RESEARCH CENTER

FIELD
Algorithmics, Programming, Software and Architecture

Activity Report 2012

Section Scientific Foundations

Edition: 2013-04-24

ARIC Team

3. Scientific Foundations

3.1. Applications

Whether its purpose is to design better operators or to make the best use of existing ones, computer arithmetic is strongly connected to applications. Some application domains are particularly in demand for high-quality arithmetic: high-performance computing (HPC) for floating-point, accounting for decimal, digital signal processing (DSP) for fixed-point, embedded systems for application-specific operators, cryptography for finite fields. Each domain comes with its specific constraints and quality metric. For example, cryptography has a specific need of resistance to attacks that impact the design of the operators themselves: a good operator for cryptography should have electromagnetic emissions and power consumption patterns independent of the data it manipulates. Another example is very large-scale HPC, which in some cases is reaching the limits of the accuracy provided by the prevalent double-precision floating-point arithmetic.

The regional (Rhône-Alpes) context is especially strong in embedded systems, with the Minalogic Competitivity Centre, major players such as STMicroelectronics, CEA and Inria, and strong startups such as Kalray. This is also true at the European level, with the HiPEAC European network of excellence. This network addresses hardware issues, but also software and compiler issues.

Indeed, the bridge between the application and the underlying hardware arithmetic is usually the compiler. Therefore, more and more arithmetic expertise should be integrated within the compiler. This goes on par with the current trend to automate arithmetic core generation. In the long term, working at the compiler level opens optimization perspective beyond what compilers traditionally perform, for instance ad-hoc generation and optimization in context of application-specific functional cores.

However, much of computer arithmetic research still focuses on the implementation of standard computing cores (such as elementary functions, linear algebra operators, or DSP filters), although this implementation is more and more automated as illustrated by projects such as ATLAS, Spiral, FFTW, and others.

Cryptography is an active field of research where there is a strong demand for efficient arithmetic operators. Practical schemes such as hash functions, public-key encryption and digital signatures may be used in constrained environments, leading to interesting arithmetic problems. Common examples are long integer arithmetic (RSA) and arithmetic of algebraic curves and finite fields of medium sizes (elliptic curve cryptography, including pairing-based cryptography), and small finite fields (code-based cryptography and lattice-based cryptography).

3.2. Technology

The traditional arithmetic operators are small, low-level, close-to-the-silicon hardware building bricks, and it is therefore important to anticipate the evolutions of the technology to address the new challenges these evolutions will bring.

It is well known now that Moore's law is no longer what it used to be. It continues to bring more transistors on a chip with each new generation, but the speed of these transistors no longer increases, and their power consumption no longer decreases. With more integration come also more reliability issues.

These are the driving forces behind the shift to multicore processors, and to coarser and more complex processing units in these processors: single-instruction, multiple data (SIMD) instructions, fused multiply-and-add, and soon dot-product operations. It also led to the emergence of new massively parallel computing devices such as graphical processing units (GPU) and field-programmable gate arrays (FPGAs). Both are increasingly being used for general purpose computing.

In the shift to massively parallel multicores and GPUs, the real challenge is how to program them. With respect to computer arithmetic, the main problem is the control of numerical precision: the order of the elementary operations is changed in a parallel execution, and will very often not even be deterministic if the main objective is performance. Assessing or guaranteeing numerical quality in the face of this uncertainty is an open problem, all the more as SIMD units and limited data bandwidth encourage the use of mixed precision where possible.

Concerning FPGAs, their programming model is that of a digital circuit which may be application-specific, and even change in the lifetime of an application. The challenge here is to design arithmetic operators that exploit this reconfigurability, which is their main strength. Whereas processor operators have to be as general-purpose as possible, in an FPGA an operator can be designed specifically for a given application's context. A related challenge is to convince application designers that they should use these operators, which may be radically different from those they are used to see in processors. The C-to-hardware community addresses this challenge by hiding the FPGA behind a classical C programming model. This raises the arithmetic problem of automatically extracting from a piece of C code a fragment that is suitable for implementation as an application-specific operator in an FPGA.

In traditional circuit design, power consumption is no longer a concern only for embedded, battery-powered applications: heat dissipation is now the main issue limiting the frequency of high-performance processors. The nature of power consumption is also changing: it used to be caused mostly by the active switching transistors, but leakage power is now as much of a concern. All this impacts the design of operators, but also their use: the energy-per-computation metric will become more and more important and will orient algorithmic choices, for instance inviting us to reassess the benefits of pre-computing values.

Finally, the industry is preparing to address, within a decade or two, the end of silicon-based Moore's law. In addition to the physical limits (it is believed so far that we need at least one atom to build a transistor), the raising cost of fabrication plants at each generation has led to increasing concentration in fewer and fewer foundries. There will therefore be an economic limit when the number of foundries is down to one. Silicon replacement alternative are emerging in laboratories, without a clear winner yet. When these alternatives reach the integrated circuit, they may be expected to drastically change the rules by which arithmetic operators are designed.

3.3. Numbers and Number Representation

The first issue addressed by computer arithmetic is the representation of numbers in the computer. There are many possible representations, and a representation typically has many parameters. For instance, for integers, the decimal representation and the binary representation belong to the same family, only differing by the radix, 10 or 2. Another parameter of this representation is the number of digits considered.

A good representation is one that enables good computing. Here the measures of quality are numerous, sometimes conflicting, and application-dependent. For instance, the classical representation of integers is compact, but addition involves a carry propagation. There exists another classical family of integer representations which are redundant, therefore less compact, but allow for carry-free, thus faster, addition. Many other quality measures are possible, for instance power consumption, or silicon area.

Research on number representation for integers and reals is no longer very active, and it may be that there is little left to find in this field. The corresponding expertise now belongs to the common culture of the computer arithmetic community. For the integers, from time to time, a new context revives interest in an exotic number representation. For the reals, the indisputable advantages of a widespread and shared standard (the IEEE 754 floating-point standard) weigh strongly against innovation. However, for barely more complex datatypes, such as complex numbers or real intervals (each of which can be represented by a pair of reals), there is no such consensus yet.

Finally, research on number representation is still very active for datatypes related to more recent application fields, most notably in cryptography. For instance, the elliptic curve number system has been introduced because it allowed to use smaller keys for similar security, and research is still active to find representations of elliptic curves that enable efficient computation on this number system. This research tries to improve on

the usual quality metrics (performance, resource consumption, power), and in addition we have two more context-specific metrics: the key size, and the security level.

3.4. Arithmetic Algorithms

Each year, new algorithms are still published for basic operations (from addition to division), but the main focus of the computer arithmetic community has long shifted to more complex objects: examples are sums of many numbers, arithmetic on complex numbers, and evaluation of algebraic and transcendental functions.

The latter typically reduces to polynomial evaluation, with two sub-problems: firstly, one must find a good approximation polynomial. Secondly, one must evaluate it as fast as possible under some accuracy constraint.

When looking for good approximation polynomials, "good" has various possible meanings. For arbitrary precision implementations, polynomials must be built at runtime, so "good" means "simple" (for both the polynomial and the error term). Typical techniques in this case are based on Taylor or Chebyshev formulae. For fixed-precision implementations (for instance for the functions of the standard floating-point mathematical library), the polynomial is static, and we may afford to spend much more effort to build it. In this case, we may aim for better polynomials, in the sense that they minimize the approximation error over a complete interval: such polynomials are given by Remez' algorithm [56]. However, the coefficients of Remez polynomials will be arbitrary reals, and for implementation purpose we are more interested in the class of polynomials with machine-representable coefficients. An even better polynomial is therefore one that minimizes the approximation error among this class, a problem addressed in the Sollya toolbox developed in Arénaire (http://sollya.gforge.inria.fr/). In some cases it is useful to impose even more constraints on the polynomial. For instance, if the function is even, one classically wants to force to zero the coefficients of the odd powers in its polynomial approximation. Although this may require a higher degree approximation for the same accuracy, it reduces operation counts, and also increases the numerical stability of the evaluation.

Then, there are many ways to evaluate a polynomial, corresponding to many ways to rewrite it. The Horner scheme minimizes operation count and, in most practical cases, rounding errors, but it is a sequential scheme entailing a long execution time on modern processors. There exists parallel evaluation schemes that improve this latency, but degrade operation count and accuracy. The optimal scheme depends on details of the target architecture, and is best found by programmed exploration, as demonstrated by Intel on Itanium, and by Arénaire on the ST200 processor.

Thus, both polynomial approximation and polynomial evaluation illustrate the need for "meta-algorithms": i.e., algorithms designed to build arithmetic algorithms. In our example, the meta-algorithms in turn rely on linear algebra, integer linear programming, and Euclidean lattices. Other approaches may also lead to successful meta-algorithms, for instance the SPIRAL project (http://www.spiral.net/) uses algebraic rewriting to implement and optimize linear transforms. This approach has potential in arithmetic design, too.

3.5. Euclidean Lattice Reduction and Applications

A Euclidean lattice is the set of *integer* linear combinations of a finite set of real vectors. Typically, lattices occur when linear algebra questions are asked with discreteness constraints. In the last decade, they have become a classical ingredient in the computer arithmetic toolbox, along with other number-theoretic techniques (continued fractions, diophantine approximation, etc.). Indeed, integers (scaled by powers of the radix) are the essence of the fixed-point and floating-point representations of the real numbers. If the macroscopic properties of floating-point numbers are close to those of the real numbers, the finer properties are definitely related to questions over the integers. Thus, lattices have been successfully used in computer arithmetic to find constrained polynomial approximations to functions, and to attack the Table Maker's Dilemma. They have a potential for further arithmetic applications, for instance the design of digital filters.

Besides, the algorithms on Euclidean lattices are a rich experimentation laboratory for different types of arithmetics. The basis vectors are often represented exactly with long integer arithmetic. Furthermore, the fastest algorithms find the operations to be performed on the basis vectors via approximate computations, typically an approximate Gram-Schmidt orthogonalisation. These approximate computations may be performed with fixed-precision or arbitrary precision floating-point arithmetics. In some time-consuming applications of lattice algorithms, such as cryptanalyses of variants of RSA or lattice-based cryptosystems, integer linear programming, or even for solving the Table Maker's Dilemma, the practical run-time is of utmost importance. This motivates strong optimizations for the underlying arithmetics.

Further, aside from this strong relationship between lattices and arithmetics, the understanding of lattice-based cryptology is developing at a quick pace; making it efficient while remaining secure will require a thorough study, which must involve experts in both arithmetics and cryptology.

3.6. Reliability and Accuracy

Having basic arithmetic operators that are well-specified by standards leads to two directions. The first is to provide a guarantee that the implementations of these operators match their specification. The second is to use these operators as building blocks of well-specified computations, in other words to build upon these operators to obtain guarantees on the results of larger computing cores.

The approaches used to get such a guarantee vary greatly. Some computations are performed exactly, and in this case the results are considered to be intrinsically correct. However, exact values may not be finitely representable in the chosen number system and format: they must then be approximated. When an approximate value is computed using floating-point arithmetic, the specification of this arithmetic is employed to establish a bound on the roundoff errors, or to check that no exceptional situation occurred. For instance, the IEEE-754 standard for floating-point arithmetic implies useful properties, e.g., Dekker's error-free multiplication for various radices and precisions, the faithfulness of Horner's polynomial evaluation, etc.

Another possibility is that a simple final computation, still performed using floating-point arithmetic, enables to check whether a computed result is a reasonable approximation of the exact (unknown) result. Typically, to check that, for instance, a computed matrix R is close to the inverse of the initial matrix A, it suffices to check whether the product RA is close enough to the identity matrix. Such a simple, a posteriori, computation is called a *certificate*.

When considering more complicated functions, e.g., elementary functions, another issue arises. These functions have to be approximated, in general by polynomials. It no longer suffices to bound the rounding errors of the computations and check that no underflow/overflow may occur. One also has to take into account the approximation errors: certifying tight error bounds is quite a challenge. One usually talks of *verified computations* in this case.

Safety is typically based on interval arithmetic: what is computed is an interval which provably encloses the sought values. Naive interval arithmetic evaluates an expression as it is written, which does not take into account the dependencies between variables. This leads to irrelevant interval bloat. To address this problem, a solution is sometimes to rewrite the expression, a technique used for instance by the Gappa tool (http://gappa.gforge.inria.fr/) initially developed in Arénaire. Another systematic method is to use extensions to interval arithmetic. For instance, affine arithmetic has been used to optimize the data-path width of FPGA computing cores, and is also used in the Fluctuat tool to diagnose numerical instabilities in programs. When working with functions, Taylor models are a relevant extension: they represent a function as the sum of a polynomial of fixed degree and of an interval enclosing all errors (approximation as well rounding errors). This approach is very useful for computations involving function approximations, and has for instance been used successfully for the computation of the supremum norm of a function in one variable. The issue here is to devise algorithms that do not overestimate too much the result. It may be necessary to mix interval arithmetic and variable precision to reach the required level of guarantee and accuracy. In general, determining the right precision is difficult: the precision must be high enough to yield accurate results, but not too high since the computing time increases with the computing precision.

The complexity of some computer arithmetic algorithms, the intrinsic complexity of the floating-point model, the use of floating-point for critical applications, strongly advocate for the use of *formal proof* in computer arithmetic: a proof checker checks every step of the proof obtained by any means mentioned above. Even circuit manufacturers often provide a formal proof of the critical parts of their floating-point algorithms. For instance, the Intel divide and square root algorithms for the Itanium were formally proven. The expertise of the French community (which includes several ex-Arénaire members) in proving floating-point algorithms is well recognized. However, even the lower properties of the arithmetic are still challenging. For instance, with the specification of decimal arithmetic in the new version of the IEEE 754 standard, many theorems established in radix two have to be generalized to other radices.

CARAMEL Project-Team

3. Scientific Foundations

3.1. Cryptography, arithmetic: hardware and software

One of the main topics for our project is public-key cryptography. After 20 years of hegemony, the classical public-key algorithms (whose security is based on integer factorization or discrete logarithm in finite fields) are currently being overtaken by elliptic curves. The fundamental reason for this is that the best-known algorithms for factoring integers or for computing discrete logarithms in finite fields have a subexponential complexity, whereas the best known attack for elliptic-curve discrete logarithms has exponential complexity. As a consequence, for a given security level 2^n, the key sizes must grow linearly with n for elliptic curves, whereas they grow like n^3 for RSA-like systems. As a consequence, several governmental agencies, like the NSA or the BSI, now recommend to use elliptic-curve cryptosystems for new products that are not bound to RSA for backward compatibility.

Besides RSA and elliptic curves, there are several alternatives currently under study. There is a recent trend to promote alternate solutions that do not rely on number theory, with the objective of building systems that would resist a quantum computer (in contrast, integer factorization and discrete logarithms in finite fields and elliptic curves have a polynomial-time quantum solution). Among them, we find systems based on hard problems in lattices (NTRU is the most famous), those based on coding theory (McEliece system and improved versions), and those based on the difficulty to solve multivariate polynomial equations (HFE, for instance). None of them has yet reached the same level of popularity as RSA or elliptic curves for various reasons, including the presence of unsatisfactory features (like a huge public key), or the non-maturity (system still alternating between being fixed one day and broken the next day).

Returning to number theory, an alternative to RSA and elliptic curves is to use other curves and in particular genus-2 curves. These so-called hyperelliptic cryptosystems have been proposed in 1989 [17], soon after the elliptic ones, but their deployment is by far more difficult. The first problem was the group law. For elliptic curves, the elements of the group are just the points of the curve. In a hyperelliptic cryptosystem, the elements of the group are points on a 2-dimensional variety associated to the genus-2 curve, called the Jacobian variety. Although there exist polynomial-time methods to represent and compute with them, it took some time before getting a group law that could compete with the elliptic one in terms of speed. Another question that is still not yet fully answered is the computation of the group order, which is important for assessing the security of the associated cryptosystem. This amounts to counting the points of the curve that are defined over the base field or over an extension, and therefore this general question is called point-counting. In the past ten years there have been major improvements on the topic, but there are still cases for which no practical solution is known.

Another recent discovery in public-key cryptography is the fact that having an efficient bilinear map that is hard to invert (in a sense that can be made precise) can lead to powerful cryptographic primitives. The only examples we know of such bilinear maps are associated with algebraic curves, and in particular elliptic curves: this is the so-called Weil pairing (or its variant, the Tate pairing). Initially considered as a threat for elliptic-curve cryptography, they have proven to be quite useful from a constructive point of view, and since the beginning of the decade, hundreds of articles have been published, proposing efficient protocols based on pairings. A long-lasting open question, namely the construction of a practical identity-based encryption scheme, has been solved this way. The first standardization of pairing-based cryptography has recently occurred (see ISO/IEC 14888-3 or IEEE P1363.3), and a large deployment is to be expected in the next years.

Despite the raise of elliptic curve cryptography and the variety of more or less mature other alternatives, classical systems (based on factoring or discrete logarithm in finite fields) are still going to be widely used in the next decade, at least, due to resilience: it takes a long time to adopt new standards, and then an even longer time to renew all the software and hardware that is widely deployed.

This context of public-key cryptography motivates us to work on integer factorization, for which we have acquired expertise, both in factoring moderate-sized numbers, using the ECM (Elliptic Curve Method) algorithm, and in factoring large RSA-like numbers, using the number field sieve algorithm. The goal is to follow the transition from RSA to other systems and continuously assess its security to adjust key sizes. We also want to work on the discrete-logarithm problem in finite fields. This second task is not only necessary for assessing the security of classical public-key algorithms, but is also crucial for the security of pairing-based cryptography.

We also plan to investigate and promote the use of pairing-based and genus-2 cryptosystems. For pairings, this is mostly a question of how efficient can such a system be in software, in hardware, and using all the tools from fast implementation to the search for adequate curves. For genus 2, as said earlier, constructing an efficient cryptosystem requires some more fundamental questions to be solved, namely the point-counting problem.

We summarize in the following table the aspects of public-key cryptography that we address in the CARAMEL team.

public-key primitive	cryptanalysis	design	implementation
RSA	X	–	–
Finite Field DLog	X	–	–
Elliptic Curve DLog	–	–	Soft
Genus 2 DLog	–	X	Soft
Pairings	X	X	Soft/Hard

Another general application for the project is computer algebra systems (CAS), that rely in many places on efficient arithmetic. Nowadays, the objective of a CAS is not only to have more and more features that the user might wish, but also to compute the results fast enough, since in many cases, the CAS are used interactively, and a human is waiting for the computation to complete. To tackle this question, more and more CAS use external libraries, that have been written with speed and reliability as first concern. For instance, most of today's CAS use the GMP library for their computations with big integers. Many of them will also use some external Basic Linear Algebra Subprograms (BLAS) implementation for their needs in numerical linear algebra.

During a typical CAS session, the libraries are called with objects whose sizes vary a lot; therefore being fast on all sizes is important. This encompasses small-sized data, like elements of the finite fields used in cryptographic applications, and larger structures, for which asymptotically fast algorithms are to be used. For instance, the user might want to study an elliptic curve over the rationals, and as a consequence, check its behaviour when reduced modulo many small primes; and then [s]he can search for large torsion points over an extension field, which will involve computing with high-degree polynomials with large integer coefficients.

Writing efficient software for arithmetic as it is used typically in CAS requires the knowledge of many algorithms with their range of applicability, good programming skills in order to spend time only where it should be spent, and finally good knowledge of the target hardware. Indeed, it makes little sense to disregard the specifics of the possible hardware platforms intended, even more so since in the past years, we have seen a paradigm shift in terms of available hardware: so far, it used to be reasonable to consider that an end-user running a CAS would have access to a single-CPU processor. Nowadays, even a basic laptop computer has a multi-core processor and a powerful graphics card, and a workstation with a reconfigurable coprocessor is no longer science-fiction.

In this context, one of our goals is to investigate and take advantage of these influences and interactions between various available computing resources in order to design better algorithms for basic arithmetic objects. Of course, this is not disconnected from the others goals, since they all rely more or less on integer or polynomial arithmetic.

CASCADE Project-Team

3. Scientific Foundations

3.1. Provable Security

Since the beginning of public-key cryptography, with the seminal Diffie-Hellman paper [75], many suitable algorithmic problems for cryptography have been proposed and many cryptographic schemes have been designed, together with more or less heuristic proofs of their security relative to the intractability of the underlying problems. However, many of those schemes have thereafter been broken. The simple fact that a cryptographic algorithm withstood cryptanalytic attacks for several years has often been considered as a kind of validation procedure, but schemes may take a long time before being broken. An example is the Chor-Rivest cryptosystem [74], based on the knapsack problem, which took more than 10 years to be totally broken [90], whereas before this attack it was believed to be strongly secure. As a consequence, the lack of attacks at some time should never be considered as a full security validation of the proposal.

A completely different paradigm is provided by the concept of "provable" security. A significant line of research has tried to provide proofs in the framework of complexity theory (a.k.a. "reductionist" security proofs): the proofs provide reductions from a well-studied problem (factoring, RSA or the discrete logarithm) to an attack against a cryptographic protocol.

At the beginning, researchers just tried to define the security notions required by actual cryptographic schemes, and then to design protocols which could achieve these notions. The techniques were directly derived from complexity theory, providing polynomial reductions. However, their aim was essentially theoretical. They were indeed trying to minimize the required assumptions on the primitives (one-way functions or permutations, possibly trapdoor, etc) [78], without considering practicality. Therefore, they just needed to design a scheme with polynomial-time algorithms, and to exhibit polynomial reductions from the basic mathematical assumption on the hardness of the underlying problem into an attack of the security notion, in an asymptotic way. However, such a result has no practical impact on actual security. Indeed, even with a polynomial reduction, one may be able to break the cryptographic protocol within a few hours, whereas the reduction just leads to an algorithm against the underlying problem which requires many years. Therefore, those reductions only prove the security when very huge (and thus maybe unpractical) parameters are in use, under the assumption that no polynomial time algorithm exists to solve the underlying problem.

For a few years, more efficient reductions have been expected, under the denomination of either "exact security" [71] or "concrete security" [83], which provide more practical security results. The perfect situation is reached when one is able to prove that, from an attack, one can describe an algorithm against the underlying problem, with almost the same success probability within almost the same amount of time: "tight reductions". We have then achieved "practical security" [67].

Unfortunately, in many cases, even just provable security is at the cost of an important loss in terms of efficiency for the cryptographic protocol. Thus, some models have been proposed, trying to deal with the security of efficient schemes: some concrete objects are identified with ideal (or black-box) ones. For example, it is by now usual to identify hash functions with ideal random functions, in the so-called "random-oracle model", informally introduced by Fiat and Shamir [76], and later formalized by Bellare and Rogaway [70]. Similarly, block ciphers are identified with families of truly random permutations in the "ideal cipher model" [68]. A few years ago, another kind of idealization was introduced in cryptography, the black-box group, where the group operation, in any algebraic group, is defined by a black-box: a new element necessarily comes from the addition (or the subtraction) of two already known elements. It is by now called the "generic model" [82], [89]. Some works even require several ideal models together to provide some new validations [73].

More recently, the new trend is to get provable security, without such ideal assumptions (there are currently a long list of publications showing "without random oracles" in their title), but under new and possibly stronger computational assumptions. As a consequence, a cryptographer has to deal with the three following important steps:

computational assumptions, which are the foundations of the security. We thus need to have a strong evidence that the computational problems are reasonably hard to solve. We study several assumptions, by improving algorithms (attacks), and notably using lattice reductions. We furthermore contribute to the list of "potential" hard problems.

security model, which makes precise the security notions one wants to achieve, as well as the means the adversary may be given. We contribute to this point, in several ways:

- by providing a security model for many primitives and protocols, and namely group-oriented protocols, which involve many parties, but also many communications (group key exchange, group signatures, etc);
- by enhancing some classical security models;
- by considering new means for the adversary, such as side-channel information.

design of new schemes/protocols, or more efficient, with additional features, etc.

security proof, which consists in exhibiting a reduction.

For a long time, the security proofs by reduction used classical techniques from complexity theory, with a direct description of the reduction, and then a long and quite technical analysis for providing the probabilistic estimates. Such analysis is unfortunately error-prone. Victor Shoup proposed a nice way to organize the proofs, and eventually obtain the probabilities, using a sequence of games [88], [69], [84] which highlights the computational assumptions, and splits the analysis in small independent problems. We early adopted and developed this technique, and namely in [77]. We applied this methodology to various kinds of systems, in order to achieve the highest security properties: authenticity, integrity, confidentiality, privacy, anonymity. Nevertheless, efficiency was also a basic requirement.

However, such reductions are notoriously error-prone: errors have been found in many published protocols. Security errors can have serious consequences, such as loss of money in the case of electronic commerce. Moreover, security errors cannot be detected by testing, because they appear only in the presence of a malicious adversary.

Security protocols are therefore an important area for formal verification.

3.2. Cryptanalysis

Because there is no absolute proof of security, it is essential to study cryptanalysis, which is roughly speaking the science of code-breaking. As a result, key-sizes are usually selected based on the state-of-the-art in cryptanalysis. The previous section emphasized that public-key cryptography required hard computational problems: if there is no hard problem, there cannot be any public-key cryptography either. If any of the computational problems mentioned above turns out to be easy to solve, then the corresponding cryptosystems can be broken, as the public key would actually disclose the private key. This means that one obvious way to cryptanalyze is to solve the underlying algorithmic problems, such as integer factorization, discrete logarithm, lattice reduction, Gröbner bases, *etc.* Here, we mean a study of the computational problem in its full generality. The project-team has a strong expertise (both in design and analysis) on the best algorithms for lattice reduction, which are also very useful to attack classical schemes based on factorization or discrete logarithm.

Alternatively, one may try to exploit the special properties of the cryptographic instances of the computational problem. Even if the underlying general problem is NP-hard, its cryptographic instances may be much easier, because the cryptographic functionalities typically require a specific mathematical structure. In particular, this means that there might be an attack which can only be used to break the scheme, but not to solve the underlying problem in general. This happened many times in knapsack cryptography and multivariate cryptography. Interestingly, generic tools to solve the general problem perform sometimes even much better on cryptographic instances (this happened for Gröbner bases and lattice reduction).

However, if the underlying computational problem turns out to be really hard both in general and for instances of cryptographic interest, this will not necessarily imply that the cryptosystem is secure. First of all, it is not even clear what is meant exactly by the term *secure* or *insecure*. Should an encryption scheme which leaks the first bit of the plaintext be considered secure? Is the secret key really necessary to decrypt ciphertexts or to sign messages? If a cryptosystem is theoretically secure, could there be potential security flaws for its implementation? For instance, if some of the temporary variables (such as pseudo-random numbers) used during the cryptographic operations are partially leaked, could it have an impact on the security of the cryptosystem? This means that there is much more into cryptanalysis than just trying to solve the main algorithmic problems. In particular, cryptanalysts are interested in defining and studying realistic environments for attacks (adaptive chosen-ciphertext attacks, side-channel attacks, *etc.*), as well as goals of attacks (key recovery, partial information, existential forgery, distinguishability, *etc.*). As such, there are obvious connections with provable security. It is perhaps worth noting that cryptanalysis also proved to be a good incentive for the introduction of new techniques in cryptology. Indeed, several mathematical objects now considered invaluable in cryptographic design were first introduced in cryptology as cryptanalytic tools, including lattices and pairings. The project-team has a strong expertise in cryptanalysis: many schemes have been broken, and new techniques have been developed.

3.3. Symmetric Cryptography

Even if asymmetric cryptography has been a major breakthrough in cryptography, and a key element in its recent development, conventional cryptography (a.k.a. symmetric, or secret key cryptography) is still required in any application: asymmetric cryptography is much more powerful and convenient, since it allows signatures, key exchange, etc. However, it is not well-suited for high-rate communication links, such as video or audio streaming. Therefore, block-ciphers remain a fundamental primitive. However, since the AES Competition (which started in January 1997, and eventually selected the Rijndael algorithm in October 2000), this domain has become less active, even though some researchers are still trying to develop new attacks. On the opposite, because of the lack of widely admitted stream ciphers (able to encrypt high-speed streams of data), ECRYPT (the European Network of Excellence in Cryptology) launched the eSTREAM project, which investigated research on this topic, at the international level: many teams proposed candidates that have been analyzed by the entire cryptographic community. Similarly, in the last few years, hash functions [86], [85], [80], [81], [79], which are an essential primitive in many protocols, received a lot of attention: they were initially used for improving efficiency in signature schemes, hence the requirement of collision-resistance. But afterwards, hash functions have been used for many purposes, such as key derivation, random generation, and random functions (random oracles [70]). Recently, a bunch of attacks [72], [91], [92], [93], [94], [96], [95] have shown several drastic weaknesses on all known hash functions. Knowing more (how weak they are) about them, but also building new hash functions are major challenges. For the latter goal, the first task is to formally define a security model for hash functions, since no realistic formal model exists at the moment: in a way, we expect too much from hash functions, and it is therefore impossible to design such "ideal" functions. Because of the high priority of this goal (the design of a new hash function), the NIST has launched an international competition, called SHA-3 (similar to the AES competition 10 years ago), in order to select and standardize a hash function. Keccak has been officially chosen on October 2nd, 2012.

One way to design new hash functions may be a new mode of operation, which would involve a block cipher, iterated in a specific manner. This is already used to build stream ciphers and message authentication codes (symmetric authentication). Under some assumptions on the block cipher, it might be possible to apply the above methodology of provable security in order to prove the validity of the new design, according to a specific security model.

GALAAD Project-Team

3. Scientific Foundations

3.1. Introduction

Our scientific activity is structured according to three broad topics:

1. **Algebraic representations for geometric modeling**.
2. **Algebraic algorithms for geometric computing**,
3. **Symbolic-numeric methods for analysis**,

3.2. Algebraic representations for geometric modeling

Compact, efficient and structured descriptions of shapes are required in many scientific computations in engineering, such as "Isogeometric" Finite Elements methods, point cloud fitting problems or implicit surfaces defined by convolution. Our objective is to investigate new algebraic representations (or improve the existing ones) together with their analysis and implementations.

We are investigating representations, based on semi-algebraic models. Such non-linear models are able to capture efficiently complexes shapes, using few data. However, they required specific methods to solve the underlying non-linear problems, which we are investigating.

Effective algebraic geometry is a natural framework for handling shape representations. This framework not only provides tools for modeling but it also allows to exploit rich geometric properties.

The above-mentioned tools of effective algebraic geometry make it possible to analyse in detail and separately algebraic varieties. We are interested in problems where collections of piecewise algebraic objects are involved. The properties of such geometrical structures are still not well controlled, and the traditional algorithmic geometry methods do not always extend to this context, which requires new investigations.

The use of piecewise algebraic representations also raises problems of approximation and reconstruction, on which we are working on. In this direction, we are studying B-spline function spaces with specified regularity associated to domain partitions.

Many geometric properties are, by nature, independent from the reference one chooses for performing analytic computations. This leads naturally to invariant theory. We are interested in exploiting these invariant properties, to develop compact and adapted representations of shapes.

3.3. Algebraic algorithms for geometric computing

This topic is directly related to polynomial system solving and effective algebraic geometry. It is our core expertise and many of our works are contributing to this area.

Our goal is to develop algebraic algorithms to efficiently perform geometric operations such as computing the intersection or self-intersection locus of algebraic surface patches, offsets, envelopes of surfaces, ...

The underlying representations behind the geometric models we consider are often of algebraic type. Computing with such models raises algebraic questions, which frequently appear as bottlenecks of the geometric problems.

In order to compute the solutions of a system of polynomial equations in several variables, we analyse and take advantage of the structure of the quotient ring, defined by these polynomials. This raises questions of representing and computing normal forms in such quotient structures. The numerical and algebraic computations in this context lead us to study new approaches of normal form computations, generalizing the well-known Gröbner bases.

Geometric objects are often described in a parametric form. For performing efficiently on these objects, it can also be interesting to manipulate implicit representations. We consider particular projections techniques based on new resultant constructions or syzygies, which allow to transform parametric representations into implicit ones. These problems can be reformulated in terms of linear algebra. We investigate methods which exploit this matrix representation based on resultant constructions.

They involve structured matrices such as Hankel, Toeplitz, Bezoutian matrices or their generalization in several variables. We investigate algorithms that exploit their properties and their implications in solving polynomial equations.

We are also interested in the "effective" use of duality, that is, the properties of linear forms on the polynomials or quotient rings by ideals. We undertake a detailed study of these tools from an algorithmic perspective, which yields the answer to basic questions in algebraic geometry and brings a substantial improvement on the complexity of resolution of these problems.

We are also interested in subdivision methods, which are able to efficiently localise the real roots of polynomial equations. The specificities of these methods are local behavior, fast convergence properties and robustness. Key problems are related to the analysis of multiple points.

An important issue while developing these methods is to analyse their practical and algorithmic behavior. Our aim is to obtain good complexity bounds and practical efficiency by exploiting the structure of the problem.

3.4. Symbolic numeric analysis

While treating practical problems, noisy data appear and incertitude has to be taken into account. The objective is to devise adapted techniques for analyzing the geometric properties of the algebraic models in this context.

Analysing a geometric model requires tools for structuring it, which first leads to study its singularities and its topology. In many context, the input representation is given with some error so that the analysis should take into account not only one model but a neighborhood of models.

The analysis of singularities of geometric models provides a better understanding of their structures. As a result, it may help us better apprehend and approach modeling problems. We are particularly interested in applying singularity theory to cases of implicit curves and surfaces, silhouettes, shadows curves, moved curves, medial axis, self-intersections, appearing in algorithmic problems in CAGD and shape analysis.

The representation of such shapes is often given with some approximation error. It is not surprising to see that symbolic and numeric computations are closely intertwined in this context. Our aim is to exploit the complementarity of these domains, in order to develop controlled methods.

The numerical problems are often approached locally. However, in many situations it is important to give global answers, making it possible to certify computation. The symbolic-numeric approach combining the algebraic and analytical aspects, intends to address these local-global problems. Especially, we focus on certification of geometric predicates that are essential for the analysis of geometrical structures.

The sequence of geometric constructions, if treated in an exact way, often leads to a rapid complexification of the problems. It is then significant to be able to approximate the geometric objects while controlling the quality of approximation. We investigate subdivision techniques based on the algebraic formulation of our problems which allow us to control the approximation, while locating interesting features such as singularities.

According to an engineer in CAGD, the problems of singularities obey the following rule: less than 20% of the treated cases are singular, but more than 80% of time is necessary to develop a code allowing to treat them correctly. Degenerated cases are thus critical from both theoretical and practical perspectives. To resolve these difficulties, in addition to the qualitative studies and classifications, we also study methods of *perturbations* of symbolic systems, or adaptive methods based on exact arithmetics.

The problem of decomposition and factorisation is also important. We are interested in a new type of algorithms that combine the numerical and symbolic aspects, and are simultaneously more effective and reliable. A typical problem in this direction is the problem of approximate factorization, which requires to analyze perturbations of the data, which enables us to break up the problem.

GEOMETRICA Project-Team

3. Scientific Foundations

3.1. Mesh Generation and Geometry Processing

Meshes are becoming commonplace in a number of applications ranging from engineering to multimedia through biomedecine and geology. For rendering, the quality of a mesh refers to its approximation properties. For numerical simulation, a mesh is not only required to faithfully approximate the domain of simulation, but also to satisfy size as well as shape constraints. The elaboration of algorithms for automatic mesh generation is a notoriously difficult task as it involves numerous geometric components: Complex data structures and algorithms, surface approximation, robustness as well as scalability issues. The recent trend to reconstruct domain boundaries from measurements adds even further hurdles. Armed with our experience on triangulations and algorithms, and with components from the CGAL library, we aim at devising robust algorithms for 2D, surface, 3D mesh generation as well as anisotropic meshes. Our research in mesh generation primarily focuses on the generation of simplicial meshes, i.e. triangular and tetrahedral meshes. We investigate both greedy approaches based upon Delaunay refinement and filtering, and variational approaches based upon energy functionals and associated minimizers.

The search for new methods and tools to process digital geometry is motivated by the fact that previous attempts to adapt common signal processing methods have led to limited success: Shapes are not just another signal but a new challenge to face due to distinctive properties of complex shapes such as topology, metric, lack of global parameterization, non-uniform sampling and irregular discretization. Our research in geometry processing ranges from surface reconstruction to surface remeshing through curvature estimation, principal component analysis, surface approximationand surface mesh parameterization. Another focus is on the robustness of the algorithms to defect-laden data. This focus stems from the fact that acquired geometric data obtained through measurements or designs are rarely usable directly by downstream applications. This generates bottlenecks, i.e., parts of the processing pipeline which are too labor-intensive or too brittle for practitioners. Beyond reliability and theoretical foundations, our goal is to design methods which are also robust to raw, unprocessed inputs.

3.2. Topological and Geometric Inference

Due to the fast evolution of data acquisition devices and computational power, scientists in many areas are asking for efficient algorithmic tools for analyzing, manipulating and visualizing more and more complex shapes or complex systems from approximative data. Many of the existing algorithmic solutions which come with little theoretical guarantee provide unsatisfactory and/or unpredictable results. Since these algorithms take as input discrete geometric data, it is mandatory to develop concepts that are rich enough to robustly and correctly approximate continuous shapes and their geometric properties by discrete models. Ensuring the correctness of geometric estimations and approximations on discrete data is a sensitive problem in many applications.

Data sets being often represented as point sets in high dimensional spaces, there is a considerable interest in analyzing and processing data in such spaces. Although these point sets usually live in high dimensional spaces, one often expects them to be located around unknown, possibly non linear, low dimensional shapes. These shapes are usually assumed to be smooth submanifolds or more generally compact subsets of the ambient space. It is then desirable to infer topological (dimension, Betti numbers,...) and geometric characteristics (singularities, volume, curvature,...) of these shapes from the data. The hope is that this information will help to better understand the underlying complex systems from which the data are generated. In spite of recent promising results, many problems still remain open and to be addressed, need a tight collaboration between mathematicians and computer scientists. In this context, our goal is to contribute to the development of new mathematically well founded and algorithmically efficient geometric tools for data analysis and processing of complex geometric objects. Our main targeted areas of application include machine learning, data mining, statistical analysis, and sensor networks.

3.3. Data Structures and Robust Geometric Computation

GEOMETRICA has a large expertise of algorithms and data structures for geometric problems.We are pursuing efforts to design efficient algorithms from a theoretical point of view, but we also put efforts in the effective implementation of these results.

In the past years, we made significant contributions to algorithms for computing Delaunay triangulations (which are used by meshes in the above paragraph). We are still working on the practical efficiency of existing algorithms to compute or to exploit classical Euclidean triangulations in 2 and 3 dimensions, but the current focus of our research is more aimed towards extending the triangulation efforts in several new directions of research.

One of these directions is the triangulation of non Euclidean spaces such as periodic or projective spaces, with various potential applications ranging from astronomy to granular material simulation.

Another direction is the triangulation of moving points, with potential applications to fluid dynamics where the points represent some particles of some evolving physical material, and to variational methods devised to optimize point placement for meshing a domain with a high quality elements.

Increasing the dimension of space is also a stimulating direction of research, as triangulating points in medium dimension (say 4 to 15) has potential applications and makes new challenges to trade exponential complexity of the problem in the dimension for the possibility to reach effective and practical results in reasonably small dimensions.

On the complexity analysis side, we pursue efforts to obtain complexity analysis in some practical situations involving randomized or stochastic hypotheses. On the algorithm design side, we are looking for new paradigms to exploit parallelism on modern multicore hardware architectures.

Finally, all this work is done while keeping in mind concerns related to effective implementation of our work, practical efficiency and robustness issues which have become a background task of all different works made by GEOMETRICA.

GRACE Team (section vide)

LFANT Project-Team

3. Scientific Foundations

3.1. Number fields, class groups and other invariants

Participants: Bill Allombert, Athanasios Angelakis, Karim Belabas, Julio Brau, Jean-Paul Cerri, Henri Cohen, Jean-Marc Couveignes, Andreas Enge, Pierre Lezowski, Nicolas Mascot, Aurel Page.

Modern number theory has been introduced in the second half of the 19th century by Dedekind, Kummer, Kronecker, Weber and others, motivated by Fermat's conjecture: There is no non-trivial solution in integers to the equation $x^n + y^n = z^n$ for $n \geqslant 3$. For recent textbooks, see [6]. Kummer's idea for solving Fermat's problem was to rewrite the equation as $(x+y)(x+\zeta y)(x+\zeta^2 y)\cdots(x+\zeta^{n-1}y) = z^n$ for a primitive n-th root of unity ζ, which seems to imply that each factor on the left hand side is an n-th power, from which a contradiction can be derived.

The solution requires to augment the integers by *algebraic numbers*, that are roots of polynomials in $\mathbb{Z}[X]$. For instance, ζ is a root of $X^n - 1$, $\sqrt[3]{2}$ is a root of $X^3 - 2$ and $\frac{\sqrt{3}}{5}$ is a root of $25X^2 - 3$. A *number field* consists of the rationals to which have been added finitely many algebraic numbers together with their sums, differences, products and quotients. It turns out that actually one generator suffices, and any number field K is isomorphic to $\mathbb{Q}[X]/(f(X))$, where $f(X)$ is the minimal polynomial of the generator. Of special interest are *algebraic integers*, "numbers without denominators", that are roots of a monic polynomial. For instance, ζ and $\sqrt[3]{2}$ are integers, while $\frac{\sqrt{3}}{5}$ is not. The *ring of integers* of K is denoted by $\mathcal{O}_K$; it plays the same role in K as $\mathbb{Z}$ in $\mathbb{Q}$.

Unfortunately, elements in $\mathcal{O}_K$ may factor in different ways, which invalidates Kummer's argumentation. Unique factorisation may be recovered by switching to *ideals*, subsets of $\mathcal{O}_K$ that are closed under addition and under multiplication by elements of $\mathcal{O}_K$. In $\mathbb{Z}$, for instance, any ideal is *principal*, that is, generated by one element, so that ideals and numbers are essentially the same. In particular, the unique factorisation of ideals then implies the unique factorisation of numbers. In general, this is not the case, and the *class group* Cl_K of ideals of $\mathcal{O}_K$ modulo principal ideals and its *class number* $h_K = |\,\mathrm{Cl}_K\,|$ measure how far $\mathcal{O}_K$ is from behaving like $\mathbb{Z}$.

Using ideals introduces the additional difficulty of having to deal with *units*, the invertible elements of $\mathcal{O}_K$: Even when $h_K = 1$, a factorisation of ideals does not immediately yield a factorisation of numbers, since ideal generators are only defined up to units. For instance, the ideal factorisation $(6) = (2)\cdot(3)$ corresponds to the two factorisations $6 = 2\cdot 3$ and $6 = (-2)\cdot(-3)$. While in $\mathbb{Z}$, the only units are 1 and -1, the unit structure in general is that of a finitely generated $\mathbb{Z}$-module, whose generators are the *fundamental units*. The *regulator* R_K measures the "size" of the fundamental units as the volume of an associated lattice.

One of the main concerns of algorithmic algebraic number theory is to explicitly compute these invariants (Cl_K and h_K, fundamental units and R_K), as well as to provide the data allowing to efficiently compute with numbers and ideals of $\mathcal{O}_K$; see [36] for a recent account.

The *analytic class number formula* links the invariants h_K and R_K (unfortunately, only their product) to the ζ-function of K, $\zeta_K(s) := \prod_{\mathfrak{p} \text{ prime ideal of } \mathcal{O}_K} (1 - \mathrm{N}\,\mathfrak{p}^{-s})^{-1}$, which is meaningful when $\Re(s) > 1$, but which may be extended to arbitrary complex $s \neq 1$. Introducing characters on the class group yields a generalisation of ζ- to L-functions. The *generalised Riemann hypothesis (GRH)*, which remains unproved even over the rationals, states that any such L-function does not vanish in the right half-plane $\Re(s) > 1/2$. The validity of the GRH has a dramatic impact on the performance of number theoretic algorithms. For instance, under GRH, the class group admits a system of generators of polynomial size; without GRH, only exponential bounds are known. Consequently, an algorithm to compute Cl_K via generators and relations (currently the only viable practical approach) either has to assume that GRH is true or immediately becomes exponential.

When $h_K = 1$ the number field K may be norm-Euclidean, endowing $\mathcal{O}_K$ with a Euclidean division algorithm. This question leads to the notions of the Euclidean minimum and spectrum of K, and another task in algorithmic number theory is to compute explicitly this minimum and the upper part of this spectrum, yielding for instance generalised Euclidean gcd algorithms.

3.2. Function fields, algebraic curves and cryptology

Participants: Karim Belabas, Julio Brau, Jean-Marc Couveignes, Andreas Enge, Nicolas Mascot, Jérôme Milan, Damien Robert, Vincent Verneuil.

Algebraic curves over finite fields are used to build the currently most competitive public key cryptosystems. Such a curve is given by a bivariate equation $\mathcal{C}(X, Y) = 0$ with coefficients in a finite field $\mathbb{F}_q$. The main classes of curves that are interesting from a cryptographic perspective are *elliptic curves* of equation $\mathcal{C} = Y^2 - (X^3 + aX + b)$ and *hyperelliptic curves* of equation $\mathcal{C} = Y^2 - (X^{2g+1} + \cdots)$ with $g \geqslant 2$.

The cryptosystem is implemented in an associated finite abelian group, the *Jacobian*$\mathrm{Jac}_\mathcal{C}$. Using the language of function fields exhibits a close analogy to the number fields discussed in the previous section. Let $\mathbb{F}_q(X)$ (the analogue of $\mathbb{Q}$) be the *rational function field* with subring $\mathbb{F}_q[X]$ (which is principal just as $\mathbb{Z}$). The *function field* of $\mathcal{C}$ is $K_\mathcal{C} = \mathbb{F}_q(X)[Y]/(\mathcal{C})$; it contains the *coordinate ring*$\mathcal{O}_\mathcal{C} = \mathbb{F}_q[X, Y]/(\mathcal{C})$. Definitions and properties carry over from the number field case $K/\mathbb{Q}$ to the function field extension $K_\mathcal{C}/\mathbb{F}_q(X)$. The Jacobian $\mathrm{Jac}_\mathcal{C}$ is the divisor class group of $K_\mathcal{C}$, which is an extension of (and for the curves used in cryptography usually equals) the ideal class group of $\mathcal{O}_\mathcal{C}$.

The size of the Jacobian group, the main security parameter of the cryptosystem, is given by an L-function. The GRH for function fields, which has been proved by Weil, yields the Hasse–Weil bound $(\sqrt{q} - 1)^{2g} \leqslant |\mathrm{Jac}_\mathcal{C}| \leqslant (\sqrt{q} + 1)^{2g}$, or $|\mathrm{Jac}_\mathcal{C}| \approx q^g$, where the *genus* g is an invariant of the curve that correlates with the degree of its equation. For instance, the genus of an elliptic curve is 1, that of a hyperelliptic one is $\frac{\deg_X \mathcal{C}-1}{2}$. An important algorithmic question is to compute the exact cardinality of the Jacobian.

The security of the cryptosystem requires more precisely that the *discrete logarithm problem* (DLP) be difficult in the underlying group; that is, given elements D_1 and $D_2 = xD_1$ of $\mathrm{Jac}_\mathcal{C}$, it must be difficult to determine x. Computing x corresponds in fact to computing $\mathrm{Jac}_\mathcal{C}$ explicitly with an isomorphism to an abstract product of finite cyclic groups; in this sense, the DLP amounts to computing the class group in the function field setting.

For any integer n, the *Weil pairing* e_n on $\mathcal{C}$ is a function that takes as input two elements of order n of $\mathrm{Jac}_\mathcal{C}$ and maps them into the multiplicative group of a finite field extension $\mathbb{F}_{q^k}$ with $k = k(n)$ depending on n. It is bilinear in both its arguments, which allows to transport the DLP from a curve into a finite field, where it is potentially easier to solve. The *Tate-Lichtenbaum pairing*, that is more difficult to define, but more efficient to implement, has similar properties. From a constructive point of view, the last few years have seen a wealth of cryptosystems with attractive novel properties relying on pairings.

For a random curve, the parameter k usually becomes so big that the result of a pairing cannot even be output any more. One of the major algorithmic problems related to pairings is thus the construction of curves with a given, smallish k.

3.3. Complex multiplication

Participants: Karim Belabas, Henri Cohen, Jean-Marc Couveignes, Andreas Enge, Nicolas Mascot, Enea Milio, Aurel Page, Damien Robert.

Complex multiplication provides a link between number fields and algebraic curves; for a concise introduction in the elliptic curve case, see [41], for more background material, [40]. In fact, for most curves $\mathcal{C}$ over a finite field, the endomorphism ring of $\mathrm{Jac}_\mathcal{C}$, which determines its L-function and thus its cardinality, is an order in a special kind of number field K, called *CM field*. The CM field of an elliptic curve is an imaginary-quadratic field $\mathbb{Q}(\sqrt{D})$ with $D < 0$, that of a hyperelliptic curve of genus g is an imaginary-quadratic extension of a totally real number field of degree g. Deuring's lifting theorem ensures that $\mathcal{C}$ is the reduction modulo some prime of a curve with the same endomorphism ring, but defined over the *Hilbert class field* H_K of K.

Algebraically, H_K is defined as the maximal unramified abelian extension of K; the Galois group of H_K/K is then precisely the class group Cl_K. A number field extension H/K is called *Galois* if $H \simeq K[X]/(f)$ and H contains all complex roots of f. For instance, $\mathbb{Q}(\sqrt{2})$ is Galois since it contains not only $\sqrt{2}$, but also the second root $-\sqrt{2}$ of X^2-2, whereas $\mathbb{Q}(\sqrt[3]{2})$ is not Galois, since it does not contain the root $e^{2\pi i/3}\sqrt[3]{2}$ of X^3-2. The *Galois group* $\mathrm{Gal}_{H/K}$ is the group of automorphisms of H that fix K; it permutes the roots of f. Finally, an *abelian* extension is a Galois extension with abelian Galois group.

Analytically, in the elliptic case H_K may be obtained by adjoining to K the *singular value* $j(\tau)$ for a complex valued, so-called *modular* function j in some $\tau \in \mathcal{O}_K$; the correspondence between $\mathrm{Gal}_{H/K}$ and Cl_K allows to obtain the different roots of the minimal polynomial f of $j(\tau)$ and finally f itself. A similar, more involved construction can be used for hyperelliptic curves. This direct application of complex multiplication yields algebraic curves whose L-functions are known beforehand; in particular, it is the only possible way of obtaining ordinary curves for pairing-based cryptosystems.

The same theory can be used to develop algorithms that, given an arbitrary curve over a finite field, compute its L-function.

A generalisation is provided by *ray class fields*; these are still abelian, but allow for some well-controlled ramification. The tools for explicitly constructing such class fields are similar to those used for Hilbert class fields.

POLSYS Project-Team

3. Scientific Foundations

3.1. Introduction

Polynomial system solving is a fundamental problem in Computer Algebra with many applications in cryptography, robotics, biology, error correcting codes, signal theory, ... Among all available methods for solving polynomial systems, computation of Gröbner bases remains one of the most powerful and versatile method since it can be applied in the continuous case (rational coefficients) as well as in the discrete case (finite fields). Gröbner bases is also a building blocks for higher level algorithms who compute real sample points in the solution set of polynomial systems, decide connectivity queries and quantifier elimination over the reals. The major challenge facing the designer or the user of such algorithms is the intrinsic exponential behaviour of the complexity for computing Gröbner bases. The current proposal is an attempt to tackle these issues in a number of different ways: improve the efficiency of the fundamental algorithms (even when the complexity is exponential), develop high performance implementation exploiting parallel computers, and investigate new classes of structured algebraic problems where the complexity drops to polynomial time.

3.2. Fundamental Algorithms and Structured Systems

Participants: Jean-Charles Faugère, Mohab Safey El Din, Elias Tsigaridas, Guénaël Renault, Dongming Wang, Jérémy Berthomieu, Pierre-Jean Spaenlehauer, Chenqi Mou, Jules Svartz, Louise Huot, Thibault Verron.

Efficient algorithms F_4/F_5 [1] for computing the Gröbner basis of a polynomial system rely heavily on a connection with linear algebra. Indeed, these algorithms reduce the Gröbner basis computation to a sequence of Gaussian eliminations on several submatrices of the so-called Macaulay matrix in some degree. Thus, we expect to improve the existing algorithms by
(i) developing dedicated linear algebra routines performing the Gaussian elimination steps: this is precisely the objective 2 described below;
(ii) generating smaller or simpler matrices to which we will apply Gaussian elimination.
We describe here our goals for the latter problem. First, we focus on algorithms for computing a Gröbner basis of *general polynomial systems*. Next, we present our goals on the development of dedicated algorithms for computing Gröbner bases of *structured polynomial systems* which arise in various applications.

Algorithms for general systems. Several degrees of freedom are available to the designer of a Gröbner basis algorithm to generate the matrices occurring during the computation. For instance, it would be desirable to obtain matrices which would be almost triangular or very sparse. Such a goal can be achieved by considering various interpretations of the F_5 algorithm with respect to different monomial orderings. To address this problem, the tight complexity results obtained for F_5 will be used to help in the design of such a general algorithm. To illustrate this point, consider the important problem of solving boolean polynomial systems; it might be interesting to preserve the sparsity of the original equations and, at the same time, using the fact that overdetermined systems are much easier to solve.

Algorithms dedicated to *structured* polynomial systems. A complementary approach is to exploit the structure of the input polynomials to design specific algorithms. Very often, problems coming from applications are not random but are highly structured. The specific nature of these systems may vary a lot: some polynomial systems can be sparse (when the number of terms in each equation is low), overdetermined (the number of the equations is larger than the number of variables), invariants by the action of some finite groups, multi-linear (each equation is linear w.r.t. to one block of variables) or more generally multihomogeneous. In each case, the ultimate goal is to identify large classes of problems whose theoretical/practical complexity drops and to propose in each case dedicated algorithms.

[1] J.-C. Faugère. *A new efficient algorithm for computing Gröbner bases without reduction to zero (F5).* In Proceedings of ISSAC '02, pages 75-83, New York, NY, USA, 2002. ACM.

3.3. Solving Systems over the Reals and Applications.

Participants: Mohab Safey El Din, Daniel Lazard, Elias Tsigaridas, Pierre-Jean Spaenlehauer, Aurélien Greuet, Simone Naldi.

We will develop algorithms for solving polynomial systems over complex/real numbers. Again, the goal is to extend significantly the range of reachable applications using algebraic techniques based on Gröbner bases and dedicated linear algebra routines. Targeted application domains are global optimization problems, stability of dynamical systems (e.g. arising in biology or in control theory) and theorem proving in computational geometry.

The following functionalities shall be requested by the end-users:
(i) deciding the emptiness of the real solution set of systems of polynomial equations and inequalities,
(ii) quantifier elimination over the reals or complex numbers,
(iii) answering connectivity queries for such real solution sets.
We will focus on these functionalities.

We will develop algorithms based on the so-called critical point method to tackle systems of equations and inequalities (problem *(i)*) . These techniques are based on solving 0-dimensional polynomial systems encoding "critical points" which are defined by the vanishing of minors of jacobian matrices (with polynomial entries). Since these systems are highly structured, the expected results of Objective 1 and 2 may allow us to obtain dramatic improvements in the computation of Gröbner bases of such polynomial systems. This will be the foundation of practically fast implementations (based on singly exponential algorithms) outperforming the current ones based on the historical Cylindrical Algebraic Decomposition (CAD) algorithm (whose complexity is doubly exponential in the number of variables). We will also develop algorithms and implementations that allow us to analyze, at least locally, the topology of solution sets in some specific situations. A long-term goal is obviously to obtain an analysis of the global topology.

3.4. Low level implementation and Dedicated Algebraic Computation and Linear Algebra.

Participants: Jean-Charles Faugère, Christian Eder, Elias Tsigaridas, F. Martani.

Here, the primary objective is to focus on *dedicated* algorithms and software for the linear algebra steps in Gröbner bases computations and for problems arising in Number Theory. As explained above, linear algebra is a key step in the process of computing efficiently Gröbner bases. It is then natural to develop specific linear algebra algorithms and implementations to further strengthen the existing software. Conversely, Gröbner bases computation is often a key ingredient in higher level algorithms from Algebraic Number Theory. In theses cases, the algebraic problems are very particular and specific. Hence dedicated Gröbner bases algorithms and implementations would provide a better efficiency.

Dedicated linear algebra tools. FGB is an efficient library for Gröbner bases computations which can be used, for instance, via MAPLE. However, the library is sequential. A goal of the project is to extend its efficiency to new trend parallel architectures such as clusters of multi-processor systems in order to tackle a broader class of problems for several applications. Consequently, our first aim is to provide a durable, long term software solution, which will be the successor of the existing FGB library. To achieve this goal, we will first develop a high performance linear algebra package (under the LGPL license). This could be organized in the form of a collaborative project between the members of the team. The objective is not to develop a general library similar to the LINBOX project but to propose a dedicated linear algebra package taking into account the specific properties of the matrices generated by the Gröbner bases algorithms. Indeed these matrices are sparse (the actual sparsity depends strongly on the application), almost block triangular and not necessarily of full rank. Moreover, most of the pivots are known at the beginning of the computation. In practice, such matrices are huge (more than 10^6 columns) but taking into account their shape may allow us to speed up the computations by one or several orders of magnitude. A variant of a Gaussian elimination algorithm together with a corresponding C implementation has been presented. The main peculiarity is the order in which the operations are performed. This will be the kernel of the new linear library that will be developed.

Fast linear algebra packages would also benefit to the transformation of a Gröbner basis of a zero–dimensional ideal with respect to a given monomial ordering into a Gröbner basis with respect to another ordering. In the generic case at least, the change of ordering is equivalent to the computation of the minimal polynomial of a so-called multiplication matrix. By taking into account the sparsity of this matrix, the computation of the Gröbner basis can be done more efficiently using variant of the Wiedemann algorithm. Hence, our goal is also to obtain a dedicated high performance library for transforming (i.e. change ordering) Gröbner bases.

Dedicated algebraic tools for Algebraic Number Theory. Recent results in Algebraic Number Theory tend to show that the computation of Gröbner bases is a key step toward the resolution of difficult problems in this domain [2]. Using existing resolution methods is simply not enough to solve relevant problems. The main algorithmic lock to overcome is to adapt the Gröbner basis computation step to the specific problems. Typically, problems coming from Algebraic Number Theory usually have a lot of symmetries or the input systems are very structured. This is the case in particular for problems coming from the algorithmic theory of Abelian varieties over finite fields [3] where the objects are represented by polynomial system and are endowed with intrinsic group actions. The main goal here is to provide dedicated algebraic resolution algorithms and implementations for solving such problems. We do not restrict our focus on problems in positive characteristic. For instance, tower of algebraic fields can be viewed as triangular sets; more generally, related problems (e.g. effective Galois theory) which can be represented by polynomial systems will receive our attention. This is motivated by the fact that, for example, computing small integer solutions of Diophantine polynomial systems in connection with Coppersmith's method would also gain in efficiency by using a dedicated Gröbner bases computations step.

3.5. Solving Systems in Finite Fields, Applications in Cryptology and Algebraic Number Theory.

Participants: Jean-Charles Faugère, Ludovic Perret, Guénaël Renault, Louise Huot, Frédéric de Portzamparc, Rina Zeitoun.

Here, we focus on solving polynomial systems over finite fields (i.e. the discrete case) and the corresponding applications (Cryptology, Error Correcting Codes, ...). Obviously this objective can be seen as an application of the results of the two previous objectives. However, we would like to emphasize that it is also the source of new theoretical problems and practical challenges. We propose to develop a systematic use of *structured systems* in *algebraic cryptanalysis*.
(i) So far, breaking a cryptosystem using algebraic techniques could be summarized as modeling the problem by algebraic equations and then computing a, usually, time consuming Gröbner basis. A new trend in this field is to require a theoretical complexity analysis. This is needed to explain the behavior of the attack but also to help the designers of new cryptosystems to propose actual secure parameters.
(ii) To assess the security of several cryptosystems in symmetric cryptography (block ciphers, hash functions, ...), a major difficulty is the size of the systems involved for this type of attack. More specifically, the bottleneck is the size of the linear algebra problems generated during a Gröbner basis computation.

We propose to develop a systematic use of *structured systems* in *algebraic cryptanalysis*.

[2] P. Gaudry, *Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem*, Journal of Symbolic Computation 44,12 (2009) pp. 1690-1702
[3] e.g. point counting, discrete logarithm, isogeny.

The first objective is to build on the recent breakthrough in attacking McEliece's cryptosystem: it is the first structural weakness observed on one of the oldest public key cryptosystem. We plan to develop a well founded framework for assessing the security of public key cryptosystems based on coding theory from the algebraic cryptanalysis point of view. The answer to this issue is strongly related to the complexity of solving bihomogeneous systems (of bidegree $(1, d)$). We also plan to use the recently gained understanding on the complexity of structured systems in other areas of cryptography. For instance, the MinRank problem – which can be modeled as an overdetermined system of bilinear equations – is at the heart of the structural attack proposed by Kipnis and Shamir against HFE (one of the most well known multivariate public cryptosystem). The same family of structured systems arises in the algebraic cryptanalysis of the Discrete Logarithmic Problem (DLP) over curves (defined over some finite fields). More precisely, some bilinear systems appear in the polynomial modeling the points decomposition problem. Moreover, in this context, a natural group action can also be used during the resolution of the considered polynomial system.

Dedicated tools for linear algebra problems generated during the Gröbner basis computation will be used in algebraic cryptanalysis. The promise of considerable algebraic computing power beyond the capability of any standard computer algebra system will enable us to attack various cryptosystems or at least to propose accurate secure parameters for several important cryptosystems. Dedicated linear tools are thus needed to tackle these problems. From a theoretical perspective, we plan to further improve the theoretical complexity of the hybrid method and to investigate the problem of solving polynomial systems with noise, i.e. some equations of the system are incorrect. The hybrid method is a specific method for solving polynomial systems over finite fields. The idea is to mix exhaustive search and Gröbner basis computation to take advantage of the over-determinacy of the resulting systems.

Polynomial system with noise is currently emerging as a problem of major interest in cryptography. This problem is a key to further develop new applications of algebraic techniques; typically in side-channel and statistical attacks. We also emphasize that recently a connection has been established between several classical lattice problems (such as the Shortest Vector Problem), polynomial system solving and polynomial systems with noise. The main issue is that there is no sound algorithmic and theoretical framework for solving polynomial systems with noise. The development of such framework is a long-term objective.

SECRET Project-Team

3. Scientific Foundations

3.1. Scientific foundations

Our research work is mainly devoted to the design and analysis of cryptographic algorithms. Our approach on the previous problems relies on a competence whose impact is much wider than cryptology. Our tools come from information theory, discrete mathematics, probabilities, algorithmics... Most of our work mix fundamental aspects (study of mathematical objects) and practical aspects (cryptanalysis, design of algorithms, implementations). Our research is mainly driven by the belief that discrete mathematics and algorithmics of finite structures form the scientific core of (algorithmic) data protection.

VEGAS Project-Team (section vide)

ALF Project-Team

3. Scientific Foundations

3.1. Motivations

Multicores have become mainstream in general-purpose as well as embedded computing in the last few years. The integration technology trend allows to anticipate that a 1000-core chip will become feasible before 2020. On the other hand, while traditional parallel application domains, e.g. supercomputing and transaction servers, are benefiting from the introduction of multicores, there are very few new parallel applications that have emerged during the last few years.

In order to allow the end-user to benefit from the technological breakthrough, new architectures have to be defined for the 2020's many-cores, new compiler and code generation techniques as well as new performance prediction/guarantee techniques have to be proposed .

3.2. The context

3.2.1. Technological context: The advent of multi- and many- cores architecture

For almost 30 years since the introduction of the first microprocessor, the processor industry was driven by the Moore's law till 2002, delivering performance that doubled every 18-24 months on a uniprocessor. However since 2002 , and despite new progress in integration technology, the efforts to design very aggressive and very complex wide issue superscalar processors have essentially been stopped due to poor performance returns, as well as power consumption and temperature walls.

Since 2002-2003, the microprocessor industry has followed a new path for performance: the so-called multicore approach, i.e., integrating several processors on a single chip. This direction has been followed by the whole processor industry. At the same time, most of the computer architecture research community has taken the same path, focusing on issues such as scalability in multicores, power consumption, temperature management and new execution models, e.g. hardware transactional memory.

In terms of integration technology, the current trend will allow to continue to integrate more and more processors on a single die. Doubling the number of cores every two years will soon lead to up to a thousand processor cores on a single chip. The computer architecture community has coined these future processor chips as many-cores.

3.2.2. The application context: multicores, but few parallel applications

For the past five years, small scale parallel processor chips (hyperthreading, dual and quad-core) have become mainstream in general-purpose systems. They are also entering the high-end embedded system market. At the same time, very few (scalable) mainstream parallel applications have been developed. Such development of scalable parallel applications is still limited to niche market segments (scientific applications, transaction servers).

3.2.3. The overall picture

Till now, the end-user of multicores is experiencing improved usage comfort because he/she is able to run several applications at the same time. Eventually, in the near future with the 8-core or the 16-core generation, the end-user will realize that he/she is not experiencing any functionality improvement or performance improvement on current applications. The end-user will then realize that he/she needs more effective performance rather than more cores. The end-user will then ask either for parallel applications or for more effective performance on sequential applications.

3.3. Technology induced challenges

3.3.1. The power and temperatures walls

The power and the temperature walls largely contributed to the emergence of the small-scale multicores. For the past five years, mainstream general-purpose multicores have been built by assembling identical superscalar cores on a chip (e.g. IBM Power series). No new complex power hungry mechanisms were introduced in the core architectures, while power saving techniques such as power gating, dynamic voltage and frequency scaling were introduced. Therefore, since 2002, the designers have been able to keep the power consumption budget and the temperature of the chip within reasonable envelopes while scaling the number of cores with the technology.

Unfortunately, simple and efficient power saving techniques have already caught most of the low hanging fruits on energy consumption. Complex power and thermal management mechanisms are now becoming mainstream; e.g. the Intel Montecito (IA64) featured an adjunct (simple) core which unique mission is to manage the power and temperature on two cores. Processor industry will require more and more heroic efforts on this power and temperature management policy to maintain its current performance scaling path. Hence the power and temperature walls might slow the race towards 100's and 1000's cores unless the processor industry takes a new paradigm shift from the current "replicating complex cores" (e.g. Intel Nehalem) towards many simple cores (e.g. Intel Larrabee) or heterogeneous manycores (e.g. new GPUs, IBM Cell).

3.3.2. The memory wall

For the past 20 years, the memory access time has been one of the main bottlenecks for performance in computer systems. This was already true for uniprocessors. Complex memory hierarchies have been defined and implemented in order to limit the visible memory access time as well as the memory traffic demands. Up to three cache levels are implemented for uniprocessors. For multi- and many-cores the problems are even worse. The memory hierarchy must be replicated for each core, memory bandwidth must be shared among the distinct cores, data coherency must be maintained. Maintaining cache coherency for up to 8 cores can be handled through relatively simple bus protocols. Unfortunately, these protocols do not scale for large numbers of cores, and there is no consensus on coherency mechanism for manycore systems. Moreover there is no consensus on core organization (flat ring? flat grid? hierarchical ring or grid?).

Therefore, organizing and dimensioning the memory hierarchy will be a major challenge for the computer architects. The successful architecture will also be determined by the abilitty of the applications (i.e., the programmers or the compilers or the run-time) to efficiently place data in the memory hierarchy and achieve high performance.

Finally new technology opportunities may demand to revisit the memory hierarchy. As an example, 3D memory stacking enables a huge last-level cache (maybe several gigabytes) with huge bandwidth (several Kbits/ processor cycle). This dwarfs the main memory bandwidth and may lead to other architectural tradeoffs.

3.4. Need for efficient execution of parallel applications

Achieving high performance on future multicores will require the development of parallel applications, but also an efficient compiler/runtime tool chain to adapt codes to the execution platform.

3.4.1. The diversity of parallelisms

Many potential execution parallelism patterns may coexist in an application. For instance, one can express some parallelism with different tasks achieving different functionalities. Within a task, one can expose different granularities of parallelism; for instance a first layer message passing parallelism (processes executing the same functionality on different parts of the data set), then a shared memory thread level parallelism and fine grain loop parallelism (a.k.a vector parallelism).

Current multicores already feature hardware mechanisms to address these different parallelisms: physically distributed memory — e.g. the new Intel Nehalem already features 6 different memory channels — to address task parallelism, thread level parallelism — e.g. on conventional multicores, but also on GPUs or on Cell-based machines —, vector/SIMD parallelism — e.g. multimedia instructions. Moreover they also attack finer instruction level parallelism and memory latency issues. Compilers have to efficiently discover and manage all these forms to achieve effective performance.

3.4.2. Portability is the new challenge

Up to now, most parallel applications were developed for specific application domains in high end computing. They were used on a limited set of very expensive hardware platforms by a limited number of expert users. Moreover, they were executed in batch mode.

In contrast, the expectation of most end-users of the future mainstream parallel applications running on multicores will be very different. The mainstream applications will be used by thousands, maybe millions of non-expert users. These users consider functional portability of codes as granted. They will expect their codes to run faster on new platforms featuring more cores. They will not be able to tune the application environment to optimize performance. Finally, multiple parallel applications may have to be executed concurrently.

The variety of possible hardware platforms, the lack of expertise of the end-users and the varying run-time execution environments will represent major difficulties for applications in the multicore era.

First of all, the end user considers functional portability without recompilation as granted, this is a major challenge on parallel machines. Performance portability/scaling is even more challenging. It will become inconceivable to rewrite/retune each application for each new parallel hardware platform generation to exploit them. Therefore, apart from the initial development of parallel applications, the major challenge for the next decade will be to *efficiently* run parallel applications on hardware architectures radically different from their original hardware target.

3.4.3. The need for performance on sequential code sections

3.4.3.1. Most software will exhibit substantial sequential code sections

For the foreseeable future, the majority of applications will feature important sequential code sections.

First, many legacy codes were developed for uniprocessors. Most of these codes will not be completely redeveloped as parallel applications, but will evolve to applications using parallel sections for the most compute-intensive parts. Second, the overwhelming majority of the programmers have been educated to program in a sequential programming style. Parallel programming is much more difficult, time consuming and error prone than sequential programming. Debugging and maintaining a parallel code is a major issue. Investing in the development of a parallel application will not be cost-effective for the vast majority of software developments. Therefore, sequential programming style will continue to be dominant in the foreseeable future. Most developers will rely on the compiler to parallelize their application and/or use some software components from parallel libraries.

3.4.3.2. Future parallel applications will require high performance sequential processing on 1000's cores chip

With the advent of universal parallel hardware in multicores, large diffusion parallel applications will have to run on a broad spectrum of parallel hardware platforms. They will be used by non-expert users who will not be able to tune the application environment to optimize performance. They will be executed concurrently with other processes which may be interactive.

The variety of possible hardware platforms, the lack of expertise of the end-user and the varying run-time execution environments are major difficulties for parallel applications. This tends to constrain the programming style and therefore reinforces the sequential structure of the control of the application.

Therefore, *most future parallel applications will rely on a single main thread or a few main threads in charge of distinct functionalities of the application. Each main thread will have a general sequential control and can initiate and control the parallel execution of parallel tasks.*

In 1967, Amdahl [34] pointed out that, if only a portion of an application is accelerated, the execution time cannot be reduced below the execution time of the residual part of the application. Unfortunately, even highly parallelized applications exhibit some residual sequential part. For parallel applications, this indicates that the effective performance of the future 1000's cores chip will significantly depend on their ability to be efficient on the execution of the control portions of the main thread as well as on the execution of sequential portions of the application.

3.4.3.3. The success of 1000's cores architecture will depend on single thread performance

While the current emphasis of computer architecture research is on the definition of scalable multi- many- core architectures for highly parallel applications, we believe that the success of the future 1000-core architecture will depend not only on their performance on parallel applications including sequential sections, but also on their performance on single thread workloads.

3.5. Performance evaluation/guarantee

Predicting/evaluating the performance of an application on a system without explicitly executing the application on the system is required for several usages. Two of these usages are central to the research of the ALF project-team: microarchitecture research (the system to be be evaluated does not exist) and Worst Case Execution Time estimation for real-time systems (the numbers of initial states or possible data inputs is too large).

When proposing a micro-architecture mechanism, its impact on the overall processor architecture has to be evaluated in order to assess its potential performance advantages. For microarchitecture research, this evaluation is generally done through the use of cycle-accurate simulation. Developing such simulators is quite complex and microarchitecture research was helped but also biased by some popular public domain research simulators (e.g. Simplescalar [36]). Such simulations are CPU consuming and simulations cannot be run on a complete application. Sampling representative slices of the application was proposed [5] and popularized by the Simpoint [45] framework.

Real-time systems need a different use of performance prediction; on hard real-time systems, timing constraints must be respected independently from the data inputs and from the initial execution conditions. For such a usage, the Worst Case Execution Time (WCET) of an application must be evaluated and then checked against the timing constraints. While safe and tight WCET estimation techniques and tools exist for reasonably simple embedded processors (e.g. techniques based on abstract interpretation such as [38]), accurate evaluation of the WCET of an algorithm on a complex uniprocessor system is a difficult problem. Accurately modelling data cache behavior [4] and complex superscalar pipelines are still research questions as illustrated by the presence of so-called *timing anomalies* in dynamically scheduled processors, resulting from complex interactions between processor elements (among others, interactions between caching and instruction scheduling) [42].

With the advance of multicores, evaluating / guaranteeing a computer system response time is becoming much more difficult. Interactions between processes occurs at different levels. The execution time on each core depends on the behavior of the other cores. Simulations of 1000's cores micro-architecture will be needed in order to evaluate future many-core proposals. While a few multiprocessor simulators are available for the community, these simulators cannot handle realistic 1000's cores micro-architecture. New techniques have to be invented to achieve such simulations. WCET estimations on multicore platforms will also necessitate radically new techniques, in particular, there are predictability issues on a multicore where many resources are shared; those resources include the memory hierarchy, but also the processor execution units and all the hardware resources if SMT is implemented [49].

3.6. General research directions

The overall performance of a 1000's core system will depend on many parameters including architecture, operating system, runtime environment, compiler technology and application development. In the ALF project, we will essentially focus on architecture, compiler/execution environment as well as performance

predictability, and in particular WCET estimation. Moreover, architecture research, and to a smaller extent, compiler and WCET estimation researches rely on processor simulation. A significant part of the effort in ALF will be devoted to define new processor simulation techniques.

3.6.1. Microarchitecture research directions

The overall performance of a multicore system depends on many parameters including architecture, operating system, runtime environment, compiler technology and application development. Even the architecture dimension of a 1000's core system cannot be explored by a single research project. Many research groups are exploring the parallel dimension of the multicores essentially targeting issues such as coherency and scalability.

We have identified that high performance on single threads and sequential codes is one of the key issues for enabling overall high performance on a 1000's core system and we anticipate that the general architecture of such 1000's core chip will feature many simple cores and a few very complex cores.

Therefore our research in the ALF project will focus on refining the microarchitecture to achieve high performance on single process and/or sequential code sections within the general framework of such an heteregeneous architecture. This leads to two main research directions 1) enhancing the microarchitecture of high-end superscalar processors, 2) exploiting/modifying heterogeneous multicore architecture on a single process. The temperature wall is also a major technological/architectural issue for the design of future processor chips.

3.6.1.1. Enhancing complex core microarchitecture

Research on wide issue superscalar processors was merely stopped around 2002 due to limited performance returns and the power consumption wall.

When considering a heterogeneous architecture featuring hundreds of simple cores and a few complex cores, these two obstacles will partially vanish: 1) the complex cores will represent only a fraction of the chip and a fraction of its power consumption. 2) any performance gain on (critical) sequential threads will result in a performance gain of the whole system

On the complex core, the performance of a sequential code is limited by several factors. At first, on current architectures, it is limited by the peak performance of the processor. To push back this first limitation, we will explore new microarchitecture mechanisms to increase the potential peak performance of a complex core enabling larger instruction issue width. The processor performance is also limited by control dependencies. To push back this limitation, we will explore new branch prediction mechanisms as well as new directions for reducing branch misprediction penalties [14], [13]. As data dependencies may strongly limit performance, we will revisit data prediction. Processor performance is also often highly dependent on the presence or absence of data in a particular level of the memory hierarchy. For the ALF multicore, we will focus on sharing the access to the memory hierarchy in order to adapt the performance of the main thread to the performance of the other cores. All these topics should be studied with the new perspective of quasi unlimited silicon budget.

3.6.1.2. Exploiting heterogeneous multicores on single process

When executing a sequential section on the complex core, the simple cores will be free. Two main research directions to exploit thread level parallelism on a sequential thread have been initiated in late 90's within the context of simultaneous multithreading and early chip multiprocessor proposals: helper threads and speculative multithreading.

Helper threads were initially proposed to improve the performance of the main threads on simultaneous multithreaded architectures [37]. The main idea of helper threads is to execute codes that will accelerate the main thread without modifying its semantic.

In many cases, the compiler cannot determine if two code sections are independent due to some unresolved memory dependency. When no dependency occurs at execution time, the code sections can be executed in parallel. Thread-Level Speculation has been proposed to exploit coarse grain speculative parallelism. Several hardware-only proposals were presented [44], but the most promising solutions integrate hardware support for software thread-level speculation [47].

In the context of future manycores, thread-level speculation and helper threads should be revisited. Many simple cores will be available for executing helper threads or speculative thread execution during the execution of sequential programs or sequential code sections. The availability of these many cores is an opportunity as well as a challenge. For example, one can try to use the simple cores to execute many different helper threads that could not be implemented within a simultaneous multithreaded processor. For thread level speculation, the new challenge is the use of less powerful cores for speculative threads. Moreover the availability of many simple cores may lead to the use of helper threads and thread level speculation at the same time.

3.6.1.3. Temperature issues

Temperature is one of the constraints that have prevented the processor clock frequency to be increased in recent years. Besides techniques to decrease the power consumption, the temperature issue can be tackled with *dynamic thermal management* [10] through techniques such as clock gating or throttling and *activity migration* [46][7].

Dynamic thermal management (DTM) is now implemented on existing processors. For high performance, processors are dimensioned according to the average situation rather than to the worst case situation. Temperature sensors are used on the chip to trigger dynamic thermal management actions, for instance thermal throttling whenever necessary. On multicores, it is possible to migrate the activity from one core to another in order to limit temperature.

A possible way to increase sequential performance is to take advantage of the smaller gate delay that comes with miniaturization, which permits in theory to increase the clock frequency. However increasing the clock frequency generally requires to increase the instantaneous power density. This is why DTM and activity migration will be key techniques to deal with Amdahl's law in future many-core processors.

3.6.2. Processor simulation research

Architecture studies, and in particular microarchitecture studies, require extensive validations through detailed simulations. Cycle accurate simulators are needed to validate the microarchitectural mechanisms.

Within the ALF project, we can distinguish two major requirements on the simulation: 1) single process and sequential code simulations 2) parallel code sections simulations.

For simulating parallel code sections, a cycle-accurate microarchitecture simulator of a 1000-core architecture will be unacceptably slow. In [9], we showed that mixing analytical modeling of the global behavior of a processor with detailed simulation of a microarchitecture mechanism allows to evaluate this mechanism. Karkhanis and Smith [39] further developed a detailed analytical simulation model of a superscalar processor. Building on top of these preliminary researches, simulation methodology mixing analytical modeling of the simple cores with a more detailed simulation of the complex cores is appealing. The analytical model of the simple cores will aim at approximately modeling the impact of the simple core execution on the shared resources (e.g. data bandwidth, memory hierarchy) that are also used by the complex cores.

Other techniques such as regression modeling [40] can also be used for decreasing the time required to explore the large space of microarchitecture parameter values. We will explore these techniques in the context of many-core simulation.

In particular, research on temperature issues will require the definition and development of new simulation tools able to simulate several minutes or even hours of processor execution, which is necessary for modeling thermal effects faithfully.

3.6.3. Compiler research directions

3.6.3.1. General directions

Compilers are keystone solutions for any approach that deals with high performance on 100+ processors systems. But general-purpose compilers try to embrace so many domains and try to serve so many constraints that they frequently fail to achieve very high performance. They need to be deeply revisited. We identify four main compiler/software related issues that must be addressed in order to allow efficient use of multi- and many-cores: 1) programming 2) resource management 3) application deployment 4) portable performance. Addressing these challenges will require to revisit parallel programming and code generation extensively.

The past of parallel programming is scattered with hundreds of parallel languages. Most of these languages were designed to program homogeneous architectures and were targeting a small and well-trained community of HPC programmers. With the new diversity of parallel hardware platforms and the new community of non-expert developers, expressing parallelism is not sufficient anymore. Resource management, application deployment and portable performance are intermingled issues that require to be addressed holistically.

As many decisions should be taken according to the available hardware, resource management cannot be separated from parallel programming. Deploying applications on various systems without having to deal with thousands of hardware configurations (different numbers of cores, accelerators, ...) will become a major concern for software distribution. The grail of parallel computing is to be able to provide portable performance on a large set of parallel machines and varying execution contexts.

Recent techniques are showing promises. Iterative compilation techniques, exploiting the huge CPU cycle count now available, can be used to explore the optimization space at compile-time. Second, machine-learning techniques can be used to automatically improve compilers and code generation strategies. Speculation can be used to deal with necessary but missing information at compile-time. Finally, dynamic techniques can select or generate at run-time the most efficient code adapted to the execution context and available hardware resources.

Future compilers will benefit from past research, but they will also need to combine static and dynamic techniques. Moreover, domain specific approaches might be needed to ensure success. The ALF research effort will focus on these static and dynamic techniques to address the multicore application development challenges.

3.6.3.2. Portability of applications and performance through virtualization

The life cycle is much longer for applications than for hardware. Unfortunately the multicore era jeopardizes the old binary compatibility recipe. Binaries cannot automatically exploit additional computing cores or new accelerators available on the silicon. Moreover maintaining backward binary compatibility on future parallel architectures will rapidly become a nightmare, applications will not run at all unless some kind of dynamic binary translation is at work.

Processor virtualization addresses the problem of portability of functionalities. Applications are not compiled to the final native code but to a target independent format. This is the purpose of languages such as Java and .NET. Bytecode formats are often *a priori* perceived as inappropriate for performance intensive applications and for embedded systems. However, it was shown that compiling a C or C++ program to a bytecode format produces a code size similar to dense instruction sets [3]. Moreover, this bytecode representation can be compiled to native code with performance similar to static compilation [2]. Therefore processor virtualization for high performance, i.e., for languages like C or C++, provides significant advantages: 1) it simplifies software engineering with fewer tools to maintain and upgrade; 2) it allows better code readability and easier code maintenancesince it avoids code specialization for specific targets using compile time macros such as #ifdef ; 3) the *execution code* deployed on the system is the execution code that has been debugged and validated, as opposed to the same *source code* has been recompiled for another platform; 4) new architectures will come with their JIT compiler. The JIT will (should) automatically take advantage of new architecture features such as SIMD/vector instructions or extra processors.

Our objective is to enrich processor virtualization to allow both functional portability and high performance using JIT at runtime, or bytecode-to-native code offline compiler. Split compilation can be used to annotate the bytecode with relevant information that can be helpful to the JIT at runtime or to the bytecode to native code offline compiler. Because the first compilation pass occurs offline, aggressive analyses can be run and their outcomes encoded in the bytecode. For example, such informations include vectorizability, memory references (in)dependencies, suggestions derived from iterative compilation, polyhedral analysis, or integer linear programming. Virtualization allows to postpone some optimizations to run time, either because they increase the code size and would increase the cost of an embedded system or because the actual hardware platform characteristics are unknown.

3.6.4. Performance predictability for real-time systems

While compiler and architecture research efforts often focus on maximizing average case performance, applications with real-time constraints do not need onlyhigh performance but also performance guarantees in all situations, including the worst-case situation. Worst-Case Execution Time estimates (WCET) need to be upper bounds of any possible execution time. The safety level required depends on the criticality of applications: missing a frame on a video in the airplane for passenger in seat 20B is less critical than a safety critical decision in the control of the airplane.

Within the ALF project, our objective is to study performance guarantees for both (i) sequential codes running on complex cores ; (ii) parallel codes running on the multicores. Considering the ALF base architecture, this results in two quite distinct problems.

For sequential code executing on a single core, one can expect that, in order to provide real-time possibility, the architecture will feature an execution mode where a given processor will be guaranteed to access a fixed portion of the shared resources (caches, memory bandwidth). Moreover, this guaranteed share could be optimized at compile time to enforce the respect of the time constraints. However, estimating the WCET of an application on a complex micro-architecture is still a research challenge. This is due to the complex interaction of micro-architectural elements (superscalar pipelines, caches, branch prediction, out-of-order execution) [42]. We will continue to explore pure analytical and static methods. However when accurate static hardware modeling methods cannot handle the hardware complexity, new probabilistic methods [41] might be needed to explore to obtain as safe as possible WCET estimates.

Providing performance guarantees for parallel applications executed on a multicore is a new and challenging issue. Entirely new WCET estimation methods have to be defined for these architectures to cope with dynamic resource sharing between cores, in particular on-chip memory (either local memory or caches) are shared, but also buses, network-on-chip and the access to the main memory. Current pure analytical methods are too pessimistic at capturing interferences between cores [50], therefore hardware-based or compiler methods such as [48] have to be defined to provide some degree of isolation between cores. Finally, similarly to simulation methods, new techniques to reduce the complexity of WCET estimation will be explored to cope with manycore architectures.

CAIRN Project-Team

3. Scientific Foundations

3.1. Panorama

The development of complex applications is traditionally split in three stages: a theoretical study of the algorithms, an analysis of the target architecture and the implementation. When facing new emerging applications such as high-performance, low-power and low-cost mobile communication systems or smart sensor-based systems, it is mandatory to strengthen the design flow by a joint study of both algorithmic and architectural issues [1].

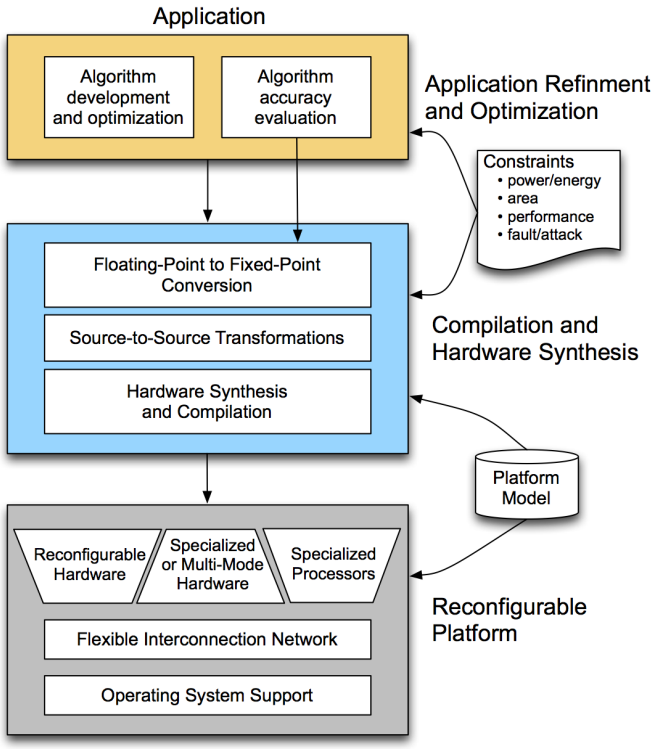


Figure 1. CAIRN*'s general design flow and related research themes*

Figure 1 shows the global design flow that we propose to develop. This flow is organized in levels which refer to our three research themes: application optimization (new algorithms, fixed-point arithmetic and advanced representations of numbers), architecture optimization (reconfigurable and specialized hardware, application-specific processors), and stepwise refinement and code generation (code transformations, hardware synthesis, compilation).

[1] Often referenced as algorithm-architecture mapping or interaction.

In the rest of this part, we briefly describe the challenges concerning **new reconfigurable platforms** in Section 3.2 , the issues on **compiler and synthesis tools** related to these platforms in Section 3.3 , and the remaining challenges in **algorithm architecture interaction** in Section 3.4 .

3.2. Reconfigurable Architecture Design

Over the last two decades, there has been a strong push of the research community to evolve static programmable processors into run-time dynamic and partial reconfigurable (DPR) architectures. Several research groups around the world have hence proposed reconfigurable hardware systems operating at various levels of granularity. For example, functional-level reconfiguration has been proposed to increase the efficiency of programmable processors without having to pay for the FPGAs penalties. These coarse-grained reconfigurable architectures (CGRAs) provide operator-level configurable functional blocks and word-level datapaths. The main goal of this class of architectures is to provide flexibility while minimizing reconfiguration overhead (there exists several recent surveys on this topic [120], [104], [85], [125]). Compared to fine-grained architectures, CGRAs benefit from a massive reduction in configuration memory and configuration delay, as well as a considerable reduction in routing and placement complexity. This, in turns, results in an improvement in the computation volume over energy cost ratio, even if it comes at the price of a loss of flexibility compared to bit-level operations. Such constraints have been taken into account in the design of DART [100][12], CRIP [88], Adres [112] or others [122]. These works have led to commercial products such as the Extreme Processor Platform (XPP) [89] from PACT or Montium [2] from Recore systems.

Another strong trend is the design of hybrid architectures which combine standard GPP or DSP cores with arrays of *configurable elements* such as the Lx [103], or of *field-configurable elements* such as the Xirisc processor [110] and more recently by commercial platforms such as the Xilinx Zynq-7000. Some of their benefits are the following: functionality on demand (set-top boxes for digital TV equipped with decoding hardware on demand), acceleration on demand (coprocessors that accelerate computationally demanding applications in multimedia or communications applications), and shorter time-to-market (products that target ASIC platforms can be released earlier using reconfigurable hardware).
Dynamic reconfiguration enables an architecture to adapt itself to various incoming tasks. This requires complex resource management and control which can be provided as services by a real-time operating system (RTOS) [111]: communication, memory management, task scheduling [99], [92][1] and task placement [19]. Such an Operating System (OS) based approach has many advantages: it provides a complete design framework, that is independent of the technology and of the underlying hardware architecture, helping to drastically reduce the full platform design time. Due to the unpredictable execution of tasks, the OS must be able to allocate resource to tasks at run-time along with mechanisms to support inter-task communication. An efficient way to support such communications is to resort to a network-on-chip [118]. The role of the communication infrastructure is then to support transactions between different components of the platform, either between macro-components – main processor, dedicated modules, dynamically reconfigurable component – or within the elements of the reconfigurable components themselves.

In CAIRN we mainly target reconfigurable system-on-chip (RSoC) defined as a set of computing and storing resources organized around a flexible interconnection network and integrated within a single silicon chip (or programmable chip such as FPGAs). The architecture is customized for an application domain, and the flexibility is provided by both hardware reconfiguration and software programmability. Computing resources are therefore highly heterogeneous and raise many issues that we discuss in the following:

- **Reconfigurable hardware blocks with a dynamic behavior** where reconfigurability can be achieved at the bit- or operator-level. Our research aims at defining new reconfigurable architectures including computing and memory resources. Since reconfiguration must happen as fast as possible (typically within a few cycles), reducing the configuration time overhead is also a key issue.

[2] http://www.recoresystems.com/technology/montium-technology

- When performance and power consumption are major constraints, it is acknowledged that optimized specialized hardware blocks (often called IPs for Intellectual Properties) are the best (and often the only) solution. Therefore, we also study architecture and tools for **specialized hardware accelerators** and for **multi-mode components**.
- Customized **processors with a specialized instruction-set** also offer a viable solution to trade between energy efficiency and flexibility. They are particularly relevant for modern FPGA platforms where many processor cores can be embedded. For this topic, we focus on the automatic generation of heterogeneous (sequential or parallel) reconfigurable processor extensions that are tightly coupled to processor cores.

3.3. Compilation and Synthesis for Reconfigurable Platforms

In spite of their advantages, reconfigurable architectures lack efficient and standardized compilation and design tools. As of today, this still makes the technology impractical for large scale industrial use. Generating and optimizing the mapping from high-level specifications to reconfigurable hardware platforms is therefore a key research issue, and the problem has received considerable interest over the last years [115], [91], [121], [124]. In the meantime, the complexity (and heterogeneity) of these platforms has also been increasing quite significantly, with complex heterogeneous multi-cores architectures becoming a *de facto* standard. As a consequence, the focus of designers is now geared toward optimizing overall system-level performance and efficiency [106], [115], [114]. Here again, existing tools are not well suited, as they fail at providing a unified programming view of the programmable and/or reconfigurable components implemented on the platform.
In this context we have been pursuing our efforts to propose tools whose design principles are based on a tight coupling between the compiler and the target hardware architectures. We build on the expertise of the team members in High Level Synthesis (HLS) [8], ASIP optimizing compilers [15] and automatic parallelization for massively parallel specialized circuits [6]. We first study how to increase the efficiency of standard programmable processor by extending their instruction set to speed-up compute intensive kernels. Our focus is on efficient and exact algorithms for the identification, selection and scheduling of such instructions [9]. We also propose techniques to synthesize reconfigurable (or multi-mode) architectures. We address these challenges by borrowing techniques from high-level synthesis, optimizing compilers and automatic parallelization, especially when dealing with nested loop kernels. The goal is then either to derive a custom fine-grain parallel architecture and/or to derive the configuration of a Coarse Grain Reconfigurable Architecture (CGRA). In addition, and independently of the scientific challenges mentioned above, proposing such flows also poses significant software engineering issues. As a consequence, we also study how leading edge Object Oriented software engineering techniques (Model Driven Engineering) can help the Computer Aided Design (CAD) and optimizing compiler communities prototyping new research ideas.
Efficient implementation of multimedia and signal processing applications (in software for DSP cores or as special-purpose hardware) often requires, for reasons related to cost, power consumption or silicon area constraints, the use of fixed-point arithmetic, whereas the algorithms are usually specified in floating-point arithmetic. Unfortunately, fixed-point conversion is very challenging and time-consuming, typically demanding up to 50% of the total design or implementation time [93]. Thus, tools are required to automate this conversion. For hardware or software implementation, the aim is to optimize the fixed-point specification. The implementation cost is minimized under a numerical accuracy or an application performance constraint. For DSP-software implementation, methodologies have been proposed [108], [113] to achieve a conversion leading to an ANSI-C code with integer data types. For hardware implementation, the best results are obtained when the word-length optimization process is coupled with the high-level synthesis [107], [96]. Evaluating the effects of finite precision is one of the major and often the most time consuming step while performing fixed-point refinement. Indeed, in the word-length optimization process, the numerical accuracy is evaluated as soon as a new word-length is tested, thus, several times per iteration of the optimization process. Classical approaches are based on fixed-point simulations [97], [119]. They lead to long evaluation times and cannot be used to explore the entire design space. Therefore, our aim is to propose closed-form expressions of errors due to fixed-point approximations that are used by a fast analytical framework for accuracy evaluation.

3.4. Interaction between Algorithms and Architectures

As CAIRN mainly targets domain-specific system-on-chip including reconfigurable capabilities, algorithmic-level optimizations have a great potential on the efficiency of the overall system. Based on the skills and experiences in "signal processing and communications" of some CAIRN's members, we conduct research on algorithmic optimization techniques under two main constraints: energy consumption and computation accuracy; and for two main application domains: fourth-generation (4G) mobile communications and wireless sensor networks (WSN). These application domains are very conducive to our research activities. The high complexity of the first one and the stringent power constraint of the second one, require the design of specific high-performance and energy-efficient SoCs. We also consider other applications such as video or bioinformatics, but this short state-of-the-art will be limited to wireless applications.

The radio in both transmit and receive modes consumes the bulk of the total power consumption of the system. Therefore, protocol optimization is one of the main sources of significant energy reduction to be able to achieve self-powered autonomous systems. Reducing power due to radio communications can be achieved by two complementary main objectives: (i) minimizing the output transmit power while maintaining sufficient wireless link quality and (ii) minimizing useless wake-up and channel hearing while still being reactive.

As the physical layer affects all higher layers in the protocol stack, it plays an important role in the energy-constrained design of WSNs. The question to answer can be summarized as: *how much signal processing can be added to decrease the transmission energy (i.e. the output power level at the antenna) such that the global energy consumption be decreased?* The temporal and spatial diversity of relay and multiple antenna techniques are very attractive due to their simplicity and their performance for wireless transmission over fading channels. Cooperative MIMO (multiple-input and multiple-output) techniques have been first studied in [101], [109] and have shown their efficiency in terms of energy consumption [98]. Our research aims at finding new energy-efficient cooperative protocols associating distributed MIMO with opportunistic and/or multiple relays and considering wireless channel impairments such as transmitters desynchronisation.

Another way to reduce the energy consumption consists in decreasing the radio activity, controlled by the medium access (MAC) layer protocols. In this regard, low duty-cycle protocols, such as preamble-sampling MAC protocols, are very efficient because they improve the lifetime of the network by reducing the unnecessary energy waste [87]. As the network parameters (data rate, topology, etc.) can vary, we propose new adaptive MAC protocols to avoid overhearing and idle listening.

Finally, MIMO precoding is now recognized as a very interesting technique to enhance the data rate in wireless systems, and is already used in Wi-Max standard (802.16e). This technique can also be used to reduce transmission energy for the same transmission reliability and the same throughput requirement. One of the most efficient precoders is based on the maximization of the minimum Euclidean distance ($\max$-d_{min}) between two received data vectors [94], but it is difficult to define the closed-form of the optimized precoding matrix for large MIMO system with high-order modulations. Our goal is to derive new generic precoders with simple expressions depending only on the channel angle and the modulation order.

CAMUS Team

3. Scientific Foundations

3.1. Research directions

The various objectives we are expecting to reach are directly related to the search of adequacy between the sofware and the new multicore processors evolution. They also correspond to the main research directions suggested by Hall, Padua and Pingali in [43]. Performance, correction and productivity must be the users' perceived effects. They will be the consequences of research works dealing with the following issues:

- Issue 1: Static parallelization and optimization
- Issue 2: Profiling and execution behavior modeling
- Issue 3: Dynamic program parallelization and optimization, virtual machine
- Issue 4: Object-oriented programming and compiling for multicores
- Issue 5: Proof of program transformations for multicores

Efficient and correct applications development for multicore processors needs stepping in every application development phase, from the initial conception to the final run.

Upstream, all potential parallelism of the application has to be exhibited. Here static analysis and transformation approaches (issue 1) must be processed, resulting in a *multi-parallel* intermediate code advising the running virtual machine about all the parallelism that can be taken advantage of. However the compiler does not have much knowledge about the execution environment. It obviously knows the instruction set, it can be aware of the number of available cores, but it does not know the effective available resources at any time during the execution (memory, number of free cores, etc.).

That is the reason why a "virtual machine" mechanism will have to adapt the application to the resources (issue 3). Moreover the compiler will be able to take advantage only of a part of the parallelism induced by the application. Indeed some program information (variables values, accessed memory adresses, etc.) being available only at runtime, another part of the available parallelism will have to be generated on-the-fly during the execution, here also, thanks to a dynamic mechanism.

This on-the-fly parallelism extraction will be performed using speculative behavior models (issue 2), such models allowing to generate speculative parallel code (issue 3). Between our behavior modeling objectives, we can add the behavior monitoring, or profiling, of a program version. Indeed current and future architectures complexity avoids assuming an optimal behavior regarding a given program version. A monitoring process will allow to select on-the-fly the best parallelization.

These different parallelizing steps are schematized on figure 1 .

The more and more widespread usage of object-oriented approaches and languages emphasizes the need for specific multicore programming tools. The object and method formalism implies specific execution schemes that translate in the final binary by quite distant elementary schemes. Hence the execution behavior control is far more difficult. Analysis and optimization, either static or dynamic, must take into account from the outset this distortion between object-oriented specification and final binary code: how can object or method parallelization be translated (issue 4).

Our project lies on the conception of a production chain for efficient execution of an application on a multicore architecture. Each link of this chain has to be formally verified in order to ensure correction as well as efficiency. More precisely, it has to be ensured that the compiler produces a correct intermediate code, and that the virtual machine actually performs the parallel execution semantically equivalent to the source code: every transformation applied to the application, either statically by the compiler or dynamically by the virtual machine, must preserve the initial semantics. They must be proved formally (issue 5).

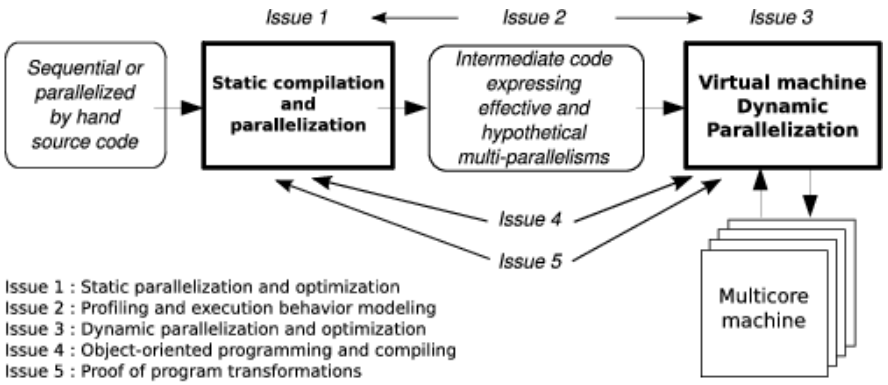


Figure 1. Automatic parallelizing steps for multicore architectures

In the following, those different issues are detailed while forming our global and long term vision of what has to be done.

3.2. Static parallelization and optimization

Participants: Vincent Loechner, Philippe Clauss, Éric Violard, Alexandra Jimborean.

Static optimizations, from source code at compile time, benefit from two decades of research in automatic parallelization: many works address the parallelization of loop nests accessing multi-dimensional arrays, and these works are now mature enough to generate efficient parallel code [26]. Low-level optimizations, in the assembly code generated by the compiler, have also been extensively dealt for single-core and require few adaptations to support multicore architectures. Concerning multicore specific parallelization, we propose to explore two research directions to take full advantage of these architectures. They are described below.

3.2.1. State of the art

Upstream, an easy interprocedural dependence analysis allows to handle complete programs (but recursivity: recursive functions must be transformed into iterative functions). Concerning iterative control we will use the polyhedral model, a formalism developped these last two decades, which allows to represent the execution of a loop nest by scanning a polytope.

When compiling an application, if it contains loop nests with affine bounds accessing scalars or arrays accessed using affine functions, the polyhedral model allows to:

- compute the dependence graph, which describes the order in which the dependent instructions must be executed [34];
- generate a schedule, which extracts some parallelism from the dependence graph [35], [36];
- generate an allocation, which assigns a processor (or a core) to a set of iterations of the loop nest to be scanned.

This last allocation step needs a thorough knowledge of the target architecture, as many crucial choices will result in performance hazards: for example, the volume and flow of inter-processor communications and synchronization; the data locality and the effects of the TLB (Translation Lookaside Buffer) and the various cache levels and distributions; or the register allocation optimizations. There are many techniques to control these parameters, and each architecture needs specific choices, of a valid schedule, of a parallel loop iterations distribution (bloc-, cyclic-, or tiled), of a loop-unrolling factor, as well as a memory data layout and a prefetch strategy (when available). They require powerful mathematical tools, such as counting the number of integer points contained in a parametric polytope.

Our own contributions in this area are significant. Concerning schedule and data placement, we proposed new advances in minimizing the number of communications for parallel architectures [54] and in cache access optimizations [53] [8]. We also proposed essential advances in parametric polytope manipulation [9], [5], developped the first algorithm to count integer points in a parametric polytope as an Ehrhart polynomial [3], and proposed successive improvements of this algorithm [10] [65]. We implemented these results in the free software *PolyLib*, utilized by many researchers around the world.

3.2.2. Adapting parallelization to multicore architecture

The first research direction to be explored is multicore specific efficient optimizations. Indeed, multicore architectures need specific optimizations, or we will get underlinear accelerations, or even decelerations. Multicore architectures may have the following properties: specific memory hierarchy, with distributed low-level cache and (possibly semi-) shared high level caches; software-controlled memory hierarchies (memory hints, local stores or scratchpads for example); optimized access to contiguous memory addresses or to separate memory banks; SIMD or vectorial execution in groups of cores, and synchronous execution; higher register allocation pressure when several threads use the same hardware (as in GPGPUs for example); etc.

A schedule and an allocation must be chosen wisely in order to obtain good performances. On NVIDIA GPGPUs, using the CUDA language, Baskaran et al. [25] obtained interesting results that have been implemented in their PLuTo compiler framework. However, they are based on many empirical and imprecise techniques, and require simulations to fine-tune the optimizations: they can be improved. Memory hierarchy efficient control is a cornerstone of tomorrow's multicore architectures performance. Compiler-optimizers have to evolve to meet this requirement.

Simulation and (partial-) profiling may however remain necessary in some cases, when static analysis reaches its intrinsic limits: when the execution of a program depends on dynamic parameters, when it uses complex pointer arithmetic, or when it performs indirect array accesses for example (as is often the case in *while* loops, out of the scope of the classical polyhedral model). In these cases, the compiler should rely on the profiler, and generate a code that interacts with the dynamic optimizer. This is the link with issues 2 and 3 of this research project.

3.2.3. Expressing many potential parallelisms

The dynamic optimizer (issue 3) must be able to exploit various parallel codes to compare them and the best one to choose, possibly swapping from a code to another during execution. The compiler must therefore generate different potentially efficient versions of a code, depending on fixed parameters such as the schedule or the data layout, and dynamic parameters such as the tile size or the unrolling factor.

The compiler then generates many variants of *effective* parallelism, formally proved by the static analyzer. It may also generate variants of code that have not been formally validated, due to the analyzer limits, and that have to be checked during execution by the dynamic optimizer: *hypothetical* parallelism. Hypothetical parallelism could be expressed as a piece of code, valid under certain conditions. Effective and hypothetical parallelisms are called *potential parallelism*. The variants of potential parallelism will be expressed in an intermediate language that has to be discovered.

Using compiler directives is an interesting way to define this intermediate language. Among the usual directives, we distinguish schedule directives for shared memory architectures (such as the OpenMP [1] *parallel* directive), and placement directives for distributed memory architectures (for example the HPF[2] *ALIGN* directive). These two types of directives are conjointly necessary to take full profit of multicore architectures. However, we have to study their complementarity and solve the interdependence or conflict that may arise between them. Moreover, new directives should allow to control data transfers between different levels of the memory hierarchy.

[1] http://www.openmp.org
[2] http://hpff.rice.edu

We are convinced that the definition of such a language is required in the next advances in compilation for multicore architectures, and there does not exist such an ambitious project to our knowledge. The OpenCL project [3], presented as an general-purpose and efficient multicore programming environment, is too low-level to be exploitable. We propose to define a new high level language based on compilation directives, that could be used by the skilled programmer or automatically generated by a compiler-optimizer (like OpenMP, recently integrated in the *gcc* compiler suite).

3.3. Profiling and execution behavior modeling

Participants: Alain Ketterlin, Philippe Clauss, Aravind Sukumaran-Rajam.

The increasing complexity of programs and hardware architectures makes it ever harder to characterize beforehand a given program's run time behavior. The sophistication of current compilers and the variety of transformations they are able to apply cannot hide their intrinsic limitations. As new abstractions like transactional memories appear, the dynamic behavior of a program strongly conditions its observed performance. All these reasons explain why empirical studies of sequential and parallel program executions have been considered increasingly relevant. Such studies aim at characterizing various facets of one or several program runs, *e.g.*, memory behavior, execution phases, etc. In some cases, such studies characterize more the compiler than the program itself. These works are of tremendous importance to highlight all aspects that escape static analysis, even though their results may have a narrow scope, due to the possible incompleteness of their input data sets.

3.3.1. Selective profiling and interaction with the compiler

In its simplest form, studying a given program's run time behavior consists in collecting and aggregating statistics, *e.g.*, counting how many times routines or basic blocks are executed, or counting the number of cache misses during a certain portion of the execution. In some cases, data can be collected about more abstract events, like the garbage-collector frequency or the number and sizes of sent and received messages. Such measures are relatively easy to obtain, are frequently used to quantify the benefits of some optimization, and may suggest some way to improve performance. These techniques are now well-known, but mostly for sequential programs.

These global studies have often been complemented by local, targeted techniques focused on some program portions, *e.g.*, where static techniques remain inconclusive for some fixed duration. These usages of profiling are usually strongly related to the optimization they complement, and are set up either by the compiler or by the execution environment. Their results may be used immediately at run time, in which case they are considered a form of run time optimization [1]. They can also be used offline to provide hints to a subsequent compilation cycle, in which case they constitute a form of profile-guided compilation, a strategy that is common in general purpose compilers.

For instance, in the context where a set of possible parallelizations have been provided by the compiler (see issue 1), a profiling component can easily be made responsible for testing some relevant condition at run time (*e.g.*, that depends on input data) and for selecting the best between various versions of the code. Beyond such simple tasks, we expect that profiling will, at the beginning of the execution, have enough resources to conduct more elaborate analyzes. We believe that combining an "open" static analysis with an integrated profiling component is a promising approach, first because it may relieve the programmer of a large part of the tedious task of implementing the distribution of computations, and second to free the compiler of the obligation to choose between several optimizations in the absence of enough relevant data. The main open question here is to define precisely the respective roles of the compiler and the profiler, and also the amount and nature of information the former can transmit to the latter.

[3] http://www.khronos.org/opencl

3.3.2. Profiling and dynamic optimization

In the context of dynamic optimization, that is, when the compiler's abilities have been exhausted, a profiler can still do useful work, provided some additional capabilities [1]. If it is able to instrument the code the way, *e.g.*, a PIN-tool does [55], it has access to the whole program, including libraries (or, for example, the code of a low-level library called from a scripting language). This means that it has access to portions of the program that were not under the compiler's control. The profiler can then perform dynamic inter-procedural analyzes, for instance to compute dependencies to detect parallelism that wasn't apparent at compile time because of a function call in the body of a loop. More generally, if the profiler is able to reconstruct at run time some representation of the whole program, as in [74] for example, it is possible to let it search for any construct that can be optimized and/or parallelized in the context of the current execution. Several virtual machines, *e.g.*, for Java or Microsoft CLR, have opened this way of optimizing programs, probably because virtual machines need to maintain an intermediate, structured representation of the running program.

The possibility of running programs on architectures that include a large number of computing cores has given rise to new abstractions [72], [46], [29]. Transactional memories, for instance, aim at simplifying the management of conflicting concurrent accesses to a shared memory, a notoriously difficult problem [48]. However, the performance of a transaction-based application heavily depends on its dynamic behavior, and too many conflicting accesses and rollbacks, severely affect performance. We bet that the need for multicore specific programming tools will lead to other abstractions based on speculative execution. Because of the very nature of speculation, all these abstractions will require run time evaluation, and maybe correction, to avoid pathological cases. The profiler has a central role here, because it can be made responsible for diagnosing inefficient use of speculative execution, and for taking corrective action, which means that it has to be integrated to the execution environment. We also think that the large scope and almost infinite potential uses of a profiling component may well suggest new parallel program abstractions, specially targeted at run time evaluation and adaptation.

3.3.3. Run time program modeling

When profiling goes beyond simple aggregation of counts, it can, for example, sample a program's behavior and split its execution into phases. These phases may help target a subsequent evaluation on a new architecture [66]. When profiling instruments the whole program to obtain a trace, *e.g.*, of memory accesses, it is possible to use this trace for:

- simulation, *e.g.*, by varying the parameters of the memory hierarchy,
- for modeling, *e.g.*, to reconstruct some specific model of the program [74], or to extract dynamic dependencies that help identifying parallel sections [62].

Handling such large execution traces, and especially compressing them, is a research topic by itself [30], [57]. Our contribution to this topic [7] is unusual in that the result of compression is a sequence of loop nests where memory accesses and loop bounds are affine functions of the enclosing loop indices. Modeling a trace this way leads to slightly better average compression rates compared to other, less expressive techniques. But more importantly, it has the advantage to provide a result in symbolic form, and this result can be further analyzed with techniques usually restricted to the static analysis of source code. We plan to apply, in the short term, similar techniques to the modeling of dynamic dependencies, so as to be able to automatically extract parallelism from program traces.

This kind of analysis is representative of a new kind of tools than could be named "parallelization assistants" [52], [62]. Properties that can't be detected by the compiler but that appear to hold in one or several executions of a program can be submitted to the programmer, maybe along a suitable reformulation of its program using some class of abstraction, *e.g.*, compiler directives. The goal is to provide help and guidance in adapting source code, in the same way a classical profiling tool helps pinpoint performance bottlenecks. Control and data dependencies are fundamental to such a tool. An execution trace provides an observed reality; for example a trace of memory addresses. If the observed dynamic dependencies provide a set of constraints, they also suggest a complete family of potential correct executions, be they parallel or sequential, and all these executions are equivalent to the reference execution. Being able to handle large traces, and representing them

in some manageable way, means being able to highlight medium to large grain parallelism, which is especially interesting on multicore architectures and often difficult for compilers to discover, for example because of the use of pointers and the difficulty of eliminating potential aliasing. This can be seen as a machine learning problem, where the goal is to recover a hidden structure from a large sequence of events. This general problem has various incarnations, depending on how much the learner knows about the original program, on the kind of data obtained by profiling, on the class of structures sought, and on the objectives of the analysis. We are convinced that such studies will enrich our understanding of the behavior of programs, and of the programming concepts that are really useful. It will also lead to useful tools, and will open up new directions for dynamic optimization.

3.4. Dynamic parallelization and optimization, virtual machine

Participants: Alexandra Jimborean, Philippe Clauss, Alain Ketterlin, Aravind Sukumaran-Rajam, Vincent Loechner.

This link in the programming chain has become essential with the advent of the new multicore architectures. Still being considered as secondary with mono-core architectures, dynamic analysis and optimization are now one of the keys for controling those new mechanisms complexity. From now on, performed instructions are not only dedicated to the application functionalities, but also to its control and its transformation, and so in its own interest. Behaving like a computer virus, such a process should rather be qualified as a “vitamin”. It perfectly knows the current characteristics of the execution environment and owns some qualitative information thanks to a behavior modeling process (issue 2). It appends a significant part of optimizing ability compared to a static compiler, while observing live resources availability evolution.

3.4.1. State of the art

Dynamic analysis and optimization, that is to say simultaneous to the program execution, have motivated a growing interest during the last decade, mainly because of the hardware architectures and applications growing complexity. Indeed, it has become more and more difficult to anticipate any program run simply from its source code, either because its control structures introduce some unknown objects before run (dynamic memory allocation, pointers, ...), or because the interaction between the target architecture and the program generates unpredictable behaviors. This is notably due to the appearance of more optimizing hardware units (prefetching units, speculative processing, code cache, branch prediction, etc.). With multicore architectures, this interest is growing even more. Works achieved in this area for mono-core processors have permitted to establish some classification of the so-called dynamic approaches, either based on the used methodologies or on the objectives.

The first objective for any dynamic approach is to extract some live information at runtime relying on a profiling process. This essential step is the main objective of issue 2 (see sub-section 3.3).

Identifying some “hotspots” thanks to profiling is then used for performance improvement optimizations. Two main approaches can be distinguished:

- the *profile-guided* approach, where analysis and optimization of profile information are performed off-line, that is to say statically. A first run is only performed to extract information for driving a re-compilation. Related to this approach, *iterative compilation* consists in running a code that has been transformed following different optimization possibilities (nature and sequencing of the applied optimizations), and then in re-compiling the transformed code guided by the collected performance information, and so on until obtaining a “best” program version. In order to promote a rapid convergence towards a better solution, some heuristics or some machine learning mechanisms are used [21], [61], [60]. The main drawback of such approaches relates to the quality of the generated code which depends on the reference profiled execution, and more precisely on the used input data set, but also on the used hardware.
- the *on-the-fly* approach consists in performing all steps at each run (profiling, analysis and transformation). The main constraint of this approach is that the time overhead has to be widely compensated

by the benefits it generates. Several works propose such approaches dedicated to specific optimizations. We personally successfully implemented a dynamic data prefetching system for the Itanium processor [1].

Although all these works provided some efficient dynamic mechanisms, their adaptation to multicore architectures yields difficult issues, and even challenges them. It is indeed necessary to control interactions between simultaneous tasks, imposing an additional complexity level which can be fateful for a dynamic system, while becoming too costly in time and space.

Some dynamic parallelizing techniques have been proposed in the last years. They are mainly focusing on parallelizing loop-nests, as programs generally spend most of their execution time in iterative structures.

The LRPD test [64] is certainly one of the foundation strategies. This method consists in speculatively parallelizing loops. Privatization and reduction transformations are applied to promote a successful application of the strategy. During execution, some tests are performed to verify the speculation validity. In case of invalid speculation, the targeted loop is re-executed sequentially. However, the application range is limited to loops accessing arrays; pointers cannot be handled. Moreover the method is not fully dynamic since an initial static analysis is needed.

In [33], Cintra and Llanos present a speculative parallel execution mechanism for loops, where iteration chunks are executed in sliding windows of n threads. The loops are not transformed and the sequential schedule remains as a reference to define a total order on the speculative threads. In order to verify whether some dependencies are violated during the program run, all data structures qualified as speculative, that is to say those being accessed in read-write mode by the threads, are duplicated for each thread and tagged following those states: *not accessed, modified, exposed loaded* or *exposed loaded and later modified*. For example, a *read-after-write* dependency has been violated if a thread owns a data tagged as *exposed loaded* or *exposed loaded and modified*, and if a predecessor thread, following the sequential total order, owns the same data but tagged as *modified* or *exposed loaded and modified*, while this data has not yet been committed in main memory. Such an approach can be memory-costly as each shared data structure is duplicated. It can be tricky to adjust verification frequencies to minimize time overhead. Some other methods based on the same principle of verifying speculation relatively to the sequential schedule have been proposed recently as in [68], where each iteration of a loop is decomposed into a prologue, a speculative body and an epilogue. The speculative bodies are performed in parallel and each body completion induces a verification. This approach seems to be only well suited for loops which bodies represent significant computation time.

Another recent work is the development of SPICE [63] which is a speculative parallelizing system where an entire first run of a loop is initially observed. This observation serves in determining the values reached by some variables during the run. During a next run of the loop, several speculative threads are launched. They consider as initial values of some variables the values that have been observed at the previous run. If a thread reaches the starting value of another thread, it stops. Thus each thread performs a different portion of the loop. But if the loop behavior changes and if another thread starting value is never reached, the run goes on sequentially until completion.

The main limits of these propositions are:

- they do not alter the initial sequential schedule since always contiguous instruction blocks are speculatively parallelized;
- their underlying parallelism is out of control: the characteristics of the generated parallel schedule are completely unknown since they randomly depend on the program instructions, their dependencies and the target machine. If bad performance is encountered, no other parallelization solution can be proposed. Moreover, the effective instruction schedule occurring at program run can significantly vary from one run to another, hence leading to a confusing performance inconsistency.

A strategy that would uniquely be based on a transactional memory mechanism, with rollbacks in the case of data races, yields a totally uncontrolable parallelism where performance can not be ensured and not even strongly expected.

While being based on efficient prediction mechanisms, a better control over parallelization will permit to provide solutions that are well suited to a varying execution context and to parallelize portions of code that can be parallelized only in some particular context. It is indeed crucial to maximize the potential parallelism of the applications to take advantage of the forthcoming processors comprising several tens of cores.

3.4.2. General objective: building a virtual machine

As it has already been mentioned, dynamic parallelization and optimization can take place inside a virtual machine. All the research objectives that are presented in the following are related to its construction.

Notice that the term of "virtual machine" is employed to group a set of dynamic analysis and optimization mechanisms taking as input a binary code, eventually enriched with specific instructions. We refer to a process virtual machine which main role is dynamic binary optimization from one instruction set to the same instruction set. The taxonomy given in [67] includes this kind of virtual machine.

Notice that this virtual machine can run in parallel on the processor cores during the four initial phases (see figure 2), but also simultaneously to the target application, either by sharing some cores with light processes, or by using cores that are useless for the target application. It will also support a transactional memory mechanism, if available. However the foreseen parallelizing strategies do not depend on such a mechanism since our speculative executions are supposed to be as reliable as possible thanks to efficient prediction models, and since they are supported by a specific and higher level rollback mechanism. Anyway if available, a transactional memory mechanism would allow to take advantage of "nearly perfect" prediction models.

The virtual machine takes as input an intermediate code expressing several kinds of parallelism on several code extracts. Those kinds of parallelism are either effective, that is to say that the corresponding parallel execution is obviously semantically correct, or hypothetical, that is to say that there is still some uncertainty on the parallelism correctness. In this case, this uncertainty will have to be resolved at run time. This intermediate "multi-parallel" code is generated by the static parallelization described subsection 3.2 . It also contains generic descriptions of parallelizing or optimizing transformations which parameters will have to be instanciated by the virtual machine, thanks to its knowledge about the target architecture and the program run-time behavior.

3.4.3. Adaptation of the intermediate code to the target architecture

The virtual machine first phase is to adapt this intermediate code to the target multicore architecture. It consists in answering the following questions:

- What is the suitable kind of parallelism?
- What is the suitable parallel task granularity?
- What is the suitable number of parallel tasks?
- Can we take advantage of a specialized instruction set for some operations?
- What are the parameter values for some parallelization or optimization?

The multi-parallel intermediate code exhibits different parameters allowing to adapt some parallelizing and optimizing transformations to the target architecture. For example, a loop unrolling will be parametrized by the number of iterations to be unrolled. This number will depend, for example, on the number of available registers and the size of the instruction cache. A parallelizing transformation will depend on several possible parallel instruction schedules. One or several schedules will be selected, for example, depending on the kind of memory hierarchy and the cache sharing among cores.

Concerning hypothetical parallelism, this first phase will reduce the number of these propositions to solutions that are well suited to the target architecture. This phase also instruments the intermediate code in order to install the dynamic mechanisms related to profiling and speculative parallel execution.

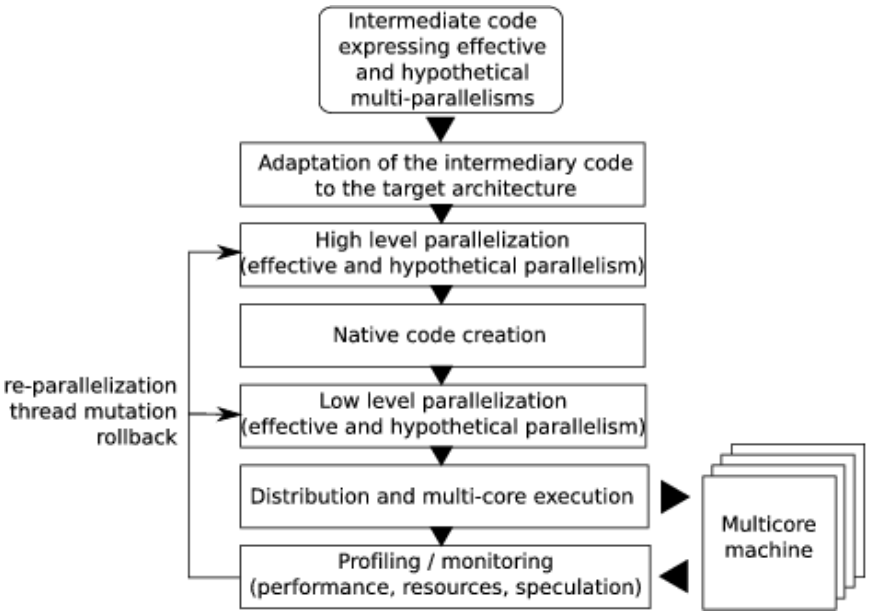


Figure 2. The virtual machine

3.4.4. High level parallelization and native code creation

From these target architecture related adaptations, a parallel intermediate code is generated. It contains instructions that are specific to the dynamic optimizing and parallelizing mechanisms, *i.e.*, instrumentation instructions to feed the profiling process as well as calls to speculative execution management procedures. A translation into native code executable by the target processor follows. This translation also allows to keep trace of the code extracts that have to be modified during the run.

3.4.5. Low level parallelization

The binary version of the code exhibits new parallelism and optimization sources that are specific to the instruction set and to the target architecture capabilities. Moreover, some dynamic optimizations are dedicated to specific instructions, or instruction blocks, as for example the memory reads which time performances can be dynamically improved by data prefetching [1]. Thus the binary code can be transformed and instrumented as well.

3.4.6. Distribution, execution and profiling

The so built executable code is then distributed among the processor cores to be run. During the run, the instrumentation instructions feed the profiler with information for execution monitoring and for behavior models construction (see subsection 3.3). An accurate knowledge of the binary code, thanks to the control of its generation, also permits at this step to dynamically control the insertion or deletion of some instrumentation instructions. Indeed it is important to manage execution monitoring through sampling based instrumentations in varying frequencies, following the changing behavior frequency (see in [1] and [73] a description of this kind of mechanism), as such instrumentations necessarily induce overheads that have to be minimized.

3.4.7. Re-parallelization, thread mutation or rollback

Depending on the information collected from instrumentation, and depending on the built prediction models, the profiling phase causes a re-transformation of some code parts, thus causing the mutation of the concerned

threads. Such re-transformation is done either on the binary code whether it consists in low level and small modifications, as for example the adjustement of a data prefetching distance, or on the intermediate code if it consists in a complete modification of the parallelizing strategy. For example, such a processing will follow the observation of a bad performance, or of a change in the computing resources availability, or will be caused by the completion of a dependency prediction model allowing the generation of a speculative parallelization. From such a speculative execution, a re-transformation can consist in rolling back to a sequential execution version when the considered hypothetical parallelism, and thus the associated prediction model, has been evaluated wrong.

3.5. Proof of program transformations for multicores

Participants: Éric Violard, Julien Narboux, Nicolas Magaud, Vincent Loechner, Alexandra Jimborean.

3.5.1. State of the art

3.5.1.1. Certification of low-level codes.

Among the languages allowing to exploit the power of multicore architectures, some of them supply the programmer a library of functions that corresponds more or less to the features of the target architecture : for example, CUDA [4] for the architectures of type GPGPU and more recently the standard OpenCL [5] that offers a unifying programming interface allowing the use of most of the existing multicore architectures or a use of heterogeneous aggregate of such architectures. The main advantage of OpenCL is that it allows the programmer to write a code that is portable on a large set of architectures (in the same spirit as the MPI library for multi-processor architectures). However, at this low level, the programming model is very close to the executing model, the control of parallelism is explicit. Proof of program correctness has to take into account low-level mechanisms such as hardware interruptions or thread preemption, which is difficult.

In [38], Feng *et al.* propose a logic inspired from the Hoare logic in order to certify such low-level programs with hardware interrupts and preempted threads. The authors specify this logic by using the meta-logic implemented in the Coq proof assistant [24].

3.5.1.2. Certification of a compiler.

The problem here is to prove that transformations or optimizations preserve the operational behaviour of the compiled programs.

Xavier Leroy in [27], [50] formalizes the analyses and optimizations performed by a C compiler: a big part of this compiler is written in the specification language of Coq and the executable (Caml) code of this compiler is obtained by automatic extraction from the specification.

Optimizing compilers are complex softwares, particularly in the case of multi-threaded programs. They apply some subtle code transformations. Therefore some errors in the compiler may occur and the compiler may produce incorrect executable codes. Work is to be done to remedy this problem. The technique of validation *a posteriori* [69], [70] is an interesting alternative to full verification of a compiler.

3.5.1.3. Semantics of directives.

As it was mentioned in subsection 3.2.3 , the use of directives is an interesting approach to adapt languages to multicore architectures. It is a syntactic means to tackle the increasing need of enriching the operational semantics of programs.

Ideally, these directives are only comments: they do not alter the correction of programs and they are a good means to improve their performance. They allow the separation of concerns: *correction* and *efficiency*.

However, using directives in that sense and in the context of automatic parallelization, raises some questions: for example, assuming that directives are not mandatory, how to ensure that directives are really taken into account? How to know if a directive is better than another? What is the impact of a directive on performance?

[4] http://www.nvidia.com/object/cuda_what_is.html
[5] http://www.khronos.org/opencl

In his thesis [40], that was supervised by Éric Violard, Philippe Gerner addresses similar questionings and states a formal framework in which the semantics of compilation directives can be defined. In this framework, any directive is encoded into one equation which is added to an algebraic specification. The semantics of the directives can be precisely defined via an order relation (called relation of *preference*) on the models of this specification.

3.5.1.4. Definition of a parallel programming model.

Classically, the good definition of a programming model is based on a semantic domain and on the definition of a "toy" language associated with a proof system, which allows to prove the correctness of the programs written in that language. Examples of such "toy" languages are CSP for control parallelism and $\mathcal{L}$ [28] for data parallelism. The proof systems associated with these two languages, are extensions of the Hoare logic.

We have done some significant works on the definition of data parallelism [11]. In particular, a crucial problem for the good definition of this programming model, is the semantics of the various syntactic constructs for data locality. We proposed a semantic domain which unifies two concepts: *alignment* (in a data-parallel language like HPF) and *shape* (in the data-parallel extensions of C).

We defined a "toy" language, called PEI, that is made of a small number of syntactic constructs. One of them, called *change of basis*, allows the programmer to exhibit parallelism in the same way as a placement or a scheduling directive [41].

3.5.1.5. Programming models for multicore architectures.

The multicore emergence questions the existing parallel programming models.

For example, with the programming model supported by OpenMP, it is difficult to master both correctness and efficiency of programs. Indeed, this model does not allow programmers to take optimal advantage of the memory hierarchy and some OpenMP directives may induce unpredictable performances or incorrect results.

Nowadays, some new programming models are experienced to help at designing both efficient and correct programs for multicores. Because memory is shared by the cores and its hierarchy has some distributed parts, some works aim at defining a hybrid model, between task parallelism and data parallelism. For example, languages like UPC (Unified Parallel C) [6] or Chapel [7] combine the advantages of several programming paradigms.

In particular, the model of memory transactions (or transactional memory [47]) retains much attention since it offers the programmer a simple operational semantics including a mutual exclusion mechanism which simplifies program design. However, much work remains to define the precise operational meaning of transactions and the interaction with the other languages features [56]. Moreover, this model leaves the compiler a lot of work to reach a safe and efficient execution on the target architecture. In particular, it is necessary to control the atomicity of transactions [39] and to prove that code transformations preserve the operational semantics.

3.5.1.6. Refinement of programs.

Refinement [22], [42] is a classical approach for gradually building correct programs: it consists in transforming an initial specification by successive steps, by verifying that each transformation preserves the correctness of the previous specification. Its basic principle is to derive simultaneously a program and its own proof. It defines a formal framework in which some rules and strategies can be elaborated to transform specifications written by using the same formalism. Such a set of rules is called a *refinement calculus*.

Unity [32] and Gamma [23] are classical examples of such formalisms, but they are not especially designed for refining programs for multicore architectures. Each of these formalisms is associated with a computing model and thus each specification can be viewed as a program. Starting with an initial specification, a proof logic allows a user to derive a specification which is more suited to the target architecture.

[6] http://upc.gwu.edu
[7] http://chapel.cs.washington.edu

Refinement applies for the programming of a large range of problems and architectures. It allows to pass the limitations of the polyhedral model and of automatic parallelization. We designed a refinement calculus to build data parallel programs [71].

3.5.2. Main objective: formal proof of analyses and transformations

Our main objective consists in certifying the critical modules of our optimization tools (the compiler and the virtual machine). First we will prove the main loop transformation algorithms which constitute the core of our system.

The optimization process can be separated into two stages: the transformations consisting in optimizing the sequential code and in exhibiting parallelism, and those consisting in optimizing the parallel code itself. The first category of optimizations can be proved within a sequential semantics. For the other optimizations, we need to work within a concurrent semantics. We expect the first stage of optimizations to produce data-race free code. For the second stage of optimizations, we will first assume that the input code is data-race free. We will prove those transformations using Appel's concurrent separation logic [44]. Proving transformations involving program which are not data-race free will constitute a longer term research goal.

3.5.3. Proof of transformations in the polyhedral model

The main code transformations used in the compiler and the virtual machine are those carried out in the polyhedral model [49], [37]. We will use the Coq proof assistant to formalize proofs of analyses and transformations based on the polyhedral model. In [31], Cachera and Pichardie formalized nested loops in Coq and showed how to prove *properties* of those loops. Our aim is slightly different as we plan to prove *transformations* of nested loops in the polyhedral model. We will first prove the simplest unimodular transformations, and later we will focus on more complex transformations which are specific to multicore architectures. We will first study scheduling optimizations and then optimizations improving data locality.

3.5.4. Validation under hypothesis

In order to prove the correction of a code transformation T it is possible to:

- prove that T is correct in general, *i.e.*, prove that for all x, $T(x)$ is equivalent to x.
- prove *a posteriori* that the applied transformation has been correct in the particular case of a code c.

The second approach relies on the definition of a program called *validator* which verifies if two pieces of program are equivalent. This program can be modeled as a function V such that, given two programs c_1 and c_2, $V(c_1, c_2) = true$ only if c_1 has the same semantics as c_2. This approach has been used in the field of optimizations certification [59], [58]. If the validator itself contains a bug then the certification process is broken. But if the validator is proved formally (as it was achieved by Tristan and Leroy for the Compcert compiler [69], [70]) then we get a transformed program which can be trusted in the same way as if the transformation is proved formally.

This second approach can be used only for the *effective parallelism*, when the static analysis provides enough information to parallelize the code. For the *hypothetical parallelism*, the necessary hypotheses have to be verified at run time.

For instance, the absence of aliases in a piece of code is difficult to decide statically but can be more easily decided at run time.

In this framework, we plan to build a *validator under hypotheses*: a function V' such that, given two programs c_1 and c_2 and an hypothesis H, if $V'(c_1, c_2, H) = true$, then H implies that c_1 has the same semantics as c_2. The validity of the hypothesis H will be verified dynamically by the virtual machine. This verification process, which is part of the virtual machine, will have to be proved as correct as well.

3.5.5. Rejecting incorrect parallelizations

The goal of the project is to exhibit potential parallelism. The source code can contain many sub-routines which could be parallelized under some hypothesis that the static analysis fails to decide. For those optimizations, the virtual machine will have to verify the hypotheses dynamically. Dynamically dealing with the potential parallelism can be complex and costly (profiling, speculative execution with rollbacks). To reduce the overhead of the virtual machine, we will have to provide efficient methods to rule out quickly incorrect parallelism. In this context, we will provide hypotheses which are easy to check dynamically and which can tell when a transformation cannot be applied, *i.e.*, hypotheses which are sufficient conditions for the non-validity of an optimization.

COMPSYS Project-Team

3. Scientific Foundations

3.1. Introduction

The embedded system design community is facing two challenges:

- The complexity of embedded applications is increasing at a rapid rate.
- The needed increase in processing power is no longer obtained by increases in the clock frequency, but by increased parallelism.

While, in the past, each type of embedded application was implemented in a separate appliance, the present tendency is toward a universal hand-held object, which must serve as a cell-phone, as a personal digital assistant, as a game console, as a camera, as a Web access point, and much more. One may say that embedded applications are of the same level of complexity as those running on a PC, but they must use a more constrained platform in terms of processing power, memory size, and energy consumption. Furthermore, most of them depend on international standards (e.g., in the field of radio digital communication), which are evolving rapidly. Lastly, since ease of use is at a premium for portable devices, these applications must be integrated seamlessly to a degree that is unheard of in standard computers.

All of this dictates that modern embedded systems retain some form of programmability. For increased designer productivity and reduced time-to-market, programming must be done in some high-level language, with appropriate tools for compilation, run-time support, and debugging. This does not mean that all embedded systems (or all of an embedded system) must be processor based. Another solution is the use of field programmable gate arrays (FPGA), which may be programmed at a much finer grain than a processor, although the process of FPGA "programming" is less well understood than software generation. Processors are better than application-specific circuits at handling complicated control and unexpected events. On the other hand, FPGAs may be tailored to just meet the needs of their application, resulting in better energy and silicon area usage. It is expected that most embedded systems will use a combination of general-purpose processors, specific processors like DSPs, and FPGA accelerators. Such a combination is already present in recent versions of the Atom Intel processor.

As a consequence, parallel programming, which has long been confined to the high-performance community, must become the common place rather than the exception. In the same way that sequential programming moved from assembly code to high-level languages at the price of a slight loss in performance, parallel programming must move from low-level tools, like OpenMP or even MPI, to higher-level programming environments. While fully-automatic parallelization is a Holy Grail that will probably never be reached in our lifetimes, it will remain as a component in a comprehensive environment, including general-purpose parallel programming languages, domain-specific parallelizers, parallel libraries and run-time systems, back-end compilation, dynamic parallelization. The landscape of embedded systems is indeed very diverse and many design flows and code optimization techniques must be considered. For example, embedded processors (micro-controllers, DSP, VLIW) require powerful back-end optimizations that can take into account hardware specificities, such as special instructions and particular organizations of registers and memories. FPGA and hardware accelerators, to be used as small components in a larger embedded platform, require "hardware compilation", i.e., design flows and code generation mechanisms to generate non-programmable circuits. For the design of a complete system-on-chip platform, architecture models, simulators, debuggers are required. The same is true for multi-cores of any kind, GPGPU ("general-purpose" graphical processing units), CGRA (coarse-grain reconfigurable architectures), which require specific methodologies and optimizations, although all these techniques converge or have connections. In other words, embedded systems need all usual aspects of the process that transforms some specification down to an executable, software or hardware. In this wide range of topics, Compsys concentrates on the code optimizations aspects in this transformation chain, restricting to compilation (transforming a program to a program) for embedded processors and to high-level synthesis (transforming a program into a circuit description) for FPGAs.

Actually, it is not a surprise to see compilation and high-level synthesis getting closer. Now that high-level synthesis has grown up sufficiently to be able to rely on place-and-route tools, or even to synthesize C-like languages, standard techniques for back-end code generation (register allocation, instruction selection, instruction scheduling, software pipelining) are used in HLS tools. At the higher level, programming languages for programmable parallel platforms share many aspects with high-level specification languages for HLS, for example, the description and manipulations of nested loops, or the model of computation/communication (e.g., Kahn process networks). In all aspects, the frontier between software and hardware is vanishing. For example, in terms of architecture, customized processors (with processor extension as proposed by Tensilica) share features with both general-purpose processors and hardware accelerators. FPGAs are both hardware and software as they are fed with "programs" representing their hardware configurations. In other words, this convergence in code optimizations explains why Compsys studies both program compilation and high-level synthesis. Besides, Compsys has a tradition of building free software tools for linear programming and optimization in general, and will continue it, as needed for our current research.

3.2. Back-End Code Optimizations for Embedded Processors

Participants: Quentin Colombet, Alain Darte, Fabrice Rastello.

Compilation is an old activity, in particular back-end code optimizations. We first give some elements that explain why the development of embedded systems makes compilation come back as a research topic. We then detail the code optimizations that we are interested in, both for aggressive and just-in-time compilation.

3.2.1. Embedded Systems and the Revival of Compilation & Code Optimizations

Applications for embedded computing systems generate complex programs and need more and more processing power. This evolution is driven, among others, by the increasing impact of digital television, the first instances of UMTS networks, and the increasing size of digital supports, like recordable DVD, and even Internet applications. Furthermore, standards are evolving very rapidly (see for instance the successive versions of MPEG). As a consequence, the industry has rediscovered the interest of programmable structures, whose flexibility more than compensates for their larger size and power consumption. The appliance provider has a choice between hard-wired structures (Asic), special-purpose processors (Asip), or (quasi) general-purpose processors (DSP for multimedia applications). Our cooperation with STMicroelectronics led us to investigate the last solution, as implemented in the ST100 (DSP processor) and the ST200 (VLIW DSP processor) family for example. Compilation and, in particular, back-end code optimizations find a second life in the context of such embedded computing systems.

At the heart of this progress is the concept of *virtualization*, which is the key for more portability, more simplicity, more reliability, and of course more security. This concept, implemented through binary translation, just-in-time compilation, etc., consists in hiding the architecture-dependent features as far as possible during the compilation process. It has been used for quite a long time for servers such as HotSpot, a bit more recently for workstations, and it is quite recent for embedded computing for reasons we now explain.

As previously mentioned, the definition of "embedded systems" is rather imprecise. However, one can at least agree on the following features:

- Even for processors that are programmable (as opposed to hardware accelerators), processors have some architectural specificities, and are very diverse;
- Many processors (but not all of them) have limited resources, in particular in terms of memory;
- For some processors, power consumption is an issue;
- In some cases, aggressive compilation (through cross-compilation) is possible, and even highly desirable for important functions.

This diversity is one of the reason why virtualization, which starts to be more mature, is becoming more and more common in programmable embedded systems, in particular through CIL (a standardization of MSIL). This implies a late compilation of programs, through just-in-time (JIT), including dynamic compilation. Some people even think that dynamic compilation, which can have more information because performed at run-time, can outperform the performances of "ahead-of-time" compilation.

Performing code generation (and some higher-level optimizations) in a late phase is potentially advantageous, as it can exploit architectural specificities and run-time program information such as constants and aliasing, but it is more constrained in terms of time and available resources. Indeed, the processor that performs the late compilation phase is, *a priori*, less powerful (in terms of memory for example) than a processor used for cross-compilation. The challenge is thus to spread the compilation process in time by deferring some optimizations ("deferred compilation") and by propagating some information for those whose computation is expensive ("split compilation"). Classically, a compiler has to deal with different intermediate representations (IR) where high-level information (i.e., more target-independent) co-exist with low-level information. The split compilation has to solve a similar problem where, this time, the compactness of the information representation, and thus its pertinence, is also an important criterion. Indeed, the IR is evolving not only from a target-independent description to a target-dependent one, but also from a situation where the compilation time is almost unlimited (cross-compilation) to one where any type of resource is limited. This is also a reason why static single assignment (SSA) is becoming specific to embedded compilation, even if it was first used for workstations. Indeed, SSA is a sparse (i.e., compact) representation of liveness information. In other words, if time constraints are common to all JIT compilers (not only for embedded computing), the benefit of using SSA is also in terms of its good ratio pertinence/storage of information. It also enables to simplify algorithms, which is also important for increasing the reliability of the compiler.

3.2.2. Aggressive and Just-in-Time Optimizations of Assembly-Level Code

Compilation for embedded processors is difficult because the architecture and the operations are specially tailored to the task at hand, and because the amount of resources is strictly limited. For instance, the potential for instruction level parallelism (SIMD, MMX), the limited number of registers and the small size of the memory, the use of direct-mapped instruction caches, of predication, but also the special form of applications [20] generate many open problems. Our goal is to contribute to their understanding and their solutions.

As previously explained, compilation for embedded processors include both aggressive and just in time (JIT) optimizations. Aggressive compilation consists in allowing more time to implement costly solutions (so, looking for complete, even expensive, studies is mandatory): the compiled program is loaded in permanent memory (ROM, flash, etc.) and its compilation time is not significant; also, for embedded systems, code size and energy consumption usually have a critical impact on the cost and the quality of the final product. Hence, the application is cross-compiled, in other words, compiled on a powerful platform distinct from the target processor. Just-in-time compilation corresponds to compiling applets on demand on the target processor. For compatibility and compactness, the source languages are CIL or Java. The code can be uploaded or sold separately on a flash memory. Compilation is performed at load time and even dynamically during execution. Used heuristics, constrained by time and limited resources, are far from being aggressive. They must be fast but smart enough.

Our aim is, in particular, to develop exact or heuristic solutions to *combinatorial* problems that arise in compilation for VLIW and DSP processors, and to integrate these methods into industrial compilers for DSP processors (mainly ST100, ST200, Strong ARM). Such combinatorial problems can be found for example in register allocation, in opcode selection, or in code placement for optimization of the instruction cache. Another example is the problem of removing the multiplexer functions (known as ϕ functions) that are inserted when converting into SSA form. These optimizations are usually done in the last phases of the compiler, using an assembly-level intermediate representation. In industrial compilers, they are handled in independent phases using heuristics, in order to limit the compilation time. Our initial goal was to develop a more global understanding of these optimization problems to derive both aggressive heuristics and JIT techniques, the main tool being the SSA representation.

In particular, we investigated the interaction of register allocation, coalescing, and spilling, with the different code representations, such as SSA. One of the challenging features of today's processors is predication [27], which interferes with all optimization phases, as the SSA form does. Many classical algorithms become inefficient for predicated code. This is especially surprising, since, beside giving a better trade-off between the number of conditional branches and the length of the critical path, converting control dependences into data dependences increases the size of basic blocks and hence creates new opportunities for local optimization

algorithms. One has to adapt classical algorithms to predicated code [29] and also to study the impact of predicated code on the whole compilation process.

As mentioned in Section 2.3 , a lot of progress has already been done in this direction in our past collaborations with STMicroelectronics. In particular, the goal of the Sceptre project was to revisit, in the light of SSA, some code optimizations in an aggressive context, i.e., by looking for the best performances without limiting, *a priori*, the compilation time and the memory usage. One of the major results of this collaboration was to propose to exploit SSA so as to design a register allocator in two phases, with one spilling phase relatively target-independent, then the allocator itself, which takes into account architectural constraints and optimizes other aspects (in particular, coalescing). This new way of considering register allocation has shown its interest for aggressive static compilation. But it offered three other perspectives:

- A simplification of the allocator, which again goes toward a more reliable compiler design, based on static single assignment.
- The possibility to handle the hardest part, the spilling phase, as a preliminary phase, thus a good candidate for split compilation.
- The possibility of a fast allocator, with a much higher quality than usual JIT approaches such as "linear scan", thus suitable for virtualization and JIT compilation.

These additional possibilities have been the heart of our research on back-end optimizations in Compsys II. The objective of the Mediacom project with STMicroelectronics was to address them. More generally, in Compsys II, our goal was to continue to develop our activity on code optimizations, exploiting SSA properties, following our two-phases strategy:

- First, revisit code optimizations in an aggressive context to develop better strategies, without eliminating too quickly solutions that may have been considered as too expensive in the past.
- Then, exploit the new concepts introduced in the aggressive context to design better algorithms in a JIT context, focusing on the speed of algorithms and their memory footprint, without compromising too much on the quality of the generated code.

An important challenge was also to consider more code optimizations and more architectural features, such as registers with aliasing, predication, and, possibly in a longer term, vectorization/parallelization.

3.3. Program Analysis and Transformations for High-Level Synthesis

Participants: Christophe Alias, Alain Darte, Paul Feautrier, Laure Gonnord [Compsys/LIFL], Alexandru Plesco [Compsys/Zettice].

3.3.1. High-Level Synthesis Context

High-level synthesis has become a necessity, mainly because the exponential increase in the number of gates per chip far outstrips the productivity of human designers. Besides, applications that need hardware accelerators usually belong to domains, like telecommunications and game platforms, where fast turn-around and time-to-market minimization are paramount. We believe that our expertise in compilation and automatic parallelization can contribute to the development of the needed tools.

Today, synthesis tools for FPGAs or ASICs come in many shapes. At the lowest level, there are proprietary Boolean, layout, and place-and-route tools, whose input is a VHDL or Verilog specification at the structural or register-transfer level (RTL). Direct use of these tools is difficult, for several reasons:

- A structural description is completely different from an usual algorithmic language description, as it is written in term of interconnected basic operators. One may say that it has a spatial orientation, in place of the familiar temporal orientation of algorithmic languages.
- The basic operators are extracted from a library, which poses problems of selection, similar to the instruction selection problem in ordinary compilation.
- Since there is no accepted standard for VHDL synthesis, each tool has its own idiosyncrasies and reports its results in a different format. This makes it difficult to build portable HLS tools.

- HLS tools have trouble handling loops. This is particularly true for logic synthesis systems, where loops are systematically unrolled (or considered as sequential) before synthesis. An efficient treatment of loops needs the polyhedral model. This is where past results from the automatic parallelization community are useful.
- More generally, a VHDL specification is too low level to allow the designer to perform, easily, higher-level code optimizations, especially on multi-dimensional loops and arrays, which are of paramount importance to exploit parallelism, pipelining, and perform communication and memory optimizations.

Some intermediate tools exist that generate VHDL from a specification in restricted C, both in academia (such as SPARK, Gaut, UGH, CloogVHDL), and in industry (such as C2H), CatapultC, Pico-Express. All these tools use only the most elementary form of parallelization, equivalent to instruction-level parallelism in ordinary compilers, with some limited form of block pipelining. Targeting one of these tools for low-level code generation, while we concentrate on exploiting loop parallelism, might be a more fruitful approach than directly generating VHDL. However, it may be that the restrictions they impose preclude efficient use of the underlying hardware.

Our first experiments with these HLS tools reveal two important issues. First, they are, of course, limited to certain types of input programs so as to make their design flows successful. It is a painful and tricky task for the user to transform the program so that it fits these constraints and to tune it to get good results. Automatic or semi-automatic program transformations can help the user achieve this task. Second, users, even expert users, have only a very limited understanding of what back-end compilers do and why they do not lead to the expected results. An effort must be done to analyze the different design flows of HLS tools, to explain what to expect from them, and how to use them to get a good quality of results. Our first goal is thus to develop high-level techniques that, used in front of existing HLS tools, improve their utilization. This should also give us directions on how to modify them.

More generally, we want to consider HLS as a more global parallelization process. So far, no HLS tools is capable of generating designs with communicating *parallel* accelerators, even if, in theory, at least for the scheduling part, a tool such as Pico-Express could have such capabilities. The reason is that it is, for example, very hard to automatically design parallel memories and to decide the distribution of array elements in memory banks to get the desired performances with parallel accesses. Also, how to express communicating processes at the language level? How to express constraints, pipeline behavior, communication media, etc.? To better exploit parallelism, a first solution is to extend the source language with parallel constructs, as in all derivations of the Kahn process networks model, including communicating regular processes (CRP, see later). The other solution is a form of automatic parallelization. However, classical methods, which are mostly based on scheduling, are not directly applicable, firstly because they pay poor attention to locality, which is of paramount importance in hardware. Besides, their aim is to extract all the parallelism in the source code; they rely on the runtime system to tailor the parallelism degree to the available resources. Obviously, there is no runtime system in hardware. The real challenge is thus to invent new scheduling algorithms that take both resource and locality into account, and then to infer the necessary hardware from the schedule. This is probably possible only for programs that fit into the polyhedral model.

In summary, as for our activity on back-end code optimizations, which is decomposed into two complementary activities, aggressive and just-in-time compilation, we focus our activity on high-level synthesis on two aspects:

- Developing high-level transformations, especially for loops and memory/communication optimizations, that can be used in front of HLS tools so as to improve their use.
- Developing concepts and techniques in a more global view of high-level synthesis, starting from specification languages down to hardware implementation.

We now give more details on the program optimizations and transformations we want to consider and on our methodology.

3.3.2. Specifications, Transformations, Code Generation for High-Level Synthesis

Before contributing to high-level synthesis, one has to decide which execution model is targeted and where to intervene in the design flow. Then one has to solve scheduling, placement, and memory management problems. These three aspects should be handled as a whole, but present state of the art dictates that they be treated separately. One of our aims will be to find more comprehensive solutions. The last task is code generation, both for the processing elements and the interfaces between FPGAs and the host processor.

There are basically two execution models for embedded systems: one is the classical accelerator model, in which data is deposited in the memory of the accelerator, which then does its job, and returns the results. In the streaming model, computations are done on the fly, as data flow from an input channel to the output. Here, data is never stored in (addressable) memory. Other models are special cases, or sometimes compositions of the basic models. For instance, a systolic array follows the streaming model, and sometimes extends it to higher dimensions. Software radio modems follow the streaming model in the large, and the accelerator model in detail. The use of first-in first-out queues (FIFO) in hardware design is an application of the streaming model. Experience shows that designs based on the streaming model are more efficient that those based on memory. One of the point to be investigated is whether it is general enough to handle arbitrary (regular) programs. The answer is probably negative. One possible implementation of the streaming model is as a network of communicating processes either as Kahn process networks (FIFO based) or as our more recent model of communicating regular processes (CRP, memory based). It is an interesting fact that several researchers have investigated translation from process networks [21] and to process networks [30], [31].

Kahn process networks (KPN) were introduced 30 years ago as a notation for representing parallel programs. Such a network is built from processes that communicate via perfect FIFO channels. Because the channel histories are deterministic, one can define a semantics and talk meaningfully about the equivalence of two implementations. As a bonus, the dataflow diagrams used by signal processing specialists can be translated on-the-fly into process networks. The problem with KPNs is that they rely on an asynchronous execution model, while VLIW processors and FPGAs are synchronous or partially synchronous. Thus, there is a need for a tool for synchronizing KPNs. This is best done by computing a schedule that has to satisfy data dependences within each process, a causality condition for each channel (a message cannot be received before it is sent), and real-time constraints. However, there is a difficulty in writing the channel constraints because one has to count messages in order to establish the send/receive correspondence and, in multi-dimensional loop nests, the counting functions may not be affine. In order to bypass this difficulty, one can define another model, *communicating regular processes* (CRP), in which channels are represented as write-once/read-many arrays. One can then dispense with counting functions. One can prove that the determinacy property still holds [22]. As an added benefit, a communication system in which the receive operation is not destructive is closer to the expectations of system designers.

The main difficulty with this approach is that ordinary programs are usually not constructed as process networks. One needs automatic or semi-automatic tools for converting sequential programs into process networks. One possibility is to start from array dataflow analysis [23]. Each statement (or group of statements) may be considered a process, and the source computation indicates where to implement communication channels. Another approach attempts to construct threads, i.e., pieces of sequential code with the smallest possible interactions. In favorable cases, one may even find outermost parallelism, i.e., threads with no interactions whatsoever. Here, communications are associated to so-called uncut dependences, i.e., dependences which cross thread boundaries. In both approaches, the main question is whether the communications can be implemented as FIFOs, or need a reordering memory. One of our research directions will be to try to take advantage of the reordering allowed by dependences to force a FIFO implementation.

Whatever the chosen solution (FIFO or addressable memory) for communicating between two accelerators or between the host processor and an accelerator, the problems of optimizing communication between processes and of optimizing buffers have to be addressed. Many local memory optimization problems have already been solved theoretically. Some examples are loop fusion and loop alignment for array contraction and for minimizing the length of the reuse vector [24], techniques for data allocation in scratch-pad memory, or techniques for folding multi-dimensional arrays [19]. Nevertheless, the problem is still largely open.

Some questions are: how to schedule a loop sequence (or even a process network) for minimal scratch-pad memory size? How is the problem modified when one introduces unlimited and/or bounded parallelism? How does one take into account latency or throughput constraints, or bandwidth constraints for input and output channels? All loop transformations are useful in this context, in particular loop tiling, and may be applied either as source-to-source transformations (when used in front of HLS tools) or as transformations to generate directly VHDL codes. One should keep in mind that theory will not be sufficient to solve these problems. Experiments are required to check the relevance of the various models (computation model, memory model, power consumption model) and to select the most important factors according to the architecture. Besides, optimizations do interact: for instance, reducing memory size and increasing parallelism are often antagonistic. Experiments will be needed to find a global compromise between local optimizations.

Finally, there remains the problem of code generation for accelerators. It is a well-known fact that modern methods for program optimization and parallelization do not generate a new program, but just deliver blueprints for program generation, in the form, e.g., of schedules, placement functions, or new array subscripting functions. A separate code generation phase must be crafted with care, as a too naïve implementation may destroy the benefits of high-level optimization. There are two possibilities here as suggested before; one may target another high-level synthesis tool, or one may target directly VHDL. Each approach has its advantages and drawbacks. However, in both situations, all such tools require that the input program respects some strong constraints on the code shape, array accesses, memory accesses, communication protocols, etc. Furthermore, to get the tool to do what the user wants requires a lot of program tuning, i.e., of program rewriting. What can be automated in this rewriting process? Semi-automated?

AOSTE Project-Team

3. Scientific Foundations

3.1. Models of Computation and Communication (MoCCs)

Participants: Charles André, Robert de Simone, Jean-Vivien Millo, Dumitru Potop Butucaru.

Esterel, SyncCharts, synchronous formalisms, Process Networks, Marked Graphs, Kahn networks, compilation, synthesis, formal verification, optimization, allocation, refinement, scheduling

Formal Models of Computation form the basis of our approach to Embedded System Design. Because of the growing importance of communication handling, it is now associated with the name, MoCC in short. The appeal of MoCCs comes from the fact that they combine features of mathematical models (formal analysis, transformation, and verification) with this of executable specifications (close to code level, simulation, and implementation). Examples of MoCCs in our case are mainly synchronous reactive formalisms and dataflow process networks. Various extensions or specific restrictions enforce respectively greater expressivity or more focused decidable analysis results.

DataFlow Process Networks and Synchronous Reactive Languages such as ESTEREL/SYNCCHARTS and SIGNAL/POLYCHRONY [53], [54], [48], [15], [4], [13] share one main characteristics: they are specified in a self-timed or loosely timed fashion, in the asynchronous data-flow style. But formal criteria in their semantics ensure that, under good correctness conditions, a sound synchronous interpretation can be provided, in which all treatments (computations, signaling communications) are precisely temporally mapped. This is refered to as clock calculus in synchronous reactive systems, and leads to a large body of theoretical studies and deep results in the case of DataFlow Process Networks [49], [47] (consider SDF balance equations for instance [56]).

As a result, explicit schedules become an important ingredient of design, which ultimately can be considered and handled by the designer him/herself. In practice such schedules are sought to optimize other parts of the design, mainly buffering queues: production and consumption of data can be regulated in their relative speeds. This was specially taken into account in the recent theories of Latency-Insensitive Design [50], or N-synchronous processes [51], with some of our contributions [6].

Explicit schedule patterns should be pictured in the framework of low-power distributed mapping of embedded applications onto manycore architectures, where they could play an important role as theoretical formal models on which to compute and optimize allocations and performances. We describe below two lines of research in this direction. Striking in these techniques is the fact that they include time and timing as integral parts of early functional design. But this original time is logical, multiform, and only partially ordering the various functional computations and communications. This approach was radically generalized in our team to a methodology for logical time based design, described next (see 3.2).

3.1.1. K-periodic static scheduling and routing in Process Networks

In the recent years we focused on the algorithm treatments of ultimately k-periodic schedule regimes, which are the class of schedules obtained by many of the theories described above. An important breakthrough occurred when realizing that the type of ultimatelly periodic binary words that were used for reporting *static scheduling* results could also be employed to record a completely distinct notion of ultimately k-periodic route switching patterns, and furthermore that commonalities of representation could ease combine them together. A new model, by the name of K-periodical Routed marked Graphs (KRG) was introduced, and extensively studied for algebraic and algorithmic properties [5].

The computations of optimized static schedules and other optimal buffering configurations in the context of latency-insensitive design led to the K-Passa software tool development 5.2 .

3.1.2. Endochrony and GALS implementation of conflict-free polychronous programs

The possibility of exploring various schedulings for a given application comes from the fact that some behaviors are truly concurrent, and mutually *conflict-free* (so they can be executed independently, with any choice of ordering). Discovering potential asynchronous inside synchronous reactive specifications then becomes something highly desirable. It can benefit to potential distributed implementation, where signal communications are restricted to a minimum, as they usually incur loss in performance and higher power consumption. This general line of research has come to be known as Endochrony, with some of our contributions [11].

3.2. Logical Time in Model-Driven Embedded System Design

Participants: Charles André, Julien deAntoni, Frédéric Mallet, Marie-Agnès Peraldi Frati, Robert de Simone.

Starting from specific needs and opportunities for formal design of embedded systems as learned from our work on MoCCs (see 3.1), we developed a Logical Time Model as part of the official OMG UML profile MARTE for Modeling and Analysis of Real-Time Embedded systems. With this model is associated a Clock Constraint Specification Language (CCSL), which allows to provide loose or strict logical time constraints between design ingredients, be them computations, communications, or any kind of events whose repetitions can be conceived as generating a logical conceptual clock (or activation condition). The definition of CCSL is provided in [1].

Our vision is that many (if not all) of the timing constraints generally expressed as physical prescriptions in real-time embedded design (such as periodicity, sporadicity) could be expressed in a logical setting, while actually many physical timing values are still unknown or unspecified at this stage. On the other hand, our logical view may express much more, such as loosely stated timing relations based on partial orderings or partial constraints.

So far we have used CCSL to express important phenonema as present in several formalisms: AADL (used in avionics domain), EAST-ADL2 (proposed for the AutoSar automotive electronic design approach), IP-Xact (for System-on-Chip (*SoC*) design). The difference here comes from the fact that these formalisms were formerly describing such issues in informal terms, while CCSL provides a dedicated formal mathematical notation. Close connections with synchronous and polychronous languages, especially Signal, were also established; so was the ability of CCSL to model dataflow process network static scheduling.

In principle the MARTE profile and its Logical Time Model can be used with any UML editor supporting profiles. In practice we focused on the PAPYRUS open-source editor, mainly from CEA LIST. We developed under Eclipse the TIMESQUARE solver and emulator for CCSL constraints (see 5.1), with its own graphical interface, as a stand-alone software module, while strongly coupled with MARTE and Papyrus.

While CCSL constraints may be introduced as part of the intended functionality, some may also be extracted from requirements imposed either from real-time user demands, or from the resource limitations and features from the intended execution platform. Sophisticated detailed descriptions of platform architectures are allowed using MARTE, as well as formal allocations of application operations (computations and communications) onto platform resources (processors and interconnects). This is of course of great value at a time where embedded architectures are becoming more and more heterogeneous and parallel or distributed, so that application mapping in terms of spatial allocation and temporal scheduling becomes harder and harder. This approach is extensively supported by the MARTE profile and its various models. As such it originates from the Application-Architecture-Adequation (AAA) methodology, first proposed by Yves Sorel, member of Aoste. AAA aims at specific distributed real-time algorithmic methods, described next in 3.3 .

Of course, while logical time in design is promoted here, and our works show how many current notions used in real-time and embedded systems synthesis can naturally be phrased in this model, there will be in the end a phase of validation of the logical time assumptions (as is the case in synchronous circuits and SoC design with timing closure issues). This validation is usually conducted from Worst-Case Execution Time (WCET) analysis on individual components, which are then used in further analysis techniques to establish the validity of logical time assumptions (as partial constraints) asserted during the design.

3.3. The AAA (Algorithm-Architecture Adequation) methodology and Real-Time Scheduling

Participants: Laurent George, Dumitru Potop Butucaru, Yves Sorel.

Note: The AAA methodology and the SynDEx environment are fully described at http://www.syndex.org/, together with relevant publications.

3.3.1. Algorithm-Architecture Adequation

The AAA methodology relies on distributed real-time scheduling and relevant optimization to connect an Algorithm/Application model to an Architectural one. We now describe its premises and benefits.

The Algorithm model is an extension of the well known data-flow model from Dennis [52]. It is a directed acyclic hyper-graph (DAG) that we call "conditioned factorized data dependence graph", whose vertices are "operations" and hyper-edges are directed "data or control dependences" between operations. The data dependences define a partial order on the operations execution. The basic data-flow model was extended in three directions: first infinite (resp. finite) repetition of a sub-graph pattern in order to specify the reactive aspect of real-time systems (resp. in order to specify the finite repetition of a sub-graph consuming different data similar to a loop in imperative languages), second "state" when data dependences are necessary between different infinite repetitions of the sub-graph pattern introducing cycles which must be avoided by introducing specific vertices called "delays" (similar to z^{-n} in automatic control), third "conditioning" of an operation by a control dependence similar to conditional control structure in imperative languages, allowing the execution of alternative subgraphs. Delays combined with conditioning allow the programmer to specify automata necessary for describing "mode changes".

The Architecture model is a directed graph, whose vertices are of two types: "processor" (one sequencer of operations and possibly several sequencers of communications) and "medium" (support of communications), and whose edges are directed connections.

The resulting implementation model [9] is obtained by an external compositional law, for which the architecture graph operates on the algorithm graph. Thus, that result is a set of algorithm graphs, "architecture-aware", corresponding to refinements of the initial algorithm graph, by computing spatial (distribution) and timing (scheduling) allocations of the operations according to the architecture graph resource availability. In that context "Adequation" refers to some search amongst the solution space of resulting algorithm graphs, labelled by timing characteristics, for one which verifies timing constraints and optimizes some criteria, usually the total execution time and the number of computing resources (but other criteria may exist). The next section describes distributed real-time schedulability analysis and optimization techniques for that purpose.

3.3.2. Distributed Real-Time Scheduling and Optimization

We address two main issues: monoprocessor real-time scheduling and multiprocessor real-time scheduling where constraints must mandatorily be met otherwise dramatic consequences may occur (hard real-time) and where resources must be minimized because of embedded features.

In our monoprocessor real-time scheduling work, beside the classical deadline constraint, often equal to a period, we take into consideration dependences beetween tasks and several, possibly related, latencies. A latency is a generalization of the typical "end-to-end" constraint. Dealing with multiple real-time constraints raises the complexity of that issue. Moreover, because the preemption leads to a waste of resources due to its approximation in the WCET (Worst Execution Time) of every task as proposed by Liu and Leyland [57], we first studied non-preemtive real-time scheduling with dependences, periodicities, and latencies constraints. Although a bad approximation may have dramatic consequences on real-time scheduling, there are only few researches on this topic. We have been investigating preemptive real-time scheduling since few years, but seeking the exact cost of the preemption such that it can be integrated in schedulability conditions, and in the corresponding scheduling algorithms. More generally, we are interested in integrating in the schedulability analyses the cost of the RTOS (Real-Time Operating System), for which the exact cost of preemption is the most difficult part because it varies according to the instance of each task [10]. Finally, we investigate also the problem of mixing hard real-time and soft real-time constraints that arises in the most complex applications.

The second research area is devoted to distributed real-time scheduling with embedding constraints. We use the results obtained in the monoprocessor case in order to derive solutions for the problem of multiprocessor (distributed) real-time scheduling. In addition to satisfy the multiple real-time constraints mentioned in the monoprocessor case, we have to minimize the total execution time (makespan) since we deal with automatic control applications involving feedback. Furthermore, the domain of embedded systems leads to solving minimization resources problems. Since these optimization problems are of NP-hard complexity we develop exact algorithms (B & B, B & C) which are optimal for simple problems, and heuristics which are sub-optimal for realistic problems corresponding to industrial needs. Long time ago we proposed a very fast "greedy" heuristics [8] whose results were regularly improved, and extended with local neighborhood heuristics, or used as initial solutions for metaheuristics such as variants of "simulated annealing".

In addition to the spatial dimension (distributed) of the real-time scheduling problem, other important dimensions are the type of communication mechanisms (shared memory vs. message passing), or the source of control and synchronization (event-driven vs. time-triggered). We explore real-time scheduling on architectures corresponding to all combinations of the above dimensions. This is of particular impact in application domains such as automotive and avionics (see 4.2).

Since real-time distributed systems are often safety-critical we address dependability issues, to tolerate faults in processors and communication interconnects. We maily focus on software redondancy, rather than hardware, to ensure real-time behaviour preservation in presence of faulty processors and/or communication media (where possible failures are predictively specified by the designer). We investigate fail silent, transient, intermittent, and Byzantine faults.

CONVECS Team

3. Scientific Foundations

3.1. New Formal Languages and their Concurrent Implementations

We aim at proposing and implementing new formal languages for the specification, implementation, and verification of concurrent systems. In order to provide a complete, coherent methodological framework, two research directions must be addressed:

- *Model-based specifications*: these are operational (i.e., constructive) descriptions of systems, usually expressed in terms of processes that execute concurrently, synchronize together and communicate. Process calculi are typical examples of model-based specification languages. The approach we promote is based on LOTOS NT (LNT for short), a formal specification language that incorporates most constructs stemming from classical programming languages, which eases its acceptance by students and industry engineers. LNT [36] is derived from the ISO standard E-LOTOS (2001), of which it represents the first successful implementation, based on a source-level translation from LNT to the former ISO standard LOTOS (1989). We are working both on the semantic foundations of LNT (enhancing the language with module interfaces and timed/probabilistic/stochastic features, compiling the m among n synchronization, etc.) and on the generation of efficient parallel and distributed code. Once equipped with these features, LNT will enable formally verified asynchronous concurrent designs to be implemented automatically.
- *Property-based specifications*: these are declarative (i.e., non-constructive) descriptions of systems, which express *what* a system should do rather than *how* the system should do it. Temporal logics and μ-calculi are typical examples of property-based specification languages. The natural models underlying value-passing specification languages, such as LNT, are Labeled Transition Systems (LTSs or simply *graphs*) in which the transitions between states are labeled by actions containing data values exchanged during handshake communications. In order to reason accurately about these LTSs, temporal logics involving data values are necessary. The approach we promote is based on MCL (*Model Checking Language*) [56], which extends the modal μ-calculus with data-handling primitives, fairness operators encoding generalized Büchi automata, and a functional-like language for describing complex transition sequences. We are working both on the semantic foundations of MCL (extending the language with new temporal and hybrid operators, translating these operators into lower-level formalisms, enhancing the type system, etc.) and also on improving the MCL on-the-fly model checking technology (devising new algorithms, enhancing ergonomy by detecting and reporting vacuity, etc.).

We address these two directions simultaneously, yet in a coherent manner, with a particular focus on applicable concurrent code generation and computer-aided verification.

3.2. Parallel and Distributed Verification

Exploiting large-scale high-performance computers is a promising way to augment the capabilities of formal verification. The underlying problems are far from trivial, making the correct design, implementation, fine-tuning, and benchmarking of parallel and distributed verification algorithms long-term and difficult activities. Sequential verification algorithms cannot be reused as such for this task: they are inherently complex, and their existing implementations reflect several years of optimizations and enhancements. To obtain good speedup and scalability, it is necessary to invent new parallel and distributed algorithms rather than to attempt a parallelization of existing sequential ones. We seek to achieve this objective by working along two directions:

- *Rigorous design:* Because of their high complexity, concurrent verification algorithms should themselves be subject to formal modeling and verification, as confirmed by recent trends in the certification of safety-critical applications. To facilitate the development of new parallel and distributed verification algorithms, we promote a rigorous approach based on formal methods and verification. Such algorithms will be first specified formally in LNT, then validated using existing model checking algorithms of the CADP toolbox. Second, parallel or distributed implementations of these algorithms will be generated automatically from the LNT specifications, enabling them to be experimented on large computing infrastructures, such as clusters and grids. As a side-effect, this "bootstrapping" approach would produce new verification tools that can later be used to self-verify their own design.
- *Performance optimization:* In devising parallel and distributed verification algorithms, particular care must be taken to optimize performance. These algorithms will face concurrency issues at several levels: grids of heterogeneous clusters (architecture-independence of data, dynamic load balancing), clusters of homogeneous machines connected by a network (message-passing communication, detection of stable states), and multi-core machines (shared-memory communication, thread synchronization). We will seek to exploit the results achieved in the parallel and distributed computing field to improve performance when using thousands of machines by reducing the number of connections and the messages exchanged between the cooperating processes carrying out the verification task. Another important issue is the generalization of existing LTS representations (explicit, implicit, distributed) in order to make them fully interoperable, such that compilers and verification tools can handle these models transparently.

3.3. Timed, Probabilistic, and Stochastic Extensions

Concurrent systems can be analyzed from a *qualitative* point of view, to check whether certain properties of interest (e.g., safety, liveness, fairness, etc.) are satisfied. This is the role of functional verification, which produces Boolean (yes/no) verdicts. However, it is often useful to analyze such systems from a *quantitative* point of view, to answer non-functional questions regarding performance over the long run, response time, throughput, latency, failure probability, etc. Such questions, which call for numerical (rather than binary) answers, are essential when studying the performance and dependability (e.g., availability, reliability, etc.) of complex systems.

Traditionally, qualitative and quantitative analyses are performed separately, using different modeling languages and different software tools, often by distinct persons. Unifying these separate processes to form a seamless design flow with common modeling languages and analysis tools is therefore desirable, for both scientific and economic reasons. Technically, the existing modeling languages for concurrent systems need to be enriched with new features for describing quantitative aspects, such as probabilities, weights, and time. Such extensions have been well-studied and, for each of these directions, there exist various kinds of automata, e.g., discrete-time Markov chains for probabilities, weighted automata for weights, timed automata for hard real-time, continuous-time Markov chains for soft real-time with exponential distributions, etc. Nowadays, the next scientific challenge is to combine these individual extensions altogether to provide even more expressive models suitable for advanced applications.

Many such combinations have been proposed in the literature, and there is a large amount of models adding probabilities, weights, and/or time. However, an unfortunate consequence of this diversity is the confuse landscape of software tools supporting such models. Dozens of tools have been developed to implement theoretical ideas about probabilities, weights, and time in concurrent systems. Unfortunately, these tools do not interoperate smoothly, due both to incompatibilities in the underlying semantic models and to the lack of common exchange formats.

To address these issues, CONVECS follows two research directions:

- *Unifying the semantic models*. Firstly, we will perform a systematic survey of the existing semantic models in order to distinguish between their essential and non-essential characteristics, the goal being to propose a unified semantic model that is compatible with process calculi techniques for specifying and verifying concurrent systems. There are already proposals for unification either

theoretical (e.g., Markov automata) or practical (e.g., PRISM and MODEST modeling languages), but these languages focus on quantitative aspects and do not provide high-level control structures and data handling features (as LNT does, for instance). Work is therefore needed to unify process calculi and quantitative models, still retaining the benefits of both worlds.

- *Increasing the operability of analysis tools*. Secondly, we will seek to enhance the interoperability of existing tools for timed, probabilistic, and stochastic systems. Based on scientific exchanges with developers of advanced tools for quantitative analysis, we plan to evolve the CADP toolbox as follows: extending its perimeter of functional verification with quantitative aspects; enabling deeper connections with external analysis components for probabilistic, stochastic, and timed models; and introducing architectural principles for the design and integration of future tools, our long-term goal being the construction of a European collaborative platform encompassing both functional and non-functional analyses.

3.4. Component-Based Architectures for On-the-Fly Verification

On-the-fly verification fights against state explosion by enabling an incremental, demand-driven exploration of LTSs, thus avoiding their entire construction prior to verification. In this approach, LTS models are handled implicitly by means of their *post* function, which computes the transitions going out of given states and thus serves as basis for any forward exploration algorithm. On-the-fly verification tools are complex software artifacts, which must be designed as modularly as possible to enhance their robustness, reduce their development effort, and facilitate their evolution. To achieve such a modular framework, we undertake research in several directions:

- *New interfaces for on-the-fly LTS manipulation*. The current application programming interface (API) for on-the-fly graph manipulation, named OPEN/CAESAR [42], provides an "opaque" representation of states and actions (transitions labels): states are represented as memory areas of fixed size and actions are character strings. Although appropriate to the pure process algebraic setting, this representation must be generalized to provide additional information supporting an efficient construction of advanced verification features, such as: handling of the types, functions, data values, and parallel structure of the source program under verification, independence of transitions in the LTS, quantitative (timed/probabilistic/stochastic) information, etc.
- *Compositional framework for on-the-fly LTS analysis*. On-the-fly model checkers and equivalence checkers usually perform several operations on graph models (LTSs, Boolean graphs, etc.), such as exploration, parallel composition, partial order reduction, encoding of model checking and equivalence checking in terms of Boolean equation systems, resolution and diagnostic generation for Boolean equation systems, etc. To facilitate the design, implementation, and usage of these functionalities, it is necessary to encapsulate them in software components that could be freely combined and replaced. Such components would act as graph transformers, that would execute (on a sequential machine) in a way similar to coroutines and to the composition of lazy functions in functional programming languages. Besides its obvious benefits in modularity, such a component-based architecture will also enable to take advantage of multi-core processors.
- *New generic components for on-the-fly verification*. The quest for new on-the-fly components for LTS analysis must be pursued, with the goal of obtaining a rich catalogue of interoperable components serving as building blocks for new analysis features. A long-term goal of this approach is to provide an increasingly large catalogue of interoperable components covering all verification and analysis functionalities that appear to be useful in practice. It is worth noticing that some components can be very complex pieces of software (e.g., the encapsulation of an on-the-fly model checker for a rich temporal logic). Ideally, it should be possible to build a novel verification or analysis tool by assembling on-the-fly graph manipulation components taken from the catalogue. This would provide a flexible means of building new verification and analysis tools by reusing generic, interoperable model manipulation components.

3.5. Real-Life Applications and Case Studies

We believe that theoretical studies and tool developments must be confronted with significant case studies to assess their applicability and to identify new research directions. Therefore, we seek to apply our languages, models, and tools for specifying and verifying formally real-life applications, often in the context of industrial collaborations.

DART Project-Team

3. Scientific Foundations

3.1. Introduction

The main research topic of the DaRT team-project concerns the hardware/software codesign of embedded systems with high performance processing units like DSP or SIMD processors. A special focus is put on multi processor architectures on a single chip (System-on-Chip). The contribution of DaRT is organized around the following items:

Co-modeling for High Performance SoC design: We define our own metamodels to specify application, architecture, and (software hardware) association. These metamodels present new characteristics as high level data parallel constructions, iterative dependency expression, data flow and control flow mixing, hierarchical and repetitive application and architecture models. All these metamodels are implemented with respect to the MARTE standard profile of the OMG group, which is dedicated to the modeling of embedded and real-time systems.

Model-based optimization and compilation techniques: We develop automatic transformations of data parallel constructions. They are used to map and to schedule an application on a particular architecture. This architecture is by nature heterogeneous and appropriate techniques used in the high performance community can be adapted. We developed new heuristics to minimize the power consumption. This new objective implies to specify multi criteria optimization techniques to achieve the mapping and the scheduling.

SoC simulation, verification and synthesis: We develop a SystemC based simulation environment at different abstraction levels for accurate performance estimation and for fast simulation. To address an architecture and the applications mapped on it, we simulate in SystemC at different abstraction levels the result of the SoC design. This simulation allows us to verify the adequacy of the mapping and the schedule, e.g., communication delay, load balancing, memory allocation. We also support IP (Intellectual Property) integration with different levels of specification. On the other hand, we use formal verification techniques in order to ensure the correctness of designed systems by particularly considering the synchronous approach. Finally, we transform MARTE models of data intensive algorithms in VHDL, in order to synthesize a hardware implementation.

3.2. Co-modeling for HP-SoC design

Modeling, UML, Marte, MDE, Transformation, Model, Metamodel

The main research objective is to build a set of metamodels (application, hardware architecture, association, deployment and platform specific metamodels) to support a design flow for SoC design. We use a MDE (Model Driven Engineering) based approach.

3.2.1. Foundations

3.2.1.1. System-on-Chip Design

SoC (System-on-Chip) can be considered as a particular case of embedded systems. SoC design covers a lot of different viewpoints including the application modeling by the aggregation of functional components, the assembly of existing physical components, the verification and the simulation of the modeled system, and the synthesis of a complete end-product integrated into a single chip.

The model driven engineering is appropriate to deal with the multiple abstraction levels. Indeed, a model allows several viewpoints on information defined only once and the links or transformation rules between the abstraction levels permit the re-use of the concepts for a different purpose.

3.2.1.2. Model-driven engineering

Model Driven Engineering (MDE) [68] is now recognized as a good approach for dealing with System on Chip design issues such as the quick evolution of the architectures or always growing complexity. MDE relies on the model paradigm where a model represents an abstract view of the reality. The abstraction mechanism avoids dealing with details and eases reusability.

A common MDE development process is to start from a high level of abstraction and to go to a targeted model by flowing through intermediate levels of abstraction. Usually, high level models contain only domain specific concepts, while technological concepts are introduced smoothly in the intermediate levels. The targeted levels are used for different purposes: code generation, simulation, verification, or as inputs to produce other models, etc. The clear separation between the high level models and the technological models makes it easy to switch to a new technology while re-using the previous high level designs. Transformations allow to go from one model at a given abstraction level to another model at another level, and to keep the different models synchronized

In an MDE approach, a SoC designer can use the same language to design application and architecture. Indeed, MDE is based on proved standards: UML 2 [38] for modeling, the MOF (Meta Object Facilities [63]) for metamodel expression and QVT [64] for transformation specifications. Some profiles, i.e. UML extensions, have been defined in order to express the specificities of a particular domain. In the context of embedded system, the MARTE profile in which we contribute follows the OMG standardization process.

3.2.1.3. Models of computation

We briefly present our reference models of computation that consist of the Array-OL language and the synchronous model. The former allows us to express the parallelism in applications while the latter favors the formal validation of the design.

Array-OL. The Array-OL language [52], [53], [48], [47] is a mixed graphical-textual specification language dedicated to express multidimensional intensive signal processing applications. It focuses on expressing all the potential parallelism in the applications by providing concepts to express data-parallel access in multidimensional arrays by regular tilings. It is a single assignment first-order functional language whose data structures are multidimensional arrays with potentially cyclic access.

The synchronous model. The synchronous approach [46] proposes formal concepts that favor the trusted design of embedded real-time systems. Its basic assumption is that computation and communication are instantaneous (referred to as "synchrony hypothesis"). The execution of a system is seen through the chronology and simultaneity of observed events. This is a main difference from visions where the system execution is rather considered under its chronometric aspect (i.e., duration has a significant role). There are different synchronous languages with strong mathematical foundations. These languages are associated with tool-sets that have been successfully used in several critical domains, e.g. avionics, nuclear power plants.

In the context of the DaRT project, we consider declarative languages such as Lustre [50] and Signal [61] to model various refinements of Array-OL descriptions in order to deal with the control aspect as well as the temporal aspect present in target applications. The first aspect is typically addressed by using concepts such as mode automata, which are proposed as an extension mechanism in synchronous declarative languages. The second aspect is studied by considering temporal projections of array dimensions in synchronous languages based on clock notion. The resulting synchronous models are analyzable using the formal techniques and tools provided by the synchronous technology.

3.2.2. Past contributions of the team on topics continued in 2012

The new team DaRT has been created in order to finalize the works started in the DaRT EPI, and also to explore new topics. We here remind the past contributions of the team on the topics we continued to work on during 2012.

Our proposal is partially based upon the concepts of the "Y-chart" [57]. The MDE contributes to express the model transformations which correspond to successive refinements between the abstraction levels.

Metamodeling brings a set of tools which enable us to specify our application and hardware architecture models using UML tools, to reuse functional and physical IPs, to ensure refinements between abstraction levels via mapping rules, to initiate interoperability between the different abstraction levels used in a same codesign, and to ensure the opening to other tools, like verification tools, thought the use of standards.

The application and the hardware architecture are modeled separately using similar concepts inspired by Array-OL to express the parallelism. The placement and scheduling of the application on the hardware architecture is then expressed in an association model.

All the previously defined models, application, architecture and association, are platform independent and they conform to the MARTE OMG Profil (figure 1). No component is associated with an execution, simulation or synthesis technology. Such an association targets a given technology (OpenMP, OpenCL, SystemC/PA, VHDL, etc.). Once all the components are associated with some IPs of the GasparLib library, the deployment is fully realized. This result can be transformed to further abstraction level models via some model transformations (figure 2).

The simulation results can lead to a refinement of the initial application, hardware architecture, association and deployment models. We propose a methodology to work with all these different models. The design steps are:

1. Separation of application and hardware architecture modeling.
2. Association with semi-automatic mapping and scheduling.
3. Selection of IPs from libraries for each element of application/architecture models, to achieve the deployment.
4. Automatic generation of the various platform specific simulation or execution models.
5. Automatic simulation or execution code generation with calls to the IPs.
6. Refinement at the highest level taking account of the simulation results.

3.2.2.1. High-level modeling in Gaspard2

In Gaspard2, models are described by using the recent OMG standard MARTE profile combined with a few native UML concepts and some extensions.

The new release of Gaspard2 uses different packages of MARTE for UML modeling. The Hardware Resource Model (HRM) concepts of MARTE enable to describe the hardware part of a system. The Repetitive Structure Modeling (RSM) concepts allow one to describe repetitive structures (DaRT team was the main contributor of this MARTE package definition). Finally, the Generic Component Modeling (GCM) concepts are used as the base for component modeling.

The above concepts are expressive enough to permit the modeling of different aspects of an embedded system:

- functionality (or applicative part): the focus is mainly put on the expression of data dependencies between components in order to describe an algorithm. Here, the manipulated data are mainly multidimensional arrays. Furthermore, a form of reactive control can be described in modeled applications via the notion of execution modes. This last aspect is modeled with the help of some native UML notions in addition to MARTE.
- hardware architecture: similar mechanisms are also used here to describe regular architectures in a compact way. Regular parallel computation units are more and more present in embedded systems, especially in SoCs. HRM is fully used to model these concepts. Some extensions are proposed for NoC design and FPGA specifications. The GPU have a particular memory hierarchy. In order to model the memory details, we extend the MARTE metamodel to describe low level characteristics of the memory.
- association of functionality with hardware architecture: the main issues concern the allocation of the applicative part of a system onto the available computation resources, and the scheduling. Here also, the allocation model takes advantage of the repetitive and hierarchical representation offered by MARTE to enable the association at different granularity levels, in a factorized way.

In addition to the above usual design aspects, Gaspard2 also defines a notion of *deployment* specification (see Figure 1) in order to select compilable IPs from libraries, at this time models can produce codes. The corresponding package defines concepts that (i) enable to describe the relation between a MARTE representation of an elementary component (a box with ports) to a text-based code (and Intellectual Property - IP, or a function with arguments), and (ii) allow one to inform the Gaspard2 transformations of specific behaviors of each component (such as average execution time, power consumption...) in order to generate a high abstraction level simulation in adequacy with the real system. Recently this package was extended to design reconfigurable systems using dynamical deployment.

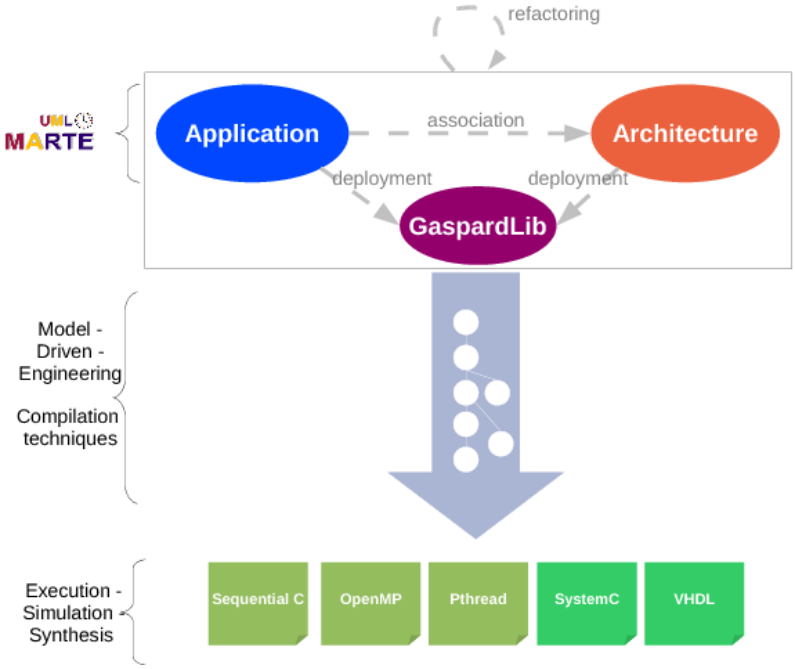


Figure 1. Overview of the design concepts.

3.2.2.2. Intermediate concept modeling and transformations

Gaspard2 targets different technologies for various purposes: formal verification, high-performance computing, simulation and hardware synthesis (Figure 1). This is achieved via model transformations that relate intermediate representations towards the final target representations.

- A metamodel for procedural language with OpenMP (OpenMP in Figure 1). It is inspired by the ANSI C and Fortran grammars and extended by OpenMP statements [41]. The aim of this metamodel is to use the same model to represent Fortran and C code. Thus, from an OpenMP model, it is possible to generate OpenMP/Fortran or OpenMP/C. The generated code includes parallelism directives and control loops to distribute task (IPs code) repetitions over processors [70].
- A VHDL metamodel (VHDL in Figure 1). It gathers the necessary concepts to describe hardware accelerators at the RTL (Register Transfer Level) level, which allows the hardware execution of applications. This metamodel introduces, *e.g.*, the notions of *clock* and *register* in order to manipulate some of the usual hardware design concepts. It is precise enough to enable the generation of synthetizable HDL code [60].

- The two metamodels SystemC and Pthread was redefined to implement both a multi-thread execution model. These are described in the " New results" part.
- Synchronous metamodel (Synchronous Equational). It was used to benefit of the verification tools of synchronous languages. It is not yet maintained in the new release of Gaspard2.

The transformation scheme. In order to target these metamodels, several transformations have been developed (Figure 2). *MartePortInstance* introduces into the MARTE metamodel the concept of *PortInstance* corresponding to an instance of port associated to a part. The *ExplicitAllocation* transformation explicits the association of each application part on the processing units, according to the association of other elements in the application hierarchy. The *LinkTopologyTask* transformation replaces the connectors between a component and an inner repeated part by a task managing the data (*TilerTask*). The scheduling of the application tasks is decomposed into three transformations, *Synchronisation* that associates, to each application component, a local graph of tasks corresponding to its parts; *GlobalSynchronization* that computes a global graph of tasks for the complete application from the local graphs of tasks; and *Scheduling* that schedules the tasks from the global graph. *TilerMapping* maps the *TilerTasks* onto processors. The management of the data in the memory is performed through two transformations. *MemoryMapping* maps the data into memory *i.e.* creates the variables and allocates address spaces. *AddressComputation* computes addresses for each variable. Finally, some transformations are dedicated to targets: *Functional* introduces the concepts relative to procedural languages. *pThread* transforms MARTE elementary tasks into threads and the connectors into buffers. *SystemC* traduces the MARTE architecture into concepts of the SystemC language.

3.2.2.3. MARTE extensions for reconfigurable based systems

Reconfigurable FPGA based Systems-on-Chip (SoC) architectures are increasingly becoming the preferred solution for implementing modern embedded systems. However due to the tremendous amount of hardware resources available in these systems, new design methodologies and tools are required to reduce their design complexity.

In previous work, we provided an initial contribution to the modeling of these systems by extending MARTE profile to incorporate significant design criteria such as power consumption.

In its current version, MARTE lacks dynamic reconfiguration concepts. Even these later are necessary to model and implement rapid prototypes for complex systems.

Our objective is to define all necessary concepts for dynamic reconfiguration issues regarding configuration latency, resources number, etc. Afterwards, these concepts will be integrated to MARTE to obtain an extended and complete profile, which can be called Reconfigurable MARTE (RecoMARTE).

Our current proposals permit us to model fine grain reconfigurable FPGA architectures with an initial extension of the MARTE profile to model Dynamic Reconfiguration at a high-level description.

Since a controller is essential for managing a dynamically reconfigurable region, we modeled a state machine at high abstraction levels using UML state machine diagrams. This state machine is responsible for switching between the available configurations.

As a future work, we will analyze the reconfigurable design flow of Xilinx from the design partitioning to the bitstream generation stage. It is a starting point for understanding how to generate configuration files. Then, we will extract relevant data to define our own design flow.

3.2.2.4. Traceability

We use the transformation mechanism to assist a tester in the mutation analysis process dedicated to model transformations. The mutation analysis aims to qualify a test model set. More precisely, errors are voluntary injected in transformation and the ability of the test models set to highlight these errors is analyzed. If the number of highlighted errors, *i.e.* if the test model set is not enough qualified, new models have to be added in order to raise the set quality [62]. Our approach relies on the hypothesis that it is easier to modify an existing model than to create a new one from scratch. The local trace, coupled to a mutation matrix, helps the tester to identify adequate test models and their relevant parts to modify in order to improve the test data set.

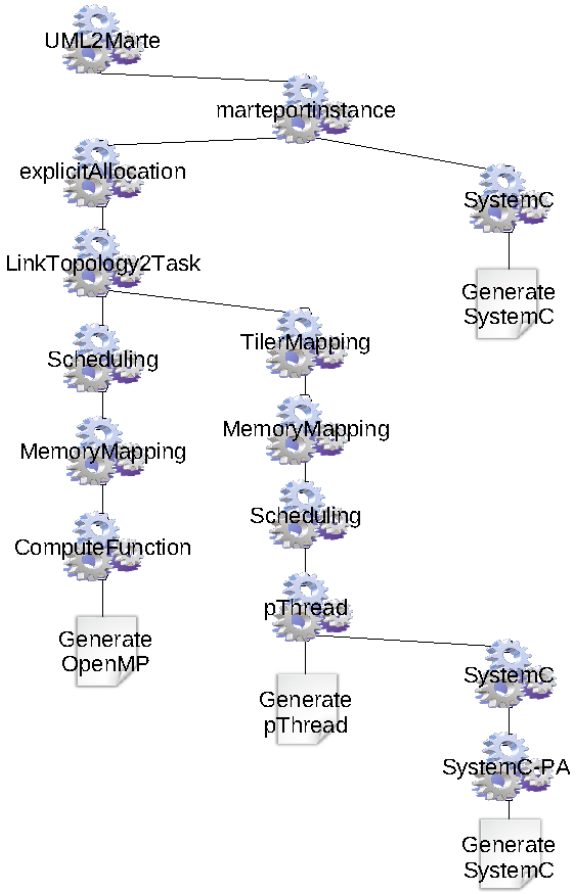


Figure 2. Overview of the transformation chains.

We propose a semi-automation approach that can automatically generate new test model in some cases and efficiently assist the testers in others cases [45].

3.3. Model-based optimization and compilation techniques

Scheduling, Mapping, Compilation, Optimization, Heuristics, Power Consumption, Data-parallelism

3.3.1. Foundations

3.3.1.1. Optimization for parallelism

We study optimization techniques to produce "good" schedules and mappings of a given application onto a hardware SoC architecture. These heuristic techniques aim at fulfilling the requirements of the application, whether they be real time, memory usage or power consumption constraints. These techniques are thus multi-objective and target heterogeneous architectures.

We aim at taking advantage of the parallelism (both data-parallelism and task parallelism) expressed in the application models in order to build efficient heuristics.

Our application model has some good properties that can be exploited by the compiler: it expresses all the potential parallelism of the application, it is an expression of data dependencies –so no dependence analysis is needed–, it is in a single assignment form and unifies the temporal and spatial dimensions of the arrays. This gives to the optimizing compiler all the information it needs and in a readily usable form.

3.3.1.2. Transformation and traceability

Model to model transformations are at the heart of the MDE approach. Anyone wishing to use MDE in its projects is sooner or later facing the question: how to perform the model transformations? The standardization process of Query View Transformation [64] was the opportunity for the development of transformation engine as Viatra, Moflon or Sitra. However, since the standard has been published, only few of investigating tools, such as ATL [1] (a transformation dedicated tool) or Kermeta [2] (a generalist tool with facilities to manipulate models) are powerful enough to execute large and complex transformations such as in the Gaspard2 framework. None of these engine is fully compliant with the QVT standard. To solve this issue, new engine relying on a subset of the standard recently emerged such as QVTO [3] and smartQVT. These engines implement the QVT Operational language.

Traceability may be used for different purposes such as understanding, capturing, tracking and verification on software artifacts during the development life cycle [58]. MDE has as main principle that everything is a model, so trace information is mainly stored as models. Solutions are proposed to keep the trace information in the initials models source or target [71]. The major drawbacks of this solution are that it pollutes the models with additional information and it requires adaptation of the metamodels in order to take into account traceability. Using a separate trace model with a specific semantics has the advantage of keeping trace information independent of initial models [59].

3.3.2. Past contributions of the team on topics continued in 2012

The new team DaRT has been created in order to finalize the works started in the DaRT EPI, and also to explore new topics. We here remind the past contributions of the team on the topics we continued to work on during 2012.

[1] http://www.eclipse.org/m2m/atl
[2] http://www.kermeta.org
[3] http://www.eclipse.org/m2m/qvto/doc

3.3.2.1. Transformation techniques

In the previous version of Gaspard2, model transformations were complex and monolithic. They were thus hardly evolvable, reusable and maintainable. We thus proposed to decompose complex transformations into smaller ones jointly working in order to build a single output model [56]. These transformations involve different parts of the same input metamodel (e.g. the MARTE metamodel); their application field is localized. The localization of the transformation was ensured by the definition of the intermediary metamodels as delta. The delta metamodel only contains the few concepts involved in the transformation (i.e. modified, or read). The specification of the transformations only uses the concepts of these deltas. We defined the Extend operator to build the complete metamodel from the delta and transposed the corresponding transformations. The complete metamodel corresponds to the merge between the delta and the MARTE metamodel or an intermediary metamodel. The transformation then becomes the chaining of metamodel shifts and the localized transformation. This way to define the model transformations has been used in the Gaspard2 environment. It allowed a better modularity and thus also reusability between the various chains.

3.3.2.2. Traceability

Our traceability solution relies on two models the Local and the Global Trace metamodels. The former is used to capture the traces between the inputs and the outputs of one transformation. The Global Trace metamodel is used to link Local Traces according to the transformation chain. The local trace also proposes an alternative "view" to the common traceability mechanism that does not refers to the execution trace of the transformation engine. It can be used whatever the used transformation language and can easily complete an existing traceability mechanism by providing a more finer grain traceability [43].

Furthermore, based on our trace metamodels, we developed algorithms to ease the model transformation debug. Based on the trace, the localization of an error is eased by reducing the search field to the sequence of the transformation rule calls [44].

3.3.2.3. Modeling for GPU

The model described in UML with Marte profile model is chained in several inout transformations that adds and/or transforms elements in the model. For adding memory allocation concepts to the model, a QVT transformation based on «Memory Allocation Metamodel» provides information to facilitate and optimize the code generation. Then a model to text transformation allows to generate the C code for GPU architecture. Before the standard releases, Acceleo is appropriate to get many aspects from the application and architecture model and transform it in CUDA (.cu, .cpp, .c, .h, Makefile) and OpenCL (.cl, .cpp, .c, .h, Makefile) files. For the code generation, it's required to take into account intrinsic characteristics of the GPUs like data distribution, contiguous memory allocation, kernels and host programs, blocks of threads, barriers and atomic functions.

3.3.2.4. GPGPU code production

The solution of large, sparse systems of linear equations « Ax=b » presents a bottleneck in sequential code executing on CPU. To solve a system bound to Maxwell's equations on Finite Element Method (FEM), a version of conjugate gradient iterative method was implemented in CUDA and OpenCL as well. The aim is to accelerate and verify the parallel code on GPUs. The first results showed a speedup around 6 times against sequential code on CPU. Another approach uses an algorithm that explores the sparse matrix storage format (by rows and by columns). This one did not increase the speedup but it allows to evaluate the impact of the access to the memory.

3.3.2.5. From MARTE to OpenCL.

We have proposed an MDE approach to generate OpenCL code. From an abstract model defined using UML/MARTE, we generate a compilable OpenCL code and then, a functional executable application. As MDE approach, the research results provide, additionally, a tool for project reuse and fast development for not necessarily experts. This approach is an effective operational code generator for the newly released OpenCL standard. Further, although experimental examples use mono device(one GPU) example, this approach provides resources to model applications running on multi devices (homogeneously configured). Moreover, we provide two main contributions for modeling with UML profile to MARTE. On the one hand, an approach to model distributed memory simple aspects, i.e. communication and memory allocations. On the other hand,

an approach for modeling the platform and execution models of OpenCL. During the development of the transformation chain, an hybrid metamodel was proposed for specifying of CPU and GPU programming models. This allows generating other target languages that conform the same memory, platform and execution models of OpenCL, such as CUDA language. Based on other created model to text templates, future works will exploit this multi language aspect. Additionally, intelligent transformations can determine optimization levels in data communication and data access. Several studies show that these optimizations increase remarkably the application performance.

3.3.2.6. Formal techniques for construction, compilation and analysis of domain-specific languages

The increasing complexity of software development requires rigorously defined *domain specific modelling languages* (DSML). Model-driven engineering (MDE) allows users to define their language's syntax in terms of *metamodels*. Several approaches for defining operational semantics of DSML have also been proposed [69], [51], [42], [49], [65]. We have also proposed one such approach, based on representing models and metamodels as algebraic specifications, and operational semantics as rewrite rules over those specifications [54], [67]. These approaches allow, in principle, for model execution and for formal analyses of the DSML. However, most of the time, the executions/analyses are performed via transformations to other languages: code generation, resp. translation to the input language of a model checker. The consequence is that the results (e.g., a program crash log, or a counterexample returned by a model checker) may not be straightforward to interpret by the users of a DSML. We have proposed in [66] a formal and operational framework for tracing such results back to the original DSML's syntax and operational semantics, and have illustrated it on SPEM, a language for timed process management.

ESPRESSO Project-Team

3. Scientific Foundations

3.1. Introduction

Embedded systems are not new, but their pervasive introduction in ordinary-life objects (cars, telephone, home appliances) brought a new focus onto design methods for such systems. New development techniques are needed to meet the challenges of productivity in a competitive environment. Synchronous languages rely on the *synchronous hypothesis*, which lets computations and behaviors be divided into a discrete sequence of *computation steps* which are equivalently called *reactions* or *execution instants*. In itself this assumption is rather common in practical embedded system design.

But the synchronous hypothesis adds to this the fact that, *inside each instant*, the behavioral propagation is well-behaved (causal), so that the status of every signal or variable is established and defined prior to being tested or used. This criterion, which may be seen at first as an isolated technical requirement, is in fact the key point of the approach. It ensures strong semantic soundness by allowing universally recognized mathematical models to be used as supporting foundations. In turn, these models give access to a large corpus of efficient optimization, compilation, and formal verification techniques. The synchronous hypothesis also guarantees full equivalence between various levels of representation, thereby avoiding altogether the pitfalls of non-synthesizability of other similar formalisms. In that sense the synchronous hypothesis is, in our view, a major contribution to the goal of *model-based design* of embedded systems.

We shall describe the synchronous hypothesis and its mathematical background, together with a range of design techniques enpowered by the approach. Declarative formalisms implementing the synchronous hypothesis can be cast into a model of computation [8] consisting of a domain of traces or behaviors and of semi-lattice structure that renders the synchronous hypothesis using a timing equivalence relation: clock equivalence. Asynchrony [32] can be superimposed on this model by considering a flow equivalence relation as well as heterogeneous systems [33] by parameterizing composition with arbitrary timing relations.

3.2. Polychronous model of computation

We consider a partially-ordered set of tags t to denote instants seen as symbolic periods in time during which a reaction takes place. The relation $t_1 \leq t_2$ says that t_1 occurs before t_2. Its minimum is noted 0. A totally ordered set of tags C is called a *chain* and denotes the sampling of a possibly continuous or dense signal over a countable series of causally related tags. Events, signals, behaviors and processes are defined as follows:

- an *evente* is a pair consisting of a value v and a tag t,
- a *signals* is a function from a *chain* of tags to a set of values,
- a *behaviorb* is a function from a set of names x to signals,
- a *processp* is a set of behaviors that have the same domain.

In the remainder, we write $\mathrm{tags}(s)$ for the tags of a signal s, $\mathrm{vars}(b)$ for the domain of b, $b|_X$ for the projection of a behavior b on a set of names X and b/X for its complementary.

Figure 1 depicts a behavior b over three signals named x, y and z. Two frames depict timing domains formalized by chains of tags. Signals x and y belong to the same timing domain: x is a down-sampling of y. Its events are synchronous to odd occurrences of events along y and share the same tags, e.g. t_1. Even tags of y, e.g. t_2, are ordered along its chain, e.g. $t_1 < t_2$, but absent from x. Signal z belongs to a different timing domain. Its tags are not ordered with respect to the chain of y.

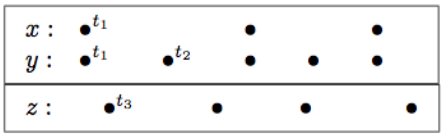


Figure 1. Behavior b over three signals x, y and z in two clock domains

3.2.1. Composition

Synchronous composition is noted $p \,|\, q$ and defined by the union $b \cup c$ of all behaviors b (from p) and c (from q) which hold the same values at the same tags $b|_I = c|_I$ for all signal $x \in I = \mathrm{vars}(b) \cap \mathrm{vars}(c)$ they share. Figure 2 depicts the synchronous composition (Figure 2 , right) of the behaviors b (Figure 2 , left) and the behavior c (Figure 2 , middle). The signal y, shared by b and c, carries the same tags and the same values in both b and c. Hence, $b \cup c$ defines the synchronous composition of b and c.

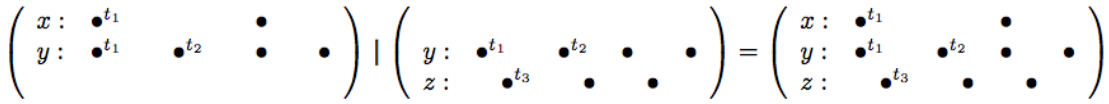


Figure 2. Synchronous composition of $b \in p$ and $c \in q$

3.2.2. Scheduling

A scheduling structure is defined to schedule the occurrence of events along signals during an instant t. A scheduling $\rightarrow$ is a pre-order relation between dates x_t where t represents the time and x the location of the event. Figure 3 depicts such a relation superimposed to the signals x and y of Figure 1 . The relation $y_{t_1} \rightarrow x_{t_1}$, for instance, requires y to be calculated before x at the instant t_1. Naturally, scheduling is contained in time: if $t < t'$ then $x_t \rightarrow^b x_{t'}$ for any x and b and if $x_t \rightarrow^b x_{t'}$ then $t' \neg < t$.

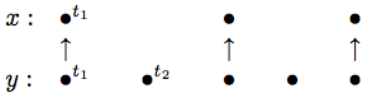


Figure 3. Scheduling relations between simultaneous events

3.2.3. Structure

A synchronous structure is defined by a semi-lattice structure to denote behaviors that have the same timing structure. The intuition behind this relation is depicted in Figure 4 . It is to consider a signal as an elastic with

ordered marks on it (tags). If the elastic is stretched, marks remain in the same relative (partial) order but have more space (time) between each other. The same holds for a set of elastics: a behavior. If elastics are equally stretched, the order between marks is unchanged.

In Figure 4 , the time scale of x and y changes but the partial timing and scheduling relations are preserved. Stretching is a partial-order relation which defines clock equivalence. Formally, a behavior c is a *stretching* of b of same domain, written $b \leq c$, iff there exists an increasing bijection on tags f that preserves the timing and scheduling relations. If so, c is the image of b by f. Last, the behaviors b and c are said *clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$.

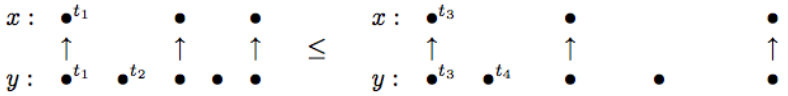


Figure 4. Relating synchronous behaviors by stretching.

3.3. A declarative design language

Signal [34], [48], [49], [41] is a declarative design language expressed within the polychronous model of computation. In Signal, a process P is an infinite loop that consists of the synchronous composition $P \,|\, Q$ of simultaneous equations $x := y\ f\ z$ over signals named x, y, z. The restriction of a signal name x to a process P is noted P/x.

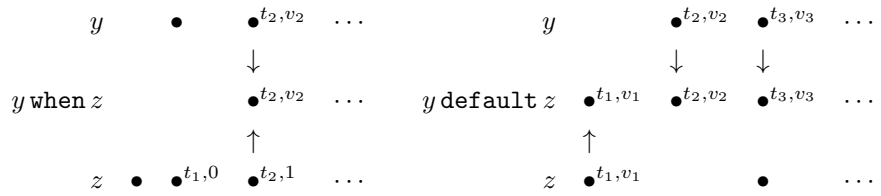
$$P, Q ::= x := y\ f\ z \ \mid\ P/x \ \mid\ P \,|\, Q$$

Equations $x := y\ f\ z$ in Signal more generally denote processes that define timing relations between input and output signals. There are four primitive combinators in Signal:

- delay $x := y \,\$\, \texttt{init}\ v$, initially defines the signal x by the value v and then by the previous value of the signal y. The signal y and its delayed copy $x = y \,\$\, \texttt{init}\ v$ are synchronous: they share the same set of tags $t_1, t_2, \cdots$. Initially, at t_1, the signal x takes the declared value v and then, at tag t_n, the value of y at tag t_{n-1}.

$$\begin{array}{rcccc} y & \bullet^{t_1,v_1} & \bullet^{t_2,v_2} & \bullet^{t_3,v_3} & \ldots \\ y \,\$\, \texttt{init}\ v & \bullet^{t_1,v} & \bullet^{t_2,v_1} & \bullet^{t_3,v_2} & \ldots \end{array}$$

- sampling $x := y\ \texttt{when}\ z$, defines x by y when z is true (and both y and z are present); x is present with the value v_2 at t_2 only if y is present with v_2 at t_2 and if z is present at t_2 with the value true. When this is the case, one needs to schedule the calculation of y and z before x, as depicted by $y_{t_2} \rightarrow x_{t_2} \longleftarrow z_{t_2}$.
- merge $x := y\ \texttt{default}\ z$, defines x by y when y is present and by z otherwise. If y is absent and z present with v_1 at t_1 then x holds (t_1, v_1). If y is present (at t_2 or t_3) then x holds its value whether z is present (at t_2) or not (at t_3).



The structuring element of a Signal specification is a process. A process accepts input signals originating from possibly different clock domains to produce output signals when needed. This allows, for instance, to specify a counter where the inputs `tick` and `reset` and the output `value` have independent clocks. The body of `counter` consists of one equation that defines the output signal `value`. Upon the event `reset`, it sets the count to 0. Otherwise, upon a `tick` event, it increments the count by referring to the previous value of `value` and adding 1 to it. Otherwise, if the count is solicited in the context of the counter process (meaning that its clock is active), the counter just returns the previous count without having to obtain a value from the `tick` and `reset` signals.

```
process counter = (? event tick, reset; ! integer value;)
    (| value := (0 when reset)
        default ((value$ init 0 + 1) when tick)
        default (value$ init 0)
    |);
```

A Signal process is a structuring element akin to a hierarchical block diagram. A process may structurally contain sub-processes. A process is a generic structuring element that can be specialized to the timing context of its call. For instance, the definition of a synchronized counter starting from the previous specification consists of its refinement with synchronization. The input tick and reset clocks expected by the process `counter` are sampled from the boolean input signals `tick` and `reset` by using the `when tick` and `when reset`†expressions. The count is then synchronized to the inputs by the equation `reset ^= tick ^= value`.

```
process synccounter = (? boolean tick, reset; ! integer value;)
    (| value := counter (when tick, when reset)
     | reset ^= tick ^= value
     |);
```

3.4. Compilation of Signal

Sequential code generation starting from a Signal specification starts with an analysis of its implicit synchronization and scheduling relations. This analysis yields the control and data-flow graphs that define the class of sequentially executable specifications and allow to generate code.

3.4.1. Synchronization and scheduling specifications

In Signal, the clock $\hat{}x$ of a signal x denotes the set of instants at which the signal x is present. It is represented by a signal that is true when x is present and that is absent otherwise. Clock expressions represent control. The clock `when` x (resp. `when not` x) represents the time tags at which a boolean signal x is present and true (resp. false).

The empty clock is written $\hat{}0$ and clock expressions e combined using conjunction, disjunction and symmetric difference. Clock equations E are Signal processes: the equation $e\hat{} = e'$ synchronizes the clocks e and e' while $e\hat{} < e'$ specifies the containment of e in e'. Explicit scheduling relations $x \rightarrow y$ `when` e allow to schedule the calculation of signals (e.g. x after y at the clock e).

$$
\begin{array}{rcll}
e & ::= & \hat{}x \mid \texttt{when}\, x \mid \texttt{not}\, x \mid e\hat{}+e' \mid e\hat{}-e' \mid e\hat{}*e' \mid \hat{}0 & \text{(clock expression)} \\
E & ::= & () \mid e\hat{}=e' \mid e\hat{}<e' \mid x \to y\, \texttt{when}\, e \mid E \,|\, E' \mid E/x & \text{(clock relations)}
\end{array}
$$

3.4.2. Synchronization and scheduling analysis

A Signal process P corresponds to a system of clock and scheduling relations E that denotes its timing structure. It can be defined by induction on the structure of P using the inference system $P : E$ of Figure 5 .

```
x := y$ init v   : ^x ^= ^y
x := y when z    : ^x ^= ^y when z | y -> x when z
x := y default z : ^x ^= ^y default ^z | y -> x when ^y | z -> x when ^z ^- ^y
```

Figure 5. Clock inference system

3.4.3. Hierarchization

The clock and scheduling relations E of a process P define the control-flow and data-flow graphs that hold all necessary information to compile a Signal specification upon satisfaction of the property of *endochrony*. A process is said endochronous iff, given a set of input signals and flow-equivalent input behaviors, it has the capability to reconstruct a unique synchronous behavior up to clock-equivalence: the input and output signals are ordered in clock-equivalent ways.

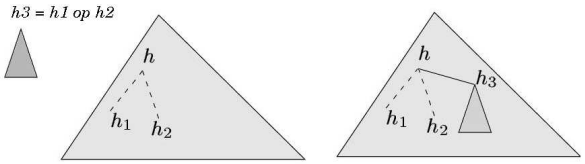


Figure 6. Hierarchization of clocks

To determine the order $x \preceq y$ in which signals are processed during the period of a reaction, clock relations E play an essential role. The process of determining this order is called hierarchization and consists of an insertion algorithm which hooks elementary control flow graphs (in the form of if-then-else structures) one to the others. Figure 6 , right, let h3 be a clock computed using h1 and h2. Let h be the head of a tree from which h1 and h2 are computed (an if-then-else), h3 is computed after h1 and h2 and placed under h [28].

MUTANT Project-Team

3. Scientific Foundations

3.1. Real-time Machine Listening

When human listeners are confronted with musical sounds, they rapidly and automatically find their way in the music. Even musically untrained listeners have an exceptional ability to make rapid judgments about music from short examples, such as determining music style, performer, beating, and specific events such as instruments or pitches. Making computer systems capable of similar capabilities requires advances in both music cognition, and analysis and retrieval systems employing signal processing and machine learning.

In a panel session at the 13th National Conference on Artificial Intelligence in 1996, Rodney Brooks (noted figure in robotics) remarked that while automatic speech recognition was a highly researched domain, there had been few works trying to build machines able to understand "non-speech sound". He went further to name this as one of the biggest challenges faced by Artificial Intelligence [41]. More than 15 years have passed. Systems now exist that are able to analyze the contents of music and audio signals and communities such as International Symposium on Music Information Retrieval (MIR) and Sound and Music Computing (SMC) have formed. But we still lack reliable Real-Time machine listening systems.

The first thorough study of machine listening appeared in Eric Scheirer's PhD thesis at MIT Media Lab in 2001 [40] with a focus on low-level listening such as pitch and musical tempo, paving the way for a decade of research. Since the work of Scheirer, the literature has focused on task-dependent methods for machine listening such as pitch estimation, beat detection, structure discovery and more. Unfortunately, the majority of existing approaches are designed for information retrieval on large databases or off-line methods. Whereas the very act of listening is real-time, very little literature exists for supporting real-time machine listening. This argument becomes more clear while looking at the yearly Music Information Retrieval Evaluation eXchange (MIREX), with different retrieval tasks and submitted systems from international institutions, where almost no emphasis exists on real-time machine listening. Most MIR contributions focus on off-line approaches to information retrieval (where the system has access to future data) with less focus on on-line and realtime approaches to information decoding.

On another front, most MIR algorithms suffer from modeling of temporal structures and temporal dynamics specific to music (where most algorithms have roots in speech or biological sequence without correct adoption to temporal streams such as music). Despite tremendous progress using modern signal processing and statistical learning, there is much to be done to achieve the same level of abstract understanding for example in text and image analysis on music data. On another hand, it is important to notice that even untrained listeners are easily able to capture many aspects of formal and symbolic structures from an audio stream in realtime. Realtime machine listening is thus still a major challenge for artificial sciences that should be addressed both on application and theoretical fronts.

In the MUTANT project, we focus on realtime and online methods of music information retrieval out of audio signals. One of the primary goals of such systems is to fill in the gap between *signal representation* and *symbolic information* (such as pitch, tempo, expressivity, etc.) contained in music signals. MUTANT's current activities focus on two main applications: *score following* or realtime audio-to-score alignment [2], and realtime transcription of music signals [20] with impacts both on signal processing using machine learning techniques and their application in real-world scenarios.

The team-project will focus on two aspects of realtime machine listening:

1. **Application-Driven Approach:** First, to enhance and foster existing application-driven approaches within the team such as realtime alignment algorithms and polyphonic pitch transcription. Our contributions on this line correspond to extensions of existing algorithmic approaches to realtime audio alignment and transcription to create new interactive application paradigms with new algorithmic

approaches. Arshia Cont's ongoing realtime alignment in *Antescofo* as well as realtime transcription using non-negative factorization methods [20] are examples of this.

2. **Music Information Geometry:** In parallel to concrete applications, we hope to theoretically contribute to the problem of signal representations of audio streams for effortless retrieval of high-level information structures. We have recently shown in [4] that the gap between the symbolic/semantic and signal aspects of music information mostly lies on constructing a well-behaved representational space before any algorithmic considerations, by employing the emerging methods of *information geometry*. Arnaud Dessein's ongoing PhD thesis is focused on this aspect of the project.

3.2. Synchronous and realtime programming for computer music

The second aspect of an interactive music system is to *react* to extracted high-level and low-level music information based on pre-defined actions. The simplest scenario is *automatic accompaniment*, delegating the interpretation of one or several musical voices to a computer, in interaction with a live solo (or ensemble) musician(s). The most popular form of such systems is the automatic accompaniment of an orchestral recording with that of a soloist in the classical music repertoire (concertos for example). In the larger context of interactive music systems, the "notes" or musical elements in the accompaniment are replaced by "programs" that are written during the phase of composition and are evaluated in realtime in reaction and relative to musicians' performance. The programs in question here can range from sound playback, to realtime sound synthesis by simulating physical models, and realtime transformation of musician's audio and gesture.

Such musical practice is commonly referred to as the *realtime school* in computer music, developed naturally with the invention of the first score following systems, and led to the invention of the first prototype of realtime digital signal processors [28] and subsequents [31], and the realtime graphical programming environment *Max* for their control [37] at Ircam. With the advent and availability of DSPs in personal computers, integrated realtime event and signal processing graphical language *MaxMSP* was developed [38] at Ircam, which today is the worldwide standard platform for realtime interactive arts programming. This approach to music making was first formalized by composers such as Philippe Manoury and Pierre Boulez, in collaboration with researchers at Ircam, and soon became a standard in musical composition with computers.

Besides realtime performance and implementation issues, little work has underlined the formal aspects of such practices in realtime music programming, in accordance to the long and quite rich tradition of musical notations. Recent progress has convinced both the researcher and artistic bodies that this programming paradigm is close to *synchronous reactive programming languages*, with concrete analogies between both: parallel synchrony and concurrency is equivalent to musical polyphony, periodic sampling to rhythmic patterns, hierarchical structures to micro-polyphonies, and demands for novel hybrid models of time among others. *Antescofo* is therefore an early response to such demands that needs further explorations and studies.

Within the MUTANT project, we propose to tackle this aspect of the research within three consecutive lines:

- **Development of a Synchronous DSL for Real Time Musician-Computer Interaction:** Ongoing and continuous extensions of the *Antescofo* language following user requests and by inscribing them within a coherent framework for the handling of temporal musical relationships. José Echeveste's ongoing PhD thesis focuses on the research and development of these aspects. Recent formalizations of the *Antescofo* language has been published in [6].
- **Formal Methods:** Failure during an artistic performance should be avoided. This naturally leads to the use of formal methods, like static analysis or model checking, to ensure formally that the execution of an *Antescofo* program will satisfy some expected property. The checked properties may also provide some assistance to the composer especially in the context of "non deterministic score" in an interactive framework.

3.3. Off-the-shelf Operating Systems for Real-time Audio

While operating systems shield the computer hardware from all other software, it provides a comfortable environment for program execution and evades offensive use of hardware by providing various services related

to essential tasks. However, integrating discrete and continuous multimedia data demands additional services, especially for real-time processing of continuous-media such as audio and video. To this end interactive systems are sometimes referred to as off-the-shelf operating systems for real-time audio. The difficulty in providing correct real-time services has much to do with human perception. Correctness for real-time audio is more stringent than video because human ear is more sensitive to audio gaps and glitches than human eye is to video jitter [43]. Here we expose the foundations of existing sound and music operating systems and focus on their major drawbacks with regards to today practices.

An important aspect of any real-time operating system is fault-tolerance with regards to short-time failure of continuous-media computation, delivery delay or missing deadlines. Existing multimedia operating systems are soft real-time where missing a deadline does not necessarily lead to system failure and have their roots in pioneering work in [42]. Soft real-time is acceptable in simple applications such as video-on-demand delivery, where initial delay in delivery will not directly lead to critical consequences and can be compensated (general scheme used for audio-video synchronization), but with considerable consequences for Interactive Systems: Timing failure in interactive systems will heavily affect inter-operability of models of computation, where incorrect ordering can lead to unpredictable and unreliable results. Moreover, interaction between computing and listening machines (both dynamic with respect of internal computation and physical environment) requires tighter and explicit temporal semantics since interaction between physical environment and the system can be continuous and not demand-driven.

Fulfilling timing requirements of continuous media demands explicit use of scheduling techniques. As shown earlier, existing Interactive Music Systems rely on combined event/signal processing. In real-time, scheduling techniques aim at gluing the two engines together with the aim of timely delivery of computations between agents and components, from the physical environment, as well as to hardware components. The first remark in studying existing system is that they all employ static scheduling, whereas interactive computing demands more and more time-aware and context-aware dynamic methods. The scheduling mechanisms are neither aware of time, nor the nature and semantics of computations at stake. Computational elements are considered in a functional manner and reaction and execution requirements are simply ignored. For example, *Max* scheduling mechanisms can delay message delivery when many time-critical tasks are requested within one cycle [38]. *SuperCollider* uses Earliest-Deadline-First (EDF) algorithms and cycles can be simply missed [35]. This situation leads to non-deterministic behavior with deterministic components and poses great difficulties for preservation of underlying techniques, art pieces, and algorithms. The situation has become worse with the demand for nomad physical computing where individual programs and modules are available but no action coordination or orchestration is proposed to design integrated systems. System designers are penalized for expressivity, predictability and reliability of their design despite potentially reliable components.

Existing systems have been successful in programing and executing small system comprised of few programs. However, severe problems arise when scaling from program to system-level for moderate or complex programs leading to unpredictable behavior. Computational elements are considered as functions and reaction and execution requirements are simply ignored. System designers have uniformly chosen to hide timing properties from higher abstractions, and despite its utmost importance in multimedia computing, timing becomes an accident of implementation. This confusing situation for both artists and system designers, is quite similar to the one described in Dr. Edward Lee's seminal paper "Computing needs time" stating: "general-purpose computers are increasingly asked to interact with physical processes through integrated media such as audio. [...] and they don't always do it well. The technological basis that engineers have chosen for general-purpose computing [...] does not support these applications well. Changes that ensure this support could improve them and enable many others" [30].

Despite all shortcomings, one of the main advantages of environments such as *Max* and *PureData* to other available systems, and probably the key to their success, is their ability to handle both synchronous processes (such as audio or video delivery and processing) within an asynchronous environment (user and environmental interactions). Besides this fact, multimedia service scheduling at large has a tendency to go more and more towards computing besides mere on-time delivery. This brings in the important question of hybrid scheduling of heterogeneous time and computing models in such environments, a subject that has had very few studies

in multimedia processing but studied in areas such simulation applications. We hope to address this issue scientifically by first an explicit study of current challenges in the domain, and second by proposing appropriate methods for such systems. This research is inscribed in the three year ANR project INEDIT coordinated by the team leader (started in September 2012).

PARKAS Project-Team

3. Scientific Foundations

3.1. Presentation and originality of the PARKAS team

Our project is founded on our expertise in three complementary domains: (1) synchronous functional programming and its extensions to deal with features such as communication with bounded buffers and dynamic process creation; (2) mathematical models for synchronous circuits; (3) compilation techniques for synchronous languages and optimizing/parallelizing compilers.

A strong point of the team is its experience and investment in the development of languages and compilers. Members of the team also have direct collaborations for several years with major industrial companies in the field and several of our results are integrated in successful products. Our main results are briefly summarized below.

3.1.1. Synchronous functional programming

In [19], Paul Caspi and Marc Pouzet introduced *synchronous Kahn networks* as those Kahn networks that can be statically scheduled and executed with bounded buffers. This was the origin of the language LUCID SYNCHRONE, [1][2] an ML extension of the synchronous language LUSTRE with higher-order features, dedicated type systems (clock calculus as a type system [19], [29], initialization analysis [30] and causality analysis [31]). The language integrates original features that are not found in other synchronous languages: such as combinations of data flow, control flow, hierarchical automata and signals [28], [27], and modular code generation [20], [17].

In 2000, Marc Pouzet started to collaborate with the SCADE team of Esterel-Technologies on the design of a new version of SCADE. [3] Several features of LUCID SYNCHRONE are now integrated into SCADE 6, which has been distributed since 2008, including the programming constructs `merge`, `reset`, the clock calculus and the type system. Several results have been developed jointly with Jean-Louis Colaço and Bruno Pagano from Esterel-Technologies, such as ways of combining data-flow and hierarchical automata, and techniques for their compilation, initialization analysis, etc.

Dassault-Systèmes (Grenoble R&D center, part of Delmia-automation) developed the language LCM, a variant of LUCID SYNCHRONE that is used for the simulation of factories. LCM follows closely the principles and programming constructs of LUCID SYNCHRONE (higher-order, type inference, mix of data-flow and hierarchical automata). The team in Grenoble is integrating this development into a new compiler for the language Modelica. [4]

In parallel, the goal of REACTIVEML[5] was to integrate a synchronous concurrency model into an existing ML language, with no restrictions on expressiveness, so as to program a large class of reactive systems, including efficient simulations of millions of communicating processes (e.g., sensor networks), video games with many interactions, physical simulations, etc. For such applications, the synchronous model simplifies system design and implementation, but the expressiveness of the algorithmic part of the language is just as essential, as is the ability to create or stop a process dynamically.

The development of REACTIVEML was started by Louis Mandel during his PhD thesis [42], [38] and is ongoing. The language extends OCAML[6] with Esterel-like synchronous primitives — synchronous composition, broadcast communication, pre-emption/suspension — applying the solution of Boussinot [18] to solve causality issues.

[1] http://www.di.ens.fr/~pouzet/lucid-synchrone
[2] The name is a reference to Lustre which stands for "Lucid Synchrone et Temps réel".
[3] http://www.esterel-technologies.com/products/scade-suite/
[4] http://www.3ds.com/products/catia/portfolio/dymola/overview/
[5] http://rml.lri.fr/
[6] More precisely a subset of OCAML without objects or functors.

Several open problems have been solved by Louis Mandel: the interaction between ML features (higher-order) and reactive constructs with a proper type system; efficient simulation that avoids busy waiting. The latter problem is particularly difficult in synchronous languages because of possible reactions to the absence of a signal. In the REACTIVEML implementation, there is no busy waiting: inactive processes have no impact on the overall performance. It turns out that this enables REACTIVEML to simulate millions of (logical) parallel processes and to compete with the best event-driven simulators [43].

REACTIVEML has been used for simulating routing protocols in ad-hoc networks [37] and large scale sensor networks [53]. The designer benefits from a real programming language that gives precise control of the level of simulation (e.g., each network layer up to the MAC layer) and programs can be connected to models of the physical environment programmed with LUTIN [52]. REACTIVEML is used since 2006 by the synchronous team at VERIMAG, Grenoble (in collaboration with France-Telecom) for the development of low-consumption routing protocols in sensor networks.

3.1.2. Relaxing synchrony with buffer communication

In the data-flow synchronous model, the clock calculus is a static analysis that ensures execution in bounded memory. It checks that the values produced by a node are instantaneously consumed by connected nodes (synchronous constraint). To program Kahn process networks with bounded buffers (as in video applications), it is thus necessary to explicitly place nodes that implement buffers. The buffers sizes and the clocks at which data must be read or written have to be computed manually. In practice, it is done with simulation or successive tries and errors. This task is difficult and error prone. The aim of the n-synchronous model is to automatically compute at compile time these values while insuring the absence of deadlock.

Technically, it allows processes to be composed whenever they can be synchronized through a bounded buffer [21], [22]. The new flexibility is obtained by relaxing the clock calculus by replacing the equality of clocks by a sub-typing rule. The result is a more expressive language which still offers the same guarantees as the original. The first version of the model was based on clocks represented as ultimately periodic binary words [57]. It was algorithmically expensive and limited to periodic systems. In [25], an abstraction mechanism is proposed which permits direct reasoning on sets of clocks that are defined as a rational slope and two shifts. An implementation of the n-synchronous model, named LUCY-N, was developed in 2009 [39], as was a formalization of the theory in COQ [26]. We also worked on low-level compiler and runtime support to parallelize the execution of relaxed synchronous systems, proposing a portable intermediate language and runtime library called ERBIUM [44].

This work started as a collaboration between Marc Pouzet (LIP6, Paris, then LRI and Inria Proval, Orsay), Marc Duranton (Philips Research then NXP, Eindhoven), Albert Cohen (Inria Alchemy, Orsay) and Christine Eisenbeis (Inria Alchemy, Orsay) on the real-time programming of video stream applications in set-top boxes. It was significantly extended by Louis Mandel and Florence Plateau during her PhD thesis [47] (supervised by Marc Pouzet and Louis Mandel). Low-level support has been investigated with Cupertino Miranda, Philippe Dumont (Inria Alchemy, Orsay) and Antoniu Pop (Mines ParisTech).

3.1.3. Polyhedral compilation and optimizing compilers

Despite decades of progress, the best parallelizing and optimizing compilers still fail to extract parallelism and to perform the necessary optimizations to harness multi-core processors and their complex memory hierarchies. *Polyhedral compilation* aims at facilitating the construction of more effective optimization and parallelization algorithms. It captures the flow of data between individual instances of statements in a loop nest, allowing to accurately model the behavior of the program and represent complex parallelizing and optimizing transformations. Affine multidimensional scheduling is one of the main tools in polyhedral compilation [32]. Albert Cohen, in collaboration with Cédric Bastoul, Sylvain Girbal, Nicolas Vasilache, Louis-Noël Pouchet and Konrad Trifunovic (LRI and Inria Alchemy, Orsay) has contributed to a large number of research, development and transfer activities in this area.

The relation between polyhedral compilation and data-flow synchrony has been identified through data-flow array languages [36], [35], [54], [33] and the study of the scheduling and mapping algorithms for these

languages. We would like to deepen the exploration of this link, embedding polyhedral techniques into the compilation flow of data-flow, relaxed synchronous languages.

Our previous work led to the design of a theoretical and algorithmic framework rooted in the polyhedral model of compilation, and to the implementation of a set of tools based on production compilers (Open64, GCC) and source-to-source prototypes (PoCC, http://pocc.sourceforge.net). We have shown that not only does this framework simplify the problem of building complex loop nest optimizations, but also that it scales to real-world benchmarks [23], [34], [50], [49]. The polyhedral model has finally evolved into a mature, production-ready approach to solve the challenges of maximizing the scalability and efficiency of loop-based computations on a variety of high performance and embedded targets.

After an initial experiment with Open64 [24], [23], we ported these techniques to the GCC compiler [48], [56], [55], applying them to multi-level parallelization and optimization problems, including vectorization and exploitation of thread-level parallelism. Independently, we made significant progress in the design of effective optimization heuristics, working on the interactions between the semantics of the compiler's intermediate representation and the structure of the optimization space [50], [49], [51]. These results open opportunities for complex optimizations that target larger problems, such as the scheduling and placement of process networks, or the offloading of computational kernels to hardware accelerators (such as GPUs).

3.1.4. Automatic compilation of high performance circuits

For both cost and performance reasons, computing systems tightly couple parts realized in hardware with parts realized in software. The boundary between hardware and software keeps moving with the underlying technology and the external economic pressure. Moreover, thanks to FPGA technology, hardware itself has become programmable. There is now a pressing need from industry for hardware/software co-design, and for tools which automatically turn software code into hardware circuits, or more usually, into hybrid code that simultaneously targets GPUs, multiple cores, encryption ASICs, and other specialized chips.

Departing from customary C-to-VHDL compilation, we trust that sharper results can be achieved from source programs that specify bit-wise time/space behavior in a rigorous synchronous language, rather than just the I/O behavior in some (ill-specified) subset of C. This specification allows the designer to also program the (asynchronous) environment in which to operate the entire system, and to profile/measure/control each variable of the design.

At any time, the designer can edit a single specification of the system, from which both the software and the hardware are automatically compiled, and guaranteed to be compatible. Once correct (functionally and with respect to the behavioral specification), the application can be automatically deployed (and tested) on a hard/soft hybrid co-design support.

Key aspects of the advocated methodology were validated by Jean Vuillemin in the design of a PAL2HDTV video sampler [45], [46]. The circuit was automatically compiled from a synchronous source specification, decorated and guided by a few key hints to the hardware back-end, that targetted an FPGA running at real-time video specifications: a tightly-packed highly-efficient design at 240MHz, generated 100% automatically from the application specification source code, and including all run-time/debug/test/validate ancillary software. It was subsequently commercialized on FPGA by LetItWave, and then on ASIC by Zoran. This successful experience underlines our research perspectives on parallel synchronous programming.

POP ART Project-Team

3. Scientific Foundations

3.1. Embedded systems and their safe design

3.1.1. Safe Design of Embedded Real-time Control Systems

The context of our work is the area of embedded real-time control systems, at the intersection between control theory and computer science. Our contribution consists of methods and tools for their safe design. The systems we consider are intrinsically safety-critical because of the interaction between the embedded, computerized controller, and a physical process having its own dynamics. Such systems are known under various names, notably *cyberphysical systems* and *embedded control systems*. What is important is to design and to analyze the safe behavior of the whole system, which introduces an inherent complexity. This is even more crucial in the case of systems whose malfunction can have catastrophic consequences, for example in transport systems (avionics, railways, automotive), production, medical, or energy production systems (nuclear).

Therefore, there is a need for methods and tools for the design of safe systems. The definition of adequate mathematical models of the behavior of the systems allows the definition of formal calculi. They in turn form a basis for the construction of algorithms for the analysis, but also for the transformation of specifications towards an implementation. They can then be implemented in software environments made available to the users. A necessary complement is the setting-up of software engineering, programming, modeling, and validation methodologies. The motivation of these problems is at the origin of significant research activity, internationally and, in particular, in the European IST network of excellence ARTISTDESIGN (Advanced Real-Time Systems).

3.1.2. Models, Methods and Techniques

The state of the art upon which we base our contributions is twofold.

From the point of view of discrete control, there is a set of theoretical results and tools, in particular in the synchronous approach, often founded on finite or infinite labeled transition systems [31], [39]. During the past years, methodologies for the formal verification [70], [41], control synthesis [72] and compilation, as well as extensions to timed and hybrid systems [69], [32] have been developed. Asynchronous models consider the interleaving of events or messages, and are often applied in the field of telecommunications, in particular for the study of protocols.

From the point of view of verification, we use the methods and tools of symbolic model-checking and of abstract interpretation. From symbolic model-checking, we use BDD techniques [37] for manipulating Boolean functions and sets, and their MTBDD extension for more general functions. Abstract interpretation [43] is used to formalize complex static analysis, in particular when one wants to analyze the possible values of variables and pointers of a program. Abstract interpretation is a theory of approximate solving of fix-point equations applied to program analysis. Most program analysis problems, among which reachability analysis, come down to solving a fix-point equation on the state space of the program. The exact computation of such an equation is generally not possible for undecidability (or complexity) reasons. The fundamental principles of abstract interpretation are: (i) to substitute to the state-space of the program a simpler domain and to transpose the equation accordingly (static approximation); and (ii) to use extrapolation (widening) to force the convergence of the iterative computation of the fix-point in a finite number of steps (dynamic approximation). Examples of static analyses based on abstract interpretation are linear relation analysis [44] and shape analysis [40].

The synchronous approach [6] [60], [61] to reactive systems design gave birth to complete programming environments, with languages like ARGOS, LUSTRE[7], ESTEREL[8], SIGNAL/ POLYCHRONY[9], LUCIDSYNCHRONE[10], SYNDEX[11], or Mode Automata. This approach is characterized by the fact that it considers periodically sampled systems whose global steps can, by synchronous composition, encompass a set of events (known as simultaneous) on the resulting transition. Generally speaking, formal methods are often used for analysis and verification; they are much less often integrated into the compilation or generation of executives (in the sense of executables of tasks combined with the host real-time operating system). They are notoriously difficult to use by end-users, who are usually experts in the application domain, not in formal techniques. This is why encapsulating formal techniques into an automated framework can dramatically improve their diffusion, acceptance, and hence impact. Our work is precisely oriented towards this direction.

3.2. Issues in Design Automation for Complex Systems

3.2.1. Hard Problems

The design of safe real-time control systems is difficult due to various issues, among them their complexity in terms of the number of interacting components, their parallelism, the difference of the considered time scales (continuous or discrete), and the distance between the various theoretical concepts and results that allow the study of different aspects of their behaviors, and the design of controllers.

A currently very active research direction focuses on the models and techniques that allow the automatic use of formal methods. In the field of verification, this concerns in particular the technique of model checking. The verification takes place after the design phase, and requires, in case of problematic diagnostics, expensive backtracks on the specification. We want to provide a more constructive use of formal models, employing them to derive correct executives by formal computation and synthesis, integrated in a compilation process. We therefore use models throughout the design flow from specification to implementation, in particular by automatic generation of embeddable executives.

3.2.2. Applicative Needs

Applicative needs initially come from the fields of safety-critical systems (*e.g.*, avionics, nuclear) and complex systems (telecommunications), embedded in an environment with which they strongly interact (comprising aspects of computer science and control theory). Fields with less criticality, or which support variable degrees of quality of service, such as in the multi-media domain, can also take advantage of methodologies that improve the quality and reliability of software, and reduce the costs of test and correction in the design.

Industrial acceptance, the dissemination, and the deployment of the formal techniques inevitably depend on the usability of such techniques by specialists in the application domain — and not in formal techniques themselves — and also on the integration in the whole design process, which concerns very different problems and techniques. Application domains where the actors are ready to employ specialists in formal methods or advanced control theory are still uncommon. Even then, design methods based on the systematic application of these theoretical results are not ripe. In fields like industrial control, where the use of PLC (Programmable Logic Controller [29]) is dominant, this question can be decisive.

Essential elements in this direction are the proposal of realistic formal models, validated by experiments, of the usual entities in control theory, and functionalities (*i.e.*, algorithms) that correspond indeed to services useful for the designer. Take, for example, the compilation and optimization taking into account the platforms of execution, the possible failures, or the interactions between the defined automatic control and its implementation. In these areas, there are functionalities that commercial tools do not have yet, and to which our results contribute.

[6] http://www.synalp.org
[7] http://www-verimag.imag.fr/SYNCHRONE
[8] http://www.inria.fr/equipes/aoste
[9] http://www.irisa.fr/espresso/Polychrony
[10] http://www.di.ens.fr/~pouzet/lucid-synchrone/
[11] http://www-rocq.inria.fr/syndex

3.2.3. Our Approach

We are proposing effective trade-offs between, on the one hand, expressiveness and formal power, and on the other hand, usability and automation. We focus on the area of specification and construction of correct real-time executives for discrete and continuous control, while keeping an interest in tackling major open problems, relating to the deployment of formal techniques in computer science, especially at the border with control theory. Regarding the applications, we propose new automated functionalities, to be provided to the users in integrated design and programming environments.

3.3. Main Research Directions

The overall consistency of our approach comes from the fact that the main research directions address, under different aspects, the specification and generation of safe real-time control executives based on *formal models*.

We explore this field by linking, on the one hand, the techniques we use, with on the other hand, the functionalities we want to offer. We are interested in questions related to:

Component-Based Design. We investigate two main directions: (i) compositional analysis and design techniques; (ii) adapter synthesis and converter verification.

Programming for embedded systems. Programming for embedded real-time systems is considered within POP ART along three axes: (i) synchronous programming languages, (ii) aspect-oriented programming, (iii) static analysis (type systems, abstract interpretation, ...).

Dependable embedded systems. Here we address the following research axes: (i) static multiprocessor scheduling for fault-tolerance, (ii) multi-criteria scheduling for reliability, (iii) automatic program transformations, (iv) formal methods for fault-tolerant real-time systems.

The creation of easily usable models aims at giving the user the role rather of a pilot than of a mechanics *i.e.*, to offer her/him pre-defined functionalities which respond to concrete demands, for example in the generation of fault tolerant or distributed executives, by the intermediary use of dedicated environments and languages.

The proposal of validated models with respect to their faithful representation of the application domain is done through case studies in collaboration with our partners, where the typical multidisciplinarity of questions across control theory and computer science is exploited.

3.3.1. Component-Based Design

Component-based construction techniques are crucial to overcome the complexity of embedded systems design. However, two major obstacles need to be addressed: the heterogeneous nature of the models, and the lack of results to guarantee correction of the composed system.

The heterogeneity of embedded systems comes from the need to integrate components using different models of computation, communication, and execution, at different levels of abstraction and different time scales. The BIP component framework [5] has been designed, in cooperation with VERIMAG, to support this heterogeneous nature of embedded systems.

Our work focuses on the underlying analysis and construction algorithms, in particular compositional techniques and approaches ensuring correctness by construction (adapter synthesis, strategy mapping). This work is motivated by the strong need for formal, heterogeneous component frameworks in embedded systems design.

3.3.2. Programming for Embedded Systems

Programming for embedded real-time systems is considered along three directions: (i) synchronous programming languages to implement real-time systems; (ii) aspect-oriented programming to specify non-functional properties separately from the base program; (iii) abstract interpretation to ensure safety properties of programs at compile time. We advocate the need for well defined programming languages to design embedded real-time systems with correct-by-construction guarantees, such as bounded time and bounded memory execution. Our original contribution resides in programming languages inheriting features from both synchronous languages

and functional languages. We contribute to the compiler of the HEPTAGON language (whose main inventor is Marc Pouzet, ENS Paris, PARKAS team), the key features of which are: data-flow formal synchronous semantics, strong typing, modular compilation. In particular, we are working on type systems for the clock calculus and the spatial modular distribution.

The goal of Aspect-Oriented Programming (AOP) is to isolate aspects (such as security, synchronization, or error handling) that cross-cut the program basic functionality and whose implementation usually yields tangled code. In AOP, such aspects are specified *separately* and integrated into the program by an automatic transformation process called *weaving*. Although this paradigm has great practical potential, it still lacks formalization, and undisciplined uses make reasoning on programs very difficult. Our work on AOP addresses these issues by studying foundational issues of AOP (semantics, analysis, verification) and by considering domain-specific aspects (availability or fault tolerance aspects) as formal properties.

Finally, the aim of the verification activity in POP ART is to check safety properties on programs, with emphasis on the analysis of the values of data variables (numerical variables, memory heap), mainly in the context of embedded and control-command systems that exibit concurrency features. The applications are not only the proof of functional properties on programs, but also test selection and generation, program transformation, controller synthesis, and fault-tolerance. Our approach is based on abstract interpretation, which consists in inferring properties of the program by solving semantic equations on abstract domains. Much effort is spent on implementing developed techniques in tools for experimentation and diffusion.

3.3.3. Dependable Embedded Systems

Embedded systems must often satisfy safety critical constraints. We address this issue by providing methods and algorithms to design embedded real-time systems with guarantees on their fault-tolerance and/or reliability level.

A first research direction concerns static multiprocessor scheduling of an application specification on a distributed target architecture. We increase the fault-tolerance level of the system by replicating the computations and the communications, and we schedule the redundant computations according to the faults to be tolerated. We also optimize the schedule *w.r.t.* several criteria, including the schedule length, the reliability, and the power consumption.

A second research direction concerns the fault-tolerance management, by reconfigurating the system (for instance by migrating the tasks that were running on a processor upon the failure of this processor) following objectives of fault-tolerance, consistent execution, functionality fulfillment, boundedness and optimality of response time. We base such formal methods on discrete controller synthesis.

A third research direction concerns AOP to weave fault-tolerance aspects in programs and electronic circuits (seen as synthesizable HDL programs) as mentioned in the previous section. A first step in this direction has been the design of automatic transformation method for fault tolerance, which implement a limited (but nonetheless interesting) form of AOP.

S4 Project-Team

3. Scientific Foundations

3.1. Scientific Foundations

The research work of the team is built on top of solid foundations, mainly, algebraic, combinatorial or logical theories of transition systems. These theories cover several sorts of systems which have been studied during the last thirty years: sequential, concurrent, synchronous or asynchronous. They aim at modeling the behavior of finite or infinite systems (usually by abstracting computations on data), with a particular focus on the control flow which rules state changes in these systems. Systems can be autonomous or reactive, that is, embedded in an environment with which the system interacts, both receiving an input flow, and emitting an output flow of events and data. System specifications can be explicit (for instance, when the system is specified by an automaton, extensively defined by a set of states and a set of transitions), or implicit (symbolic transition rules, usually parameterized by state or control variables; partially-synchronized products of finite transition systems; Petri nets; systems of equations constraining the transitions of synchronous reactive systems, according to their input flows; etc.). Specifications can be non-ambiguous, meaning that they fully define at most one system (this holds in the previous cases), or they can be ambiguous, in which case more than one system is conforming to the specification (for instance, when the system is described by logical formulas in the modal mu-calculus, or when the system is described by a set of scenario diagrams, such as *Sequence Diagrams* or *Message Sequence Charts*).

Systems can be described in two ways: either the state structure is described, or only the behavior is described. Both descriptions are often possible (this is the case for formal languages, automata, products of automata, or Petri nets), and moving from one representation to the other is achieved by folding/unfolding operations.

Another taxonomy criteria is the concurrency these models can encompass. Automata usually describe sequential systems. Concurrency in synchronous systems is usually not considered. In contrast, Petri nets or partially-synchronized products of automata are concurrent. When these models are transformed, concurrency can be either preserved, reflected or even, infused. An interesting case is whenever the target architecture requires distributing events among several processes. There, communication-efficient implementations require that concurrency is preserved as far as possible and that, at the same time, causality relations are also preserved. These notions of causality and independence are best studied in models such as concurrent automata, Petri nets or Mazurkiewicz trace languages.

Here are our sources of inspiration regarding formal mathematical tools:

1. Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science - Volume B: Formal Models and Semantics*, Elsevier, 1990.
2. Jörg Desel, Wolfgang Reisig and Grzegorz Rozenberg (eds.), *Lectures on Concurrency and Petri nets*, Lecture Notes in Computer Science, Vol. 3098, Springer, 2004.
3. Volker Diekert and Grzegorz Rozenberg (eds.), *The Book of Traces*, World Scientific, 1995.
4. André Arnold and Damian Niwinski, *Rudiments of Mu-Calculus*, North-Holland, 2001.
5. Gérard Berry, *Synchronous languages for hardware and software reactive systems - Hardware Description Languages and their Applications*, Chapman and Hall, 1997.

Our research exploits decidability or undecidability results on these models (for instance, inclusion of regular languages, bisimilarity on automata, reachability on Petri nets, validity of a formula in the mu-calculus, etc.) and also, representation theorems which provide effective translations from one model to another. For instance, Zielonka's theorem yields an algorithm which maps regular trace languages to partially-synchronized products of finite automata. Another example is the theory of regions, which provides methods for mapping finite or infinite automata, languages, or even *High-Level Message Sequence Charts* to Petri nets. A further example concerns the mu-calculus, in which algorithms computing winning strategies for parity games can be used to synthesize supervisory control of discrete event systems.

Our research aims at providing effective representation theorems, with a particular emphasis on algorithms and tools which, given an instance of one model, synthesize an instance of another model. In particular we have contributed a theory, several algorithms and a tool for synthesizing Petri nets from finite or infinite automata, regular languages, or languages of *High-Level Message Sequence Charts*. This also applies to our work on supervisory control of discrete event systems. In this framework, the problem is to compute a system (the controller) such that its partially-synchronized product with a given system (the plant) satisfies a given behavioral property (control objective, such as a regular language or satisfaction of a mu-calculus formula).

Software engineers often face problems similar to *service adaptation* or *component interfacing*, which in turn, often reduce to particular instances of system synthesis or supervisory control problems.

TRIO Project-Team

3. Scientific Foundations

3.1. Fondation

In order to check for the timing behavior and the reliability of distributed systems, the TRIO team developed several techniques based on deterministic approaches ; in particular, we apply and extend analytical evaluation of worst case response times and when necessary, e.g. for large-scale communication systems as Internet based applications, we use techniques based on network calculus.

When the environment might lead to hazards (e.g. electromagnetic interferences causing transmission errors and bit-flips in memory), or when some characteristics of the system are not perfectly known or foreseeable beforehand, we model and analyze the uncertainties using stochastic models, for instance, models of the frame transmission patterns or models of the transmission errors. In the context of real time computing, we are in general much more interested by worst-case results over a given time window than by average and asymptotic results, and dedicated analyses in that area have been developed in our team over the last 10 years.

In the design of discrete event systems with hard real time constraints, the scheduling of the system's activities is of crucial importance. This means that we have to devise scheduling policies that ensure the respect of time constraints on line and / or optimize the behavior of the system according to some other application-dependent performance criteria.

In order to foster the best quality for programs, their understanding has to be automated, or at made significantly easier. Thus, we focus on analyzing and modeling program code, program structure and program behavior, and presenting these pieces of information to the user (in our case, program designers and program developers). Modeling user interaction is to come as well.

In the design of embedded, autonomous systems, power and energy usage is of paramount importance. We thus strive to model energy usage, based on actual hardware, and derive context-aware optimizations to decrease peak power and overall energy usage.

VERTECS Project-Team

3. Scientific Foundations

3.1. Underlying models

The formal models we use are mainly automata-like structures such as labelled transition systems (LTS) and some of their extensions: an LTS is a tuple $M = (Q, \Lambda, \rightarrow, q_o)$ where Q is a non-empty set of states; $q_o \in Q$ is the initial state; A is the alphabet of actions, $\rightarrow \subseteq Q \times \Lambda \times Q$ is the transition relation. These models are adapted for testing and controller synthesis.

To model reactive systems in the testing context, we use Input/Output labeled transition systems (IOLTS for short). In this setting, the interactions between the system and its environment (where the tester lies) must be partitioned into inputs (controlled by the environment), outputs (observed by the environment), and internal (non observable) events modeling the internal behavior of the system. The alphabet Λ is then partitioned into $\Lambda_! \cup \Lambda_? \cup \mathcal{T}$ where $\Lambda_!$ is the alphabet of outputs, $\Lambda_?$ the alphabet of inputs, and $\mathcal{T}$ the alphabet of internal actions.

In the controller synthesis theory, we also distinguish between controllable and uncontrollable events ($\Lambda = \Lambda_c \cup \Lambda_{uc}$), observable and unobservable events ($\Lambda = \Lambda_O \cup \mathcal{T}$).

In the context of verification, we also use Timed Automata. A timed automaton is a tuple $A = (L, X, E, \mathcal{I})$ where L is a set of locations, X is a set of clocks whose valuations are positive real numbers, $E \subseteq L \times \mathcal{G}(X) \times 2^X \times L$ is a finite set of edges composed of a source and a target state, a guard given by a finite conjunction of expressions of the form $x \sim c$ where x is a clock, c is a natural number and $\sim \in \{<, \leq, =, \geq, >\}$, a set of resetting clocks, and $\mathcal{I} : \mathcal{L} \rightarrow \mathcal{G}(\mathcal{X})$ assigns an invariant to each location [22]. The semantics of a timed automaton is given by a (infinite states) labelled transition system whose states are composed of a location and a valuation of clocks.

Also, for verification purposes, we use graph grammars that are a general tool to define families of graphs. Such grammars are formed by a set of rules whose left-hand sides are hyperedges and right-hand sides are hypergraphs. For graphs with finite degree, these grammars characterise transition graphs of pushdown automata (the correspondence between graphs generated by grammars and transition graphs of pushdown automata is bijective). Graph grammars provide a simple yet powerful setting to define and study infinite state systems.

In order to cope with models closer to practical specification languages, we also need higher level models encompassing both control and data aspects. We defined (input-output) symbolic transition systems ((IO)STS), which are extensions of (IO)LTS that convey or operate on data (i.e., program variables, communication parameters, symbolic constants) through message passing, guards, and assignments. Formally, an IOSTS is a tuple (V, Θ, Σ, T), where V is a set of variables (including a counter variable encoding the control structure), Θ is the initial condition defined by a predicate on V, Σ is the finite alphabet of actions, where each action has a signature (just like in IOLTS, Σ can be partitioned as e.g. $\Sigma_? \cup \Sigma_! \cup \Sigma_\tau$), T is a finite set of symbolic transitions of the form $t = (a, p, G, A)$ where a is an action (possibly with a polarity reflecting its input/output/internal nature), p is a tuple of communication parameters, G is a guard defined by a predicate on p and V, and A is an assignment of variables. The semantics of IOSTS is defined in terms of (IO)LTS where states are vectors of values of variables, and transitions between them are labelled with instantiated actions (action with valued communication parameter). This (IO)LTS semantics allows us to perform syntactical transformations at the (IO)STS level while ensuring semantical properties at the (IO)LTS level. We also consider extensions of these models with added features such as recursion, fifo channels, etc. An alternative to IOSTS for specifying systems with data variables is the model of synchronous dataflow equations.

Our research is based on well established theories: conformance testing, supervisory control, abstract interpretation, and theorem proving. Most of the algorithms that we employ take their origins in these theories:

- graph traversal algorithms (breadth first, depth first, strongly connected components, ...). We use these algorithms for verification as well as test generation and control synthesis.
- BDDs (Binary Decision Diagrams) algorithms, for manipulating Boolean formulae, and their MTBDDs (Multi-Terminal Decision Diagrams) extension for manipulating more general functions. We use these algorithms for verification, test generation and control.
- abstract interpretation algorithms, specifically in the abstract domain of convex polyhedra (for example, Chernikova's algorithm for the computation of dual forms). Such algorithms are used in verification and test generation.
- logical decision algorithms, such as satisfiability of formulas in Presburger arithmetics. We use these algorithms during generation and execution of symbolic test cases.

3.2. Verification

Verification in its full generality consists in checking that a system, which is specified by a formal model, satisfies a required property. Verification takes place in our research in two ways: on the one hand, a large part of our work, and in particular controller synthesis and conformance testing, relies on the ability to solve some verification problems. Many of these problems reduce to reachability and coreachability questions on a formal model (a state s is *reachable from an initial state* s_i if an execution starting from s_i can lead to s; s is *coreachable from a final state* s_f if an execution starting from s can lead to s_f). These are important cases of verification problems, as they correspond to the verification of safety properties.

On the other hand we investigate verification on its own in the context of complex systems. For expressivity purposes, it is necessary to be able to describe faithfully and to deal with complex systems. Some particular aspects require the use of infinite state models. For example asynchronous communications with unknown transfer delay (and thus arbitrary large number of messages in transit) are correctly modeled by unbounded FIFO queues, and real time systems require the use of continuous variables which evolve with time. Apart from these aspects requiring infinite state data structure, systems often include uncertain or random behaviours (such as failures, actions from the environment), which it make sense to model through probabilities. To encompass these aspects, we are interested in the verification of systems equipped with infinite data structures and/or probabilistic features.

When the state space of the system is infinite, or when we try to evaluate performances, standard model-checking techniques (essentially graph algorithms) are not sufficient. For large or infinite state spaces, symbolic model-checking or approximation techniques are used. Symbolic verification is based on efficient representations of sets of states and permits exact model-checking of some well-formed infinite-state systems. However, for feasibility reasons, it is often mandatory to use approximate computations, either by computing a finite abstraction and resort to graph algorithms, or preferably by using more sophisticated abstract interpretation techniques. For systems with stochastic aspects, a quantitative analysis has to be performed, in order to evaluate the performances. Here again, either symbolic techniques (e.g. by grouping states with similar behaviour) or approximation techniques should be used.

We detail below verification topics we are interested in: abstract interpretation, quantitative model-checking and analysis of systems defined by graph grammars.

3.2.1. Abstract interpretation and data handling

Most problems in test generation or controller synthesis reduce to state reachability and state coreachability problems which can be solved by fixpoint computations of the form $x = F(x), x \in C$ where C is a lattice. In the case of reachability analysis, if we denote by S the state space of the considered program, C is the lattice $\wp(S)$ of sets of states, ordered by inclusion, and F is roughly the "*successor states*" function defined by the program.

The big change induced by taking into account the data and not only the (finite) control of the systems under study is that the fixpoints become uncomputable. The undecidability is overcome by resorting to approximations, using the theoretical framework of Abstract Interpretation [24]. The fundamental principles of Abstract Interpretation are:

1. to substitute to the *concrete domain*C a simpler *abstract domain*A (static approximation) and to transpose the fixpoint equation into the abstract domain, so that one has to solve an equation $y = G(y), y \in A$;
2. to use a *widening operator* (dynamic approximation) to make the iterative computation of the least fixpoint of G converge after a finite number of steps to some upper-approximation (more precisely, a post-fixpoint).

Approximations are conservative so that the obtained result is an upper-approximation of the exact result. In simple cases the state space that should be abstracted has a simple structure, but this may be more complicated when variables belong to different data types (Booleans, numerics, arrays) and when it is necessary to establish *relations* between the values of different types.

3.2.2. Model-checking quantitative systems

Model-checking techniques for finite-state systems are now quite developed, and a current challenge is to adapt them as much as possible to infinite-state systems. We detail below two types of models we are interested in: timed automata and infinite-state probabilistic systems.

Model-checking timed automata The model of timed automata, introduced by Alur and Dill in the 90's [22] is commonly used to represent real-time systems. Timed automata consist of an extension of finite automata with continuous variables, called clocks, that evolve synchronously with time, and can be tested and reset along an execution. Despite their uncountable state space, checking reachability, and more generally ω-regular properties, is decidable *via* the construction of a finite abstraction, the so-called region automaton. The recent developments in model-checking timed automata have aimed at modelling and verifying quantitative aspects encompassing timing constraints, for example costs, probabilities, frequencies. These quantitative questions demand advanced techniques that go far beyond the classical methods.

Model-checking infinite state probabilistic systems Model-checking techniques for finite state probabilistic systems are now quite developed. Given a finite state Markov chain, for example, one can check whether some property holds almost surely (i.e. the set of executions violating the property is negligible), and one can even compute (or at leat approximate as close as wanted) the probability that some property holds. In general, these techniques cannot be adapted to infinite state probabilistic systems, just as model-checking algorithms for finite state systems do not carry over to infinite state systems. For systems exhibiting complex data structures (such as unbounded queues, continuous clocks) and uncertainty modeled by probabilities, it can thus be hard to design model-checking algorithms. However, in some cases, especially when considering qualitative verification, symbolic methods can lead to exact results. Qualitative questions aim neither at computing nore at approximating a probability, but are only concerned with almost-sure or non neglectible behaviours (that is events of probability either one or non zero). In some cases, qualitative model-checking can be derived from a combination of techniques for infinite state systems (such as abstractions) with methods for finite state probabilistic systems. However, when one is interested in computing (or rather approximating) precise probability values (neither 0 nor 1), exact methods are scarce. To deal with these questions, we either try to restrict to classes of systems where exact computations can be made, or look for approximation algorithms.

3.2.3. Analysis of infinite state systems defined by graph grammars

Currently, many techniques (reachability, model checking, ...) from finite state systems have been generalised to pushdown systems, that can be modeled by graph grammars. Several such extensions heavily depend on the actual definition of the pushdown automata, for example, how many top stack symbols may be read, or whether the existence of ε-transitions (silent transitions) is allowed. Many of these restrictions do not affect the actual structure of the graph, and interesting properties like reachability or satisfiability (of a formula) only depend on the structure of a graph.

Deterministic graph grammars enable us to focus on structural properties of systems. The connection with finite graph algorithms is often straightforward: for example reachability is obtained by iterating the finite graph algorithm iterated on the right hand sides of the rules. On the other hand, extending these grammars with time or probabilities is not straightforward: qualitative values associated to different copies (in the graph) of the same vertex (in the grammar) may differ, introducing complex equations. Furthermore, the fact that the left-hand sides of rules are single hyperarcs is a strong restriction. But removing this restriction would lead to non-recursive graphs. Identifying decidable families of graphs defined by contextual graph grammars is also very challenging.

3.3. Automatic test generation

We are mainly interested in conformance testing, which consists in checking whether a black box implementation under test (the real system that is only known by its interface) behaves correctly with respect to its specification (the reference which specifies the intended behavior of the system). In the line of model-based testing, we use formal specifications and their underlying models to unambiguously define the intended behavior of the system, to formally define conformance and to design test case generation algorithms. The difficult problems are to generate test cases that correctly identify faults (the oracle problem) and, as exhaustiveness is impossible to reach in practice, to select an adequate subset of test cases that are likely to detect faults. Hereafter we detail some elements of the models, theories and algorithms we use.
We use IOLTS (or IOSTS) as formal models for specifications, implementations, test purposes, and test cases. We adapt a well established theory of conformance testing [30], which formally defines conformance as a relation between formal models of specifications and implementations. This conformance relation, called **ioco** compares the visible behaviors (called *suspension traces*) of the implementation I (denoted by $STraces(I)$) with those of the specification S ($STraces(S)$). Suspension traces are sequence of inputs, outputs or quiescence (absence of action denoted by δ), thus abstracting away internal behaviors that cannot be observed by testers. Intuitively, I $ioco$ S if after a suspension trace of the specification, the implementation I can only show outputs and quiescences of the specification S. We re-formulated ioco as a partial inclusion of visible behaviors as follows:

$$I \; ioco \; S \Leftrightarrow STraces(I) \cap [STraces(S).\Lambda_!^\delta \setminus STraces(S)] = \varnothing.$$

In other words, suspension traces of I which are suspension traces of S prolongated by an output or quiescence, should still be suspension traces of S.

Interestingly, this characterization presents conformance with respect to S as a safety property of suspension traces of I. The negation of this property is charaterized by a *canonical tester*$Can(S)$ which recognizes exactly $[STraces(S).\Lambda_!^\delta \setminus STraces(S)]$, the set of non-conformant suspension traces. This *canonical tester* also serves as a basis for test selection.

Test cases are processes executed against implementations in order to detect non-conformance. They are also formalized by IOLTS (or IOSTS) with special states indicating *verdicts*. The execution of test cases against implementations is formalized by a parallel composition with synchronization on common actions. A *Fail* verdict means that the implementation under test (IUT) is rejected and should correspond to non-conformance, a *Pass* verdict means that the IUT exhibited a correct behavior and some specific targeted behaviour has been observed, while an *Inconclusive* verdict is given to a correct behavior that is not targeted.

Test suites (sets of test cases) are required to exhibit some properties relating the verdict they produce to the conformance relation. Soundness means that only non conformant implementations should be rejected by a test suite and exhaustiveness means that every non conformant implementation may be rejected by the test suite. Soundness is not difficult to obtain, but exhaustiveness is not possible in practice and one has to select test cases.

Test selection is often based on the coverage of some criteria (state coverage, transition coverage, etc). But test cases are often associated with *test purposes* describing some abstract behaviors targeted by a test case. In our framework, test purposes are specified as IOLTS (or IOSTS) associated with marked states or dedicated variables, giving them the status of automata or observers accepting runs (or sequences of actions or suspension traces). Selection of test cases amounts to selecting traces of the canonical tester accepted by the test purpose. The resulting test case is then both an observer of the negation of a safety property (non-conformance wrt. S), and an observer of a reachability property (acceptance by the test purpose). Selection can be reduced to a model-checking problem where one wants to identify states (and transitions between them) which are both reachable from the initial state and co-reachable from the accepting states. We have proved that these algorithms ensure soundness. Moreover the (infinite) set of all possibly generated test cases is also exhaustive. Apart from these theoretical results, our algorithms are designed to be as efficient as possible in order to be able to scale up to real applications.

Our first test generation algorithms are based on enumerative techniques, thus adapted to IOLTS models, and optimized to fight the state-space explosion problem. On-the-fly algorithms where designed and implemented in the TGV tool, which consist in computing co-reachable states from a target state during a lazy exploration of the set of reachable states in a product of the specification and the test purpose [25]. However, this enumerative technique suffers from some limitations when specification models contain data.

More recently, we have explored symbolic test generation techniques for IOSTS specifications [29]. The objective is to avoid the state space explosion problem induced by the enumeration of values of variables and communication parameters. The idea consists in computing a test case under the form of an *IOSTS*, i.e., a reactive program in which the operations on data are kept in a symbolic form. Test selection is still based on test purposes (also described as IOSTS) and involves syntactical transformations of IOSTS models that should ensure properties of their IOLTS semantics. However, most of the operations involved in test generation (determinisation, reachability, and coreachability) become undecidable. For determinisation we employ heuristics that allow us to solve the so-called bounded observable non-determinism (i.e., the result of an internal choice can be detected after finitely many observable actions). The product is defined syntactically. Finally test selection is performed as a syntactical transformation of transitions which is based on a semantical reachability and co-reachability analysis. As both problems are undecidable for IOSTS, syntactical transformations are guided by over-approximations using abstract interpretation techniques. Nevertheless, these over-approximations still ensure soundness of test cases [26]. These techniques are implemented in the STG tool (see 5.1), with an interface with NBAC used for abstract interpretation.

3.4. Control synthesis

The supervisory control problem is concerned with ensuring (not only checking) that a computer-operated system works correctly. More precisely, given a system model and a required property, the problem is to control the model's behavior, by coupling it to a supervisor, such that the controlled system satisfies the property [28]. The models used are LTSs and the associated languages, where one makes a distinction between *controllable* and *non-controllable* actions and between *observable* and *non-observable* actions. Typically, the controlled system is constrained by the supervisor, which can block on the system's controllable actions in order to force it to behave as specified by the property. The control synthesis problem can be seen as a constructive verification problem: building a supervisor that prevents the system from violating a property. Several kinds of properties can be enforced such as reachability, invariance (i.e. safety), attractivity, etc. Techniques adapted from model checking are used to compute the supervisor. Optimality must be taken into account as one often wants to obtain a supervisor that constrains the system as few as possible.

Supervisory control theory overview. Supervisory control theory deals with control of Discrete Event Systems. In this theory, the behavior of the system S is assumed not to be fully satisfactory. Hence, it has to be reduced by means of a feedback control (named Supervisor or Controller) in order to achieve a given set of requirements [28]. Namely, if S denotes the model of the system and Φ a safety property to be enforced on S, the problem consists of computing a supervisor $\mathcal{C}$ such that

$$S \| \mathcal{C} \models \Phi \tag{1}$$

where $\|$ is the classical parallel composition of LTSs. Given S, some events of S are said to be uncontrollable (Σ_{uc}), i.e., the occurrence of these events cannot be prevented by a supervisor, while the others are controllable (Σ_c). It means that all the supervisors satisfying (1) are not good candidates. The behavior of the controlled system must respect an additional condition that happens to be similar to the $ioco$ conformance relation previously defined in 3.3 . This condition is called the *controllability condition* and it may be stated as

$$\mathcal{L}(S \| \mathcal{C}) \Sigma_{uc} \cap \mathcal{L}(S) \subseteq \mathcal{L}(S \| \mathcal{C}) \tag{2}$$

Namely, when acting on S, a supervisor is not allowed to disable uncontrollable events. Given a safety property Φ, that can be modeled by an LTS A_Φ, there actually exist many different supervisors satisyfing both (1) and (2). Among all the valid supervisors, we are interested in computing the supremal one, ie the one that restricts the system as few as possible. It has been shown in [28] that such a supervisor always exists and is unique. It gives access to a behavior of the controlled system that is called the supremal controllable sub-language of A_Φ w.r.t. S and Σ_{uc}. In some situations, it may also be interesting to force the controlled system to be non-blocking (See [28] for details).

The underlying techniques are similar to the ones used for Automatic Test Generation. They consist of computing the product of the system model and A_Φ and to remove the states of the product that may lead to subsequent states violating the property by triggering only uncontrollable events.

ABSTRACTION Project-Team

3. Scientific Foundations

3.1. Abstract Interpretation Theory

The abstract interpretation theory [41], [32], [42], is the main scientific foundation of the work of the ABSTRACTION project-team. Its main current application is on the safety and security of complex hardware and software computer systems either sequential [41], [34], or parallel [36] with shared memory [33], [35], [44] or synchronous message [43] communication.

Abstract interpretation is a theory of sound approximation of mathematical structures, in particular those involved in the behavior of computer systems. It allows the systematic derivation of sound methods and algorithms for approximating undecidable or highly complex problems in various areas of computer science (semantics, verification and proof, model-checking, static analysis, program transformation and optimization, typing, software steganography, etc...) and system biology (pathways analysis).

3.2. Formal Verification by Abstract Interpretation

The *formal verification* of a program (and more generally a computer system) consists in proving that its *semantics* (describing "what the program executions actually do") satisfies its *specification* (describing "what the program executions are supposed to do").

Abstract interpretation formalizes the idea that this formal proof can be done at some level of abstraction where irrelevant details about the semantics and the specification are ignored. This amounts to proving that an *abstract semantics* satisfies an *abstract specification*. An example of abstract semantics is Hoare logic while examples of abstract specifications are invariance, partial, or total correctness. These examples abstract away from concrete properties such as execution times.

Abstractions should preferably be *sound* (no conclusion derived from the abstract semantics is wrong with respect to the program concrete semantics and specification). Otherwise stated, a proof that the abstract semantics satisfies the abstract specification should imply that the concrete semantics also satisfies the concrete specification. Hoare logic is a sound verification method, debugging is not (since some executions are left out), bounded model checking is not either (since parts of some executions are left out). Unsound abstractions lead to *false negatives* (the program may be claimed to be correct/non erroneous with respect to the specification whereas it is in fact incorrect). Abstract interpretation can be used to design sound semantics and formal verification methods (thus eliminating all false negatives).

Abstractions should also preferably be *complete* (no aspect of the semantics relevant to the specification is left out). So if the concrete semantics satisfies the concrete specification this should be provable in the abstract. However program proofs (for non-trivial program properties such as safety, liveness, or security) are undecidable. Nevertheless, we can design tools that address undecidable problems by allowing the tool not to terminate, to be driven by human intervention, to be unsound (e.g. debugging tools omit possible executions), or to be incomplete (e.g. static analysis tools may produce false alarms). Incomplete abstractions lead to *false positives* or *false alarms* (the specification is claimed to be potentially violated by some program executions while it is not). Semantics and formal verification methods designed by abstract interpretation may be complete (e.g. [39], [40], [47]) or incomplete (e.g. [2]).

Sound, automatic, terminating and precise tools are difficult to design. Complete automatic tools to solve non-trivial verification problems cannot exist, by undecidability. However static analysis tools producing very few or no false alarms have been designed and used in industrial contexts for specific families of properties and programs [45]. In all cases, abstract interpretation provides a systematic construction method based on the effective approximation of the concrete semantics, which can be (partly) automated and/or formally verified.

Abstract interpretation aims at:

- providing a basic coherent and conceptual theory for understanding in a unified framework the multiplicity of ideas, concepts, reasonings, methods, and tools on formal program analysis and verification [41], [42];
- guiding the correct formal design of *abstract semantics* [40], [47] and automatic tools for *program analysis* (computing an abstract semantics) and *program verification* (proving that an abstract semantics satisfies an abstract specification) [37].

Abstract interpretation theory studies semantics (formal models of computer systems), abstractions, their soundness, and completeness.

In practice, abstract interpretation is used to design analysis, compilation, optimization, and verification tools which must automatically and statically determine properties about the runtime behavior of programs. For example the ASTRÉE static analyzer (Section 5.2), which was developed by the team over the last decade, aims at proving the absence of runtime errors in programs written in the C programming language. It was originally used in the aerospace industry to verify very large, synchronous, time-triggered, real-time, safety-critical, embedded software and its scope of application was later broadly widened. ASTRÉE is now industrialized by AbsInt Angewandte Informatik GmbH and is commercially available.

3.3. Advanced Introductions to Abstract Interpretation

A short, informal, and intuitive introduction to the theory of abstract interpretation can be found in [37], see also "AbstractInterpretationinaNutshell" [1] on the web. A more comprehensive introduction is available online [2]. The paper entitled "Basicconceptsofabstractinterpretation" [38] and an elementary "courseonabstract interpretation" [3] can also be found on the web.

[1] www.di.ens.fr/~cousot/AI/IntroAbsInt.html
[2] www.di.ens.fr/~cousot/AI/
[3] web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/

ATEAMS Project-Team

3. Scientific Foundations

3.1. Research method

We are inspired by formal methods and logic to construct new tools for software analysis, transformation and generation. We try and proof the correctness of new algorithms using any means necessary.

Nevertheless we mainly focus on the study of existing (large) software artifacts to validate the effectiveness of new tools. We apply the scientific method. To (in)validate our hypothesis we often use detailed manual source code analysis, or we use software metrics, and we have started to use more human subjects (programmers).

Note that we maintain ties with the CWI spinoff "Software Improvement Group" which services most of the Dutch software industry and government and many European companies as well. This provides access to software systems and information about software systems that is valuable in our research.

3.2. Software analysis

This research focuses on source code; to analyze it and to transform it. Each analysis or transformation begins with fact extraction. After the we may analyze specific software systems or large bodies of software systems. Our goal is to improve software systems by learning to understand the causes of complexity.

The mother and father of fact extraction techniques are probably Lex, a scanner generator, and AWK, a language intended for fact extraction from textual records and report generation. Lex is intended to read a file character-by-character and produce output when certain regular expressions (for identifiers, floating point constants, keywords) are recognized. AWK reads its input line-by-line and regular expression matches are applied to each line to extract facts. User-defined actions (in particular print statements) can be associated with each successful match. This approach based on regular expressions is in wide use for solving many problems such as data collection, data mining, fact extraction, consistency checking, and system administration. This same approach is used in languages like Perl, Python, and Ruby. Murphy and Notkin have specialized the AWK-approach for the domain of fact extraction from source code. The key idea is to extend the expressivity of regular expressions by adding context information, in such a way that, for instance, the begin and end of a procedure declaration can be recognized. This approach has, for instance, been used for call graph extraction but becomes cumbersome when more complex context information has to be taken into account such as scope information, variable qualification, or nested language constructs. This suggests using grammar-based approaches as will be pursued in the proposed project. Another line of research is the explicit instrumentation of existing compilers with fact extraction capabilities. Examples are: the GNU C compiler GCC, the CPPX C++ compiler, and the Columbus C/C++ analysis framework. The Rigi system provides several fixed fact extractors for a number of languages. The extracted facts are represented as tuples (see below). The CodeSurfer source code analysis tool extracts a standard collection of facts that can be further analyzed with built-in tools or user-defined programs written in Scheme. In all these cases the programming language as well as the set of extracted facts are fixed thus limiting the range of problems that can be solved.

The approach we want to explore is the use of syntax-related program patterns for fact extraction. An early proposal for such a pattern-based approach is described in: a fixed base language (either C or PL/1 variant) is extended with pattern matching primitives. In our own previous work on RScript we have already proposed a query algebra to express direct queries on the syntax tree. It also allows the querying of information that is attached to the syntax tree via annotations. A unifying view is to consider the syntax tree itself as "facts" and to represent it as a relation. This idea is already quite old. For instance, Linton proposes to represent all syntactic as well as semantic aspects of a program as relations and to use SQL to query them. Due to the lack of expressiveness of SQL (notably the lack of transitive closures) and the performance problems encountered, this approach has not seen wider use.

Another approach is proposed by de Moor and colleagues and uses path expressions on the syntax tree to extract program facts and formulate queries on them. This approach builds on the work of Paige and attempts to solve a classic problem: how to incrementally update extracted program facts (relations) after the application of a program transformation.

Parsing is a fundamental tool for fact extraction for source code. Our group has longstanding contributions in the field of Generalized LR parsing and Scannerless parsing. Such generalized parsing techniques enable generation of parsers for a wide range of real (legacy) programming languages, which is highly relevant for experimental research and validation.

3.2.1. Goals

The main goal is to replace labour-intensive manual programming of fact extractors by automatic generation from annotated grammars or other concise and formal notation. There is a wide open scientific challenge here: to create a uniform and generic framework for fact extraction that is superior to current more ad-hoc approaches. We expect to develop new ideas and techniques for generic (language-parametric) fact extraction from source code and other software artifacts.

Given the advances made in fact extraction we are starting to apply our techniques to observe source code and analyze it in detail.

3.3. Relational paradigm

For any source code analysis or transformation, after fact extraction comes elaboration, aggregation or other further analyses of these facts. For fact analysis, we base our entire research on the simple formal concept of a “relation”.

There are at least three lines of research that have explored the use of relations. First, in SQL, n-ary relations are used as basic data type and queries can be formulated to operate on them. SQL is widely used in database applications and a vast literature on query optimization is available. There are, however, some problems with SQL in the applications we envisage: (a) Representing facts about programs requires storing program fragments (e.g., tree-structured data) and that is not easy given the limited built-in datatypes of SQL; (b) SQL does not provide transitive closures, which are essential for computing many forms of derived information; (c) More generally, SQL does not provide fixed-point computations that help to solve sets of equations. Second, in Prolog, Horn clauses can be used to represent relational facts and inference rules for deriving new facts. Although the basic paradigm of Prolog is purely declarative, actual Prolog implementations add imperative features that increase the efficiency of Prolog programs but hide the declarative nature of the language. Extensions of Prolog with recursion have resulted in Datalog in many variations [AHV95]. In F(p)–L a Prolog database and a special-purpose language are used to represent and query program facts.

Third, in SETL, the basic data type was the set. Since relations can easily be represented as sets of tuples, relation-based computations can, in principle, be expressed in SETL. However, SETL as a language was very complicated and has not survived. However, work on programming with sets, bags and lists has continued well into the 90’s and has found a renewed interested with the revival of Lisp dialects in 2008 and 2009.

We have already mentioned the relational program representation by Linton. In Rigi, a tuple format (RSF) is introduced to represent untyped relations and a language (RCL) to manipulate them. Relational algebra is used in GROK, Crocopat and Relation Partition Algebra (RPA) to represent basic facts about software systems and to query them. In GUPRO graphs are used to represent programs and to query them. Relations have also been proposed for software manufacture, software knowledge management, and program slicing. Sometimes, set constraints are used for program analysis and type inference. More recently, we have carried out promising experiments in which the relational approach is applied to problems in software analysis and feature analysis. Typed relations can be used to decouple extraction, analysis and visualization of source code artifacts. These experiments confirm the relevance and viability of the relational approach to software analysis, and also indicate a certain urgency of the research direction of this team.

3.3.1. Goals

- New ideas and techniques for the efficient implementation of a relation-based specification formalism.
- Design and prototype implementation of a relation-based specification language that supports the use of extracted facts (Rascal).
- We target at uniform reformulations of existing techniques and algorithms for software analysis as well as the development of new techniques using the relational paradigm.
- We apply the above in the reformulation of refactorings for Java and domain specific languages.

3.4. Refactoring and Transformation

The final goal, to be able to safely refactor or transform source code can be realized in strong collaboration with extraction and analysis.

Software refactoring is usually understood as changing software with the purpose of increasing its readability and maintainability rather than changing its external behavior. Refactoring is an essential tool in all agile software engineering methodologies. Refactoring is usually supported by an interactive refactoring tool and consists of the following steps:

- Select a code fragment to refactor.
- Select a refactoring to apply to it.
- Optionally, provide extra parameter needed by the refactoring (e.g., a new name in a renaming).

The refactoring tool will now test whether the preconditions for the refactoring are satisfied. Note that this requires fact extraction from the source code. If this fails the user is informed.The refactoring tool shows the effects of the refactoring before effectuating them. This gives the user the opportunity to disable the refactoring in specific cases.The refactoring tool applies the refactoring for all enabled cases. Note that this implies a transformation of the source code. Some refactorings can be applied to any programming language (e.g., rename) and others are language specific (e.g., Pull Up Method). At http://www.refactoring.com an extensive list of refactorings can be found.

There is hardly any general and pragmatic theory for refactoring, since each refactoring requires different static analysis techniques to be able to check the preconditions. Full blown semantic specification of programming languages have turned out to be infeasible, let alone easily adaptable to small changes in language semantics. On the other hand, each refactoring is an instance of the extract, analyze and transform paradigm. Software transformation regards more general changes such as adding functionality and improving non-functional properties like performance and reliability. It also includes transformation from/to the same language (source-to-source translation) and transformation between different languages (conversion, translation). The underlying techniques for refactoring and transformation are mostly the same. We base our source code transformation techniques on the classical concept of term rewriting, or aspects thereof. It offers simple but powerful pattern matching and pattern construction features (list matching, AC Matching), and type-safe heterogenous data-structure traversal methods that are certainly applicable for source code transformation.

3.4.1. Goals

Our goal is to integrate the techniques from program transformation completely with relational queries. Refactoring and transformation form the Achilles Heel of any effort to change and improve software. Our innovation is in the strict language-parametric approach that may yield a library of generic analyses and transformations that can be reused across a wide range of programming and application languages. The challenge is to make this approach scale to large bodies of source code and rapid response times for precondition checking.

3.5. The Rascal Meta-programming language

The Rascal Domain Specific Language for Source code analysis and Transformation is developed by ATeams. It is a language specifically designed for any kind of meta programming.

Meta programming is a large and diverse area both conceptually and technologically. There are plentiful libraries, tools and languages available but integrated facilities that combine both source code analysis and source code transformation are scarce. Both domains depend on a wide range of concepts such as grammars and parsing, abstract syntax trees, pattern matching, generalized tree traversal, constraint solving, type inference, high fidelity transformations, slicing, abstract interpretation, model checking, and abstract state machines. Examples of tools that implement some of these concepts are ANTLR, ASF+SDF, CodeSurfer, Crocopat, DMS, Grok, Stratego, TOM and TXL. These tools either specialize in analysis or in transformation, but not in both. As a result, combinations of analysis and transformation tools are used to get the job done. For instance, ASF+SDF relies on RScript for querying and TXL interfaces with databases or query tools. In other approaches, analysis and transformation are implemented from scratch, as done in the Eclipse JDT. The TOM tool adds transformation primitives to Java, such that libraries for analysis can be used directly. In either approach, the job of integrating analysis with transformation has to be done over and over again for each application and this requires a significant investment.

We propose a more radical solution by completely merging the set of concepts for analysis and transformation of source code into a single language called Rascal. This language covers the range of applications from pure analyses to pure transformations and everything in between. Our contribution does not consist of new concepts or language features *per se*, but rather the careful collaboration, integration and cross-fertilization of existing concepts and language features.

3.5.1. Goals

The goals of Rascal are: (a) to remove the cognitive and computational overhead of integrating analysis and transformation tools, (b) to provide a safe and interactive environment for constructing and experimenting with large and complicated source code analyses and transformations such as, for instance, needed for refactorings, and (c) to be easily understandable by a large group of computer programming experts. Rascal is not limited to one particular object programming language, but is generically applicable. Reusable, language specific, functionality is realized as libraries.

CARTE Project-Team

3. Scientific Foundations

3.1. Computer Virology

From a historical point of view, the first official virus appeared in 1983 on Vax-PDP 11. At the very same time, a series of papers was published which always remains a reference in computer virology: Thompson [70], Cohen [39] and Adleman [29]. The literature which explains and discusses practical issues is quite extensive [43], [45]. However, there are only a few theoretical/scientific studies, which attempt to give a model of computer viruses.

A virus is essentially a self-replicating program inside an adversary environment. Self-replication has a solid background based on works on fixed point in λ-calculus and on studies of von Neumann [75]. More precisely we establish in [35] that Kleene's second recursion theorem [58] is the cornerstone from which viruses and infection scenarios can be defined and classified. The bottom line of a virus behavior is

1. a virus infects programs by modifying them,
2. a virus copies itself and can mutate,
3. it spreads throughout a system.

The above scientific foundation justifies our position to use the word virus as a generic word for self-replicating malwares. There is yet a difference. A malware has a payload, and virus may not have one. For example, worms are an autonous self-replicating malware and so fall into our definition. In fact, the current malware taxonomy (virus, worms, trojans, ...) is unclear and subject to debate.

3.2. Computation over continuous structures

Classical recursion theory deals with computability over discrete structures (natural numbers, finite symbolic words). There is a growing community of researchers working on the extension of this theory to continuous structures arising in mathematics. One goal is to give foundations of numerical analysis, by studying the limitations of machines in terms of computability or complexity, when computing with real numbers. Classical questions are : if a function $f : \mathbb{R} \to \mathbb{R}$ is computable in some sense, are its roots computable? in which time? Another goal is to investigate the possibility of designing new computation paradigms, transcending the usual discrete-time, discrete-space computer model initiated by the Turing machine that is at the base of modern computers.

While the notion of a computable function over discrete data is captured by the model of Turing machines, the situation is more delicate when the data are continuous, and several non-equivalent models exist. In this case, let us mention computable analysis, which relates computability to topology [42], [74]; the Blum-Shub-Smale model (BSS), where the real numbers are treated as elementary entities [34]; the General Purpose Analog Computer (GPAC) introduced by Shannon [68]with continuous time.

3.3. Rewriting

The rewriting paradigm is now widely used for specifying, modelizing, programming and proving. It allows to easily express deduction systems in a declarative way, and to express complex relations on infinite sets of states in a finite way, provided they are countable. Programming languages and environments with a rewriting based semantics have been developed ; see ASF+SDF [36], MAUDE [38], and TOM [64].

For basic rewriting, many techniques have been developed to prove properties of rewrite systems like confluence, completeness, consistency or various notions of termination. Proof methods have also been proposed for extensions of rewriting such as equational extensions, consisting of rewriting modulo a set of axioms, conditional extensions where rules are applied under certain conditions only, typed extensions, where rules are applied only if there is a type correspondence between the rule and the term to be rewritten, and constrained extensions, where rules are enriched by formulas to be satisfied [31], [41], [69].

An interesting aspect of the rewriting paradigm is that it allows automatable or semi-automatable correctness proofs for systems or programs: the properties of rewriting systems as those cited above are translatable to the deduction systems or programs they formalize and the proof techniques may directly apply to them.

Another interesting aspect is that it allows characteristics or properties of the modelled systems to be expressed as equational theorems, often automatically provable using the rewriting mechanism itself or induction techniques based on completion [40]. Note that the rewriting and the completion mechanisms also enable transformation and simplification of formal systems or programs.

Applications of rewriting-based proofs to computer security are various. Approaches using rule-based specifications have recently been proposed for detection of computer viruses [72], [73]. For several years, in our team, we have also been working in this direction. We already have proposed an approach using rewriting techniques to abstract program behaviors for detecting suspicious or malicious programs [32].

CASSIS Project-Team

3. Scientific Foundations

3.1. Introduction

Our main goal is to design techniques and to develop tools for the verification of (safety-critical) systems, such as programs or protocols. To this end, we develop a combination of techniques based on automated deduction for program verification, constraint resolution for test generation, and reachability analysis for the verification of infinite-state systems.

3.2. Automated Deduction

The main goal is to prove the validity of assertions obtained from program analysis. To this end, we develop techniques and automated deduction systems based on rewriting and constraint solving. The verification of recursive data structures relies on inductive reasoning or the manipulation of equations and it also exploits some form of reasoning modulo properties of selected operators (such as associativity and/or commutativity).

Rewriting, which allows us to simplify expressions and formulae, is a key ingredient for the effectiveness of many state-of-the-art automated reasoning systems. Furthermore, a well-founded rewriting relation can be also exploited to implement reasoning by induction. This observation forms the basis of our approach to inductive reasoning, with high degree of automation and the possibility to refute false conjectures.

The constraints are the key ingredient to postpone the activity of solving complex symbolic problems until it is really necessary. They also allow us to increase the expressivity of the specification language and to refine theorem-proving strategies. As an example of this, the handling of constraints for unification problems or for the orientation of equalities in the presence of interpreted operators (e.g., commutativity and/or associativity function symbols) will possibly yield shorter automated proofs.

Finally, decision procedures are being considered as a key ingredient for the successful application of automated reasoning systems to verification problems. A decision procedure is an algorithm capable of efficiently deciding whether formulae from certain theories (such as Presburger arithmetic, lists, arrays, and their combination) are valid or not. We develop techniques to build and to combine decision procedures for the domains which are relevant to verification problems. We also perform experimental evaluation of the proposed techniques by combining propositional reasoning (implemented by means of Boolean solvers, e.g. SAT solvers) and decision procedures to get solvers for the problem of Satisfiability Modulo Theories (SMT).

3.3. Synthesizing and Solving Constraints

Applying constraint logic programming technology in the validation and verification area is currently an active way of research. It usually requires the design of specific solvers to deal with the description language's vocabulary. For instance, we are interested in applying a solver for set constraints [6] to evaluate set-oriented formal specifications. By evaluation, we mean the encoding of the formal model into a constraint system, and the ability for the solver to verify the invariant on the current constraint graph, to propagate preconditions or guards, and to apply a substitution calculus on this graph. The constraint solver is used for animating specifications and automatically generating abstract test cases.

3.4. Rewriting-based Safety Checking

Invariant checking and strenghtening is the dual of reachability analysis, and can thus be used for verifying safety properties of infinite-state systems. In fact, many infinite-state systems are just parameterized systems which become finite state systems when parameters are instantiated. Then, the challenge is to automatically discharge the maximal number of proof obligations coming from the decomposition of the invariance conditions. For parameterized systems, we are interested in a deductive approach where states are defined by first order formulae with equality, and proof obligations are checked by SMT solvers.

CELTIQUE Project-Team

3. Scientific Foundations

3.1. Static program analysis

Static program analysis is concerned with obtaining information about the run-time behaviour of a program without actually running it. This information may concern the values of variables, the relations among them, dependencies between program values, the memory structure being built and manipulated, the flow of control, and, for concurrent programs, synchronisation among processes executing in parallel. Fully automated analyses usually render approximate information about the actual program behaviour. The analysis is correct if the information includes all possible behaviour of a program. Precision of an analysis is improved by reducing the amount of information describing spurious behaviour that will never occur.

Static analysis has traditionally found most of its applications in the area of program optimisation where information about the run-time behaviour can be used to transform a program so that it performs a calculation faster and/or makes better use of the available memory resources. The last decade has witnessed an increasing use of static analysis in software verification for proving invariants about programs. The Celtique project is mainly concerned with this latter use. Examples of static analysis include:

- Data-flow analysis as it is used in optimising compilers for imperative languages. The properties can either be approximations of the values of an expression ("the value of variable x is greater than 0" or x is equal to y at this point in the program") or more intensional information about program behaviour such as "this variable is not used before being re-defined" in the classical "dead-variable" analysis [71].
- Analyses of the memory structure includes shape analysis that aims at approximating the data structures created by a program. Alias analysis is another data flow analysis that finds out which variables in a program addresses the same memory location. Alias analysis is a fundamental analysis for all kinds of programs (imperative, object-oriented) that manipulate state, because alias information is necessary for the precise modelling of assignments.
- Control flow analysis will find a safe approximation to the order in which the instructions of a program are executed. This is particularly relevant in languages where parameters or functions can be passed as arguments to other functions, making it impossible to determine the flow of control from the program syntax alone. The same phenomenon occurs in object-oriented languages where it is the class of an object (rather than the static type of the variable containing the object) that determines which method a given method invocation will call. Control flow analysis is an example of an analysis whose information in itself does not lead to dramatic optimisations (although it might enable in-lining of code) but is necessary for subsequent analyses to give precise results.

Static analysis possesses strong **semantic foundations**, notably abstract interpretation [51], that allow to prove its correctness. The implementation of static analyses is usually based on well-understood constraint-solving techniques and iterative fixpoint algorithms. In spite of the nice mathematical theory of program analysis and the solid algorithmic techniques available one problematic issue persists, *viz.*, the *gap* between the analysis that is proved correct on paper and the analyser that actually runs on the machine. While this gap might be small for toy languages, it becomes important when it comes to real-life languages for which the implementation and maintenance of program analysis tools become a software engineering task. A *certified static analysis* is an analysis that has been formally proved correct using a proof assistant.

In previous work we studied the benefit of using abstract interpretation for developing **certified static analyses** [49], [74]. The development of certified static analysers is an ongoing activity that will be part of the Celtique project. We use the Coq proof assistant which allows for extracting the computational content of a constructive proof. A Caml implementation can hence be extracted from a proof of existence, for any program, of a correct approximation of the concrete program semantics. We have isolated a theoretical framework based on abstract interpretation allowing for the formal development of a broad range of static analyses. Several case studies for the analysis of Java byte code have been presented, notably a memory usage analysis [50]. This work has recently found application in the context of Proof Carrying Code and have also been successfully applied to particular form of static analysis based on term rewriting and tree automata [3].

3.1.1. Static analysis of Java

Precise context-sensitive control-flow analysis is a fundamental prerequisite for precisely analysing Java programs. Bacon and Sweeney's Rapid Type Analysis (RTA) [42] is a scalable algorithm for constructing an initial call-graph of the program. Tip and Palsberg [80] have proposed a variety of more precise but scalable call graph construction algorithms *e.g.,* MTA, FTA, XTA which accuracy is between RTA and 0'CFA. All those analyses are not context-sensitive. As early as 1991, Palsberg and Schwartzbach [72], [73] proposed a theoretical parametric framework for typing object-oriented programs in a context-sensitive way. In their setting, context-sensitivity is obtained by explicit code duplication and typing amounts to analysing the expanded code in a context-insensitive manner. The framework accommodates for both call-contexts and allocation-contexts.

To assess the respective merits of different instantiations, scalable implementations are needed. For Cecil and Java programs, Grove *et al.,* [60], [59] have explored the algorithmic design space of contexts for benchmarks of significant size. Latter on, Milanova *et. al.,* [66] have evaluated, for Java programs, a notion of context called *object-sensitivity* which abstracts the call-context by the abstraction of the `this` pointer. More recently, Lhotak and Hendren [64] have extended the empiric evaluation of object-sensitivity using a BDD implementation allowing to cope with benchmarks otherwise out-of-scope. Besson and Jensen [46] proposed to use DATALOG in order to specify context-sensitive analyses. Whaley and Lam [81] have implemented a context-sensitive analysis using a BDD-based DATALOG implementation.

Control-flow analyses are a prerequisite for other analyses. For instance, the security analyses of Livshits and Lam [65] and the race analysis of Naik, Aiken [67] and Whaley [68] both heavily rely on the precision of a control-flow analysis.

Control-flow analysis allows to statically prove the absence of certain run-time errors such as "message not understood" or cast exceptions. Yet it does not tackle the problem of "null pointers". Fahnrich and Leino [55] propose a type-system for checking that after object creation fields are non-null. Hubert, Jensen and Pichardie have formalised the type-system and derived a type-inference algorithm computing the most precise typing [63]. The proposed technique has been implemented in a tool called NIT [62]. Null pointer detection is also done by bug-detection tools such as FindBugs [62]. The main difference is that the approach of findbugs is neither sound nor complete but effective in practice.

3.1.2. Quantitative aspects of static analysis

Static analyses yield qualitative results, in the sense that they compute a safe over-approximation of the concrete semantics of a program, w.r.t. an order provided by the abstract domain structure. Quantitative aspects of static analysis are two-sided: on one hand, one may want to express and verify (compute) quantitative properties of programs that are not captured by usual semantics, such as time, memory, or energy consumption; on the other hand, there is a deep interest in quantifying the precision of an analysis, in order to tune the balance between complexity of the analysis and accuracy of its result.

The term of quantitative analysis is often related to probabilistic models for abstract computation devices such as timed automata or process algebras. In the field of programming languages which is more specifically addressed by the Celtique project, several approaches have been proposed for quantifying resource usage: a non-exhaustive list includes memory usage analysis based on specific type systems [61], [41], linear

logic approaches to implicit computational complexity [43], cost model for Java byte code [37] based on size relation inference, and WCET computation by abstract interpretation based loop bound interval analysis techniques [52].

We have proposed an original approach for designing static analyses computing program costs: inspired from a probabilistic approach [75], a quantitative operational semantics for expressing the cost of execution of a program has been defined. Semantics is seen as a linear operator over a dioid structure similar to a vector space. The notion of long-run cost is particularly interesting in the context of embedded software, since it provides an approximation of the asymptotic behaviour of a program in terms of computation cost. As for classical static analysis, an abstraction mechanism allows to effectively compute an over-approximation of the semntics, both in terms of costs and of accessible states [48]. An example of cache miss analysis has been developed within this framework [79].

3.1.3. Semantic analysis for test case generation

The semantic analysis of programs can be combined with efficient constraint solving techniques in order to extract specific information about the program, *e.g.*, concerning the accessibility of program points and feasibility of execution paths [76], [54]. As such, it has an important use in the automatic generation of test data. Automatic test data generation received considerable attention these last years with the development of efficient and dedicated constraint solving procedures and compositional techniques [58].

We have made major contributions to the development of **constraint-based testing**, which is a two-stage process consisting of first generating a constraint-based model of the program's data flow and then, from the selection of a testing objective such as a statement to reach or a property to invalidate, to extract a constraint system to be solved. Using efficient constraint solving techniques allows to generate test data that satisfy the testing objective, although this generation might not always terminate. In a certain way, these constraint techniques can be seen as efficient decision procedures and so, they are competitive with the best software model checkers that are employed to generate test data.

3.2. Software certification

The term "software certification" has a number of meanings ranging from the formal proof of program correctness via industrial certification criteria to the certification of software developers themselves! We are interested in two aspects of software certification:

- industrial, mainly process-oriented certification procedures
- software certificates that convey semantic information about a program

Semantic analysis plays a role in both varieties.

Criteria for software certification such as the Common criteria or the DOA aviation industry norms describe procedures to be followed when developing and validating a piece of software. The higher levels of the Common Criteria require a semi-formal model of the software that can be refined into executable code by traceable refinement steps. The validation of the final product is done through testing, respecting criteria of coverage that must be justified with respect to the model. The use of static analysis and proofs has so far been restricted to the top level 7 of the CC and has not been integrated into the aviation norms.

3.2.1. Process-oriented software certification

The testing requirements present in existing certification procedures pose a challenge in terms of the automation of the test data generation process for satisfying functional and structural testing requirements. For example, the standard document which currently governs the development and verification process of software in airborne system (DO-178B) requires the coverage of all the statements, all the decisions of the program at its higher levels of criticality and it is well-known that DO-178B structural coverage is a primary cost driver on avionics project. Although they are widely used, existing marketed testing tools are currently restricted to test

coverage monitoring and measurements [1] but none of these tools tries to find the test data that can execute a given statement, branch or path in the source code. In most industrial projects, the generation of structural test data is still performed manually and finding automatic methods for this problem remains a challenge for the test community. Building automatic test case generation methods requires the development of precise semantic analysis which have to scale up to software that contains thousands of lines of code.

Static analysis tools are so far not a part of the approved certification procedures. For this to change, the analysers themselves must be accepted by the certification bodies in a process called "Qualification of the tools" in which the tools are shown to be as robust as the software it will certify. We believe that proof assistants have a role to play in building such certified static analysis as we have already shown by extracting provably correct analysers for Java byte code.

3.2.2. Semantic software certificates

The particular branch of information security called "language-based security" is concerned with the study of programming language features for ensuring the security of software. Programming languages such as Java offer a variety of language constructs for securing an application. Verifying that these constructs have been used properly to ensure a given security property is a challenge for program analysis. One such problem is confidentiality of the private data manipulated by a program and a large group of researchers have addressed the problem of tracking information flow in a program in order to ensure that *e.g.*, a credit card number does not end up being accessible to all applications running on a computer [78], [45]. Another kind of problems concern the way that computational resources are being accessed and used, in order to ensure that a given access policy is being implemented correctly and that a given application does not consume more resources that it has been allocated. Members of the Celtique team have proposed a verification technique that can check the proper use of resources of Java applications running on mobile telephones [47]. **Semantic software certificates** have been proposed as a means of dealing with the security problems caused by mobile code that is downloaded from foreign sites of varying trustworthiness and which can cause damage to the receiving host, either deliberately or inadvertently. These certificates should contain enough information about the behaviour of the downloaded code to allow the code consumer to decide whether it adheres to a given security policy.

Proof-Carrying Code (PCC) [69] is a technique to download mobile code on a host machine while ensuring that the code adheres to a specified security policy. The key idea is that the code producer sends the code along with a proof (in a suitably chosen logic) that the code is secure. Upon reception of the code and before executing it, the consumer submits the proof to a proof checker for the logic. Our project focus on two components of the PCC architecture: the proof checker and the proof generator.

In the basic PCC architecture, the only components that have to be trusted are the program logic, the proof checker of the logic, and the formalization of the security property in this logic. Neither the mobile code nor the proposed proof—and even less the tool that generated the proof—need be trusted.

In practice, the *proof checker* is a complex tool which relies on a complex Verification Condition Generator (VCG). VCGs for real programming languages and security policies are large and non-trivial programs. For example, the VCG of the Touchstone verifier represents several thousand lines of C code, and the authors observed that "there were errors in that code that escaped the thorough testing of the infrastructure" [70]. Many solutions have been proposed to reduce the size of the trusted computing base. In the *foundational proof carrying code* of Appel and Felty [40], [39], the code producer gives a direct proof that, in some "foundational" higher-order logic, the code respects a given security policy. Wildmoser and Nipkow [83], [82]. prove the soundness of a *weakest precondition* calculus for a reasonable subset of the Java bytecode. Necula and Schneck [70] extend a small trusted core VCG and describe the protocol that the untrusted verifier must follow in interactions with the trusted infrastructure.

One of the most prominent examples of software certificates and proof-carrying code is given by the Java byte code verifier based on *stack maps*. Originally proposed under the term "lightweight Byte Code Verification"

[1] Coverage monitoring answers to the question: what are the statements or branches covered by the test suite ? While coverage measurements answers to: how many statements or branches have been covered ?

by Rose [77], the techniques consists in providing enough typing information (the stack maps) to enable the byte code verifier to check a byte code in one linear scan, as opposed to inferring the type information by an iterative data flow analysis. The Java Specification Request 202 provides a formalization of how such a verification can be carried out.

Inspired by this, Albert *et al.* [38] have proposed to use static analysis (in the form of abstract interpretation) as a general tool in the setting of mobile code security for building a proof-carrying code architecture. In their *abstraction-carrying code* framework, a program comes equipped with a machine-verifiable certificate that proves to the code consumer that the downloaded code is well-behaved.

3.2.3. Certified static analysis

In spite of the nice mathematical theory of program analysis (notably abstract interpretation) and the solid algorithmic techniques available one problematic issue persists, *viz.*, the *gap* between the analysis that is proved correct on paper and the analyser that actually runs on the machine. While this gap might be small for toy languages, it becomes important when it comes to real-life languages for which the implementation and maintenance of program analysis tools become a software engineering task.

A *certified static analysis* is an analysis whose implementation has been formally proved correct using a proof assistant. Such analysis can be developed in a proof assistant like Coq [36] by programming the analyser inside the assistant and formally proving its correctness. The Coq extraction mechanism then allows for extracting a Caml implementation of the analyser. The feasibility of this approach has been demonstrated in [5].

We also develop this technique through certified reachability analysis over term rewriting systems. Term rewriting systems are a very general, simple and convenient formal model for a large variety of computing systems. For instance, it is a very simple way to describe deduction systems, functions, parallel processes or state transition systems where rewriting models respectively deduction, evaluation, progression or transitions. Furthermore rewriting can model every combination of them (for instance two parallel processes running functional programs).

Depending on the computing system modelled using rewriting, reachability (and unreachability) permits to achieve some verifications on the system: respectively prove that a deduction is feasible, prove that a function call evaluates to a particular value, show that a process configuration may occur, or that a state is reachable from the initial state. As a consequence, reachability analysis has several applications in equational proofs used in the theorem provers or in the proof assistants as well as in verification where term rewriting systems can be used to model programs.

For proving unreachability, i.e. safety properties, we already have some results based on the over-approximation of the set of reachable terms [56], [57]. We defined a simple and efficient algorithm [53] for computing exactly the set of reachable terms, when it is regular, and construct an over-approximation otherwise. This algorithm consists of a *completion* of a *tree automaton*, taking advantage of the ability of tree automata to finitely represent infinite sets of reachable terms.

To certify the corresponding analysis, we have defined a checker guaranteeing that a tree automaton is a valid fixpoint of the completion algorithm. This consists in showing that for all term recognised by a tree automaton all his rewrites are also recognised by the same tree automaton. This checker has been formally defined in Coq and an efficient Ocaml implementation has been automatically extracted [3]. This checker is now used to certify all analysis results produced by the regular completion tool as well as the optimised version of [44].

COMETE Project-Team

3. Scientific Foundations

3.1. Probability and information theory

Participants: Miguel Andrés, Nicolás Bordenabe, Konstantinos Chatzikokolakis, Ehab ElSalamouny, Sardaouna Hamadou, Catuscia Palamidessi, Marco Stronati.

Much of the research of Comète focuses on security and privacy. In particular, we are interested in the problem of the leakage of secret information through public observables.

Ideally we would like systems to be completely secure, but in practice this goal is often impossible to achieve. Therefore, we need to reason about the amount of information leaked, and the utility that it can have for the adversary, i.e. the probability that the adversary be able to exploit such information.

The recent tendency is to use information theoretic approach to model the problem and define the leakage in a quantitative way. The idea is to consider that system as an information-theoretic *channel*. The input represents the secret, the output represents the observable, and the correlation between the input and output (*mutual information*) represents the information leakage.

Information theory depends on the notion of entropy. Most of the proposals in the literature use *Shannon entropy*, which is the most established measure of uncertainty. From the security point of view, this measure corresponds to a particular model of attack and a particular way of estimating the security threat (vulnerability of the secret). We consider also other notions, in particular the Rényi min-entropy, which seem to be more appropriate for security in common scenarios like the one-try attacks.

3.2. The probabilistic asynchronous π-calculus

Participants: Konstantinos Chatzikokolakis, Marco Giunti, Catuscia Palamidessi, Frank Valencia, Lili Xu.

We will focus our efforts on a probabilistic variant of the asynchronous π-calculus, which is a formalism designed for mobile and distributed computation. A characteristic of our calculus is the presence of both probabilistic and nondeterministic aspects. This combination is essential to represent probabilistic algorithms and protocols, and express their properties in presence of unpredictable (nondeterministic) users and adversaries.

3.3. Expressiveness issues

Participants: Andrés Aristizábal, Catuscia Palamidessi, Luis Fernando Pino Duque, Frank Valencia.

We intend to study models and languages for concurrent, probabilistic and mobile systems, with a particular attention to expressiveness issues. We aim at developing criteria to assess the expressive power of a model or formalism in a distributed setting, to compare existing models and formalisms, and to define new ones according to an intended level of expressiveness, taking also into account the issue of (efficient) implementability.

3.4. Concurrent constraint programming

Participants: Andrés Aristizábal, Sophia Knight, Luis Fernando Pino Duque, Frank Valencia.

Concurrent constraint programming (ccp) is a well-established process calculus [43] for modeling systems where agents interact by adding and asking information in a global store. This information is represented as first-order logic formulae, called constraints, on the shared variables of the system (e.g., $X > 42$). The most distinctive and appealing feature of ccp is perhaps that it unifies in a single formalism the operational view of processes based upon process calculi with a declarative one based upon first-order logic. It also has an elegant denotational semantics that interprets processes as closure operators (over the set of constraints ordered by entailment). In other words, any ccp process can be seen as an idempotent, increasing, and monotonic function from stores to stores. Consequently, ccp processes can be viewed at the same time as computing agents, formulae in the underlying logic, and closure operators. This allows ccp to benefit from the large body of techniques of process calculi, logic and domain theory.

Our research in ccp develops along the following two lines:

1. The study of a bisimulation semantics for ccp. The advantage of bisimulation, over other kinds of semantics, is that it can be efficiently verified.
2. Enriching ccp with epistemic constructs, which will allow to reason about the knowledge of agents.

3.5. Model checking

Participants: Miguel Andrés, Catuscia Palamidessi.

We plan to develop model-checking techniques and tools for verifying properties of systems and protocols specified in the above formalisms.

Model checking addresses the problem of establishing whether the model (for instance, a finite-state machine) of a certain specification satisfies a certain logical formula.

We intend to concentrate our efforts on aspects that are fundamental for the verification of security protocols, and that are not properly considered in existing tools. Namely, we will focus on:

(a) the combination of probability and mobility, which is not provided by any of the current model checkers,

(b) the interplay between nondeterminism and probability, which in security present subtleties that cannot be handled with the traditional notion of scheduler,

(c) the development of a logic for expressing security (in particular privacy) properties.

Concerning the last point (the logic), we should capture both probabilistic and epistemological aspects, the latter being necessary for treating the knowledge of the adversary.

Logics of this kind have been already developed, but the investigation of the relation with the models coming from process calculi, and their utilization in model checking, is still in its infancy.

CONTRAINTES Project-Team

3. Scientific Foundations

3.1. Rule-based Modeling Languages

Logic programming in a broad sense is a declarative programming paradigm based on mathematical logic with the following identifications:

$$program = logical\ formula,$$

$$execution = proof\ search,$$

In Constraint Satisfaction Problems (CSP), the logical formulae are conjunctions of constraints (i.e. relations on variables expressing partial information) and the satisfiability proofs are computed by constraint solving procedures.

In Constraint Logic Programming (CLP), the logical formulae are Horn clauses with constraints (i.e. one headed rules for the inductive definitions of relations on variables) and the satisfiability proofs combine constraint solving and clause resolution. Gnu-Prolog and its modular extension EMoP that we develop, belong to this family of languages. We use them for solving combinatorial problems and for implementing Biocham.

In Concurrent Constraint Programming (CCP), CLP resolution is extended with a synchronization mechanism based on constraint entailment. The variables play the role of transmissible dynamically created communication channels. An agent may add constraints to the store or read the store to decide whether a constraint guard is entailed by the current store. Sicstus-Prolog and SWI-Prolog belong the this family of languages. We use them for solving combinatorial optimization problems and defining new global constraints.

CCP execution can be identified to deduction in J.Y. Girard's Linear Logic by interpreting multisets of constraints and agents as tensor product conjunctions and guards and rules as linear implications [1]. The logical completeness of CCP in LL continues to hold when considering linear logic constraint systems, i.e. constraint systems where constraints can be consumed by implication. This extension, named Linear Logic Concurrent Constraint Programming (LLCC), allows for a non-monotonic evolution of the store of constraints and can encode multi-headed rules like the Constraint Handling Rules (CHR) language of T. Frühwirth.

All these rule-based languages, of increasing expressivity, involve some form of *multiset rewriting*. We have designed and continue developing the following modeling languages:

- Rules2CP, a rule-based modeling language for solving constraint optimization problems, developed for non-programmers,
- SiLCC, our experimental implementation of LLCC,
- the Biochemical Abstract Machine BIOCHAM, a rule-based modeling language dedicated to Systems Biology, in which biochemical reactions between multisets of reactants and products are expressed with multi-headed rules (somewhat similar to CHR rules) and augmented with *kinetic expressions* from which one can derive quantitative interpretations by Ordinary Differential Equations (ODE), Continuous-Time Markov Chains (CTMC) or Hybrid Automata.

3.2. Constraint Solving Techniques

Constraint propagation algorithms use constraints actively during search for filtering the domains of variables and reducing the search space. These domain reductions are the only way constraints communicate between each other. Our research involves different constraint domains, namely:

- booleans: binary decision diagrams and SAT solvers;

[1] F. Fages, P. Ruet, S. Soliman. *Linear concurrent constraint programming: operational and phase semantics*, in "Information and Control", 2001, vol. 165(1), pp.14-41.

- finite domains (bounded natural numbers): membership, arithmetic, reified [20], higher order and global constraints;
- reals: polyhedral libraries for linear constraints and interval methods;
- terms: subtyping constraints;
- graphs: subgraph epimorphism (SEPI) and isomorphism constraints; acyclicity constraint;
- Petri nets: P/T-invariants [5], siphons and traps [10];
- Kripke structures: temporal logic constraints (first-order Computation Tree Logic constraints over the reals).

We develop new constraints and domain filtering algorithms by using already existing constraint solving algorithms and implementations. For instance, we use the Parma Polyhedra Library PPL with its interface with Prolog for solving temporal logic constraints over the reals. Similarly, we use standard finite domain constraints for developing solvers for the new SEPI graph constraint.

3.3. Formal Methods for Systems Biology

At the end of the 90s, research in Bioinformatics evolved, passing from the analysis of the genomic sequence to the analysis of post-genomic interaction networks (expression of RNA and proteins, protein-protein interactions, transport, etc.). Systems biology is the name given to a pluridisciplinary research field involving biology, computer science, mathematics, physics, to illustrate this change of focus towards system-level understanding of high-level functions of living organisms from their biochemical bases at the molecular level.

Our group was among the first ones in 2002 to apply formal methods from computer science to systems biology in order to reason on large molecular interaction networks and get over complexity walls. The *logical paradigm for systems biology* that we develop can be summarized by the following identifications :

$$\textit{biological model} = \textit{rule-based transition system},$$
$$\textit{biological property} = \textit{temporal logic formula},$$
$$\textit{model validation} = \textit{model-checking},$$
$$\textit{model inference} = \textit{constraint solving}.$$

Rule-based dynamical models of biochemical reaction networks are composed of a reaction graph (bipartite graph with vertices for species and reactions) where the reaction vertices are given with kinetic expressions (mass action law, Michaelis-Menten, Hill, etc.). Most of our work consists in analysing the *interplay between the structure* (reaction graphs) *and the dynamics* (ODE, CTMC or hybrid interpretations derived from the kinetic expressions).

Besides this logical paradigm, we use the theory of abstract interpretation to relate the different interpretations of rule-based models and organize them in a hierarchy of semantics from the most concrete (CTMC stochastic semantics) to the most abstract (asynchronous Boolean transition system). This allows us to prove for instance that if a behavior is not possible in the Boolean semantics of the rules then it is not possible in the stochastic semantics for any kinetic expressions and parameter values. We also use the framework of abstract interpretation to formally relate rule-based reaction models to other knowledge representation formalisms such as, for instance, ontologies of protein functions, or influence graphs between molecular species. These formal methods are used to build models of biological processes, fit models to experimental data, make predictions, and design new biological experiments.

3.4. Tight Integration of In Silico and In Vivo Approaches

Bridging the gap between the complexity of biological systems and our capacity to model and predict systems behaviors is a central challenge in quantitative systems biology. We investigate using wet and dry experiments a few challenging biological questions that necessitate a tight integration between *in vivo* and *in silico* work. Key to the success of this line of research fundamentally guided by specific biological questions is the deployment of innovative modelling and analysis methods for the *in silico* studies.

Synthetic biology, or bioengineering, aims at designing and constructing *in vivo* biological systems that performs novel, useful tasks. This is achieved by reingeneering existing natural biological systems. While the construction of simple intracellular circuits has shown the feasibility of the approach, the design of larger, multicellular systems is a major open issue. In engineered tissues for example, the behavior results from the subtle interplay between intracellular processes (signal transduction, gene expression) and intercellular processes (contact inhibition, gradient of diffusible molecule). How should cells be genetically modified such that the desired behavior robustly emerges from cell interactions? In collaboration with Dirk Drasdo (EPI BANG), we develop *abstraction methods for multiscale systems* to make the design and optimization of such systems computationally tractable and investigate the mammalian tissue homeostasis problem from a bioengineering point of view. Then, in collaboration with the Weiss lab (MIT), we construct and test *in vitro* the proposed designs in actively-growing mammalian cells.

The rational design of synthetic systems relies however on a good quantitative understanding of the functioning of the various processes involved. To acquire that knowledge, one observes the cell reaction to a range of external perturbations. However, current experimental techniques do not allow precise perturbations of cellular processes over a long time period. To make progress on this problem, we develop an experimental platform for the *closed-loop control* of intracellular processes. In collaboration with the MSC lab (CNRS/Paris Diderot U), we develop models of the controlled cellular system, generate quantitative data for parameter identification, and develop real-time control approaches. The integration of all these elements results in an original platform combining hardware (microfluidic device and microscope) and software (cell tracking and model predictive control algorithms). More specifically, by setting up an external, *in silico* feedback loop, we investigate the strengths and time scales of natural feedback loops, responsible for cell adaptation to environmental fluctuations.

DEDUCTEAM Team (section vide)

FORMES Team

3. Scientific Foundations

3.1. Rewriting and Type theory

Coq [42] is one of the most popular proof assistant, in the academia and in the industry. Based on the Extended Calculus of Inductive Constructions, Coq has four kinds of basic entities: objects are used for computations (data, programs, proofs are objects); types express properties of objects; kinds categorize types by their logical structure. Coq's type checker can decide whether a given object satisfies a given type, and if a given type has a logical structure expressed by a given kind. Because it is possible to (uniformly) define inductive types such as lists, dependent types such as lists-of-length-n, parametric types such as lists-of-something, inductive properties such as $(even\ n)$ for some natural number n, etc, writing small specifications in Coq is an easy task. Writing proofs is a harder (non-automatable) task that must be done by the user with the help of tactics. We are interested in two challenges that one has to face with the development of formal proofs in Coq: the theoretical status of equality on the one hand, and the confidence one may have in Coq's proofs on the other hand. Our answer to the first challenge is CoqMTU, which isolates equality in a theory T, which must be first order, such as Presburger Arithmetic. Our answer to the second challenge is the (manual) certification of CoqMTU in Coq.

Rewriting is at the heart of proof systems such as the Extended Calculus of Constructions on which Coq is based, since mathematical proofs are made of reasonning steps, expressed by the typing rules of a given proof system, and computational steps, expressed by its rewrite rules. The certification of a proof system involves, in particular, proving three main properties of its rewrite rules: subject reduction (rewriting should preserve types), confluence (computations should be deterministic), and termination -computations must always terminate. The fact that falsity is not provable in a given proof system such as CoqMTU follows from the previous properties, while decidability of type-checking may require further work. These meta-theoretical proofs are indeed very complex, although at the same time very repetitive, depending on both the typing rules and the rewrite rules. A challenging research question here is to develop certification tools aiming at automating these proofs. Building such tools requires new results allowing to check subject-reduction, confluence and termination of higher-order calculi that are found in proof systems. Since subject-reduction is usually easy to check while consistency and decidability of type-checking follow, *in general*, from the others, confluence and termination are two very active research topics in this area. A last challenge to achieve these goals is the formalization itself of proof systems.

3.2. Verification

Model checking is an automatic formal verification technique [38]. In order to apply the technique, users have to formally specify desired properties on an abstract model of the system under verification. Model checkers will check whether the abstract model satisfies the given properties. If model checkers are able to prove or disprove the properties on the abstract model, they report the result and terminate. In practice, however, abstract models can be extremely complicated, model checkers may not conclude with reasonable computational resources.

Compositional reasoning is a way to ameliorate the complexity in abstract models [75]. Compositional reasoning tries to prove global properties on abstract models by establishing local properties on their components. If local properties on components are easier to verify, compositional reasoning can improve the capacity of model checking by local reasoning. Experiences however suggest that local reasoning may not suffice to establish global properties. It is rare that a global property can be established without considering their interactions. In assume-guarantee reasoning, model checkers try to verify local properties under a contextual assumption of each component. If contextual assumptions faithfully capture interactions among components, model checkers can conclude the verification of global properties.

Finding contextual assumptions however is difficult and may require clairvoyance. Interestingly, a fully automated technique for computing contextual assumptions was proposed in [41]. The automated technique formalizes the contextual assumption generation problem as a learning problem. If properties and abstract models are formalized as finite automata, then a contextual assumption is nothing but an unknown finite automaton that characterizes the environment. Applying a learning algorithm for finite automata, the automated technique will generate contextual assumptions for assume-guarantee reasoning. Experimental results show that the automated technique can outperform a monolithic and explicit verification algorithm.

The success of the learning-based assume-guarantee reasoning is however not satisfactory. Most verification tools are using implicit algorithms. In fact, implicit representations such as Binary Decision Diagrams can improve the capacity of model checking algorithms in several order of magnitudes. Early learning-based techniques, on the other hand, are based on the L^* learning algorithm using explicit representations. If a contextual assumption requires hundreds of states, the learning algorithm will take too much time to infer an assumption. Subsequently, early learning-based techniques cannot compete with monolithic implicit verification [40].

Recently, we propose assume-guarantee reasoning with implicit learning [37]. Our idea is to adopt an implicit representation used in the learning-based framework. Instead of enumerating states of contextual assumptions explicitly, our new technique computes transition relations as an implicit representation of contextual assumptions. Using a learning algorithm for Boolean functions, the new technique can easily compute contextual assumptions with thousands of states. Our preliminary experimental results show that the implicit learning technique can outperform interpolation-based monolithic implicit model checking in several parametrized test cases such as synchronous bus arbiters and the MSI cache coherence protocol.

Learning Boolean functions can also be applied to loop invariant inference [53], [54]. Suppose that a programmer annotates a loop with pre- and post-conditions. We would like to compute a loop invariant to verify that the annotated loop conforms to its specification. Finding loop invariants manually is very tedious. One makes a first guess and then iteratively refines the guess by examining the loop body. This process is in fact very similar to learning an unknown formula. Applying predicate abstraction and decision procedures, a learning algorithm for Boolean functions can infer loop invariants generated by a given set of atomic predicates. Preliminary experimental results show that the learning-based technique is effective for annotated loops extracted from source codes of Linux and SPEC2000 benchmarks.

Although implicit learning techniques have been developed for assume-guarantee reasoning and loop invariant inference successfully, challenges still remain. Currently, the learning algorithm is able to infer Boolean functions over tens of Boolean variables. Contextual assumptions over tens of Boolean variables are not enough. Ideally, one would like to have contextual assumptions over hundreds (even thousands) of Boolean variables. On the other hand, it is known that learning arbitrary Boolean functions is infeasible. The scalability of implicit learning techniques cannot be improved satisfactorily by tuning the learning algorithm alone. Combining implicit learning with abstraction will be essential to improve its scalability.

Our second challenge is to extend learning-based techniques to other computation models. In addition to finite automata, probabilistic automata and timed automata are also widely used to specify abstract models. Their verification problems are much more difficult than those for finite automata. Compositional reasoning thus can improve the capacity of model checkers more significantly. Recently, the L^* algorithm is applied in assume-guarantee reasoning for probabilistic automata [46]. The new technique is unfortunately incomplete. Developing a complete learning-based assume-guarantee reasoning technique for probabilistic automata and timed automata will be very useful to their verification.

Through predicate abstraction, learning Boolean functions can be very useful in program analysis. We have successfully applied algorithmic learning to infer both quantified and quantifier-free loop invariants for annotated loops. Applying algorithmic learning to static analysis or program testing will be our last challenge. In the context of program analysis, scalability of the learning algorithm is less of an issue. Formulas over tens of atomic predicates usually suffice to characterize relation among program variables. On the other hand, learning algorithms require oracles to answer queries or generate samples. Designing such oracles necessarily

requires information extracted from program texts. How to extract information will be essential to applying algorithmic learning in static analysis or program testing.

3.3. Decision Procedures

Decision procedures are of utmost importance for us, since they are at the heart of theorem proving and verification. Research in decision procedures started several decades ago, and are now commonly used both in the academia and industry. A decision procedure [55] is an algorithm which returns a correct yes/no answer to a given input decision problem. Many real-world problems can be reduced to the decision problems, making this technique very practical. For example, Intel and AMD are developing solvers for their circuit verification tools, while Microsoft is developing decision procedures for their code analysis tools.

Mathematical logic is the appropriate tool to formulate a decision problem. Most decision problems are formulated as a decidable fragment of a first-order logic interpreted in some specific domain. On such, easy and popular fragment, is propositional (or Boolean) logic, which corresponding decision procedure is called SAT. Representing real problems in SAT often results in awkward encodings that destroy the logical structure of the original problem.

A very popular, effective recent trend is Satisfiability Modulo Theories (SMT) [74], a general technique to solve decision problems formulated as propositional formulas operating on atoms in a given background theory, for example linear real arithmetic. Existing approaches for solving SMT problems can be classified into two categories: *lazy* method [67], and *eager* method [68]. The eager method encodes an SMT problem into an equi-satisfiable SAT problem, while the lazy method employs different theory solvers for each theory and coordinates them appropriately. The eager method does allow the user to express her problem in a natural way, but does not exploit its logical structure to speed up the computation. The lazy approach is more appealing, and has prompted much interest in algorithms for the various background theories important in practice.

Our SMT solver aCiNO is based on the lazy approach. So far, it provides with two (popular) theories only: linear real arithmetic (LRA) and uninterpreted functions (UF). For efficiency consideration, the solver is implemented in an incremental way. It also invokes an online SAT solver, which is now a modified DPLL procedure, so that recovery from conflicts is possible. Our challenge here is twofold: first, to add other theories of interest for the project, we are currently working on fragments of the theory of arrays [61], [34]. The theory of arrays is important because of its use for expressing loop invariants in programs with arrays, but its full first-order theory is undecidable. We are also interested in the theory of bit vectors, very much used for hardware verification.

Theory solvers implement state-of-the-art algorithms which sophistication makes their correct implementation a delicate task. Moreover, SMT solvers themselves employ a quite complex machinery, making them error prone as well [4] We therefore strongly believe that decision procedures, and SMT provers, should come along with a formal assessment of their correctness. As usual, there are two ways: ensure the correctness of an arbitrary output by proving the code, or deliver for each input a certificate ensuring the correctness of the corresponding output when the checker says so. Developing concise certificates together with efficient certificate checkers for the various decision procedures of interest and their combination with SMT is yet another challenge which is at the heart of the project FORMES.

3.4. Simulation

The development of complex embedded systems platforms requires putting together many hardware components, processor cores, application specific co-processors, bus architectures, peripherals, etc. The hardware platform of a project is seldom entirely new. In fact, in most cases, 80 percent of the hardware components are re-used from previous projects or simply are COTS (Commercial Off-The-Shelf) components. There is no need to simulate in great detail these already proven components, whereas there is a need to run fast simulation of the software using these components.

[4] It took almost 20 years to have a correct implementation of a correct version of Shostak's algorithm for combining decision procedures, which can be seen as an ancestor of SMT.

These requirements call for an integrated, modular simulation environment where already proven components can be simulated quickly, (possibly including real hardware in the loop), new components under design can be tested more thoroughly, and the software can be tested on the complete platform with reasonable speed.

Modularity and fast prototyping also have become important aspects of simulation frameworks, for investigating alternative designs with easier re-use and integration of third party components.

The project aims at developing such a rapid prototyping, modular simulation platform, combining new hardware components modeling, verification techniques, fast software simulation for proven components, capable of running the real embedded software application without any change.

To fully simulate a complete hardware platform, one must simulate the processors, the co-processors, together with the peripherals such as network controllers, graphics controllers, USB controllers, etc. A commonly used solution is the combination of some ISS (Instruction Set Simulator) connected to a Hardware Description Language (HDL) simulator which can be implemented by software or by using a FPGA [60] simulator. These solutions tend to present slow iteration design cycles and implementing the FPGA means the hardware has already been designed at low level, which comes normally late in the project and become very costly when using large FPGA platforms. Others have implemented a co-simulation environment, using two separate technologies, typically one using a HDL and another one using an ISS [47], [50], [66]. Some communication and synchronization must be designed and maintained between the two using some inter-process communication (IPC), which slows down the process.

The idea we pursue is to combine hardware modeling and fast simulation into a fully integrated, software based (not using FPGA) simulation environment named SimSoC, which uses a single simulation loop thanks to Transaction Level Modeling (TLM) [36], [23] combined with a new ISS technology designed specifically to fit within the TLM environment.

The most challenging way to enhance simulation speed is to simulate the processors. Processor simulation is achieved with Instruction Set Simulation (ISS). There are several alternatives to achieve such simulation. In *interpretive simulation*, each instruction of the target program is fetched from memory, decoded, and executed. This method is flexible and easy to implement, but the simulation speed is slow as it wastes a lot of time in decoding. Interpretive simulation is used in Simplescalar [35]. Another technique to implement a fast ISS is *dynamic translation* [39], [65], [44] which has been favored by many [63], [44], [64], [65] in the past decade.

With dynamic translation, the binary target instructions are fetched from memory at run-time, like in interpretive simulation. They are decoded on the first execution and the simulator translates these instructions into another representation which is stored into a cache. On further execution of the same instructions, the translated cached version is used. Dynamic translation introduces a translation time phase as part of the overall simulation time. But as the resulting cached code is re-used, the translation time is amortized over time. If the code is modified during run-time, the simulator must invalidate the cached representation. Dynamic translation provides much faster simulation while keeping the advantage of interpretive simulation as it supports the simulation of programs that have either dynamic loading or self-modifying code.

There are many ways of translating binary code into cached data, which each come at a price, with different trade-offs between the translation time and the obtained speed up on cache execution. Also, simulation speed-ups usually don't come for free : most of time there is a trade-off between accuracy and speed.

There are two well known variants of the dynamic translation technology: the target code is translated either directly into machine code for the simulation host, or into an intermediate representation, independent from the host machine, that makes it possible to execute the code with faster speed. Both have pros and cons.

Processor simulation is also achieved in Virtual Machines such as QEMU [28] and GXEMUL [49] that emulate to a large extent the behavior of a particular hardware platform. The technique used in QEMU is a form of dynamic translation. The target code is translated directly into machine code using some pre-determined code patterns that have been pre-compiled with the C compiler. Both QEMU and GXEMUL include many device models of open-source C code, but this code is hard to reuse. The functions that emulate device accesses do not have the same profile. The scheduling process of the parallel hardware entities is not specified well enough to

guarantee the compatibility between several emulators or re-usability of third-party models using the standards from the electronics industry (e.g. IEEE 1666).

A challenge in the development of high performance simulators is to maintain simultaneously fast speed and simulation accuracy. In the FORMES project, we expect to develop a dynamic translation technology satisfying the following additional objectives:

- provide different levels of translation with different degrees of accuracy so that users can choose between accurate and slow (for debugging) or less accurate but fast simulation.
- to take advantage of multi-processor simulation hosts to parallelize the simulation;
- to define intermediate representations of programs that optimize the simulation speed and possibly provide a more convenient format for studying properties of the simulated programs.

Another objective of the FORMES simulation is to extract information from the simulated applications to prove properties. Running a simulation is exercising a test case. In most cases, if a test is failing, a bug has been found. One can use model checking tools to generate tests that can be run on the simulator to check whether the test fails or not on the real application. It is also a goal of FORMES simulation activity to use such formal methods tools to detect bugs, either by generating tests, or by using formal methods tools to analyze the results of simulation sessions.

3.5. Trustworthy Software

Since the early days of software development, computer scientists have been interested in designing methods for improving software quality. Formal methods based on model checking, correctness proofs, common criteria certification, all address this issue in their own way. None of these methods, however, considers the trustworthiness of a given software system as a system-level property, requiring to grasp a given software within its environment of execution.

The major challenge we want to address here is to provide a framework in which to formalize the notion of trustworthiness, to evaluate the trustworthiness of a given software, and if necessary improve it.

To make trustworthiness a fruitful concept, our vision is to formalize it via a hierarchy of observability and controllability degrees: the more the software is observable and controllable, the more its behaviors can be trusted by users. On the other hand, users from different application domains have different expectations from the software they use. For example, aerospace embedded software should be safety-critical while e-commerce software should be insensitive to attacks. As a result, trustworthiness should be domain-specific.

A main challenge is the evaluation of trustworthiness. We believe that users should be responsible for describing the level of trustworthiness they need, in the form of formal requirements that the software should satisfy. A major issue is to come up with some predefined levels of trustworthiness for the major applicative areas. Another is to use stepwise refinement techniques to achieve the appropriate level of trustworthiness. These levels would then drive the design and implementation of a software system: the objective would be to design a model with enough details (observability) to make it possible to check all requirements of that level.

The other challenge is the effective integration of results obtained from different verification methods. There are many verification techniques, like simulation, testing, model checking and theorem proving. These methods may operate on different models of the software to be then executed, while trustworthiness should measure our trust in the real software running in its real execution environment. There are also monitoring and analysis techniques to capture the characteristics of actual executions of the system. Integrating all the analysis in order to decide the trustworthiness level of a software is quite a hard task.

GALLIUM Project-Team

3. Scientific Foundations

3.1. Programming languages: design, formalization, implementation

Like all languages, programming languages are the media by which thoughts (software designs) are communicated (development), acted upon (program execution), and reasoned upon (validation). The choice of adequate programming languages has a tremendous impact on software quality. By "adequate", we mean in particular the following four aspects of programming languages:

- **Safety.** The programming language must not expose error-prone low-level operations (explicit memory deallocation, unchecked array accesses, etc) to the programmers. Further, it should provide constructs for describing data structures, inserting assertions, and expressing invariants within programs. The consistency of these declarations and assertions should be verified through compile-time verification (e.g. static type checking) and run-time checks.
- **Expressiveness.** A programming language should manipulate as directly as possible the concepts and entities of the application domain. In particular, complex, manual encodings of domain notions into programmatic notations should be avoided as much as possible. A typical example of a language feature that increases expressiveness is pattern matching for examination of structured data (as in symbolic programming) and of semi-structured data (as in XML processing). Carried to the extreme, the search for expressiveness leads to domain-specific languages, customized for a specific application area.
- **Modularity and compositionality.** The complexity of large software systems makes it impossible to design and develop them as one, monolithic program. Software decomposition (into semi-independent components) and software composition (of existing or independently-developed components) are therefore crucial. Again, this modular approach can be applied to any programming language, given sufficient fortitude by the programmers, but is much facilitated by adequate linguistic support. In particular, reflecting notions of modularity and software components in the programming language enables compile-time checking of correctness conditions such as type correctness at component boundaries.
- **Formal semantics.** A programming language should fully and formally specify the behaviours of programs using mathematical semantics, as opposed to informal, natural-language specifications. Such a formal semantics is required in order to apply formal methods (program proof, model checking) to programs.

Our research work in language design and implementation centers around the statically-typed functional programming paradigm, which scores high on safety, expressiveness and formal semantics, complemented with full imperative features and objects for additional expressiveness, and modules and classes for compositionality. The OCaml language and system embodies many of our earlier results in this area [36]. Through collaborations, we also gained experience with several domain-specific languages based on a functional core, including XML processing (XDuce, CDuce), reactive functional programming, distributed programming (JoCaml), and hardware modeling (ReFLect).

3.2. Type systems

Type systems [49] are a very effective way to improve programming language reliability. By grouping the data manipulated by the program into classes called types, and ensuring that operations are never applied to types over which they are not defined (e.g. accessing an integer as if it were an array, or calling a string as if it were a function), a tremendous number of programming errors can be detected and avoided, ranging from the trivial (misspelled identifier) to the fairly subtle (violation of data structure invariants). These restrictions are also very effective at thwarting basic attacks on security vulnerabilities such as buffer overflows.

The enforcement of such typing restrictions is called type checking, and can be performed either dynamically (through run-time type tests) or statically (at compile-time, through static program analysis). We favor static type checking, as it catches bugs earlier and even in rarely-executed parts of the program, but note that not all type constraints can be checked statically if static type checking is to remain decidable (i.e. not degenerate into full program proof). Therefore, all typed languages combine static and dynamic type-checking in various proportions.

Static type checking amounts to an automatic proof of partial correctness of the programs that pass the compiler. The two key words here are *partial*, since only type safety guarantees are established, not full correctness; and *automatic*, since the proof is performed entirely by machine, without manual assistance from the programmer (beyond a few, easy type declarations in the source). Static type checking can therefore be viewed as the poor man's formal methods: the guarantees it gives are much weaker than full formal verification, but it is much more acceptable to the general population of programmers.

3.2.1. Type systems and language design.

Unlike most other uses of static program analysis, static type-checking rejects programs that it cannot analyze safe. Consequently, the type system is an integral part of the language design, as it determines which programs are acceptable and which are not. Modern typed languages go one step further: most of the language design is determined by the *type structure* (type algebra and typing rules) of the language and intended application area. This is apparent, for instance, in the XDuce and CDuce domain-specific languages for XML transformations [46], [42], whose design is driven by the idea of regular expression types that enforce DTDs at compile-time. For this reason, research on type systems – their design, their proof of semantic correctness (type safety), the development and proof of associated type checking and inference algorithms – plays a large and central role in the field of programming language research, as evidenced by the huge number of type systems papers in conferences such as Principles of Programming Languages.

3.2.2. Polymorphism in type systems.

There exists a fundamental tension in the field of type systems that drives much of the research in this area. On the one hand, the desire to catch as many programming errors as possible leads to type systems that reject more programs, by enforcing fine distinctions between related data structures (say, sorted arrays and general arrays). The downside is that code reuse becomes harder: conceptually identical operations must be implemented several times (say, copying a general array and a sorted array). On the other hand, the desire to support code reuse and to increase expressiveness leads to type systems that accept more programs, by assigning a common type to broadly similar objects (for instance, the `Object` type of all class instances in Java). The downside is a loss of precision in static typing, requiring more dynamic type checks (downcasts in Java) and catching fewer bugs at compile-time.

Polymorphic type systems offer a way out of this dilemma by combining precise, descriptive types (to catch more errors statically) with the ability to abstract over their differences in pieces of reusable, generic code that is concerned only with their commonalities. The paradigmatic example is parametric polymorphism, which is at the heart of all typed functional programming languages. Many forms of polymorphic typing have been studied since then. Taking examples from our group, the work of Rémy, Vouillon and Garrigue on row polymorphism [53], integrated in OCaml, extended the benefits of this approach (reusable code with no loss of typing precision) to object-oriented programming, extensible records and extensible variants. Another example is the work by Pottier on subtype polymorphism, using a constraint-based formulation of the type system [50].

3.2.3. Type inference.

Another crucial issue in type systems research is the issue of type inference: how many type annotations must be provided by the programmer, and how many can be inferred (reconstructed) automatically by the typechecker? Too many annotations make the language more verbose and bother the programmer with unnecessary details. Too few annotations make type checking undecidable, possibly requiring heuristics, which is unsatisfactory. OCaml requires explicit type information at data type declarations and at component interfaces, but infers all other types.

In order to be predictable, a type inference algorithm must be complete. That is, it must not find *one*, but *all* ways of filling in the missing type annotations to form an explicitly typed program. This task is made easier when all possible solutions to a type inference problem are *instances* of a single, *principal* solution.

Maybe surprisingly, the strong requirements – such as the existence of principal types – that are imposed on type systems by the desire to perform type inference sometimes lead to better designs. An illustration of this is row variables. The development of row variables was prompted by type inference for operations on records. Indeed, previous approaches were based on subtyping and did not easily support type inference. Row variables have proved simpler than structural subtyping and more adequate for typechecking record update, record extension, and objects.

Type inference encourages abstraction and code reuse. A programmer's understanding of his own program is often initially limited to a particular context, where types are more specific than strictly required. Type inference can reveal the additional generality, which allows making the code more abstract and thus more reuseable.

3.3. Compilation

Compilation is the automatic translation of high-level programming languages, understandable by humans, to lower-level languages, often executable directly by hardware. It is an essential step in the efficient execution, and therefore in the adoption, of high-level languages. Compilation is at the interface between programming languages and computer architecture, and because of this position has had considerable influence on the designs of both. Compilers have also attracted considerable research interest as the oldest instance of symbolic processing on computers.

Compilation has been the topic of much research work in the last 40 years, focusing mostly on high-performance execution ("optimization") of low-level languages such as Fortran and C. Two major results came out of these efforts: one is a superb body of performance optimization algorithms, techniques and methodologies; the other is the whole field of static program analysis, which now serves not only to increase performance but also to increase reliability, through automatic detection of bugs and establishment of safety properties. The work on compilation carried out in the Gallium group focuses on a less investigated topic: compiler certification.

3.3.1. Formal verification of compiler correctness.

While the algorithmic aspects of compilation (termination and complexity) have been well studied, its semantic correctness – the fact that the compiler preserves the meaning of programs – is generally taken for granted. In other terms, the correctness of compilers is generally established only through testing. This is adequate for compiling low-assurance software, themselves validated only by testing: what is tested is the executable code produced by the compiler, therefore compiler bugs are detected along with application bugs. This is not adequate for high-assurance, critical software which must be validated using formal methods: what is formally verified is the source code of the application; bugs in the compiler used to turn the source into the final executable can invalidate the guarantees so painfully obtained by formal verification of the source.

To establish strong guarantees that the compiler can be trusted not to change the behavior of the program, it is necessary to apply formal methods to the compiler itself. Several approaches in this direction have been investigated, including translation validation, proof-carrying code, and type-preserving compilation. The approach that we currently investigate, called *compiler verification*, applies program proof techniques to the compiler itself, seen as a program in particular, and use a theorem prover (the Coq system) to prove that the generated code is observationally equivalent to the source code. Besides its potential impact on the critical software industry, this line of work is also scientifically fertile: it improves our semantic understanding of compiler intermediate languages, static analyses and code transformations.

3.4. Interface with formal methods

Formal methods refer collectively to the mathematical specification of software or hardware systems and to the verification of these systems against these specifications using computer assistance: model checkers, theorem provers, program analyzers, etc. Despite their costs, formal methods are gaining acceptance in the critical software industry, as they are the only way to reach the required levels of software assurance.

In contrast with several other Inria projects, our research objectives are not fully centered around formal methods. However, our research intersects formal methods in the following two areas, mostly related to program proofs using proof assistants and theorem provers.

3.4.1. Software-proof codesign

The current industrial practice is to write programs first, then formally verify them later, often at huge costs. In contrast, we advocate a codesign approach where the program and its proof of correctness are developed in interaction, and are interested in developing ways and means to facilitate this approach. One possibility that we currently investigate is to extend functional programming languages such as Caml with the ability to state logical invariants over data structures and pre- and post-conditions over functions, and interface with automatic or interactive provers to verify that these specifications are satisfied. Another approach that we practice is to start with a proof assistant such as Coq and improve its capabilities for programming directly within Coq. Finally, we also participate in the FoCaLiZe project, which designs and implements an environment for combined programming and proving [23] [52].

3.4.2. Mechanized specifications and proofs for programming languages components

We emphasize mathematical specifications and proofs of correctness for key language components such as semantics, type systems, type inference algorithms, compilers and static analyzers. These components are getting so large that machine assistance becomes necessary to conduct these mathematical investigations. We have already mentioned using proof assistants to verify compiler correctness. We are also interested in using them to specify and reason about semantics and type systems. These efforts are part of a more general research topic that is gaining importance: the formal verification of the tools that participate in the construction and certification of high-assurance software.

MARELLE Project-Team

3. Scientific Foundations

3.1. Type theory and formalization of mathematics

The calculus of inductive constructions is a branch of type theory that serves as a foundation for theorem proving tools, especially the Coq proof assistant. It is powerful enough to formalize complex mathematics, based on algebraic structures and operations. This is especially important as we want to produce proofs of logical properties for these algebraic structures, a goal that is only marginally addressed in most scientific computation systems.

The calculus of inductive constructions also makes it possible to write algorithms as recursive functional programs which manipulate tree-like data structures. A third important characteristic of this calculus is that it is also a language for manipulating proofs. All this makes this calculus a tool of choice for our investigations. However, this language is still being improved and part of our work concerns these improvements.

3.2. Verification of scientific algorithms

To produce certified algorithms, we use the following approach: instead of attempting to prove properties of an existing program written in a conventional programming language such as C or Java, we produce new programs in the calculus of constructions whose correctness is an immediate consequence of their construction. This has several advantages. First, we work at a high level of abstraction, independently of the target implementation language. Second, we concentrate on specific characteristics of the algorithm, and abstract away from the rest (for instance, we abstract away from memory management or data implementation strategies). Thus, we are able to address more high-level mathematics and to express more general properties without being overwhelmed by implementation details.

However, this approach also presents a few drawbacks. For instance, the calculus of constructions usually imposes that recursive programs should explicitly terminate for all inputs. For some algorithms, we need to use advanced concepts (for instance, well-founded relations) to make the property of termination explicit, and proofs of correctness become especially difficult in this setting.

3.3. Programming language semantics

To bridge the gap between our high-level descriptions of algorithms and conventional programming languages, we investigate the algorithms that are present in programming language implementations, for instance algorithms that are used in a compiler or a static analysis tool. For these algorithms, we generally base our work on the semantic description of a language. The properties that we attempt to prove for an algorithm are, for example, that an optimization respects the meaning of programs or that the programs produced are free of some unwanted behavior. In practice, we rely on this study of programming language semantics to propose extensions to theorem proving tools or to participate in the verification that compilers for conventional programming languages are exempt from bugs.

MEXICO Project-Team

3. Scientific Foundations

3.1. Concurrency

Participants: Benedikt Bollig, Thomas Chatain, Aiswarya Cyriac, Paul Gastin, Stefan Haar, Serge Haddad, Hernán Ponce de Léon, Stefan Schwoon.

Concurrency: Property of systems allowing some interacting processes to be executed in parallel.

Diagnosis: The process of deducing from a partial observation of a system aspects of the internal states or events of that system; in particular, *fault diagnosis* aims at determining whether or not some non-observable fault event has occurred.

Conformance Testing: Feeding dedicated input into an implemented system IS and deducing, from the resulting output of I, whether I respects a formal specification S.

3.1.1. Introduction

It is well known that, whatever the intended form of analysis or control, a *global* view of the system state leads to overwhelming numbers of states and transitions, thus slowing down algorithms that need to explore the state space. Worse yet, it often blurs the mechanics that are at work rather than exhibiting them. Conversely, respecting concurrency relations avoids exhaustive enumeration of interleavings. It allows us to focus on 'essential' properties of non-sequential processes, which are expressible with causal precedence relations. These precedence relations are usually called causal (partial) orders. Concurrency is the explicit absence of such a precedence between actions that do not have to wait for one another. Both causal orders and concurrency are in fact essential elements of a specification. This is especially true when the specification is constructed in a distributed and modular way. Making these ordering relations explicit requires to leave the framework of state/interleaving based semantics. Therefore, we need to develop new dedicated algorithms for tasks such as conformance testing, fault diagnosis, or control for distributed discrete systems. Existing solutions for these problems often rely on centralized sequential models which do not scale up well.

3.1.2. Diagnosis

Participants: Benedikt Bollig, Stefan Haar, César Rodríguez, Stefan Schwoon.

Fault Diagnosis for discrete event systems is a crucial task in automatic control. Our focus is on *event oriented* (as opposed to *state oriented*) model-based diagnosis, asking e.g. the following questions:
given a - potentially large - *alarm pattern* formed of observations,

- what are the possible *fault scenarios* in the system that *explain* the pattern ?
- Based on the observations, can we deduce whether or not a certain - invisible - fault has actually occurred ?

Model-based diagnosis starts from a discrete event model of the observed system - or rather, its relevant aspects, such as possible fault propagations, abstracting away other dimensions. From this model, an extraction or unfolding process, guided by the observation, produces recursively the explanation candidates.

In asynchronous partial-order based diagnosis with Petri nets [98], [99], [103], one unfolds the *labelled product* of a Petri net model $\mathcal{N}$ and an observed alarm pattern $\mathcal{A}$, also in Petri net form. We obtain an acyclic net giving partial order representation of the behaviors compatible with the alarm pattern. A recursive online procedure filters out those runs *(configurations)* that explain *exactly*$\mathcal{A}$. The Petri-net based approach generalizes to dynamically evolving topologies, in dynamical systems modeled by graph grammars, see [83].

3.1.2.1. Observability and Diagnosability

Diagnosis algorithms have to operate in contexts with low observability, i.e., in systems where many events are invisible to the supervisor. Checking *observability* and *diagnosability* for the supervised systems is therefore a crucial and non-trivial task in its own right. Analysis of the relational structure of occurrence nets allows us to check whether the system exhibits sufficient visibility to allow diagnosis. Developing efficient methods for both verification of *diagnosability checking* under concurrency, and the *diagnosis* itself for distributed, composite and asynchronous systems, is an important field for *MExICo*.

3.1.2.2. Distribution

Distributed computation of unfoldings allows one to factor the unfolding of the global system into smaller *local* unfoldings, by local supervisors associated with sub-networks and communicating among each other. In [99], [84], elements of a methodology for distributed computation of unfoldings between several supervisors, underwritten by algebraic properties of the category of Petri nets have been developed. Generalizations, in particular to Graph Grammars, are still do be done.

Computing diagnosis in a distributed way is only one aspect of a much vaster topic, that of *distributed diagnosis* (see [96], [110]). In fact, it involves a more abstract and often indirect reasoning to conclude whether or not some given invisible fault has occurred. Combination of local scenarios is in general not sufficient: the global system may have behaviors that do not reveal themselves as faulty (or, dually, non-faulty) on any local supervisor's domain (compare [82], [87]). Rather, the local diagnosers have to join all *information* that is available to them locally, and then deduce collectively further information from the combination of their views. In particular, even the *absence* of fault evidence on all peers may allow to deduce fault occurrence jointly, see [116], [117]. Automatizing such procedures for the supervision and management of distributed and locally monitored asynchronous systems is a mid-term goal of *MExICo*.

3.1.3. Verification of Concurrent Recursive Programs

Participants: Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, César Rodríguez, Stefan Schwoon.

3.1.3.1. Contextual nets

Assuring the correctness of concurrent systems is notoriously difficult due to the many unforeseeable ways in which the components may interact and the resulting state-space explosion. A well-established approach to alleviate this problem is to model concurrent systems as Petri nets and analyse their unfoldings, essentially an acyclic version of the Petri net whose simpler structure permits easier analysis [97].

However, Petri nets are inadequate to model concurrent read accesses to the same resource. Such situations arise naturally in many circumstances, for instance in concurrent databases or in asynchronous circuits. The encoding tricks typically used to model these cases in Petri nets make the unfolding technique inefficient.

Contextual nets, which explicitly do model concurrent read accesses, address this problem. Their accurate representation of concurrency makes contextual unfoldings up to exponentially smaller in certain situations, which promises to yield more efficient analysis procedures. In order to realize such procedures, we shall study contextual nets and their properties, in particular the efficient construction and analysis of their unfoldings, and their applications in verification, diagnosis, and planning.

3.1.3.2. Concurrent Recursive Programs

In a DIGITEO PhD project, we will study logical specification formalisms for concurrent recursive programs. With the advent of multi-core processors, the analysis and synthesis of such programs is becoming more and more important. However, it cannot be achieved without more comprehensive formal mathematical models of concurrency and parallelization. Most existing approaches have in common that they restrict to the analysis of an over- or underapproximation of the actual program executions and do not focus on a behavioral semantics. In particular, temporal logics have not been considered. Their design and study will require the combination of prior works on logics for sequential recursive programs and concurrent finite-state programs.

3.1.4. Testing

Participants: Benedikt Bollig, Paul Gastin, Stefan Haar, Hernán Ponce de Léon.

3.1.4.1. Introduction

The gap between specification and implementation is at the heart of research on formal testing. The general *conformance testing problem* can be defined as follows: Does an implementation $\mathcal{M}'$ conform a given specification $\mathcal{M}$? Here, both $\mathcal{M}$ and $\mathcal{M}'$ are assumed to have input and output channels. The formal model $\mathcal{M}$ of the specification is entirely known and can be used for analysis. On the other hand, the implementation $\mathcal{M}'$ is unknown but interacts with the environment through observable input and output channels. So the behavior of $\mathcal{M}'$ is partially controlled by input streams, and partially observable via output streams. The Testing problem consists in computing, from the knowledge of $\mathcal{M}$, *input streams* for $\mathcal{M}'$ such that observation of the resulting output streams from $\mathcal{M}'$ allows to determine whether $\mathcal{M}'$ conforms to $\mathcal{M}$ as intended.

In this project, we focus on distributed or asynchronous versions of the conformance testing problem. There are two main difficulties. First, due to the distributed nature of the system, it may not be possible to have a unique global observer for the outcome of a test. Hence, we may need to use *local* observers which will record only *partial views* of the execution. Due to this, it is difficult or even impossible to reconstruct a coherent global execution. The second difficulty is the lack of global synchronization in distributed asynchronous systems. Up to now, models were described with I/O automata having a centralized control, hence inducing global synchronizations.

3.1.4.2. Asynchronous Testing

Since 2006 and in particular during his sabbatical stay at the University of Ottawa, Stefan Haar has been working with Guy-Vincent Jourdan and Gregor v. Bochmann of UOttawa and Claude Jard of IRISA on asynchronous testing. In the synchronous (sequential) approach, the model is described by an I/O automaton with a centralized control and transitions labeled with individual input or output actions. This approach has known limitations when inputs and outputs are distributed over remote sites, a feature that is characteristic of , e.g., web computing. To account for concurrency in the system, they have developed in [105], [88] asynchronous conformance testing for automata with transitions labeled with (finite) partial orders of I/O. Intuitively, this is a "big step" semantics where each step allows concurrency but the system is synchronized before the next big step. This is already an important improvement on the synchronous setting. The non-trivial challenge is now to cope with fully asynchronous specifications using models with decentralized control such as Petri nets.

3.1.4.3. Local Testing

Message-Sequence-Charts (MSCs) provide models of behaviors of distributed systems with communicating processes. An important problem is to test whether an implementation conforms to a specification given for instance by an HMSC. In *local testing*, one proceeds by injecting messages to the local processes and observing the responses: for each process p, a local observer records the sequence of events at p. If each local observation is consistent with some MSC defined by the specification, the implementation passes the test. If local testing on individual processes suffices to check conformance, the given specification (an HMSC language) is called locally testable. Local testability turns out to be undecidable even for regular HMSC languages [82]; the main difficulty lies in the existence of implied scenarios, i.e., global behaviors which are locally consistent with different specification scenarios. There are two approaches to attack the problem of local testing in light of this bottleneck. One is to allow joint observations of tuples of processes. This gives rise to the problem of k-testability where one allows joint observations of up to k processes [87]. We will look for structural conditions on the model or the specification ensuring k-testability. Another tactic would be to recognize that practical implementations always work with bounded buffers and impose an upper bound B on the buffer size. The set of B-bounded MSCs in the k-closure of a regular MSC language is again regular, so the B-bounded k-testability problem is decidable for all regular HMSC-definable specifications. The focus could now be on efficiently identifying the smallest k for which an HMSC specification is k-testable. Another interesting problem is to identify a minimal set of tests to validate a k-testable specification.

3.1.4.4. Goals

The first step that should be reached in the near future is the completion of asynchronous testing in the setting without any big-step synchronization. In parallel, work on the problems in local testing should progress

sufficiently to allow, in a mid-term perspective, to understand the relations and possible interconnections between local (i.e. distributed) and asynchronous (centralized) testing. This is the objective of the *TECSTES* project (2011-2014), funded by a DIGITEO *DIM/LSC* grant, and which involves Hernán Ponce de Léon and Stefan Haar of *MExICo*, and Delphine Longuet at LRI, University Paris-Sud/Orsay. We have extended several well known conformance (ioco style) relations for sequential models to models that can handle concurrency (labeled event structures). Two semantics (interleaving and partial order) were presented for every relation. With the interleaving semantics, the relations we obtained boil down to the same relations defined for labeled transition systems, since they focus on sequences of actions. The only advantage of using labeled event structures as a specification formalism for testing remains in the conciseness of the concurrent model with respect to a sequential one. As far as testing is concerned, the benefit is low since every interleaving has to be tested. By contrast, under the partial order semantics, the relations we obtain allow to distinguish explicitly implementations where concurrent actions are implemented concurrently, from those where they are interleaved, i.e. implemented sequentially. Therefore, these relations will be of interest when designing distributed systems, since the natural concurrency between actions that are performed in parallel by different processes can be taken into account. In particular, the fact of being unable to control or observe the order between actions taking place on different processes will not be considered as an impediment for testing.

In [69] and a subsequent journal submission in preparation, we develop complete testing framework for concurrent systems was developped, which included the notions of test suites and test cases. We studied what kind of systems are testable in such a framework and we have proposed sufficient conditions for obtaining a complete test suite. Finally, an algorithm to construct a test suite with such properties was proposed. These result are summarized in a paper that is being prepared for a journal submission.

The mid-to long term goal (perhaps not yet to achieve in this four-year term) is the comprehensive formalization of testing and testability in asynchronous systems with distributed architecture and test protocols.

3.2. Interaction

Participants: Benedikt Bollig, Thomas Chatain, Paul Gastin, Stefan Haar, Serge Haddad.

3.2.1. Introduction

Systems and services exhibit non-trivial *interaction* between specialized and heterogeneous components. This interplay is challenging for several reasons. On one hand, a coordinated interplay of several components is required, though each has only a limited, partial view of the system's configuration. We refer to this problem as *distributed synthesis* or *distributed control*. An aggravating factor is that the structure of a component might be semi-transparent, which requires a form of *grey box management*.

Interaction, one of the main characteristics of systems under consideration, often involves an environment that is not under the control of cooperating services. To achieve a common goal, the services need to agree upon a strategy that allows them to react appropriately regardless of the interactions with the environment. Clearly, the notions of opponents and strategies fall within *game theory*, which is naturally one of our main tools in exploring interaction. We will apply to our problems techniques and results developed in the domains of distributed games and of games with partial information. We will consider also new problems on games that arise from our applications.

3.2.2. Distributed Control

Participants: Benedikt Bollig, Thomas Chatain, Paul Gastin, Stefan Haar.

Program synthesis, as introduced by Church [95] aims at deriving directly an implementation from a specification, allowing the implementation to be correct by design. When the implementation is already at hand but choices remain to be resolved at run time then the problem becomes controller synthesis. Both program and controller synthesis have been extensively studied for sequential systems. In a distributed setting, we need to synthesize a distributed program or distributed controllers that interact locally with the system components. The main difficulty comes from the fact that the local controllers/programs have only a partial view of the entire system. This is also an old problem largely considered undecidable in most settings [114], [108], [113], [100], [102].

Actually, the main undecidability sources come from the fact that this problem was addressed in a synchronous setting using global runs viewed as sequences. In a truly distributed system where interactions are asynchronous we have recently obtained encouraging decidability results [101],[25]. This is a clear witness where concurrency may be exploited to obtain positive results. It is essential to specify expected properties directly in terms of causality revealed by partial order models of executions (MSCs or Mazurkiewicz traces). We intend to develop this line of research with the ambitious aim to obtain decidability for all natural systems and specifications. More precisely, we will identify natural hypotheses both on the architecture of our distributed system and on the specifications under which the distributed program/controller synthesis problem is decidable. This should open the way to important applications, e.g., for distributed control of embedded systems.

3.2.3. Adaptation and Grey box management

Participants: Benedikt Bollig, Stefan Haar, Serge Haddad.

Contrary to mainframe systems or monolithic applications of the past, we are experiencing and using an increasing number of services that are performed not by one provider but rather by the interaction and cooperation of many specialized components. As these components come from different providers, one can no longer assume all of their internal technologies to be known (as it is the case with proprietary technology). Thus, in order to compose e.g. orchestrated services over the web, to determine violations of specifications or contracts, to adapt existing services to new situations etc, one needs to analyze the interaction behavior of *boxes* that are known only through their public interfaces. For their semi-transparent-semi-opaque nature, we shall refer to them as **grey boxes**. While the concrete nature of these boxes can range from vehicles in a highway section to hotel reservation systems, the tasks of *grey box management* have universal features allowing for generalized approaches with formal methods. Two central issues emerge:

- Abstraction: From the designer point of view, there is a need for a trade-off between transparency (no abstraction) in order to integrate the box in different contexts and opacity (full abstraction) for security reasons.
- Adaptation: Since a grey box gives a partial view about the behavior of the component, even if it is not immediately useable in some context, the design of an adaptator is possible. Thus the goal is the synthesis of such an adaptator from a formal specification of the component and the environment.

3.3. Management of Quantitative Behavior

Participants: Sandie Balaguer, Benedikt Bollig, Thomas Chatain, Paul Gastin, Stefan Haar, Serge Haddad, Benjamin Monmege.

3.3.1. Introduction

Besides the logical functionalities of programs, the *quantitative* aspects of component behavior and interaction play an increasingly important role.

- *Real-time* properties cannot be neglected even if time is not an explicit functional issue, since transmission delays, parallelism, etc, can lead to time-outs striking, and thus change even the logical course of processes. Again, this phenomenon arises in telecommunications and web services, but also in transport systems.
- In the same contexts, *probabilities* need to be taken into account, for many diverse reasons such as unpredictable functionalities, or because the outcome of a computation may be governed by race conditions.
- Last but not least, constraints on *cost* cannot be ignored, be it in terms of money or any other limited resource, such as memory space or available CPU time.

Traditional mainframe systems were proprietary and (essentially) localized; therefore, impact of delays, unforeseen failures, etc. could be considered under the control of the system manager. It was therefore natural, in verification and control of systems, to focus on *functional* behavior entirely.

With the increase in size of computing system and the growing degree of compositionality and distribution, quantitative factors enter the stage:

- calling remote services and transmitting data over the web creates *delays*;
- remote or non-proprietary components are not "deterministic", in the sense that their behavior is uncertain.

Time and *probability* are thus parameters that management of distributed systems must be able to handle; along with both, the *cost* of operations is often subject to restrictions, or its minimization is at least desired. The mathematical treatment of these features in distributed systems is an important challenge, which *MExICo* is addressing; the following describes our activities concerning probabilistic and timed systems. Note that cost optimization is not a current activity but enters the picture in several intended activities.

3.3.2. Probabilistic distributed Systems

Participants: Stefan Haar, Serge Haddad.

3.3.2.1. Non-sequential probabilistic processes

Practical fault diagnosis requires to select explanations of *maximal likelihood*; this leads therefore to the question what the probability of a given partially ordered execution is. In Benveniste et al. [86], [79], we presented a model of stochastic processes, whose trajectories are partially ordered, based on local branching in Petri net unfoldings; an alternative and complementary model based on Markov fields is developed in [104], which takes a different view on the semantics and overcomes the first model's restrictions on applicability.

Both approaches abstract away from real time progress and randomize choices in *logical* time. On the other hand, the relative speed - and thus, indirectly, the real-time behavior of the system's local processes - are crucial factors determining the outcome of probabilistic choices, even if non-determinism is absent from the system.

Recently, we started a new line of research with Anne Bouillard, Sidney Rosario, and Albert Benveniste in the DistribCom team at Inria Rennes, studying the likelihood of occurrence of non-sequential runs under random durations in a stochastic Petri net setting.

Once the properties of the probability measures thus obtained are understood, it will be interesting to relate them with the two above models in logical time, and understand their differences. Another mid-term goal, in parallel, is the transfer to *diagnosis*.

3.3.2.2. Distributed Markov Decision Processes

Distributed systems featuring non-deterministic and probabilistic aspects are usually hard to analyze and, more specifically, to optimize. Furthermore, high complexity theoretical lower bounds have been established for models like partially observed Markovian decision processes and distributed partially observed Markovian decision processes. We believe that these negative results are consequences of the choice of the models rather than the intrinsic complexity of problems to be solved. Thus we plan to introduce new models in which the associated optimization problems can be solved in a more efficient way. More precisely, we start by studying connection protocols weighted by costs and we look for online and offline strategies for optimizing the mean cost to achieve the protocol. We cooperate on this subject with Eric Fabre in the DistribCom team at Inria Rennes, in the context of the DISC project.

3.3.3. Large scale probabilistic systems

Addressing large-scale probabilistic systems requires to face state explosion, due to both the discrete part and the probabilistic part of the model. In order to deal with such systems, different approaches have been proposed:

- Restricting the synchronization between the components as in queuing networks allows to express the steady-state distribution of the model by an analytical formula called a product-form [85].
- Some methods that tackle with the combinatory explosion for discrete-event systems can be generalized to stochastic systems using an appropriate theory. For instance symmetry based methods have been generalized to stochastic systems with the help of aggregation theory [94].

- At last simulation, which works as soon as a stochastic operational semantic is defined, has been adapted to perform statistical model checking. Roughly speaking, it consists to produce a confidence interval for the probability that a random path fulfills a formula of some temporal logic [120] .

We want to contribute to these three axes: (1) we are looking for product-forms related to systems where synchronization are more involved (like in Petri nets), (2) we want to adapt methods for discrete-event systems that require some theoretical developments in the stochastic framework and, (3) we plane to address some important limitations of statistical model checking like the expressiveness of the associated logic and the handling of rare events.

3.3.4. Real time distributed systems

Nowadays, software systems largely depend on complex timing constraints and usually consist of many interacting local components. Among them, railway crossings, traffic control units, mobile phones, computer servers, and many more safety-critical systems are subject to particular quality standards. It is therefore becoming increasingly important to look at networks of timed systems, which allow real-time systems to operate in a distributed manner.

Timed automata are a well-studied formalism to describe reactive systems that come with timing constraints. For modeling distributed real-time systems, networks of timed automata have been considered, where the local clocks of the processes usually evolve at the same rate [111] [90]. It is, however, not always adequate to assume that distributed components of a system obey a global time. Actually, there is generally no reason to assume that different timed systems in the networks refer to the same time or evolve at the same rate. Any component is rather determined by local influences such as temperature and workload.

3.3.4.1. Distributed timed systems with independently evolving clocks

Participants: Benedikt Bollig, Paul Gastin.

A first step towards formal models of distributed timed systems with independently evolving clocks was done in [80]. As the precise evolution of local clock rates is often too complex or even unknown, the authors study different semantics of a given system: The *existential semantics* exhibits all those behaviors that are possible under *some* time evolution. The *universal semantics* captures only those behaviors that are possible under *all* time evolutions. While emptiness and universality of the universal semantics are in general undecidable, the existential semantics is always regular and offers a way to check a given system against safety properties. A decidable under-approximation of the universal semantics, called *reactive semantics*, is introduced to check a system for liveness properties. It assumes the existence of a *global* controller that allows the system to react upon local time evolutions. A short term goal is to investigate a *distributed* reactive semantics where controllers are located at processes and only have local views of the system behaviors.

Several questions, however, have not yet been tackled in this previous work or remain open. In particular, we plan to exploit the power of synchronization via local clocks and to investigate the *synthesis problem*: For which (global) specifications $\mathcal{S}$ can we generate a distributed timed system with independently evolving clocks $\mathcal{A}$ (over some given system architecture) such that both the reactive and the existential semantics of $\mathcal{A}$ are precisely (the semantics of) $\mathcal{S}$? In this context, it will be favorable to have partial-order based specification languages and a partial-order semantics for distributed timed systems. The fact that clocks are not shared may allow us to apply partial-order–reduction techniques.

If, on the other hand, a system is already given and complemented with a specification, then one is usually interested in controlling the system in such a way that it meets its specification. The interaction between the actual *system* and the *environment* (i.e., the local time evolution) can now be understood as a 2-player game: the system's goal is to guarantee a behavior that conforms with the specification, while the environment aims at violating the specification. Thus, building a controller of a system actually amounts to computing winning strategies in imperfect-information games with infinitely many states where the unknown or unpredictable evolution of time reflects an imperfect information of the environment. Only few efforts have been made to tackle those kinds of games. One reason might be that, in the presence of imperfect information and infinitely many states, one is quickly confronted with undecidability of basic decision problems.

3.3.4.2. Implementation of Real-Time Concurrent Systems

Participants: Sandie Balaguer, Thomas Chatain, Stefan Haar, Serge Haddad.

This is one of the tasks of the ANR ImpRo.

The objective is to provide formal guarantees on the implementation of real-time distributed systems, despite the semantic differences between the model and the code. We consider two kinds of timed models: time Petri nets [112] and networks of timed automata [81].

Time Petri Nets allow the designer to explicit concurrent parts of the system, but without having decided yet to localize the different actions on the different components. In that sense, TPNs are more abstract than networks of timed automata, which can be seen as possible (ideal) distributed implementations. This raises the question of semantical comparison of these two models in the light of preserving the maximum of concurrency.

In order to implement our models on distributed architectures, we need a way to evaluate how much the implementation preserves the concurrency that is described in the model. For this we must be able to identify concurrency in the behavior of the models. This is done by equipping the models with a concurrent semantics (unfoldings), allowing us to consider the behaviors as partial orders.

For instance, we would like to be able to transform a time Petri net into a network of timed automata, which is closer to the implementation since the processes are well identified. But we require that this transformation preserves concurrency. Yet the first works about formal comparisons of the expressivity of these models [92], [89], [91], [93], [118] do not consider preservation of concurrency.

In contrast, we aim at formalizing and automating translations that preserve both the timed semantics and the concurrent semantics. This effort is crucial for extending concurrency-oriented methods for logical time, in particular for exploiting partial order properties. In fact, validation and management - in a broad sense - of distributed systems is not realistic *in general* without understanding and control of their real-time dependent features; the link between real-time and logical-time behaviors is thus crucial for many aspects of *MExICo*'s work.

3.3.5. Weighted Automata and Weighted Logics

Participants: Benedikt Bollig, Paul Gastin, Benjamin Monmege.

Time and probability are only two facets of quantitative phenomena. A generic concept of adding weights to qualitative systems is provided by the theory of weighted automata [78]. They allow one to treat probabilistic or also reward models in a unified framework. Unlike finite automata, which are based on the Boolean semiring, weighted automata build on more general structures such as the natural or real numbers (equipped with the usual addition and multiplication) or the probabilistic semiring. Hence, a weighted automaton associates with any possible behavior a weight beyond the usual Boolean classification of "acceptance" or "non-acceptance". Automata with weights have produced a well-established theory and come, e.g., with a characterization in terms of rational expressions, which generalizes the famous theorem of Kleene in the unweighted setting. Equipped with a solid theoretical basis, weighted automata finally found their way into numerous application areas such as natural language processing and speech recognition, or digital image compression.

What is still missing in the theory of weighted automata are satisfactory connections with verification-related issues such as (temporal) logic and bisimulation that could lead to a general approach to corresponding satisfiability and model-checking problems. A first step towards a more satisfactory theory of weighted systems was done in [12]. That paper, however does not give final solutions to all the aforementioned problems. It identifies directions for future research that we will be tackling.

PAREO Project-Team

3. Scientific Foundations

3.1. Introduction

It is a common claim that rewriting is ubiquitous in computer science and mathematical logic. And indeed the rewriting concept appears from very theoretical settings to very practical implementations. Some extreme examples are the mail system under Unix that uses rules in order to rewrite mail addresses in canonical forms (see the `/etc/sendmail.cf` file in the configuration of the mail system) and the transition rules describing the behaviors of tree automata. Rewriting is used in semantics in order to describe the meaning of programming languages [28] as well as in program transformations like, for example, re-engineering of Cobol programs [36]. It is used in order to compute, implicitly or explicitly as in Mathematica or MuPAD, but also to perform deduction when describing by inference rules a logic [24], a theorem prover [26] or a constraint solver [27]. It is of course central in systems making the notion of rule an explicit and first class object, like expert systems, programming languages based on equational logic, algebraic specifications, functional programming and transition systems.

In this context, the study of the theoretical foundations of rewriting have to be continued and effective rewrite based tools should be developed. The extensions of first-order rewriting with higher-order and higher-dimension features are hot topics and these research directions naturally encompass the study of the rewriting calculus, of polygraphs and of their interaction. The usefulness of these concepts becomes more clear when they are implemented and a considerable effort is thus put nowadays in the development of expressive and efficient rewrite based programming languages.

3.2. Rule-based programming languages

Programming languages are formalisms used to describe programs, applications, or software which aim to be executed on a given hardware. In principle, any Turing complete language is sufficient to describe the computations we want to perform. However, in practice the choice of the programming language is important because it helps to be effective and to improve the quality of the software. For instance, a web application is rarely developed using a Turing machine or assembly language. By choosing an adequate formalism, it becomes easier to reason about the program, to analyze, certify, transform, optimize, or compile it. The choice of the programming language also has an impact on the quality of the software. By providing high-level constructs as well as static verifications, like typing, we can have an impact on the software design, allowing more expressiveness, more modularity, and a better reuse of code. This also improves the productivity of the programmer, and contributes to reducing the presence of errors.

The quality of a programming language depends on two main factors. First, the *intrinsic design*, which describes the programming model, the data model, the features provided by the language, as well as the semantics of the constructs. The second factor is the programmer and the application which is targeted. A language is not necessarily good for a given application if the concepts of the application domain cannot be easily manipulated. Similarly, it may not be good for a given person if the constructs provided by the language are not correctly understood by the programmer.

In the *Pareo* group we target a population of programmers interested in improving the long-term maintainability and the quality of their software, as well as their efficiency in implementing complex algorithms. Our privileged domain of application is large since it concerns the development of *transformations*. This ranges from the transformation of textual or structured documents such as XML, to the analysis and the transformation of programs and models. This also includes the development of tools such as theorem provers, proof assistants, or model checkers, where the transformations of proofs and the transitions between states play a crucial role. In that context, the *expressiveness* of the programming language is important. Indeed, complex encodings into low level data structures should be avoided, in contrast to high level notions such as abstract types and transformation rules that should be provided.

It is now well established that the notions of *term* and *rewrite rule* are two universal abstractions well suited to model tree based data types and the transformations that can be done upon them. Over the last ten years we have developed a strong experience in designing and programming with rule based languages [29], [20], [18]. We have introduced and studied the notion of *strategy* [19], which is a way to control how the rules should be applied. This provides the separation which is essential to isolate the logic and to make the rules reusable in different contexts.

To improve the quality of programs, it is also essential to have a clear description of their intended behaviors. For that, the *semantics* of the programming language should be formally specified.

There is still a lot of progress to be done in these directions. In particular, rule based programming can be made even more expressive by extending the existing matching algorithms to context-matching or to new data structures such as graphs or polygraphs. New algorithms and implementation techniques have to be found to improve the efficiency and make the rule based programming approach effective on large problems. Separating the rules from the control is very important. This is done by introducing a language for describing strategies. We still have to invent new formalisms and new strategy primitives which are both expressive enough and theoretically well grounded. A challenge is to find a good strategy language we can reason about, to prove termination properties for instance.

On the static analysis side, new formalized typing algorithms are needed to properly integrate rule based programming into already existing host languages such as Java. The notion of traversal strategy merits to be better studied in order to become more flexible and still provide a guarantee that the result of a transformation is correctly typed.

3.3. Rewriting calculus

The huge diversity of the rewriting concept is obvious and when one wants to focus on the underlying notions, it becomes quickly clear that several technical points should be settled. For example, what kind of objects are rewritten? Terms, graphs, strings, sets, multisets, others? Once we have established this, what is a rewrite rule? What is a left-hand side, a right-hand side, a condition, a context? And then, what is the effect of a rule application? This leads immediately to defining more technical concepts like variables in bound or free situations, substitutions and substitution application, matching, replacement; all notions being specific to the kind of objects that have to be rewritten. Once this is solved one has to understand the meaning of the application of a set of rules on (classes of) objects. And last but not least, depending on the intended use of rewriting, one would like to define an induced relation, or a logic, or a calculus.

In this very general picture, we have introduced a calculus whose main design concept is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *application* and *result*. We concentrate on *term* rewriting, we introduce a very general notion of rewrite rule and we make the rule application and result explicit concepts. These are the basic ingredients of the *rewriting*- or ρ-calculus whose originality comes from the fact that terms, rules, rule application and application strategies are all treated at the object level (a rule can be applied on a rule for instance).

The λ-calculus is usually put forward as the abstract computational model underlying functional programming. However, modern functional programming languages have pattern-matching features which cannot be directly expressed in the λ-calculus. To palliate this problem, pattern-calculi [34], [31], [25] have been introduced. The rewriting calculus is also a pattern calculus that combines the expressiveness of pure functional calculi and algebraic term rewriting. This calculus is designed and used for logical and semantical purposes. It could be equipped with powerful type systems and used for expressing the semantics of rule based as well as object oriented languages. It allows one to naturally express exception handling mechanisms and elaborated rewriting strategies. It can be also extended with imperative features and cyclic data structures.

The study of the rewriting calculus turns out to be extremely successful in terms of fundamental results and of applications [22]. Different instances of this calculus together with their corresponding type systems have been proposed and studied. The expressive power of this calculus was illustrated by comparing it with similar

formalisms and in particular by giving a typed encoding of standard strategies used in first-order rewriting and classical rewrite based languages like *ELAN* and *Tom*.

PARSIFAL Project-Team

3. Scientific Foundations

3.1. General overview

There are two broad approaches for computational specifications. In the *computation as model* approach, computations are encoded as mathematical structures containing nodes, transitions, and state. Logic is used to *describe* these structures, that is, the computations are used as models for logical expressions. Intensional operators, such as the modals of temporal and dynamic logics or the triples of Hoare logic, are often employed to express propositions about the change in state.

The *computation as deduction* approach, in contrast, expresses computations logically, using formulas, terms, types, and proofs as computational elements. Unlike the model approach, general logical apparatus such as cut-elimination or automated deduction becomes directly applicable as tools for defining, analyzing, and animating computations. Indeed, we can identify two main aspects of logical specifications that have been very fruitful:

- *Proof normalization*, which treats the state of a computation as a proof term and computation as normalization of the proof terms. General reduction principles such as β-reduction or cut-elimination are merely particular forms of proof normalization. Functional programming is based on normalization [44], and normalization in different logics can justify the design of new and different functional programming languages [30].
- *Proof search*, which views the state of a computation as a a structured collection of formulas, known as a *sequent*, and proof search in a suitable sequent calculus as encoding the dynamics of the computation. Logic programming is based on proof search [49], and different proof search strategies can be used to justify the design of new and different logic programming languages [48].

While the distinction between these two aspects is somewhat informal, it helps to identify and classify different concerns that arise in computational semantics. For instance, confluence and termination of reductions are crucial considerations for normalization, while unification and strategies are important for search. A key challenge of computational logic is to find means of uniting or reorganizing these apparently disjoint concerns.

An important organizational principle is structural proof theory, that is, the study of proofs as syntactic, algebraic and combinatorial objects. Formal proofs often have equivalences in their syntactic representations, leading to an important research question about *canonicity* in proofs – when are two proofs "essentially the same?" The syntactic equivalences can be used to derive normal forms for proofs that illuminate not only the proofs of a given formula, but also its entire proof search space. The celebrated *focusing* theorem of Andreoli [32] identifies one such normal form for derivations in the sequent calculus that has many important consequences both for search and for computation. The combinatorial structure of proofs can be further explored with the use of *deep inference*; in particular, deep inference allows access to simple and manifestly correct cut-elimination procedures with precise complexity bounds.

Type theory is another important organizational principle, but most popular type systems are generally designed for either search or for normalization. To give some examples, the Coq system [56] that implements the Calculus of Inductive Constructions (CIC) is designed to facilitate the expression of computational features of proofs directly as executable functional programs, but general proof search techniques for Coq are rather primitive. In contrast, the Twelf system [53] that is based on the LF type theory (a subsystem of the CIC), is based on relational specifications in canonical form (*i.e.*, without redexes) for which there are sophisticated automated reasoning systems such as meta-theoretic analysis tools, logic programming engines, and inductive theorem provers. In recent years, there has been a push towards combining search and normalization in the same type-theoretic framework. The Beluga system [54], for example, is an extension of the LF type theory with a purely computational meta-framework where operations on inductively defined LF objects can be expressed as functional programs.

The Parsifal team investigates both the search and the normalization aspects of computational specifications using the concepts, results, and insights from proof theory and type theory.

3.2. Design of two level-logic systems

The team has spent a number of years in designing a strong new logic that can be used to reason (inductively and co-inductively) on syntactic expressions containing bindings. This work has been published is a series of papers by McDowell and Miller [46] [45], Tiu and Miller [51] [57], and Gacek, Miller, and Nadathur [2] [38]. Besides presenting formal properties of these logic, these papers also documented a number of examples where this logic demonstrated superior approaches to reasoning about a number of complex formal systems, ranging from programming languages to the λ-calculus and π-calculus.

The team has also been working on three different prototype theorem proving system that are all related to this stronger logic. These systems are the following.

- Abella, which is an interactive theorem prover for the full logic.
- Bedwyr, which is a model checker for the "finite" part of the logic.
- Tac, which is a sophisticate tactic for automatically completing simple proofs involving induction and unfolding.

We are now in the process of attempting to make all of these system communicate properly. Given that these systems have been authored by different team members at different times and for different reasons, they do not formally share the same notions of syntax and proof. We are now working to revisit all of these systems and revise them so that they all work on the *same* logic and so that they can share their proofs with each other.

Currently, Chaudhuri, Miller, and Accattoli are working with our technical staff member, Heath, to redesign and restructure these systems so that they can cooperate in building proofs.

3.3. Making the case for proof certificates

The team has been considering how it might be possible to define a universal format for proofs so that any existing theorem provers can have its proofs trusted by any other prover. This is a rather ambitious project and involves a great deal of work at the infrastructure level of computational logic. As a result, we have put significant energies into considering the high-level objectives and consequences of deploying such proof certificates.

Our current thinking on this point is roughly the following. Proofs, both formal and informal, are documents that are intended to circulate within societies of humans and machines distributed across time and space in order to provide trust. Such trust might lead a mathematician to accept a certain statement as true or it might help convince a consumer that a certain software system is secure. Using this general definition of proof, we have re-examined a range of perspectives about proofs and their roles within mathematics and computer science that often appears contradictory.

Given this view of proofs as both document and object, that need to be communicated and checked, we have attempted to define a particular approach to a *broad spectrum proof certificate* format that is intended as a universal language for communicating formal proofs among computational logic systems. We identify four desiderata for such proof certificates: they must be

1. checkable by simple proof checkers,
2. flexible enough that existing provers can conveniently produce such certificates from their internal evidence of proof,
3. directly related to proof formalisms used within the structural proof theory literature, and
4. permit certificates to elide some proof information with the expectation that a proof checker can reconstruct the missing information using bounded and structured proof search.

We consider various consequences of these desiderata, including how they can mix computation and deduction and what they mean for the establishment of marketplaces and libraries of proofs. More specifics can be found in Miller's papers [8] and [47].

3.4. Combining Classical and Intuitionistic Proof Systems

In order to develop an approach to proof certificates that is as comprehensive as possible, one needs to handle theorems and proofs in both classical logic and intuitionistic logic. Yet, building two separate libraries, one for each logic, can be inconvenient and error-prone. An ideal approach would be to design a single proof system in which both classical and intuitionistic proofs can exist together. Such a proof system should allow cut-elimination to take place and should have a sensible semantic framework.

Liang and Miller have recently been working on exactly that problem. In their paper [7], they showed how to describe a general setting for specifying proofs in intuitionistic and classical logic and to achieve one framework for describing initial-elimination and cut-elimination for such these two logics. That framework allowed for some mixing of classical and intuitionistic features in one logic. A more ambitious merging of these logics was provided in their work on "polarized intuitionistic logic" in which classical and intuitionistic connectives can be used within the same formulas [14].

3.5. Deep inference

Deep inference [40], [42] is a novel methodology for presenting deductive systems. Unlike traditional formalisms like the sequent calculus, it allows rewriting of formulas deep inside arbitrary contexts. The new freedom for designing inference rules creates a richer proof theory. For example, for systems using deep inference, we have a greater variety of normal forms for proofs than in sequent calculus or natural deduction systems. Another advantage of deep inference systems is the close relationship to categorical proof theory. Due to the deep inference design one can directly read off the morphism from the derivations. There is no need for a counter-intuitive translation.

The following research problems are investigated by members of the Parsifal team:

- Find deep inference system for richer logics. This is necessary for making the proof theoretic results of deep inference accessible to applications as they are described in the previous sections of this report.
- Investigate the possibility of focusing proofs in deep inference. As described before, focusing is a way to reduce the non-determinism in proof search. However, it is well investigated only for the sequent calculus. In order to apply deep inference in proof search, we need to develop a theory of focusing for deep inference.

3.6. Proof nets and atomic flows

Proof nets and atomic flows are abstract (graph-like) presentations of proofs such that all "trivial rule permutations" are quotiented away. Ideally the notion of proof net should be independent from any syntactic formalism, but most notions of proof nets proposed in the past were formulated in terms of their relation to the sequent calculus. Consequently we could observe features like "boxes" and explicit "contraction links". The latter appeared not only in Girard's proof nets [39] for linear logic but also in Robinson's proof nets [55] for classical logic. In this kind of proof nets every link in the net corresponds to a rule application in the sequent calculus.

Only recently, due to the rise of deep inference, new kinds of proof nets have been introduced that take the formula trees of the conclusions and add additional "flow-graph" information (see e.g., [4], [3] and [41]. On one side, this gives new insights in the essence of proofs and their normalization. But on the other side, all the known correctness criteria are no longer available.

This directly leads to the following research questions investigated by members of the parsifal team:

- Finding (for classical logic) a notion of proof nets that is deductive, i.e., can effectively be used for doing proof search. An important property of deductive proof nets must be that the correctness can be checked in linear time. For the classical logic proof nets by Lamarche and Straßburger [4] this takes exponential time (in the size of the net).
- Studying the normalization of proofs in classical logic using atomic flows. Although there is no correctness criterion they allow to simplify the normalization procedure for proofs in deep inference, and additionally allow to get new insights in the complexity of the normalization.

PI.R2 Project-Team

3. Scientific Foundations

3.1. Proof theory and the Curry-Howard correspondence

3.1.1. Proofs as programs

Proof theory is the branch of logic devoted to the study of the structure of proofs. An essential contributor to this field is Gentzen [49] who developed in 1935 two logical formalisms that are now central to the study of proofs. These are the so-called "natural deduction", a syntax that is particularly well-suited to simulate the intuitive notion of reasoning, and the so-called "sequent calculus", a syntax with deep geometric properties that is particularly well-suited for proof automation.

Proof theory gained a remarkable importance in computer science when it became clear, after genuine observations first by Curry in 1958 [43], then by Howard and de Bruijn at the end of the 60's [56], [73], that proofs had the very same structure as programs: for instance, natural deduction proofs can be identified as typed programs of the ideal programming language known as λ-calculus.

This proofs-as-programs correspondence has been the starting point to a large spectrum of researches and results contributing to deeply connect logic and computer science. In particular, it is from this line of work that Coquand's Calculus of Constructions [40] stemmed out – a formalism that is both a logic and a programming language and that is at the source of the Coq system [39].

3.1.2. Towards the calculus of constructions

The λ-calculus, defined by Church [38], is a remarkably succinct model of computation that is defined via only three constructions (abstraction of a program with respect to one of its parameters, reference to such a parameter, application of a program to an argument) and one reduction rule (substitution of the formal parameter of a program by its effective argument). The λ-calculus, which is Turing-complete, i.e. which has the same expressiveness as a Turing machine (there is for instance an encoding of numbers as functions in λ-calculus), comes with two possible semantics referred to as call-by-name and call-by-value evaluations. Of these two semantics, the first one, which is the simplest to characterise, has been deeply studied in the last decades [34].

For explaining the Curry-Howard correspondence, it is important to distinguish between intuitionistic and classical logic: following Brouwer at the beginning of the 20^{th} century, classical logic is a logic that accepts the use of reasoning by contradiction while intuitionistic logic proscribes it. Then, Howard's observation is that the proofs of the intuitionistic natural deduction formalism exactly coincide with programs in the (simply typed) λ-calculus.

A major achievement has been accomplished by Martin-Löf who designed in 1971 a formalism, referred to as modern type theory, that was both a logical system and a (typed) programming language [62].

In 1985, Coquand and Huet [40], [41] in the Formel team of Inria-Rocquencourt explored an alternative approach based on Girard-Reynolds' system F [50], [68]. This formalism, called the Calculus of Constructions, served as logical foundation of the first implementation of Coq in 1984. Coq was called CoC at this time.

3.1.3. The Calculus of Inductive Constructions

The first public release of CoC dates back to 1989. The same project-team developed the programming language Caml (nowadays coordinated by the Gallium team) that provided the expressive and powerful concept of algebraic data types (a paragon of it being the type of list). In CoC, it was possible to simulate algebraic data types, but only through a not-so-natural not-so-convenient encoding.

In 1989, Coquand and Paulin [42] designed an extension of the Calculus of Constructions with a generalisation of algebraic types called inductive types, leading to the Calculus of Inductive Constructions (CIC) that started to serve as a new foundation for the Coq system. This new system, which got its current definitive name Coq, was released in 1991.

In practice, the Calculus of Inductive Constructions derives its strength from being both a logic powerful enough to formalise all common mathematics (as set theory is) and an expressive richly-typed functional programming language (like ML but with a richer type system, no effects and no non-terminating functions).

3.2. The development of Coq

Since 1984, about 40 persons have contributed to the development of Coq, out of which 7 persons have contributed to bring the system to the place it is now. First Thierry Coquand through his foundational theoretical ideas, then Gérard Huet who developed the first prototypes with Thierry Coquand and who headed the Coq group until 1998, then Christine Paulin who was the main actor of the system based on the CIC and who headed the development group from 1998 to 2006. On the programming side, important steps were made by Chet Murthy who raised Coq from the prototypical state to a reasonably scalable system, Jean-Christophe Filliâtre who turned to concrete the concept of a small trustful certification kernel on which an arbitrary large system can be set up, Bruno Barras and Hugo Herbelin who, among other extensions, reorganised Coq on a new smoother and more uniform basis able to support a new round of extensions for the next decade.

The development started from the Formel team at Rocquencourt but, after Christine Paulin got a position in Lyon, it spread to École Normale Supérieure de Lyon. Then, the task force there globally moved to the University of Orsay when Christine Paulin got a new position there. On the Rocquencourt side, the part of Formel involved in ML moved to the Cristal team (now Gallium) and Formel got renamed into Coq. Gérard Huet left the team and Christine Paulin started to head a Coq team bilocalised at Rocquencourt and Orsay. Gilles Dowek became the head of the team which was renamed into LogiCal. Following Gilles Dowek who got a position at École Polytechnique, LogiCal globally moved to Futurs with a bilocalisation on Orsay and Palaiseau. It then split again giving birth to ProVal. At the same time, the Marelle team (formerly Lemme, formerly Croap) which has been a long partner of the Formel team, invested more and more energy in both the formalisation of mathematics in Coq and in user interfaces for Coq.

After various other spreadings resulting from where the wind pushed former PhD students, the development of Coq got multi-site with the development now realised by employees of Inria, the CNAM and Paris 7.

We next briefly describe the main components of Coq.

3.2.1. The underlying logic and the verification kernel

The architecture adopts the so-called de Bruijn principle: the well-delimited *kernel* of Coq ensures the correctness of the proofs validated by the system. The kernel is rather stable with modifications tied to the evolution of the underlying Calculus of Inductive Constructions formalism. The kernel includes an interpreter of the programs expressible in the CIC and this interpreter exists in two flavours: a customisable lazy evaluation machine written in OCaml and a call-by-value bytecode interpreter written in C dedicated to efficient computations. The kernel also provides a module system.

3.2.2. Programming and specification languages

The concrete user language of Coq, called *Gallina*, is a high-level language built on top of the CIC. It includes a type inference algorithm, definitions by complex pattern-matching, implicit arguments, mathematical notations and various other high-level language features. This high-level language serves both for the development of programs and for the formalisation of mathematical theories. Coq also provides a large set of commands. Gallina and the commands together forms the *Vernacular* language of Coq.

3.2.3. Libraries

Libraries are written in the vernacular language of Coq. There are libraries for various arithmetical structures and various implementations of numbers (Peano numbers, implementation of $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ with binary digits, implementation of $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ using machine words, axiomatisation of $\mathbb{R}$). There are libraries for lists, list of a specified length, sorts, and for various implementations of finite maps and finite sets. There are libraries on relations, sets, orders.

3.2.4. Tactics

The tactics are the methods available to conduct proofs. This includes the basic inference rules of the CIC, various advanced higher level inference rules and all the automation tactics. Regarding automation, there are tactics for solving systems of equations, for simplifying ring or field expressions, for arbitrary proof search, for semi-decidability of first-order logic and so on. There is also a powerful and popular untyped scripting language for combining tactics into more complex tactics.

Note that all tactics of Coq produce proof certificates that are checked by the kernel of Coq. As a consequence, possible bugs in proof methods do not hinder the confidence in the correctness of the Coq checker. Note also that the CIC being a programming language, tactics can be written (and certified) in the own language of Coq if needed.

3.2.5. Extraction

Extraction is a component of Coq that maps programs (or even computational proofs) of the CIC to functional programs (in OCaml, Scheme or Haskell). Especially, a program certified by Coq can further be extracted to a program of a full-fledged programming language then benefiting of the efficient compilation, linking tools, profiling tools, ... of the target software.

3.3. Dependently typed programming languages

Dependently typed programming (shortly DTP) is an emerging concept referring to the diffuse and broadening tendency to develop programming languages with type systems able to express program properties finer than the usual information of simply belonging to specific data-types. The type systems of dependently-typed programming languages allow to express properties *dependent* of the input and the output of the program (for instance that a sorting program returns a list of same size as its argument). Typical examples of such languages were the Cayenne language, developed in the late 90's at Chalmers University in Sweden and the DML language developed at Boston. Since then, various new tools have been proposed, either as typed programming languages whose types embed equalities (Ωmega at Portland, ATS at Boston, ...) or as hybrid logic/programming frameworks (Agda at Chalmers University, Twelf at Carnegie, Delphin at Yale, OpTT at U. Iowa, Epigram at Nottingham, ...).

DTP contributes to a general movement leading to the fusion between logic and programming. Coq, whose language is both a logic and a programming language which moreover can be extracted to pure ML code plays a role in this movement and some frameworks for DTP have been proposed on top of Coq (Concoqtion at Rice and Colorado, Ynot at Harvard, Why in the ProVal team at Inria). It also connects to Hoare logic, providing frameworks where pre- and post-conditions of programs are tied with the programs.

DTP approached from the programming language side generally benefits of a full-fledged language (e.g. supporting effects) with efficient compilation. DTP approached from the logic side generally benefits of an expressive specification logic and of proof methods so as to certify the specifications. The weakness of the approach from logic however is generally the weak support for effects or partial functions.

3.3.1. Type-checking and proof automation

In between the decidable type systems of conventional data-types based programming languages and the full expressiveness of logically undecidable formulae an active field of research explores a spectrum of decidable or semi-decidable type systems for possible use in dependently programming languages. At the beginning of the spectrum, this includes for instance the system F's extension ML_F of the ML type system or the generalisation

of abstract data types with type constraints (G.A.D.T.) such as found in the Haskell programming language. At the other side of the spectrum, one finds arbitrary complex type specification languages (e.g. that a sorting function returns a list of type "sorted list") for which more or less powerful proof automation tools (generally first-order ones) exist.

3.3.2. Libraries

Developing libraries for programming languages takes time and generally benefits of a critical mass effect. An advantage is given to languages that start from well-established existing frameworks for which a large panel of libraries exist. Coq is such a framework.

3.4. Around and beyond the Curry-Howard correspondence

For two decades, the Curry-Howard correspondence was limited to the intuitionistic case but in 1990, an important stimulus spurred on the community following the discovery by Griffin that the correspondence was extensible to classical logic. The community then started to investigate unexplored potential fields of connection between computer science and logic. One of these fields was the computational understanding of Gentzen's sequent calculus while another one was the computational content of the axiom of choice.

3.4.1. Control operators and classical logic

Indeed, a significant extension of the Curry-Howard correspondence has been obtained at the beginning of the 90's thanks to the seminal observation by Griffin [51] that some operators known as control operators were typable by the principle of double negation elimination ($\neg\neg A \Rightarrow A$), a principle which provides classical logic.

Control operators are operators used to jump from one place of a program to another place. They were first considered in the 60's by Landin [61] and Reynolds [67] and started to be studied in an abstract way in the 80's by Felleisen *et al* [45], culminating in Parigot's $\lambda\mu$-calculus [64], a reference calculus that is in fine Curry-Howard correspondence with classical natural deduction. In this respect, control operators are fundamental pieces of the full connection between proofs and programs.

3.4.2. Sequent calculus

The Curry-Howard interpretation of sequent calculus started to be investigated at the beginning of the 90's. The main technicality of sequent calculus is the presence of *left introduction* inference rules and two kinds of interpretations of these rules are applicable. The first approach interprets left introduction rules as construction rules for a language of patterns but it does not really address the problem of the interpretation of the implication connective. The second approach, started in 1994, interprets left introduction rules as evaluation context formation rule. This line of work culminated in 2000 with the design by Hugo Herbelin and Pierre-Louis Curien of a symmetric calculus exhibiting deep dualities between the notion of programs and evaluation contexts and between the standard notions of call-by-name and call-by-value evaluation semantics.

3.4.3. Abstract machines

Abstract machines came as an intermediate evaluation device, between high-level programming languages and the computer microprocessor. The typical reference for call-by-value evaluation of λ-calculus is Landin's SECD machine [60] and Krivine's abstract machine for call-by-name evaluation [59], [57]. A typical abstract machine manipulates a state that consists of a program in some environment of bindings and some evaluation context traditionally encoded into a "stack".

3.4.4. Delimited control

Delimited control extends the expressiveness of control operators with effects: the fundamental result here is a completeness result by Filinski [46]: any side-effect expressible in monadic style (and this covers references, exceptions, states, dynamic bindings, ...) can be simulated in λ-calculus equipped with delimited control.

PROSECCO Project-Team

3. Scientific Foundations

3.1. Symbolic verification of cryptographic applications

Despite decades of experience, designing and implementing cryptographic applications remains dangerously error-prone, even for experts. This is partly because cryptographic security is an inherently hard problem, and partly because automated verification tools require carefully-crafted inputs and are not widely applicable. To take just the example of TLS, a widely-deployed and well-studied cryptographic protocol designed, implemented, and verified by security experts, the lack of a formal proof about all its details has regularly led to the discovery of major attacks (in 2003, 2008, 2009, and 2011) on both the protocol and its implementations, after many years of unsuspecting use.

As a result, the automated verification for cryptographic applications is an active area of research, with a wide variety of tools being employed for verifying different kinds of applications.

In previous work, the we have developed the following three approaches:

- ProVerif: a symbolic prover for cryptographic protocol models
- Tookan: an attack-finder for PKCS#11 hardware security devices
- F7: a security typechecker for cryptographic applications written in F#

3.1.1. Verifying cryptographic protocols with ProVerif

Given a model of a cryptographic protocol, the problem is to verify that an active attacker, possibly with access to some cryptographic keys but unable to guess other secrets, cannot thwart security goals such as authentication and secrecy [52]; it has motivated a serious research effort on the formal analysis of cryptographic protocols, starting with [50] and eventually leading to effective verification tools, such as our tool ProVerif.

To use ProVerif, one encodes a protocol model in a formal language, called the applied pi-calculus, and ProVerif abstracts it to a set of generalized Horn clauses. This abstraction is a small approximation: it just ignores the number of repetitions of each action, so ProVerif is still very precise, more precise than, say, tree automata-based techniques. The price to pay for this precision is that ProVerif does not always terminate; however, it terminates in most cases in practice, and it always terminates on the interesting class of *tagged protocols* [46]. ProVerif also distinguishes itself from other tools by the variety of cryptographic primitives it can handle, defined by rewrite rules or by some equations, and the variety of security properties it can prove: secrecy [44], [38], correspondences (including authentication) [45], and observational equivalences [43]. Observational equivalence means that an adversary cannot distinguish two processes (protocols); equivalences can be used to formalize a wide range of properties, but they are particularly difficult to prove. Even if the class of equivalences that ProVerif can prove is limited to equivalences between processes that differ only by the terms they contain, these equivalences are useful in practice and ProVerif is the only tool that proves equivalences for an unbounded number of sessions.

Using ProVerif, it is now possible to verify large parts of industrial-strength protocols such as TLS [13], JFK [39], and Web Services Security [42]. against powerful adversaries that can run an unlimited number of protocol sessions, for strong security properties expressed as correspondence queries or equivalence assertions. ProVerif is used by many teams at the international level, and has been used in more 30 research papers (references available at http://proverif.inria.fr/proverif-users.html).

3.1.2. Verifying security APIs using Tookan

Security application programming interfaces (APIs) are interfaces that provide access to functionality while also enforcing a security policy, so that even if a malicious program makes calls to the interface, certain security properties will continue to hold. They are used, for example, by cryptographic devices such as smartcards and Hardware Security Modules (HSMs) to manage keys and provide access to cryptographic functions whilst keeping the keys secure. Like security protocols, their design is security critical and very difficult to get right. Hence formal techniques have been adapted from security protocols to security APIs.

The most widely used standard for cryptographic APIs is RSA PKCS#11, ubiquitous in devices from smartcards to HSMs. A 2003 paper highlighted possible flaws in PKCS#11 [48], results which were extended by formal analysis work using a Dolev-Yao style model of the standard [49]. However at this point it was not clear to what extent these flaws affected real commercial devices, since the standard is underspecified and can be implemented in many different ways. The Tookan tool, developed by Steel in collaboration with Bortolozzo, Centenaro and Focardi, was designed to address this problem. Tookan can reverse engineer the particular configuration of PKCS#11 used by a device under test by sending a carefully designed series of PKCS#11 commands and observing the return codes. These codes are used to instantiate a Dolev-Yao model of the device's API. This model can then be searched using a security protocol model checking tool to find attacks. If an attack is found, Tookan converts the trace from the model checker into the sequence of PKCS#11 queries needed to make the attack and executes the commands directly on the device. Results obtained by Tookan are remarkable: of 18 commercially available PKCS#11 devices tested, 10 were found to be susceptible to at least one attack.

3.1.3. Verifying cryptographic applications using F7

Verifying the implementation of a protocol has traditionally been considered much harder than verifying its model. This is mainly because implementations have to consider real-world details of the protocol, such as message formats, that models typically ignore. This leads to a situation that a protocol may have been proved secure in theory, but its implementation may be buggy and insecure. However, with recent advances in both program verification and symbolic protocol verification tools, it has become possible to verify fully functional protocol implementations in the symbolic model.

One approach is to extract a symbolic protocol model from an implementation and then verify the model, say, using ProVerif. This approach has been quite successful, yielding a verified implementation of TLS in F# [13]. However, the generated models are typically quite large and whole-program symbolic verification does not scale very well.

An alternate approach is to develop a verification method directly for implementation code, using well-known program verification techniques such as typechecking. F7 [40] is a refinement typechecker for F#, developed jointly at Microsoft Research Cambridge and Inria. It implements a dependent type-system that allows us to specify security assumptions and goals as first-order logic annotations directly inside the program. It has been used for the modular verification of large web services security protocol implementations [41]. F* [53] is an extension of F7 with higher-order kinds and a certifying typechecker. Both F7 and F* have a growing user community. The cryptographic protocol implementations verified using F7 and F* already represent the largest verified cryptographic applications to our knowledge.

3.2. Computational verification of cryptographic applications

Proofs done by cryptographers in the computational model are mostly manual. Our goal is to provide computer support to build or verify these proofs. In order to reach this goal, we have already designed the automatic tool CryptoVerif, which generates proofs by sequences of games. Much work is still needed in order to develop this approach, so that it is applicable to more protocols. We also plan to design and implement techniques for proving implementations of protocols secure in the computational model, by generating them from CryptoVerif specifications that have been proved secure, or by automatically extracting CryptoVerif models from implementations.

An alternative approach is to directly verify cryptographic applications in the computational model by typing. A recent work [51] shows how to use refinement typechecking in F7 to prove computational security for protocol implementations. In this method, henceforth referred to as computational F7, typechecking is used as the main step to justify a classic game-hopping proof of computational security. The correctness of this method is based on a probabilistic semantics of F# programs and crucially relies on uses of type abstraction and parametricity to establish strong security properties, such as indistinguishability.

In principle, the two approaches, typechecking and game-based proofs, are complementary. Understanding how to combine these approaches remains an open and active topic of research.

3.3. Provably secure web applications

Web applications are fast becoming the dominant programming platform for new software, probably because they offer a quick and easy way for developers to deploy and sell their *app*s to a large number of customers. Third-party web-based apps for Facebook, Apple, and Google, already number in the hundreds of thousands and are likely to grow in number. Many of these applications store and manage private user data, such as health information, credit card data, and GPS locations. To protect this data, applications tend to use an ad hoc combination of cryptographic primitives and protocols. Since designing cryptographic applications is easy to get wrong even for experts, we believe this is an opportune moment to develop security libraries and verification techniques to help web application programmers.

As a typical example, consider commercial password managers, such as LastPass, RoboForm, and 1Password. They are implemented as browser-based web applications that, for a monthly fee, offer to store a user's passwords securely on the web and synchronize them across all of the user's computers and smartphones. The passwords are encrypted using a master password (known only to the user) and stored in the cloud. Hence, no-one except the user should ever be able to read her passwords. When the user visits a web page that has a login form, the password manager asks the user to decrypt her password for this website and automatically fills in the login form. Hence, the user no longer has to remember passwords (except her master password) and all her passwords are available on every computer she uses.

Password managers are available as browser extensions for mainstream browsers such as Firefox, Chrome, and Internet Explorer, and as downloadable apps for Android and Apple phones. So, seen as a distributed application, each password manager application consists of a web service (written in PHP or Java), some number of browser extensions (written in JavaScript), and some smartphone apps (written in Java or Objective C). Each of these components uses a different cryptographic library to encrypt and decrypt password data. How do we verify the correctness of all these components?

We propose three approaches. For client-side web applications and browser extensions written in JavaScript, we propose to build a static and dynamic program analysis framework to verify security invariants. For Android smartphone apps and web services written in Java, we propose to develop annotated JML cryptography libraries that can be used with static analysis tools like ESC/Java to verify the security of application code. For clients and web services written in F# for the .NET platform, we propose to use F7 to verify their correctness.

SECSI Project-Team

3. Scientific Foundations

3.1. Foundations

Computer security has become more and more pressing as a concern since the mid 1990s. There are several reasons to this: cryptography is no longer a *chasse réservée* of the military, and has become ubiquitous; and computer networks (e.g., the Internet) have grown considerably and have generated numerous opportunities for attacks and misbehaviors, notably.

The aim of the SECSI project is to *develop logic-based verification techniques for security properties of computer systems and networks*. Let us explain what this means, and what this does not mean.

First, the scope of the research at SECSI started as a rather broad subset of computer security, although the core of SECSI's activities has always been on verifying cryptographic protocols.

We took this for granted in 2006, and decided to concentrate on the latter. This already includes a vast number of concerns.

First, there is a plethora of distinct *security properties* one may wish to verify. Beyond the standard properties of secrecy (weak or strong forms), or authentication, one considers anonymity, fairness in contract-signing, and the subtle security properties involved in electronic voting such as accountability, receipt-freeness, resistance to coercion, or user verifiability. Some of these properties are trace properties, some are not, and are therefore more complex to state and verify.

Second, there are many available *models*. SECSI started with the rather simple symbolic models of security known today as Dolev-Yao models. One must then look at process algebra models (spi-calculus, applied pi-calculus), which allow for a symbolic treatment of more complex properties, especially those that are not trace properties. And one must also look at the computational models favored by cryptographers, e.g., the game-based approaches and the universal composability/simulatability approaches. They are more realistic in terms of security, but less directly amenable to automated verification. One of the features of computational models that makes them more complex is the need for computing, and bounding probabilities of certain events. This led us into contributing to the field of verification of probabilistic systems. One must also look at the relations between these models.

Third, there are many important *applications*. While SECSI started looking at the rather simple and now mundane confidentiality and authentication protocols, two important application domains have emerged: the verification of electronic voting protocols, and the verification of cryptographic APIs.

Apart from cryptographic protocols, the initial vision of the SECSI project was that computer security, being a global concern, should be taken as a whole, as far as possible. This is why one of the initial objectives of SECSI included topic in intrusion detection, again seen from the logical point of view.

One should remember the following. First, one of the key phrases in the SECSI motto is "logic-based". It is a founding theme of SECSI that logic matters in security, and opportunities are to be grabbed. Another key phrase is "verification techniques". The expertise of SECSI is not in designing protocols or security architectures. Verifying protocols, formally, is an arduous task already, and has proved to be an extremely rich area.

3.2. Objectives

SECSI has five objectives:

- Objective 1: symbolic verification of cryptographic protocols. Tree-automata based methods, automated deduction, and approximate/exact cryptographic protocol verification in the Dolev-Yao model. Enriching the Dolev-Yao model with algebraic theories, and associated decision problems.

- Objective 2: verification of cryptographic protocols in computational models. Computational soundness of formal models (Dolev-Yao, applied pi-calculus).
- Objective 3: security of group protocols, fair exchange, voting and other protocols. Other security properties, other security models. Security properties based on notions of indistinguishability.
- Objective 4: probabilistic transition systems. Security in the presence of probabilistic and demonic non-deterministic choices.
- Objective 5: intrusion detection, network and host protection in the large.

TASC Project-Team

3. Scientific Foundations

3.1. Overview

Basic research is guided by the challenges raised before: to classify and enrich the models, to automate reformulation and resolution, to dissociate declarative and procedural knowledge, to come up with theories and tools that can handle problems involving both continuous and discrete variables, to develop modelling tools and to come up with solving tools that scale well. On the one hand, **classification aspects** of this research are integrated within a knowledge base about combinatorial problem solving: the global constraint catalog (see http://www.emn.fr/x-info/sdemasse/gccat/index.html). On the other hand, **solving aspects** are capitalized within the constraint solving system CHOCO. Lastly, within the framework of its activities of valorisation, teaching and of partnership research, the team uses constraint programming for solving various concrete problems. The challenge is, on one side to increase the visibility of the constraints in the others disciplines of computer science, and on the other side to contribute to a broader diffusion of the constraint programming in the industry.

3.2. Fundamental Research Topics

This part presents the research topics investigated by the project:

- Global Constraints Classification, Reformulation and Filtering,
- Convergence between Discrete and Continuous,
- Dynamic, Interactive and over Constrained Problems,
- Solvers.

These research topics are in fact not independent. The work of the team thus frequently relates transverse aspects such as explained global constraints, Benders decomposition and explanations, flexible and dynamic constraints, linear models and relaxations of constraints.

3.2.1. Constraints Classification, Reformulation and Filtering

In this context our research is focused (a) first on identifying recurring combinatorial structures that can be used for modelling a large variety of optimization problems, and (b) exploit these combinatorial structures in order to come up with efficient algorithms in the different fields of optimization technology. The key idea for achieving point (b) is that many filtering algorithms both in the context of Constraint Programming, Mathematical Programming and Local Search can be interpreted as the maintenance of invariants on specific domains (e.g., graph, geometry). The systematic classification of global constraints and of their relaxation brings a synthetic view of the field. It establishes links between the properties of the concepts used to describe constraints and the properties of the constraints themselves. Together with SICS, the team develops and maintains *a catalog of global constraints*, which describes the semantics of more than 350 constraints, and proposes a unified mathematical model for expressing them. This model is based on graphs, automata and logic formulae and allows to derive filtering methods and automatic reformulation for each constraint in a unified way (see http://www.emn.fr/x-info/sdemasse/gccat/index.html). We consider hybrid methods (i.e., methods that involve more than one optimization technology such as constraint programming, mathematical programming or local search), to draw benefit from the respective advantages of the combined approaches. More fundamentally, the study of hybrid methods makes it possible to compare and connect strategies of resolution specific to each approach for then conceiving new strategies. Beside the works on classical, complete resolution techniques, we also investigate local search techniques from a mathematical point of view. These partly random algorithms have been proven very efficient in practice, although we have little theoretical knowledge on their behaviour, which often makes them problem-specific. Our research in that area is focused on a probabilistic model of

local search techniques, from which we want to derive quantified information on their behaviour, in order to use this information directly when designing the algorithms and exploit their performances better. We also consider algorithms that maintain local and global consistencies, for more specific models. Having in mind the trade off between genericity and effectiveness, the effort is put on the efficiency of the algorithms with guarantee on the produced levels of filtering. This effort results in adapting existing techniques of resolution such as graph algorithms. For this purpose we identify necessary conditions of feasibility that can be evaluated by efficient incremental algorithms. Genericity is not neglected in these approaches: on the one hand the constraints we focus on are applicable in many contexts (for example, graph partitioning constraints can be used both in logistics and in phylogeny); on the other hand, this work led to study the portability of such constraints and their independence with specific solvers. This research orientation gathers various work such as strong local consistencies, graph partitioning constraints, geometrical constraints, and optimization and soft constraints. Within the perspective to deal with complex industrial problems, we currently develop meta constraints (e.g. *geost*) handling all together the issues of large-scale problems, dynamic constraints, combination of spatial and temporal dimensions, expression of business rules.

3.2.2. Convergence between Discrete and Continuous

Many industrial problems mix continuous and discrete aspects that respectively correspond to physical (e.g., the position, the speed of an object) and logical (e.g., the identifier, the nature of an object) elements. Typical examples of problems are for instance:

- *Geometrical placement problems* where one has to place in space a set of objects subject to various geometrical constraints (i.e., non-overlapping, distance). In this context, even if the positions of the objects are continuous, the structure of optimal configurations has a discrete nature.
- *Trajectory and mission planning problems* where one has to plan and synchronize the moves of several teams in order to achieve some common goal (i.e., fire fighting, coordination of search in the context of rescue missions, surveillance missions of restricted or large areas).
- *Localization problems in mobile robotic* where a robot has to plan alone (only with its own sensors) its trajectory. This kind of problematic occurs in situations where the GPS cannot be used (e.g., under water or Mars exploration) or when it is not precise enough (e.g., indoor surveillance, observation of contaminated sites).

Beside numerical constraints that mix continuous and integer variables we also have global constraints that involve both type of variables. They typically correspond to graph problems (i.e., graph colouring, domination in a graph) where a graph is dynamically constructed with respect to geometrical and-or temporal constraints. In this context, the key challenge is avoiding decomposing the problem in a discrete and continuous parts as it is traditionally the case. As an illustrative example consider *the wireless network deployment problem*. On the one hand, the continuous part consists of finding out where to place a set of antenna subject to various geometrical constraints. On the other hand, by building an interference graph from the positions of the antenna, the discrete part consists of allocating frequencies to antenna in order to avoid interference. In the context of convergence between discrete and continuous variables, our goals are:

- First to identify and compare typical class of techniques that are used in the context of continuous and discrete solvers.
- To see how one can unify and/or generalize these techniques in order to handle in an integrated way continuous and discrete constraints within the same framework.

3.2.3. Dynamic, Interactive and over Constrained Problems

Some industrial applications are defined by a set of constraints which may change over time, for instance due to an interaction with the user. Many other industrial applications are over-constrained, that is, they are defined by set of constraints which are more or less important and cannot be all satisfied at the same time. Generic, dedicated and explanation-based techniques can be used to deal efficiently with such applications. Especially, these applications rely on the notion of *soft constraints* that are allowed to be (partially) violated. The generic concept that captures a wide variety of soft constraints is the violation measure, which is coupled with specific resolution techniques. Lastly, soft constraints allow to combinate the expressive power of global constraints with local search frameworks.

3.2.4. Solvers

Our theoretical work is systematically validated by concrete experimentations. We have in particular for that purpose the CHOCO constraint platform. The team develops and maintains CHOCO with the assistance of the laboratory e-lab of Bouygues (G. Rochart), the company Amadeus (F. Laburthe), and others researchers such as H. Cambazard (4C, INP Grenoble). The functionalities of CHOCO are gradually extended with the outcomes of our works: design of constraints, analysis and visualization of explanations, etc. The open source CHOCO library is downloaded on average 450 times each month since 2006. CHOCO is developed in line with the research direction of the team, in an open-minded scientific spirit. Contrarily to other solvers where the efficiency often relies on problem-specific algorithms, CHOCO aims at providing the users both with reusable techniques (based on an up-to-date implementation of the global constraint catalogue) and with a variety of tools to ease the use of these techniques (clear separation between model and resolution, event-based solver, management of the over-constrained problems, explanations, etc.). Since 2009 year, due to the hiring of G. Chabert, the team is also involved in the development of the continuous constraint solver IBEX. These developments led us to new research topics, suitable for the implementation of discrete and continuous constraint solving systems: portability of the constraints, management of explanations, incrementality and recalculation. They partially use aspect programming (in collaboration with the InriaASCOLA team). This work around the design and the development of solvers thus forms the fourth direction of basic research of the project.

TOCCATA Team

3. Scientific Foundations

3.1. Introduction

In the former *ProVal* project, we have been working on the design of methods and tools for deductive verification of programs. One of our originalities is our ability to conduct proofs by using automatic provers and proof assistants at the same time, depending on the difficulty of the program, and specifically the difficulty of each particular verification condition. We thus believe that we are in a good position to propose a bridge between the two families of approaches of deductive verification presented above. This is a new goal of the team: we want to provide methods and tools for deductive program verification that can offer both a high amount of proof automation and a high guarantee of validity. Toward this objective, a new axis of research that we propose is to develop certified tools: analysis tools that are themselves formally proved correct.

As mentioned above, some of the members of the team have an internationally recognized expertise on deductive program verification involving floating-point computation [5], including both interactive proving and automated solving [9]. Indeed we noticed that the verification of numerical programs is a representative case that can benefit a lot from combining automatic and interactive theorem proving [62][4]. This is why certification of numerical programs is another axis of this proposition.

The *Toccata* project emphasizes two new axes of research: certified tools and verification of numerical programs. Additionally we want to continue the fundamental studies we conducted in the past concerning deductive program verification in general. This is why our detailed scientific programme is structured into three themes:

1. Certified Programs,
2. Certified Tools,
3. Certified Numerical Programs.

The reader should be aware that the word “Certified” in this scientific programme means “verified by a formal specification and a formal proof that the program meets this specification”. This differs from the standard meaning of “Certified” in an industrial context which is that it conforms to a rigorous process and/or a norm.

3.2. Certified Programs

This theme of research builds upon our expertise on the development of methods and tools for proving programs, from source codes annotated with specifications to proofs. In the past years, we tackled programs written in mainstream programming languages, with the system *Why3* and the front-ends *Krakatoa* for Java source code, and Frama-C/Jessie for C code. However, Java and C programming languages were designed a long time ago, and certainly not with the objective of formal verification in mind. This raises a lot of difficulties when designing specification languages on top of them, and verification condition generators to analyze them. On the other hand, we designed and/or used the *Coq* and *Why3* languages and tools for performing deductive verification, but those were not designed as programming languages that can be compiled into executable programs.

Thus, a new axis of research we propose is the design of an environment that is aimed to both programming and proving, hence that will allow to develop certified programs. To achieve this goal, there are two major axes of theoretical research that needs to be conducted, concerning on the one hand methods required to support genericity and reusability of certified components, and on the other hand the automation of the proof of the verification conditions that will be generated.

3.2.1. Genericity and Reusability of Certified Components

A central ingredient for the success of deductive approaches in program verification is the ability to reuse components that are already proved. This is the only way to scale the deductive approach up to programs of larger size. As for programming languages, a key aspect that allow reusability is *genericity*. In programming languages, genericity typically means parametricity with respect to data types, e.g. *polymorphic types* in functional languages like ML, or *generic classes* in object-oriented languages. Such genericity features are essential for the design of standard libraries of data structures such as search trees, hash tables, etc. or libraries of standard algorithms such as for searching, sorting.

In the context of deductive program verification, designing reusable libraries needs also the design of *generic specifications* which typically involve parametricity not only with respect to data types but also with respect to other program components. For example, a generic component for sorting an array needs to be parametrized by the type of data in the array but also by the comparison function that will be used. This comparison function is thus another program component that is a parameter of the sorting component. For this parametric component, one needs to specify some requirements, at the logical level (such as being a total ordering relation), but also at the program execution level (like being *side-effect free*, i.e. comparing of data should not modify the data). Typically such a specification may require *higher-order* logic.

Another central feature that is needed to design libraries of data structures is the notion of data invariants. For example, for a component providing generic search trees of reasonable efficiency, one would require the trees to remain well-balanced, over all the life time of a program.

This is why the design of reusable certified components requires advanced features, such as *higher-order specifications and programs*, *effect polymorphism* and *specification of data invariants*. Combining such features is considered as an important challenge in the current state of the art (see e.g. [90]). The well-known proposals for solving it include *Separation logic* [107], *implicit dynamic frames* [105], and *considerate reasoning* [106]. Part of our recent research activities were aimed at solving this challenge: first at the level of specifications, e.g. we proposed generic specification constructs upon Java [108] or a system of theory cloning in our system *Why3* [1]; second at the level of programs, which mainly aims at controlling side-effects to avoid unexpected breaking of data invariants, thanks to advanced type checking : approaches based on *memory regions*, *linearity* and *capability-based* type systems [69], [88], [50].

A concrete challenge that should be solved in the future is: what additional constructions should we provide in a specification language like ACSL for C, in order to support modular development of reusable software components? In particular, what would be an adequate notion of module, that would provide a good notion of abstraction, both at the level of program components and at the level of specification components.

3.2.2. Automated Deduction for Program Verification

Verifying that a program meets formal specifications typically amounts to generate *verification conditions* e.g. using a weakest precondition calculus. These verification conditions are purely logical formulas—typically in first-order logic and involving arithmetic in integers or real numbers—that should be checked to be true. This can be done using either automatic provers or interactive proof assistants. Automatic provers do not need user interaction, but may run forever or give no conclusive answer.

There are several important issues to tackle. Of course, the main general objective is to improve automation as much as possible. We want to continue our efforts around our own automatic prover *Alt-Ergo* towards more expressivity, efficiency, and usability, in the context of program verification. More expressivity means that the prover should better support the various theories that we use for modeling. Toward this direction, we aim at designing specialized proof search strategies in *Alt-Ergo*, directed by rewriting rules, in the spirit of what we did for the theory of associativity and commutativity [6].

A key challenge is also in the better handling of quantifiers. SMT solvers, including *Alt-Ergo*, deal with quantifiers with a somewhat ad-hoc mechanism of heuristic instantiation of quantified hypotheses using the so-called *triggers* that can be given by hand [42], [32]. This is completely different from resolution-based provers of the TPTP category (E-prover, Vampire, etc.) which use unification to apply quantified premises.

A challenge is thus to find the best way to combine these two different approaches of quantifiers. Another challenge is to add some support for higher-order functions and predicates in this SMT context, since as said above, reusable certified components will require higher-order specifications. There are a few solutions that were proposed yet, that amount to encode higher-order goals in first-order ones [88].

Generally speaking, there are several theories, interesting for program verification, that we would like to add as built-in decision procedures in a SMT context. First, although there already exist decision procedures for variants of bit-vectors, these are not complete enough to support what is needed to reason on programs that manipulate data at the bit-level, in particular if conversions from bit-vectors to integers or floating-point numbers are involved [12]. Regarding floating-point numbers, an important challenge is to integrate in an SMT context a decision procedure like the one implemented in our tool *Gappa*.

Another goal is to improve the feedback given by automatic provers: failed proof attempts should be turned into potential counterexamples, so as to help debugging programs or specifications. A pragmatic goal would be to allow cooperation with other verification techniques. For instance, testing could be performed on unproved goals. Regarding this cooperation objective, an important goal is a deeper integration of automated procedures in interactive proofs, like it already exists in Isabelle [68]. We now have a prototype for a *Why3* tactic in *Coq* that we plan to improve.

3.2.3. An environment for both programming and proving

As said before, a new axis of research we would like to follow is to design a language and an environment for both programming and proving. We believe that this will be a fruitful approach for designing highly trustable software. This is a similar goal as projects Plaid, Trellys, ATS, or Guru, mentioned above.

The basis of this research direction is the *Why3* system, which is in fact a reimplementation from scratch of the former *Why* tool, that we started in January 2011. This new system supports our research at various levels. It is already used as an intermediate language for deductive verification.

The next step for us is to develop its use as a true programming language. Our objective is to propose a language where programs could be both executed (e.g. thanks to a compiler to, say, *OCaml*) and proved correct. The language would basically be purely applicative (i.e. without side-effects, e.g. close to ML) but incorporating specifications in its core. There are, however, some programs (e.g. some clever algorithms) where a bit of imperative programming is desirable. Thus, we want to allow some form of imperative features, but in a very controlled way: it should provide a strict form of imperative programming that is clearly more amenable to proof, in particular dealing with data invariants on complex data structures.

As already said before, reusability is a key issue. Our language should propose some form of modules with interfaces abstracting away implementation details. Our plan is to reuse the known ideas of *data refinement* [99] that was the foundation of the success of the B method. But our language will be less constrained than what is usually the case in such a context, in particular regarding the possibility of sharing data, and the constraints on composition of modules, there will be a need for advanced type systems like those based on regions and permissions.

The development of such a language will be the basis of the new theme regarding the development of certified tools, that is detailed in Section 3.3 below.

3.2.4. Extra Exploratory Axes of Research

In this theme concerning certified programs, there are a few extra exploratory topics that we plan to explore.

Concurrent Programming So far, we only investigated the verification of sequential programs. However, given the spreading of multi-core architectures nowadays, it becomes important to be able to verify concurrent programs. This is known to be a major challenge. We plan to investigate in this direction, but in a very careful way. We believe that the verification of concurrent programs should be done only under restrictive conditions on the possible interleaving of processes. In particular, the access and modification of shared data should be constrained by the programming paradigm, to allow reasonable formal specifications. In this matter, the issues are close to the ones about sharing data between components in sequential programs, and there are already some successful approaches like separation logic, dynamic frames, regions, and permissions.

Resource Analysis The deductive verification approaches are not necessarily limited to functional behavior of programs. For example, a formal termination proof typically provides a bound on the time complexity of the execution. Thus, it is potentially possible to verify resources consumption in this way, e.g. we could prove WCET (Worst Case Execution Times) of programs. Nowadays, WCET analysis is typically performed by abstract interpretation, and is applied on programs with particular shape (e.g. no unbounded iteration, no recursion). Applying deductive verification techniques in this context could allow to establish good bounds on WCET for more general cases of programs.

Other Programming Paradigms We are interested in the application of deductive methods in other cases than imperative programming à la C, Java or Ada. Indeed, in the recent years, we applied proof techniques to randomized programs [45], to cryptographic programs [49]. We plan to use proof techniques on applications related to databases. We also have plans to support low-level programs such as assembly code [81], [102] and other unstructured programming paradigm.

We are also investigating more and more applications of SMT solving, e.g. in model-checking approach (for example in Cubicle [1] [24]) or abstract interpretation techniques (new project Cafein that will start in 2013) and also for discharging proof obligations coming from other systems like Atelier B [29] (new project BWare, Section 8.2.1).

3.3. Certified Tools

The goal of this theme is to guarantee the soundness of the tools we develop. Indeed it goes beyond that: our goal is to promote our future *Why3* environment so that *others* could develop certified tools. Tools like automated provers or program analysers are good candidate case studies because they are mainly performing symbolic computations, and as such they are usually programmed in a mostly purely functional style.

We conducted several experiments of development of certified software in the past. First, we have a strong expertise in the development of *libraries* in *Coq*: the Coccinelle library [74] formalizing term rewriting systems, the Alea library [45] for the formalization of randomized algorithms, several libraries formalizing floating-point numbers (Floats [58], Gappalib [97], and now Flocq [5] which unifies the formers). Second we recently conducted the development of a certified decision procedure [93] that corresponds to a core part of *Alt-Ergo*, and a certified verification condition generator for a language [28] similar to *Why*. On-going work aims at building, still in *Coq*, a certified VC generator for C annotated in ACSL [55], based on the operational semantics formalized in the CompCert certified compiler project [92].

To go further, we have several directions of research in mind.

3.3.1. Formalization of Binders

Using the *Why3* programming language instead of *Coq* allows more freedom. For example, it should allow one to use a bit of side-effects when the underlying algorithm justify it (e.g. hash-consing, destructive unification). On the other hand, we will lose some *Coq* features like dependent types that are usually useful when formalizing languages. Among the issues that should be studied, we believe that the question of the formalization of binders is both central and challenging (as exemplified by the POPLmark international challenge [47]).

The support of binders in *Why3* should not be built-in, but should be under the form of a reusable *Why3* library, that should already contain a lot of proved lemmas regarding substitution, alpha-equivalence and such. Of course we plan to build upon the former experiments done for the POPLmark challenge. Although, it is not clear yet that the support of binders only via a library will be satisfactory. We may consider addition of built-in constructs if this shows useful. This could be a form of (restricted) dependent types as in *Coq*, or subset types as in PVS.

3.3.2. Theory Realizations, Certification of Transformations

As an environment for both programming and proving, *Why3* should come with a standard library that includes both certified libraries of programs, but also libraries of specifications (e.g. theories of sets, maps, etc.).

[1] http://cubicle.lri.fr/

The certification of those *Why3* libraries of specifications should be addressed too. *Why3* libraries for specifying models of programs are commonly expressed using first-order axiomatizations, which have the advantage of being understood by many different provers. However, such style of formalization does not offer strong guarantees of consistency. More generally, the fact that we are calling different kind of provers to discharge our verification conditions raises several challenges for certification: we typically apply various transformations to go from the *Why3* language to those of the provers, and these transformations should be certified too.

A first attempt in considering such an issue was done in [29]. It was proposed to certify the consistency of a library of specification using a so-called *realization*, which amounts to "implementing" the library in a proof assistant like *Coq*. This will be an important topic of the new ANR project BWare.

3.3.3. Certified Theorem Proving

The goal is to develop *certified* provers, in the sense that they are proved to give a correct answer. This is an important challenge since there have been a significant amount of soundness bugs discovered in the past, in many tools of this kind.

The former work on the certified core of *Alt-Ergo* [93] should be continued to support more features: more theories (full integer arithmetic, real arithmetic, arrays, etc.), quantifiers. Development of a certified prover that supports quantifiers should build upon the previous topic about binders.

In a similar way, the *Gappa* prover which is specialized to solving constraints on real numbers and floating-point numbers should be certified too. Currently, *Gappa* can be asked to produce a *Coq* proof of its given goal, so as to check *a posteriori* its soundness. Indeed, the idea of producing a trace is not contradictory with certifying the tool. For very complex decision procedures, the goal of developing a certified proof search might be too ambitious, and the production of an internal trace is a general technique that might be used as a workaround: it suffices to instrument the proof search and to develop a certified trace checker to be used by the tool before it gives an answer. We used this approach in the past for certified proofs of termination of rewriting systems [75]. This is also a technique that is used internally in CompCert for some passes of compilation [92].

3.3.4. Certified VC generation

The other kind of tools that we would like to certify are the VC generators. This will be a continuation of the on-going work on developing in *Coq* a certified VC generator for C code annotated in ACSL. We would like to develop such a generator in *Why3* instead of *Coq*. As before, this will build upon a formalization of binders.

There are various kinds of VC generators that are interesting. A generator for a simple language in the style of those of *Why3* is a first step. Other interesting cases are: a generator implementing the so-called *fast weakest preconditions* [91], and a generator for unstructured programs like assembly, that would operate on an arbitrary control-flow graph.

On a longer term, it would be interesting to be able to certify advanced verification methods like those involving refinement, alias control, regions, permissions, etc.

An interesting question is how one could certify a VC generator that involves a highly expressive logic, like higher-order logic, as it is the case of the *CFML* method [70] which allows one to use the whole *Coq* language to specify the expected behavior. One challenging aspect of such a certification is that a tool that produces *Coq* definitions, including inductive definitions and module definitions, cannot be directly proved correct in *Coq*, because inductive definitions and module definitions are not first-class objects in *Coq*. Therefore, it seems necessary to involve, in a way or another, a "deep embedding", that is, a formalization of *Coq* in *Coq*, possibly by reusing the deep embedding developed by B. Barras [52].

3.4. Certified Numerical Programs

In recent years, we demonstrated our capability towards specifying and proving properties of floating-point programs, properties which are both complex and precise about the behavior of those programs: see the

publications [65], [109], [61], [104], [64], [59], [98], [96] but also the web galleries of certified programs at our Web page [2], the Hisseo project [3], S. Boldo's page [4], and industrial case studies in the U3CAT ANR project. The ability to express such complex properties comes from models developed in *Coq* [5]. The ability to combine proof by reasoning and proof by computation is a key aspect when dealing with floating-point programs. Such a modeling provides a safe basis when dealing with C source code [4]. However, the proofs can get difficult even on short programs, and to achieve them some automation is needed, and obtained by combining SMT solvers and *Gappa* [62], [79], [46][9]. Finally, the precision of the verification is obtained thanks to precise models of floating-point computations, taking into account the peculiarities of the architecture (e.g. x87 80-bit floating-point unit) and also the compiler optimizations [66], [102].

The directions of research concerning floating-point programs that we want to pursue are the following.

3.4.1. Making Formal Verification of Floating-point Programs Easier

A first goal is to ease the formal verification of floating-point programs: the primary objective is still to improve the scope and efficiency of our methods, so as to ease further the verification of numerical programs. The on-going development of the Flocq library should be continued towards the formalization of bit-level manipulations and also of exceptional values (e.g. infinities). We believe that good candidates for applications of our techniques are smart algorithms to compute efficiently with floats, which operate at the bit-level. The formalization of real numbers need to be revamped too: higher-level numerical algorithms are usually built on some mathematical properties (e.g. computable approximations of ideal approximations), which then have to be proved during the formal verification of these algorithms.

Easing the verification of numerical programs also implies more automation. SMT solvers are generic provers well-suited for automatically discharging verification conditions, but they tend to be confused by floating-point arithmetic [31]. Our goal is to improve the arithmetic theories of *Alt-Ergo*, so that they support floating-point arithmetic along their other theories, if possible by reusing the heuristics developed for *Gappa*.

3.4.2. Continuous Quantities, Numerical Analysis

The goal is to handle floating-point programs that are related to continuous quantities. This includes numerical analysis programs we have already worked on [14] [61][3]. But our work is only a beginning: we were able to solve the difficulties to prove one particular scheme for one particular partial differential equation. We need to be able to easily prove this kind of programs. This requires new results that handle generic schemes and many partial differential equations. The idea is to design a toolbox to prove these programs with as much automation as possible. We wish this could be used by numerical analysts that are not or hardly familiar with formal methods, but are interested in the formal correctness of their schemes and their programs.

Another very interesting kind of programs (especially for industrial developers) are those based on *hybrid* systems, that is where both discrete and continuous quantities are involved. This is a longer term goal than four years, but we may try to go towards this direction. A first problem is to be able to specify hybrid systems: what are they exactly expected to do? Correctness usually means not going into a forbidden state but we may want additional behavioral properties. A second problem is the interface with continuous systems, such as sensors. How can we describe their behavior? Can we be sure that the formal specification fits? We may think about Ariane V where one piece of code was shamelessly reused from Ariane IV. Ensuring that such a reuse is allowed requires to correctly specify the input ranges and bandwidths of physical sensors.

Studying hybrid systems is among the goals of the new ANR project Cafein.

3.4.3. Certification of Floating-point Analyses

In coordination with our second theme, another objective is to port the kernel of *Gappa* into either *Coq* or *Why3*, and then extract a certified executable. Rather than verifying the results of the tool *a posteriori* with a proof checker, they would then be certified *a priori*. This would simplify the inner workings of *Gappa*, help to

[2] http://toccata.lri.fr/gallery/index.en.html
[3] http://hisseo.saclay.inria.fr/
[4] http://www.lri.fr/~sboldo/research.html

support new features (e.g. linear arithmetic, elementary functions), and make it scale better to larger formulas, since the tool would no longer need to carry certificates along its computations. Overall the tool would then be able to tackle a wider range of verification conditions.

An ultimate goal would be to develop the decision procedure for floating-point computations, for SMT context, that is mentioned in Section refsec:atp, directly as a certified program in *Coq* or *Why3*.

TYPICAL Project-Team

3. Scientific Foundations

3.1. Logical formalisms

A proof system implements a logical formalism in the way a compiler implements a programming language. Similarly, the choice of the formalism is crucial for the success of the proof system. One of the main line of research of the team is to study or invent type theories that are well-adapted to the formalization of mathematics. For instance a crucial property of a proof system is its correctness, hence the importance of the study of the models of the meta-theory of the Coq proof assistant. An other issue is the interoperability of the various proof systems used to formalize mathematics in the world-wide community of users of proof assistants, and the design of a system which could serve as a back-end to front-end implementing various formalisms and proof languages.

3.2. Libraries of formalized mathematics

It is well known that advanced mathematics can play a crucial role in the design and correctness of sophisticated and sometimes critical software. In some cases, using a proof system is the only option to mechanize the correctness of such programs; this can require the formalization of a wide variety of mathematical theories, and a careful design of these formal libraries for them to be maintainable, combinable and reusable. Furthermore, the ability to formalize advanced contemporary mathematics is still a form of ultimate quality tests for proof systems, and also a way to gain visibility. One of our objectives is to make modern and large pieces of mathematics available as usable formal libraries. Recent examples of complex proofs (Four Color Theorem, Kepler conjecture, classification of finite groups, Fermat theorem) challenge the way the mathematical literature is refereed and published. We think that the development of these formal libraries of mathematics may also change the way certain mathematical result become accepted as theorems. Crafting large bodies of formalized mathematics is a challenging task. These libraries obey similar requirements as software : modularity and usability stem from appropriate data-structures, design patterns and corpus of lemmas. But the appropriate methodology leading to the relevant solutions is often far from obvious, and this is where research has to be done and know-how has to be gained. Up to recently, formal developments were seldom collaborative and rarely benefitted from reusable previous work. The maturity of proof assistants is now sufficient to envision a more modern conception of formal software, as required by large scale verification projects like T. Hales' proof of the Kepler conjecture or the Feit-Thompson theorem. Several members of the TypiCal team are committed in such big formalization projects, or in more specific but related side projects.

3.3. Proof search and automated decision procedures

Interactive proof assistants provide a very expressive logical formalism, rich enough to allow extremely precise descriptions of complex objects like the meta theory of a programming language, a model of C compiler, or the proof of the Four Color Theorem. This description includes logical statements of the properties required by the objects of interest but also their formal proofs, checked by the merciless proof-checker of the system, which should be a small hence trusted piece of code. These systems provide the highest formal guarantee, for instance, of the correctness with respect to the mathematical specification of a code.

Proof-search is a central issue in such a formalization of mathematics. It is also a common aspect of automated reasoning and high-level programming paradigms such as Logic programming. However specific applications commonly involve specific logics or theories, like for instance linear arithmetic. Whether or not such a logical framework can express these at all, it is unlikely that its generic proof-search mechanisms can replace the methods that are specific to a logic or theory. Either because this specific domain lies outside the reach of generic proof-search or simply because generic proof-search is less efficient therein than a purpose-made procedure (typically a decision procedure).

But to enlarge the scope where a specific method applies, one can combine both generic proof search mechanisms with specific methods. We hence investigate how to craft formal proof producing decision procedures in the context of an interactive proof assistant. This activity includes understanding the impact of proof-search mechanism (polarization, focusing, etc.), the implementation of efficient connections between domain specific automated decision procedures (SMT solvers, polynomial optimization tools, etc.) with a proof assistant, and the combination of these two aspects in the design a unique logical framework where a generic notion of proof-search could serve each of the above purposes.

VERIDIS Project-Team

3. Scientific Foundations

3.1. Automated and interactive theorem proving

The VeriDis team unites experts in techniques and tools for interactive and automated verification, and specialists in methods and formalisms for the proved development of concurrent and distributed systems and algorithms. Our common objective is to advance the state of the art of combining interactive with automated methods resulting in powerful tools for the (semi-)automatic verification of distributed systems and protocols. Our techniques and tools will support methods for the formal development of trustworthy distributed systems that are grounded in mathematically precise semantics and that scale to algorithms relevant for practical applications.

The VeriDis members from Saarbrücken are developing Spass [7], one of the leading automated theorem provers for first-order logic based on the superposition calculus [33]. Recent extensions to the system include the integration of dedicated reasoning procedures for specific theories, such as linear arithmetic [44], [31], that are ubiquitous in the verification of systems and algorithms. The group also studies general frameworks for the combination of theories such as the locality principle [45] and automated reasoning mechanisms these induce.

The VeriDis members from Nancy develop veriT [1], an SMT (Satisfiability Modulo Theories [35]) solver that combines decision procedures for different fragments of first-order logic and that integrates an automatic theorem prover for full first-order logic. The veriT solver is designed to produce detailed proofs; this makes it particularly suitable as a component of a robust cooperation of deduction tools.

We rely on interactive theorem provers for reasoning about specifications at a high level of abstraction. Members of VeriDis have ample experience in the specification and subsequent machine-assisted, interactive verification of algorithms. In particular, we participate in a project at the joint MSR-Inria Centre in Saclay on the development of methods and tools for the formal proof of TLA$^+$ [41] specifications. Our prover relies on a declarative proof language and includes several automatic backends [3].

3.2. Methodology of proved system development

Powerful theorem provers are not a panacea for system verification: they support sound methodologies for modeling and verifying systems. In this respect, members of VeriDis have gained expertise and recognition in making contributions to formal methods for concurrent and distributed algorithms and systems [2], [6], and in applying them to concrete use cases. In particular, the concept of *refinement* [30], [34], [43] in state-based modeling formalisms is central to our approach. Its basic idea is to derive an algorithm or implementation by providing a series of models, starting from a high-level description that precisely states the problem, and gradually adding details in intermediate models. An important goal in designing such methods is to reduce the number of generated proof obligations and/or support their proof by automatic tools. This requires taking into account specific characteristics of certain classes of systems and tailoring the model to concrete computational models. Our research in this area is supported by carrying out case studies for academic and industrial developments. This activity benefits from and influences the development of our proof tools.

Our vision for the integration of our expertise can be resumed as follows. Based on our experience and related work on specification languages, logical frameworks, and automatic theorem proving tools, we develop an approach that is suited for specification, interactive theorem proving, and for eventual automated analysis and verification, possibly through appropriate translation methods. While specifications are developed by users inside our framework, they are analyzed for errors by our SMT based verification tools. Eventually, properties are proved by a combination of interactive and automatic theorem proving tools, potentially again with support of SMT procedures for specific sub-problems, or with the help of interactive proof guidance.

Today, the formal verification of a new algorithm is typically the subject of a PhD thesis, if it is addressed at all. This situation is not sustainable given the move towards more and more parallelism in mainstream systems: algorithm developers and system designers must be able to productively use verification tools for validating their algorithms and implementations. On a high level, the goal of VeriDis is to make formal verification standard practice for the development of distributed algorithms and systems, just as symbolic model checking has become commonplace in the development of embedded systems and as security analysis for cryptographic protocols is becoming standard practice today. Although the fundamental problems in distributed programming, such as mutual exclusion, leader election, group membership or consensus, are well-known, they pose new challenges in the context of current system paradigms, including ad-hoc and overlay networks or peer-to-peer systems.