. References and pointers are a particular impediment for scalability, as they allow apparently unrelated pieces of code to affect each other's behavior. However, several logics (and tools) have been developed to reason about single-threaded programs at source code level [Huisman PhD, Jacobs-Poll, von Oheimb]. For multi-threaded applications some initial theoretical investigations have been made into their verification [Abraham PhD, Abraham-de Roever-Steffen], though their practical feasibility still remains an open issue. Moreover, O'Hearn, Reynolds and Yang have recently developed an approach called "local reasoning" that allows dealing with pointers and references, and has the potential to scale. The key observation is that separate program texts which work on separate sections of the store can be reasoned about independently. This idea can be embedded in an appropriate substructural logic, called *separation logic* [Reynolds 02], that presents a connective to model the disjointness of two portions of the store. Small but intricate graph-manipulating programs can then be specified and proved correct with relatively little fuss. Preliminary investigations of extensions of separation logics to local concurrency [O'Hearn] suggest that it is possible to design a logic suited to reason about concurrent cooperative programming. Separation logic tracks exactly the resources held at any time by a program. As a consequence, it is an invaluable tool to detect, on the one hand resource leaks, on the other which parts of a concurrent program share data, potentially interfering.

3.2.1.1 Objective 5. Separation logic and relaxed memory models

Recently, the use of general-purpose logics such as Floyd-Hoare logics or temporal logics, that provide standard means to specify and verify functional and/or behavioral properties of programs, has been advocated to verify security properties of programs. A long term line of investigation might then be the study of techniques based on these logics, as well as on separation logic to enforce the security properties that are at the heart of the PARSEC proposal. We conjecture for instance that it is possible to use a concurrent extension of separation logic to prove properties of concurrent programs running on systems that implement relaxed memory models. More precisely, we conjecture that if a program is proved sound supposing strong consistency, then it is sound even if strong consistency is relaxed. The logic is expected to play a fundamental role here: both separation logics for concurrency and relaxed memory models are designed to support easily standard locking mechanisms, and the constraints imposed by the logic should ensure the soundness of the proof.

3.2.1.2 Objective 6. Logics for verifying multi-threaded programs

One of our more immediate aims consists in designing practical verification methods for multi-threaded programs, and in proving their soundness. Initial ongoing work in this direction has identified a promising approach to the verification of multi-threaded programs, namely that of augmenting traditional pre- and post-conditions style specifications with atomicity information. This has the advantage that we can still use existing methods and tools for verification of single-threaded code, and that we can separate the multi-threaded aspects from the functional aspects of the specification. This approach requires that we define the semantics of notions such as

atomicity, independence and immutability, and that we define appropriate proof rules for verifying them. To this end, we need to formally study the JAVA Memory Model, as described in the previous section. We will also explore ways of simplifying reasoning, investigating in particular how one can exploit the fact that an object is "local" to a single thread, and how this can be verified.

3.2.1.3 Objective 7. Logical characterizations of secure information flow

An example of using general-purpose logics to verify security properties was first given in [Barthe-D'Argenio-Rezk, Darvas-Hâhnle-Sands], where it is shown that non-interference (see below) of a program *P* can be reduced to a property about single program executions (universally quantified over all possible program inputs) of the program *P; P'*, where *P'* is a "renaming" of *P*. This idea of *self-composition* has then been further exploited in a number of papers. Huisman *et al* [Huisman-Worah-Sunesen] have recently proposed a characterization of observational determinism using this technique. Furthermore, Naumann [Naumann] has extended the technique to encode secure information flow to mutable heap object structures using ghost fields. This extension shows a direct application of self composition to verification of object oriented languages, both for secure information flow and data refinement. Also for object oriented languages, Dufay et al. [Dufay-Felty-Matwin] have experimented with verification of information flow using self-composition and JML specifications in the Krakatoa tool that generates proof obligations in the COQ theorem prover. They have illustrated this approach on data-mining algorithms. The idea of self-composition is also used by Terauchi and Aiken [Terauchi-Aiken] to formulate a notion of relaxed non-interference. They also propose a type-directed transformation as a solution for some safety analysis tools that try to solve problems semantically, and whose analysis will possibly not terminate in presence of certain predicates, e.g. predicates including complex arithmetic. Then an objective we have in our project is to pursue this line of work, aiming at logical characterizations of secure information flow for more and more complex policies and richer languages.

3.2.2 Secure information flow

Security models for mobile code typically rely on typing mechanisms that statically enforce basic safety policies (e.g. code containment or absence of pointer arithmetic), and on run-time mechanisms such as stack inspection in JAVA and .NET that dynamically enforce access control policies. While these mechanisms are intended to enforce confidentiality and integrity policies, it is difficult to assess the end-to-end policies that they guarantee. In particular, they do not guarantee information-flow policies that are commonly desirable in scenarios involving the execution of untrusted code. A similar observation can be made regarding the kind of massively concurrent applications that we mentioned in introducing our proposal. For instance, if requests sent to an Internet-scale medical database are implemented as spawning dedicated threads, then one must have guarantees that a thread issued by a trusted doctor will not inadvertently disclose confidential information about patients to another, possibly malicious client, indexing for instance the database for statistical purposes.

Secure information flow, that is the property that information is only flowing from a security level to another if this is allowed by the given security policy, is usually formalized as a *non-interference* property, relating several executions of a program in the context of memories that contain the same public information (this is actually Cohen's *strong dependency*, see [Sabelfeld-

Myers]). In an imperative language this property amounts to the absence of dependency between the values of secret variables and the values of public variables. Besides direct flows, arising from writing confidential values in the low part of the memory, one may identify several kinds of leaks: indirect flow, arising from the branching structure of the program, termination leaks, when performing a low write depends on the termination of a piece of code, that in turn depends on confidential information, or timing leaks, when observing the duration of the computation may reveal confidential information (see again [Sabelfeld-Myers]). The last two kinds of leaks typically occur in the concurrent execution of threads using a shared memory, and may depend on the scheduling strategy [Smith-Volpano]. In a process calculus setting, one is concerned with public events not being influenced by secret events while processes communicate. Here different security properties have been proposed, mostly based on trace equivalence or bisimulation, among which *nondeducibility on composition* and *persistent nondeducibility on composition* are the most popular *(cf.* [Focardi-Gorrieri] for a review).

To enforce secure information flow, information-flow type systems, that provide a means to track data flows and control dependencies, have been largely used, starting from the pioneering work by Volpano, Smith and Irvine on a core sequential imperative language [Volpano-Smith-Irvine] Since then, this methodology has become somehow classical and several type systems and other static analysis techniques have been put forward for increasingly complex languages, including features like higher order, dynamic thread creation, scheduling policies, exceptions etc., and culminating with type systems for realistic languages such as JAVA [Myers] or CAML [Pottier-Simonet]. As regards the calculus-based approach, some work on methods for static detection of insecure processes appears in [Crafa-Rossi, Hennessy-Riely, Honda-Yoshida 02, Pottier]. However, this by now well-established approach has not yet been largely used in practice, for several reasons. The notion of non-interference itself has been often criticized [Ryan & al], from various viewpoints. Its extension to non-deterministic systems, and in particular to concurrent ones, for instance, is problematic. It also forbids the use of any declassification mechanism, without which however a security-minded programming language would be useless. A generalization of non-interference dealing with declassification has recently been proposed [Almeida-Boudol].

It is a well-known fact that, while information flow type systems provide an attractive means to enforce non-interference, they often turn out to be overly conservative in practice, mainly because they usually reject many programs that are secure from the non-interference point of view. On the one hand, this is unavoidable: type systems usually provide a computable approximation for a generally undecidable property, and cannot therefore be complete. Moreover, type systems are usually compositional, and therefore reject code that contains insecure fragments, even if these fragments are never executed. On the other hand, the line of research that aims at improving the type systems or finding more precise means to ensure the property to guarantee cannot be neglected. This has motivated some work to propose program analysis methods that are closer to the property to ensure (e.g. [Barthe-D'Argenio-Rezk]). Several of the research objectives described below pertains to this line of work.

3.2.2.1 Objective 8. Improving information flow type systems (1)

For instance, the type system of [Barthe-Rezk] is inherently imprecise in its treatment of exceptions, because all instructions that can throw exceptions are considered as branching statements, and thus many secure programs are rejected. In order to retain some acceptable degree

of precision and not to reject too many programs, we intend to let the type system depend on previous analyses, including a control dependence region (cdr) analysis - already considered in [Barthe-Rezk] - a class analysis and an exception analysis. Although the need for auxiliary analyses has been recognized earlier, in particular by the implementers of Jif [Myers], we are not aware of any proof of soundness for an information flow type system parameterized by other static analyses.

3.2.2.2 Objective 9. Improving information flow type systems (2)

In a similar vein, we plan to improve type systems taking termination leaks into account. In [Boudol] we have shown that one can considerably improve some previous solutions to the static analysis of such leaks. One of these solutions consists in recording the security level of the predicate in a branching construct, so that an assignment in the low part of the memory cannot follow a conditional branching with a high test [Boudol-Castellani]. However, this is still quite restrictive, and we have shown in [Boudol] that in the case where both branches are known to terminate, one can remove this constraint while still ensuring secure information flow. However, we left open the question of determining, by a static analysis technique, an interesting class of terminating programs. Some techniques are available for showing termination of recursive programs over inductive data structures for instance, but very little is known regarding imperative (first-order) programs, and, as we said above, nothing is known about this problem for higher-order imperative programs. Then we plan to apply to the typing of termination leaks in an expressive, ML-like language, the techniques that we will develop for proving that programs are cooperative, as described in Section 2.1.

3.2.2.3 Objective 10. Secure information flow as a safety property

As we said some work has been done [Barthe-D'Argenio-Rezk, Darvas-Hâhnle-Sands] to find logical characterizations of non-interference, thus opening the way to more precise methods than type systems for ensuring this property. However, one may also think that type systems, and especially implicit ones, supporting type inference, are a very useful means to help the programmer to program in a "safe" way, and that it could therefore be interesting to look for stronger properties than non-interference, that would be closer to what is actually ensured by static analysis. To this end, we plan to formalize in an operational way what it means for information to flow, during the execution of a program, from one security level to another. Then the intention is to define secure information flow as a *safety* property, meaning that no security error occurs at run-time, where it is an error to store a value that has been computed using confidential information. We think that we can in this way deal with declassification mechanisms, and perhaps some other ways of managing security, like revoking security policies, in a natural manner. We will also have to see whether such a notion of secure information flow is adequate to deal with various notions of security leaks, and especially those arising in concurrent systems.

3.2.2.4 Objective 11. History-based information flow

A related research objective is to design a history-based notion of information flow, to formalize the dependence of interactions of sub-terms into results of programs. This follows hints given by Abadi and Fournet in [Abadi-Fournet] to enforce safety by inclusion of history into the JVM stack inspection mechanism. A typical example of what one would like to treat is the Chinese

Wall policy where a result may depend upon two individuals A and B, but not upon the interaction between A and B. Another similar example is the password example, where individual A can access to the data of B if and only if A has previously interacted with C. The old theory of the labeled A-calculus [Levy] can be used to track history of computations in a core ML language. In his forthcoming PhD Thesis [Blanc PhD], Blanc shows the expressivity of this approach; for instance, the formal semantics of the Chinese Wall or the password examples can be defined. Moreover an operational description of this policy can be expressed without labels thanks to the Church-Rosser property. By extending the A-calculus with constants and delta rules, it seems feasible to address history-based information flow in non-confluent calculi with side-effects, and to capture at least the standard notion of non-interference or to get as a corollary the static analysis based on types of [Pottier-Simonet], without exceptions. The objectives of this work are, first to get more flexible checks for information flow by mixing run-time information due to history with static information; second, to design a simple logic for reasoning about history of creation of redexes in the A-calculus or of interactions in more complex languages. A longer term goal would be to connect history-based information flow with static analysis on traces as already done in first-order imperative languages.

3.2.2.5 Objective 12. Information flow and cooperative programming

Regarding more specifically secure information flow for concurrent systems, we plan to show that cooperative concurrency is better suited to support precise analysis on information flow than preemptive concurrency (this will extend some work done in the CRISS project of the ACI Sécurité Informatique [Almeida-Boudol-Castellani]). The fact is that, unlike with preemptive scheduling, the points where scheduling decisions are taken during the execution are explicitly located in the code of the threads. Then an obvious idea is to assign security levels to these suspension points, and to analyse the flow of information that may pass through these control points. We think that this should allow for an explicit analysis of confidentiality leaks arising from the scheduling strategy. (By contrast, such an analysis is notoriously difficult with preemptive scheduling.) Another related line of research, based on a model where interactions between the threads and the scheduler are made explicit [Russo-Sabelfeld], is described in Section 2.3.

3.2.2.6 Objective 13. Language-based vs process-calculus-based information flow

Still regarding concurrency and security, we plan to compare the language-based approach to secure information flow with process calculi formalizations of the same problem. The two lines of research have been developed to a large extent independently, and in the current state of research, there is not much work bridging the gap between the two. A recent contribution in this direction was given by Focardi, Rossi and Sabelfeld in [Focardi-Rossi-Sabelfeld]. This paper considers a sequential imperative language equipped with a (time-sensitive) notion of non-interference and, using Milner's encoding of this language into CCS, proves that an external observer cannot infer any secret from (the image of) a secure program. However, a limitation of this work is that the language is purely sequential, although Milner's encoding was originally defined for a parallel language and various definitions of non-interference now exist for multi-threaded languages [Smith-Volpano, Sabelfeld-Sands, Boudol-Castellani]. We also believe that the notion of security adopted for value-passing CCS in this paper as well as in other papers (namely, persistent bisimulation-based nondeducibility on composition), is rather restrictive and not fully in accordance with that of non-interference for programming languages. Taking this

work as our starting point, we propose to develop the comparison between the language-based approach and the calculus-based approach. We shall do this for a concurrent and imperative language [Boudol-Castellani], elaborating a suitable notion of non-interference for the target process calculus. This might involve exploiting the asymmetry between inputs and outputs, which are mostly treated on an equal footing in existing studies. Then we plan to show that the translation is correct with respect to non-interference, by relating type systems for secure information flow in the source and target models.

3.2.3 Access and resource control

Access control is a well-established technology of operating systems, which however has been renewed with the advent of global computing systems. In this project we shall focus on specific issues regarding access control in new applications, as the ones previously mentioned. As usual, the problem is to control access to resources (confidential data, computing time, external devices. ..), in such a way that only code that is authorized to perform a certain operation can do so. The programmer is faced with the task of enforcing such security requirements by combining a number of programming language and operating system features such as strong typing, scope reduction of variables, sand-boxing, run-time checks on the state of execution, etc. The options are diverse and it may be difficult to estimate the consequences of a particular choice. The JAVA language for instance proposes a variety of mechanisms for controlling the access to resources.
One of our aims is to develop models for these mechanisms. The well-known stack inspection mechanism for instance is intended to support access control for applications in which code components from different protection domains have to cooperate. With this mechanism, one can enable a component to obtain information about the code that (directly or indirectly) invokes the component's methods, by letting it inspect the call stack of the run-time environment. Based on this information, the component can decide whether or not the callers have the right to access a given resource. This mechanism has been formally studied [Bartoletti-Degano-Ferrari, Besson-Jensen-Le Métayer, Jensen-Le Métayer-Thorn, Fournet-Gordon], but only limited formal results regarding the properties guaranteed by stack inspection have been obtained. Fournet and Gordon [Fournet-Gordon] for instance proved a security theorem that demonstrates that stack inspection only guarantees a weak safety property. For interactive devices such as mobile telephones, JAVA proposes a security architecture [SUN] which uses interactive querying of the user to grant on-the-fly permissions to the applet executing on a mobile phone so that it can make Internet connections, access files, send SMSs, etc. An important feature of this MIDP (Mobile Interactive Device Profile) model is the "one-shot" permissions that can be used only once for accessing a resource. This quantitative aspect of permissions raises several questions of how such permissions should be modeled (e.g., "do they accumulate?" or "which one to choose if several permissions apply?"), and how to program with such permissions in a way that respects both usability and security principles such as Least Privilege.

Besides access control, we are also interested in the *availability* aspect of security, and more precisely in means to control the use of resources by the various components. Mastering the *computational complexity* of programs is an important aspect of computer security in modern applications. For instance, in smartcards applications, and more generally in embedded software, one has to deal with bounded resources, and it may be critical to control that resource usage stays within certain bounds. Another example is mobile code. In this case, the hosting system providing computing resources needs guarantees on the time and space that incoming code might use. One approach to this problem is to monitor the resource consumption at run-time and to

raise an exception when some bound is reached. A variant of this approach is to instrument the code so that bounds are checked at a given time, but obviously, although they offer a great flexibility and precision, such run-time checks are not always appropriate (think for instance of a critical embedded system that would crash because its execution is aborted by a resource monitor). An alternative approach is to statically analyze the program to guarantee that during the execution it will respect certain resource bounds. The advantages of static analysis are that it does not introduce any overhead at run time and, perhaps more importantly, that it allows an early detection of buggy or malicious programs.

In this project we will focus on static analyses, which offer more challenging problems, while keeping in mind that the two approaches are complementary. (For instance, static analyses may be helpful in reducing the frequency of dynamic verifications.) When addressing the issue of resource control, there is a variety of properties of a program that one may check. Termination is probably the first one that comes to mind. However, in the context of interacting programs, this property should be refined into the one of "reactivity" (similar to the "productivity" property in [Hughes-Pareto-Sabry]). If a program manipulates data values of variable size such as lists, trees, or graphs, then the analysis can go beyond reactivity and, for instance, it can establish that the program "reacts" while using a *feasible* amount of resources. Our starting point in this respect is the work on the automatic extraction of resource bounds for (first-order) functional languages. One can trace back the origin of this line of work to Cobham's characterization [Cobham] of polynomial time functions by bounded recursion on notation. Later work (e.g. [Bellantoni-Cook, Hofmann, Jones, Leivant]) has developed various inference techniques that allow for efficient analyses while capturing a large range of practical algorithms. In [Bonfante-Marion-Moyen, Marion] it has been shown that polynomial time or space bounds can be obtained by combining traditional termination techniques for term rewriting systems with an analysis of the size of computed values based on the notion of quasi-interpretation. Thus, in a nutshell, resource control relies on termination and bounds on data size. A challenge here is to design static analysis techniques that would be practicable, in the sense of being feasible and not rejecting too many programs, and precise in the sense of providing as exact bounds as possible.

3.2.3.1 Objective 14. Dynamic permission management

A specific objective we have regarding access control is to develop a formal model of the above-mentioned protection mechanisms offered by the JAVA language in a fully concurrent setting -the references cited above are all limited to the sequential subset of this language. One concrete result of the success of this effort would be the development of a semantically well-founded general model that could provide an alternative for the JAVA MIDP security model. We propose to define the semantics of the model's programming constructs, and a logic for reasoning about the flow of permissions in programs using these constructs. This logic will notably allow us to prove the basic security property that a program will never attempt to access a resource for which it does not have permission. Initial steps towards this goal will appear at this year's ESORICS conference [Besson-Dufay-Jensen].

3.2.3.2 Objective 15. Resource consumption analysis

Controlling the way in which a program accesses and consumes resources usually requires additional information about the program's flow of data and control. For example, the certified memory usage analysis for JAVA Card developed by the Lande group [Cachera-Jensen-

Pichardie] requires a precise model of the program's intra- and inter-procedural control flow, and can be further enhanced by being coupled with relational data flow analyses such as the classical polyhedral analysis of relations between integer variables. We will be concerned with extending existing resource analyses to a concurrent setting and with examining their interaction with other static analyses such as those used for analyzing race conditions in JAVA programs [Abadi-Flanagan-Freund], and the resource bound analyses developed in the next objective.

3.2.3.3 Objective 16. Feasible reactivity

Regarding "feasible reactivity", we have previously addressed this issue in the context of the ACI CRISS project [Amadio-Dal Zilio]. In this work, we have been extending and adapting the above mentioned results regarding implicit computational complexity to a concurrent framework of synchronous parallel threads interacting on shared variables. Our analysis goes in three main steps. A first step is to guarantee that each instant terminates. A second step is to bound the size of the computed values as a function of the size of the parameters at the beginning of the instant. A third step is to combine the termination and size analyses. In particular, we show how to obtain polynomial bounds on the space and time needed for the execution of the system during an instant as a function of the size of the parameters at the beginning of the instant. A characteristic of the analyses is that to a great extent they make abstraction of the memory and the scheduler. This means that each thread can be analyzed separately, that the complexity of the analyses grows linearly in the number of threads, and that an incremental analysis of a dynamically changing system of threads is possible. Ongoing [Amadio-Dabrowski] and planned work aims at extending the analysis to a more general language based on a synchronous version of the 7r-calculus *(cf.* Section 2.1), at designing reasonable conditions that allow to predict the size of the system after arbitrarily many instants, and finally at experimenting the static analysis conditions on a certain number of synchronous applications. In a different direction, we plan to explore the connections between reactivity in a synchronous and an asynchronous framework. Proving reactivity in a synchronous framework appears to be easier because the programs are naturally structured into instants. Then what we need to prove is simply that each instant terminates. By contrast, termination of an asynchronous concurrent program is a less structured task and may involve proving a complicated invariant on the global structure of the system.

3.3 Machine-checking security properties

Enforcing and verifying properties of programs has always been a major concern for computer science. We have mentioned the use of program logics and static analyses, together with their correctness proofs, for this purpose. As these methods gradually come closer to real applications, the need for a machine-checked formalization, of either tool correctness or specific program verifications, is emerging. A first motivation is to gain increased assurance in the results. Indeed, large scale proofs are generally intrinsically complicated, from a methodological or combinatorial point of view, and therefore their development is error-prone. Moreover, in some applications one would like to run only programs that have been proved correct according to some criterion, but one should not have to redo the proof of correctness, which should come as a certificate attached to the code, that can be automatically checked. This is the paradigm of *proof carrying code*, which relies on the development and use of automatic or interactive theorem provers. In this section we describe our research objectives that pertain to this line of work, on machine-checked security or reliability properties.

3.3.1 Objective 17. A mechanized framework for concurrent separation logic

At the time of writing, an ongoing research project[2] is building a mechanized framework to prove the correctness of sequential programs written in a large subset of C using separation logics. Focusing on a realistic programming language raises new challenges, related to the presence of side-effects in expressions and functions, and to the complexity of the semantics. At the same time this makes it possible to prove properties of real programs. The development is done inside the COQ theorem prover. We propose to extend this framework to prove properties of concurrent programs using a concurrent extension of separation logic. We will focus on the Posix thread model, shared memory, and traditional locking mechanisms and we will put our framework at work by proving the correctness of some well-known concurrent algorithms (possibly including some lockfree algorithms). As proofs done in concurrent separation logic seem to fit well to cooperative and/or synchronous interaction models, we plan, in a second phase, to study how to extend this work to other multithreaded programming patterns developed in the PARSEC project.

3.3.2 Objective 18. Enforcement of information flow policies

We plan to pursue the work of [Barthe-Rezk], machine-check it in COQ, and improve it in several respects. As indicated above, a motivation for formalizing and proving the results of this work in COQ lies in the fact that the type system of [Barthe-Rezk] is intricate, because stack manipulation can lead to illegal flows, and because its definition relies on a control dependence region (cdr) analysis that computes the scope of branching statements. In addition, the soundness proof involves some lengthy and error-prone proofs by case analysis, as well as some unusual induction principle on the execution of programs. It is therefore important to resort to proof assistants for managing the complexity of the definitions and proofs involved in establishing non-interference for this language. We also aim at generalizing the results we obtained in [Barthe-Rezk], dealing with more expressive policies and more realistic languages. We intend to allow arbitrary lattices of confidentiality levels, and, more critically, to follow more closely the JVM in its treatment of exceptions. (A related improvement, regarding the type system, has been described in Section 2.2.) Further, we plan to use our formalization as a starting point for an investigation of information-flow policies for the JVM. In particular, we would like to extend our formalization in two ways. First, we wish to prove the soundness of an information flow type system that enforces the non-disclosure policy (NDP) [Almeida-Boudol], a bisimulation-based policy that deals with declassification. The first step will be to select a notion of control dependence region that enforces termination-sensitive non-interference, and use progress lemmas to move from input-output security policies to bisimulation-based policies. Once this is done, introducing constructs for local flows and adapting the type system to enforce NDP should be rather direct. A second line of investigations will be concerned with dealing with multi-threading along the lines of [Russo-Sabelfeld]. The semantics of programs is extended to multi-threading in a non standard fashion, namely by introducing explicitly a scheduling function that picks the new thread to be executed, and by using instructions that swap threads between pools of threads, the idea being to have a pool of threads per security level. The security policy is expressed in terms of input/output behavior of programs, and the soundness of the type system is shown using simulation lemmas akin to those of [Barthe-Rezk]. Then, we intend to study the impact of synchronization on information flow. A much more ambitious goal would be to use the

[2] involving Francesco Zappa Nardelli (MOSCOVA, INRIA Rocquencourt), Andrew Appel (Princeton U., US), Sandrine Blazy (GALLIUM, INRIA Rocquencourt) and Matthew Parkinson (Cambridge U., UK).

formalization as a basis to machine-check the results of [Barthe-Naumann-Rezk].

3.3.3 Objective 19. Type-preserving compilation

JFlow is an extension of JAVA with a flexible and expressive information flow type system [Myers]. The flexibility and expressiveness of information flow policies supported by JFlow has been exploited for modeling and analyzing the policies that underlie battleship and mental poker games [Aslarov-Sabelfeld]. Despite the richness of its language, the information-flow policies of JFlow can be enforced automatically by a constraint-based algorithm that rejects programs that may violate their policy. However, the flexibility of the type system, especially with respect to de-classification, makes it difficult to characterize formally the security properties that are verified by typable programs. Thus, subsequent work [Banerjee-Naumann 02, Banerjee-Naumann 05] has focused on developing for a fragment of JAVA that includes objects, inheritance, and methods, an information-flow type system that enforces non-interference. The benefits provided by JFlow (and [Banerjee-Naumann 05]) do not directly address mobile code security since they apply to source code, whereas JAVA applets are deployed in compiled form as JVM bytecode programs. Thus, it is desirable to develop information flow type systems at bytecode level. An extended bytecode verifier is provided by [Barthe-Rezk]; the type system guarantees secure information flow for a fragment of JAVA bytecode that includes objects, inheritance, methods, and (a simplified mechanism for) exceptions. The type systems of JFlow and of [Barthe-Rezk] have been developed and applied in isolation: JFlow offers a practical tool for developing secure applications, and in particular for ensuring to developers that applications meet high-level policies about API usage. In contrast, the type system of [Barthe-Rezk] augments the JAVA security architecture to provide assurance to users that applets respect high-level policies about API usage. These two lines of work have recently been connected via a type preservation result [Barthe-Naumann-Rezk], showing that programs typable in a suitable fragment of JFlow are compiled into bytecode programs that pass information-flow bytecode verification. The interest of such a result is to show on the one hand that applications written with JFlow can be deployed in a mobile code architecture that fulfills the promises of JFlow in terms of confidentiality, and on the other hand that the enhanced security architecture from [Barthe-Rezk] can benefit from practical tools for developing applications that meets the policy it enforces. The results of [Barthe-Naumann-Rezk] cover a fragment of the JVM with a simplified treatment of exceptions, no method calls and no multi-threading. Then we propose to extend our results in [Barthe-Naumann-Rezk, Barthe-Rezk] to a more complete treatment of the JVM.

3.4 Deliverables

Delivrables				
	Title	*Type*	*Teams*	*Date*
l	PARSEC web site	Web site	All	T0+6
2	Design of cooperative and reactive programming models (objectives 2, 3, 4)	Technical papers	MIMOSA, PPS	T0+12
3	Logics for reliability and security of concurrent systems (objectives 5, 6, 7, 14)	Technical papers	EVEREST, LANDE, MOSCOVA	T0+12
4	Improved static analysis for secure information flow (objectives 8, 9)	Technical papers	EVEREST, MIMOSA	T0+12
5	Notions of secure information flow (objectives 10, 11)	Technical papers	MIMOSA, MOSCOVA	T0+12
6	Mechanizing the verification of security properties (objectives 1, 17, 18, 19)	Technical papers	EVEREST, LANDE MOSCOVA	T0+18
7	Secure information flow for cooperative and reactive systems (objectives 12, 13)	Technical papers	MIMOSA, PPS	T0+24
8	Analysis for resource control (objectives 15, 16)	Technical papers	LANDE, PPS	TO+24

3.5 Intended results

We recall that the main aim of the project is to achieve a good understanding of what could be a *safe and secure concurrent programming model,* based on theoretical justifications. We plan to make contributions regarding two of the three classical aspects of security, namely *confidentiality* (access control, secure information flow) and *availability* (resource usage control), following a language-based approach, and addressing specifically security issues arising in concurrent systerns. Although we expect our results to be essentially of a theoretical character, our motivation is *to provide the basis for tools that are needed to program in a safe and secure way massively concurrent applications that are open to untrusted parties.* These tools will be *program logics,* possibly supported by proof assistants, and providing support for verifiable software security, *static analysis techniques* to assist the programmer in avoiding security errors and preventing execution of insecure programs, and *protection mechanisms* with

a well-understood semantics to achieve run-time checks whenever a sufficiently complete analysis is not available.

Our more specific objectives are summarized in the table above. Since the list of the deliverables of the project is supposed to be established each year for the next phase of the project, we only give this list for the first year and, tentatively, for the next one, indicating the relevant project's objectives from Section 2. Among the final results we expect to achieve is a PhD thesis for which we are requiring financial support. The thesis will focus on Task 1 (concurrent programming models) and Task 2.2.2 (secure information flow) taking as programming model the synchronous calculus which is currently under development. Concerning Task 1, it will explore in particular the issue of determinacy both in the synchronous and asynchronous context. Concerning Task 2.2.2, it will develop suitable notions of information flow for the considered model and relate them to the standard notions developed in classical approaches based on imperative languages. We are planning to recruit a "Master Recherche" student in the Fall 2006 and start the thesis work in the Fall 2007. In this way, the thesis will benefit from the exploratory work and the collaborations that will be carried on during the first year of the project. The thesis work will be directed by R. Amadio (PPS).

4 Références

[Abadi-Flanagan-Freund] M. ABADI, C.FLANAGAN, S.N. FREUND, *Types for safe locking: static race detection for Java,* ACM TOPLAS Vol. 28 No. 2 (2006) 207-255.

[Abadi-Fournet] M. ABADI, C. FOURNET, *Access control based on execution history,* NDSS'03 (2003) 107-121.

[Abraham PhD] E. ABRAHAM, *An Assertional Proof System for Multithreaded Java - Theory and Tool Support,* PhD Thesis, University of Leiden (2004).

[Abraham-de Roever-Steffen] E. ABRAHAM, W.-P. DE ROEVER, M. STEFFEN, *Tool-supported proof system for multithreaded Java,* MFCO'02, Lecture Notes in Comput. Sci. 2852 (2003) 1-32.

[Adve-Gharachorloo] S.A. ADVE, K. GHARACHORLOO, *Shared memory consistency models: a tutorial,* IEEE Computer Vol. 29 No. 12 (1996) 66-76.

[Adve-Manson-Pugh] S.A. ADVE, J. MANSON, W. PUGH, *The Java memory model,* POPL'05 (2005) 378-391.

[Adya & al] A. ADYA, J. HOWELL, M. THEIMER, W. J. BOLOSKY, H.R. DOUCEUR, *Cooperative task management without manual stack management or, Event-driven programming is not the opposite of threaded programming,* Usenix ATC (2002).

[Almeida-Boudol] A. ALMEIDA MATOS, G. BOUDOL, *On declassiûcation and the non-disclosure policy,* CSFW'05 (2005) 226-240.

[Almeida-Boudol-Castellani] A.ALMEIDA MATOS, G. BOUDOL, I. CASTELLANI, *Typing noninterference for reactive programs,* accepted for publication in the J. of Logic and Algebraic

Programming (2006).

[Amadio 05a] R. AMADIO, *Synthesis of max-plus quasi-interpretations,* Fundamenta Informaticae Vol. 65 No. 1-2 (2005) 29-60.

[Amadio 05b] R. AMADIO, *The SL synchronous language, revisited,* Technical report PPS, November 2005. To appear in the J. of Logic and Algebraic Programming (2005).

[Amadio & al] R. AMADIO, G. BOUDOL, F. BOUSSINOT, I. CASTELLANI, *Reactive programming revisited,* Proc. Workshop on *Algebraic Process Calculi: the first* 25 *years and beyond,* Bertinoro. To appear in ENTCS (2005).

[Amadio-Dabrowski] R. AMADIO, F. DABROWSKI, *Feasible reactivity for synchronous cooperative threads,* EXPRESS'05, to appear in ENTCS (2005).

[Amadio-Dal Zilio] R. AMADIO, S. DAL-ZILIO, *Resource control for synchronous cooperative threads,* CONCUR'04, Lecture Notes in Comput. Sci. 3170 (2004) 68-82.

[Anderson & al] T.E. ANDERSON, B.N. BERSHAD, E.D. LAZOWSKA, H.M. LEVY, *Scheduler activations: effective kernel support for the user-level management of parallelism,* ACM Trans, on Computer Systems Vol. 10 No. 1 (1992) 53-79.

[Aslarov-Sabelfeld] A. ASKAROV, A. SABELFELD, *Security-typed languages for implementation of cryptographic protocols: A case study,* ESORICS'05, Lecture Notes in Comput. Sci. 3679 (2005) 197-221.

[Banerjee-Naumann 02] A. BANERJEE, D. A. NAUMANN, *Secure information flow and pointer confinement in a Java-like language,* CSFW'02 (2002).

[Banerjee-Naumann 05] A. BANERJEE, D. A. NAUMANN, *Stack-based access control for secure information flow,* JFP Vol. 15, special issue on Language-Based Security (2005) 131-177.

[Banga-Druschel-Mogul] G. BANGA, P. DRUSCHEL, J.C.MOGUL, *Better operating system features for faster network servers,* ACM SIGMETRICS Performance Evaluation Review Vol. 26 No. 3 (1998) 23-30.

[Barthe-D'Argenio-Rezk] G. BARTHE, P. D'ARGENIO, T. REZK, *Secure information flow by self-composition,* CSFW'04 (2004).

[Barthe-Naumann-Rezk] G. BARTHE, D. NAUMANN, T. REZK, *Deriving an information flow checker and certifying compiler for Java,* Symposium on Security and Privacy (2006).

[Barthe-Rezk] G. BARTHE, T. REZK, *Secure information flow for a JVM-like language,* TLDI'05 (2005) 103-112.

[Bartoletti-Degano-Ferrari] M. BARTOLETTI, P. DEGANO, G.L.FERRARI, *Static analysis for stack inspection,* ENTCS Vol. 54 (2001).

[von Behren-Condit-Brewer] R. VON BERHEN, J. CONDIT, E. BREWER, *Why events are a bad idea (for high-concurrency servers),* Proceedings of HotOS IX (2003).

[von Behren & al] R. VON BERHEN, J. CONDIT, F. ZHOU, G. C. NECULA, E. BREWER, *Capriccio: scalable threads for Internet services,* SOSP'03 (2003).

[Bellantoni-Cook] S. BELLANTONI, S. COOK, *A new recursion-theoretic characterization of the poly-time functions,* Computational Complexity 2 (1992) 97-110.

[Benveniste & al] A. BENVENISTE, P. CASPI, S.A. EDWARDS, N. HALBWACHS, P. Le GUERNIC, R. de SIMONE, *The synchronous languages twelve years later,* Proc. of the IEEE, Special Issue on the Modeling and Design of Embedded Software, Vol. 91 No. 1 (2003) 64-83.

[Besson-Dufay-Jensen] F. BESSON, G. DUFAY, T. JENSEN, *A formal model of access control for mobile interactive devices,* ESORICS'06 (2006).

[Besson-Jensen-Le Métayer] F. BESSON, T. JENSEN, D. LE MÉTAYER, *Model checking security properties of control flow graphs,* J. of Computer Security Vol. 9 (2001) 217-250.

[Birrel] A.D. BIRREL, *An introduction to programming with threads,* SRC Report 35, DEC (1989).

[Blanc PhD] T. BLANC, *Politiques de sécurité dans le lambda calcul,* Thèse, École Polytechnique (2006).

[Boehm] H.-J. BOEHM, *Threads cannot be implemented as a library,* PLDI'05 (2005) 261-268.

[Bonfante-Marion-Moyen] G. BONFANTE, J.-Y.MARION, J.-Y. MOYEN, *On termination methods with space bound certifications,* Perspectives of System Informatics, Lecture Notes in Comput. Sci. 2244 (2001).

[Boudol] G. BOUDOL, *On typing information Bow,* Intern. Coll. on Theoretical Aspects of Computing, Lecture Notes in Comput. Sci. 3722 (2005) 366-380.

[Boudol-Castellani] G. BOUDOL, I. CASTELLANI, *Non-interference for concurrent programs and thread systems,* in "Merci, Maurice, A mosaic in honor of Maurice Nivat" (P.-L. Curien, Ed), Theoretical Comput. Sci. Vol. 281, No. 1 (2002) 109-130.

[Boussinot 91] F. BOUSSINOT, *Reactive-C: an extension of C to program reactive systems,* Software Practice and Experience Vol. 21, No. 4 (1991) 401-428.

[Boussinot 06] F. BOUSSINOT, *FairThreads: mixing cooperative and preemptive threads in C,* Concurrency and Computation: Practice and Experience, Vol. 18 (2006) 445-469.

[Boussinot-de Simone] F. BOUSSINOT, R de SIMONE, *The SL synchronous language,* IEEE Trans, on Software Engineering Vol. 22, No. 4 (1996) 256-266.

[Boussinot-Susini] F. BOUSSINOT, J.-F. SUSINI, *The SugarCubes tool box: a reactive* JAVA

framework, Software Practice and Experience, Vol. 28, No. 14 (1998) 1531-1550.

[Cachera-Jensen-Pichardie] D. CACHERA, T. JENSEN, D. PICHARDIE, *Certified memory usage analysis,* FM'05, Lecture Notes in Comput. Sci. 3582 (2005) 91-106.

[Cobham] A. COBHAM, *The intrinsic computational difficulty of functions,* Proceedings Logic, Methodology, and Philosophy of Science II, North Holland (1965).

[Crafa-Rossi] S. CRAFA, S. ROSSI, *A theory of noninterference for the ir-calculus,* TCG'05, Lecture Notes in Comput. Sci. 3705 (2005).

[Darvas-Hâhnle-Sands] A. DARVAS, R. HÂHNLE, D. SANDS, *A theorem proving approach to analysis of secure information flow,* WITS'03 (2003).

[Dufay-Felty-Matwin] G. DUFAY, A. FELTY, S. MATWIN, *Privacy-sensitive information flow with JML,* CADE'05, Lecture Notes in Comput. Sci. 3632 (2005) 116-130.

[Engelschall] R.S. ENGELSCHALL, *Portable multithreading, the signal stack trick for user-space thread creation,* Usenix ATC (2000).

[Flanagan-Qadeer] C. FLANAGAN, S. QADEER, *Types for atomicity,* TLDI'03 (2003) 1-12.

[Focardi-Gorrieri] R FOCARDI, R GORRIERI, *A classification of security properties for process algebras,* J. of Computer Security, Vol. 3 No. 1 (1995) 5-33.

[Focardi-Rossi-Sabelfeld] R. FOCARDI, S.ROSSI, A. SABELFELD, *Bridging language-based and process calculi security,* FOSSACS'05, Lecture Notes in Comput. Sci. 3441 (2005).

[Fournet-Gordon] C. FOURNET, A. GORDON, *Stack inspection: theory and variants,* POPL'02 (2002).

[Grossman] D. GROSSMAN, *Type-safe multithreading in Cyclone,* TLDI'03 (2003) 13-25.

[O'Hearn] P. O'HEARN, *Resources, concurrency and local reasoning,* CONCUR'04, Lecture Notes in Comput. Sci. 3170 (2004) 49-67.

[Hennessy-Riely] M. HENNESSY, J. RIELY, *Information flow vs resource access in the asynchronous 7i-calculus,* ACM TOPLAS Vol. 24 No. 5 (2002) 566-591.

[Hofmann] M. HOFMANN, *The strength of non size-increasing computation,* POPL'02 (2002).

[Honda-Yoshida 93] K. HONDA, N. YOSHIDA, *On reduction-based process semantics,* FST-TCS'93, Lecture Notes in Comput. Sci. 761 (1993) 371-387.

[Honda-Yoshida 02] K. HONDA, N. YOSHIDA, *A uniform type structure for secure information flow,* POPL'02 (2002) 81-92.

[Hughes-Pareto-Sabry] J. HUGHES, L. PARETO, A. SABRY, *Proving the correctness of reactive systems using sized types,* POPL'96 (1996) 410-423.

[Huisman PhD] M. HUISMAN, *Reasoning about Java programs in higher order logic using PVS and Isabelle,* PhD Thesis, Computing Science Institute, University of Nijmegen (2001).

[Huisman-Worah-Sunesen] M. HUISMAN, P. Wo RAH, K. SUNESEN, *A temporal logic characterisation of observational determinism,* to appear in CSFW'06 (2006).

[Jacobs-Poll] B.JACOBS, E.POLL, *A logic for the Java Modeling Language JML,* FASE'01, Lecture Notes in Comput. Sci. 2029 (2001) 284-299.

[Jensen-Le Métayer-Thorn] T. JENSEN, D. LE MÉTAYER, T. THORN, *Verification of control flow based security properties,* Proceedings of the 20th IEEE Symp. on Security and Privacy (1999) 89-103.

[Jones] N. JONES, *Computability and complexity, from a programming perspective,* MIT Press (1997).

[Leivant] D. LEIVANT, *Predicative recurrence and computational complexity i: word recurrence and poly-time,* Feasible Mathematics II, Birkhâuser (1994) 320-343.

[Levy] J.-J. LEVY, *Réductions correctes et optimales dans le lambda-calcul,* Thèse, Université Paris 7 (1978).

[Mandel-Pouzet] L. MANDEL, M. POUZET, *ReactiveML, a reactive extension to ML,* PPDP'05 (2005) 82-93.

[Marion] J.-Y. MARION, *Complexité implicite des calculs, de la théorie a la pratique,* Habilitation à diriger des recherches, Université de Nancy (2000).

[Myers] A. MYERS, *JFlow: practical mostly-static information flow control,* POPL'99 (1999).

[Naumann] D. A. NAUMANN, *From coupling relations to mated invariants for checking information Bow,* ESORICS'06 (2006).

[von Oheimb] D. VON OHEIMB, *Analyzing Java in Isabelle/HOL: formalization, type safety and Hoare Logic,* PhD Thesis, Technische Universitât Miinchen (2001).

[Ousterhout] J. OUSTERHOUT, *Why threads are a bad idea (for most purposes),* presentation given at the 1996 Usenix ATC (1996).

[Pottier] F. POTTIER, *A simple view of type-secure information flow in the ir-calculus,* CSFW'02 (2002) 320-330.

[Pottier-Simonet] F. POTTIER, V. SIMONET, *Information flow inference for ML,* ACM TOPLAS Vol. 25 No. 1 (2003) 117-158.

[Pugh] W. PUGH, *The Java memory model is fatally flawed,* Concurrency Practice and Experience, Vol. 12 No. 1 (2000) 1-11.

[Reynolds 02] J.C.REYNOLDS, *Separation logic: a logic for shared mutable data structures,*

LICS'02 (2002) 55-74.

[Reynolds 04] J.C.REYNOLDS, *Toward a grainless semantics for shared-variable concurrency,* FST-TCS'04, Lecture Notes in Comput. Sci. 3328 (2004) 35-48.

[Russo-Sabelfeld] A. Russo, A. SABELFELD, *Securing interaction between threads and the scheduler,* CSFW'06 (2006).

[Ryan & al] P. RYAN, J. MCLEAN, J. MILLEN, V. GLIGOR, *Non-interference, who needs it?,* CSFW'01 (2001).

[Sabelfeld-Myers] A. SABELFELD, A.C.MYERS, *Language-based information-flow security,* IEEE J. on Selected Areas in Communications Vol. 21 No. 1 (2003) 5-19.

[Sabelfeld-Sands] A. SABELFELD, D. SANDS, *Probabilistic noninterference for multi-threaded programs,* CSFW'OO (2000).

[Serrano-Boussinot-Serpette] M. SERRANO, F. BOUSSINOT, B. SERPETTE, *Scheme fair threads,* PPDP'04 (2004) 203-214.

[Smith-Volpano] G. SMITH, D. VOLPANO, *Secure information flow in a multi-threaded imperative language,* POPL'98 (1998).

[SUN] SUN MICROSYSTEMS, INC., *MIDP: Mobile Information Device Profile,* Specification for Java 2 Micro Edition, Version 2.0, Palo Alto/CA, USA (2002).

[Terauchi-Aiken] T. TERAUCHI, A. AIKEN, *Secure information Bow as a safety problem,* SAS'05 (2005).

[Varacca-Yoshida] D. VARACCA, N. YOSHIDA, *Typed event structures and the ir-calculus,* MFPS'06, to appear (2006).

[Volpano-Smith-Irvine] D. VOLPANO, G. SMITH, C. IRVINE, *A sound type system for secure flow analysis,* J. of Computer Security, Vol. 4 No 3 (1996) 167-187.

4.1 Bibliographical references of the researchers involved in the project

R. AMADIO, *Synthesis of max-plus quasi-interpretations,* Fundamenta Informaticae Vol. 65 No. 1-2 (2005) 29-60.

R. AMADIO, *The SL synchronous language, revisited,* Technical report PPS, November 2005. To appear in the J. of Logic and Algebraic Programming (2005).

R AMADIO, S. DAL-ZILIO, *Resource control for synchronous cooperative threads,* CON-CUR'04, Lecture Notes in Comput. Sci. 3170 (2004) 68-82.

G. BARTHE, L. PRENSA NIETO, *Formally verifying information flow type systems for concurrent and thread systems,* FMSE'04 (2004).

G. BARTHE, T. REZK, *Secure information flow for a JVM-like language,* TLDI'05 (2005).

G. BARTHE, T. REZK, A. SAABAS, *Proof obligations preserving compilation,* FAST'05 (2006).

G. BARTHE, T. REZK, A. BASU, *Security types preserving compilation,* International Journal of Computer Languages, Systems and Structures (2005).

G. BARTHE, P. D'ARGENIO, T. REZK, *Secure information flow by self-composition,* CSFW'04 (2004).

F. BESSON, T. JENSEN, D. LE MÉTAYER, T. THORN, *Model ckecking security properties of control flow graphs,* J. of Computer Security Vol. 9 (2001) 217-250.

F. BESSON, T. BLANC, C. FOURNET, A. GORDON, *From stack inspection to access control: a security analysis for libraries,* CSFW'04 (2004).

F. BESSON, T. DE GRENIER DE LATOUR, T. JENSEN, *Interfaces for stack inspection,* J. of Functional Programming Vol. 15 No. 2 (2005) 179-217.

F. BESSON, G. DUFAY, T. JENSEN, *A formal model of access control for mobile interactive devices,* ESORICS'06 (2006).

A. ALMEIDA MATOS, G. BOUDOL, *On declassiBcation and the non-disclosure policy,* CSFW'05 (2005) 226-240.

G. BOUDOL, ULM, *a core programming model for global computing,* ESOP'04, Lecture Notes in Comput. Sci. 2986 (2004) 234-248.

G. BOUDOL, *On typing information Bow,* Intern. Coll. on Theoretical Aspects of Computing, Lecture Notes in Comput. Sci. 3722 (2005) 366-380.

G. BOUDOL, I. CASTELLANI, *Non-interference for concurrent programs and thread systems,* in "Merci, Maurice, A mosaic in honour of Maurice Nivat" (P.-L. Curien, Ed), Theoretical Comput. Sci. Vol. 281, No. 1 (2002) 109-130.

A. ALMEIDA MATOS, G. BOUDOL I. CASTELLANI, *Typing noninterference for reactive programs,* accepted for publication in the J. of Logic and Algebraic Programming (2006).

R. AMADIO, G. BOUDOL, F. BOUSSINOT, I. CASTELLANI, *Reactive programming revisited,* Proc. Workshop on *Algebraic Process Calculi: the first* 25 *years and beyond,* Bertinoro. To appear in ENTCS (2005).

G. BOUDOL, I. CASTELLANI, *Non-interference for concurrent programs and thread systems,* in "Merci, Maurice, A mosaic in honour of Maurice Nivat" (P.-L. Curien, Ed), Theoretical Comput. Sci. Vol. 281, No. 1 (2002) 109-130.

M. HUISMAN, P. WORAH, K. SUNESEN, *A temporal logic characterisation of observational*

determinism, CSFW'06 (2006).

M.PAVLOVA, G. BARTHE, L. BURDY, M. HUISMAN, J.-L. LANET, *Enforcing high-level security properties for applets,* CARDIS'04 (2004).

M. HUISMAN, D. GUROV, C. SPRENGER, G. CHUGUNOV, *Checking absence of illicit applet interactions: a case study,* FASE'04, Lecture Notes in Comput. Sci. 2984 (2004) 84-98.

F. BESSON, T. JENSEN, D. LE MÉTAYER, T. THORN, *Model checking security properties of control Bow graphs,* J. of Computer Security Vol. 9 (2001) 217-250.

F. BESSON, T. DE GRENIER DE LATOUR, T. JENSEN, *Interfaces for stack inspection,* J. of Functional Programming Vol. 15 No. 2 (2005) 179-217.

F. BESSON, G. DUFAY, T. JENSEN, *A formal model of access control for mobile interactive devices,* ESORICS'06 (2006).

M. ABADI, B.W. LAMPSON, J.-J. LEVY, *Analysis and caching of dependencies,* ICFP'96 (1996) 83-91.

J.-J. LEVY, *Réductions correctes et optimales dans le lambda-calcul,* Thèse, Université Paris 7 (1978).

G. CASTAGNA, R. DE NICOLA, D. VARACCA, *Semantic subtyping for the IT-calculus,* LICS'05 (2005).

G. CASTAGNA, M. DEZANI, D. VARACCA, *Encoding CDuce in the Cir-calculus,* CONCUR'06, to appear (2006).

D. VARACCA, N. YOSHIDA, *Typed event structures and the ir-calculus,* MFPS'06, to appear (2006).

M. MERRO, F. ZAPPA NARDELLI, *Behavioural theory for Mobile Ambients,* Journal of ACM Vol. 52 No. 6 (2005) 961-1023.

P. SEWELL, J. J. LEIFER, F. ZAPPA NARDELLI, M. ALLEN-WILLIAMS, P. HABOUZIT, V. VAFEIADIS, *Acute: High-level programming language design for distributed computation,* ICFP'05 (2005).