RESEARCH CENTER

FIELD
Algorithmics, Programming, Software and Architecture

Activity Report 2014

Section Scientific Foundations

Edition: 2015-03-24

ARIC Project-Team

3. Research Program

3.1. Lattice-based cryptography

Lattice-based cryptography (LBC) is an utterly promising, attractive (and competitive) research ground in cryptography, thanks to a combination of unmatched properties:

- **Improved performance.** LBC primitives have low asymptotic costs, but remain cumbersome in practice (e.g., for parameters achieving security against computations of up to 2100 bit operations). To address this limitation, a whole branch of LBC has evolved where security relies on the restriction of lattice problems to a family of more structured lattices called *ideal lattices*. Primitives based on such lattices can have quasi-optimal costs (i.e., quasi-constant amortized complexities), outperforming all contemporary primitives. This asymptotic performance sometimes translates into practice, as exemplified by NTRUEncrypt.
- **Improved security.** First, lattice problems seem to remain hard even for quantum computers. Moreover, the security of most of LBC holds under the assumption that standard lattice problems are hard in the worst case. Oppositely, contemporary cryptography assumes that specific problems are hard with high probability, for some precise input distributions. Many of these problems were artificially introduced for serving as a security foundation of new primitives.
- **Improved flexibility.** The master primitives (encryption, signature) can all be realized based on worst-case (ideal) lattice assumptions. More evolved primitives such as ID-based encryption (where the public key of a recipient can be publicly derived from its identity) and group signatures, that were the playing-ground of pairing-based cryptography (a subfield of elliptic curve cryptography), can also be realized in the LBC framework, although less efficiently and with restricted security properties. More intriguingly, lattices have enabled long-wished-for primitives. The most notable example is homomorphic encryption, enabling computations on encrypted data. It is the appropriate tool to securely outsource computations, and will help overcome the privacy concerns that are slowing down the rise of the cloud.

We will work on three directions, detailed now.

3.1.1. Lattice algorithms

All known lattice reduction algorithms follow the same design principle: perform a sequence of small elementary steps transforming a current basis of the input lattice, where these steps are driven by the Gram-Schmidt orthogonalisation of the current basis.

In the short term, we will fully exploit this paradigm, and hopefully lower the cost of reduction algorithms with respect to the lattice dimension. We aim at asymptotically fast algorithms with complexity bounds closer to those of basic and normal form problems (matrix multiplication, Hermite normal form). In the same vein, we plan to investigate the parallelism potential of these algorithms.

Our long term goal is to go beyond the current design paradigm, to reach better trade-offs between run-time and shortness of the output bases. To reach this objective, we first plan to strengthen our understanding of the interplay between lattice reduction and numerical linear algebra (how far can we push the idea of working on approximations of a basis?), to assess the necessity of using the Gram-Schmidt orthogonalisation (e.g., to obtain a weakening of LLL-reduction that would work up to some stage, and save computations), and to determine whether working on generating sets can lead to more efficient algorithms than manipulating bases. We will also study algorithms for finding shortest non-zero vectors in lattices, and in particular look for quantum accelerations.

We will implement and distribute all algorithmic improvements, e.g., within the fplll library. We are interested in high performance lattice reduction computations (see application domains below), in particular in connection/continuation with the HPAC ANR project (algebraic computing and high performance consortium).

3.1.2. Lattice-based cryptography

Our long term goal is to demonstrate the superiority of lattice-based cryptography over contemporary public-key cryptographic approaches. For this, we will 1- Strengthen its security foundations, 2- Drastically improve the performance of its primitives, and 3- Show that lattices allow to devise advanced and elaborate primitives.

The practical security foundations will be strengthened by the improved understanding of the limits of lattice reduction algorithms (see last section). On the theoretical side, we plan to attack two major open problems: Are ideal lattices (lattices corresponding to ideals in rings of integers of number fields) computationally as hard to handle as arbitrary lattices? What is the quantum hardness of lattice problems?

Lattice-based primitives involve two types of operations: sampling from discrete Gaussian distributions (with lattice supports), and arithmetic in polynomial rings such as $(\mathbb{Z}/q\mathbb{Z})[x]/(x^n+1)$ with n a power of 2. When such polynomials are used (which is the case in all primitives that have the potential to be practical), then the underlying algorithmic problem that is assumed hard involves ideal lattices. This is why it is crucial to precisely understand the hardness of lattice problems for this family. We will work on improving both types of operations, both in software and in hardware, concentrating on values of q and n providing security. As these problems are very arithmetic in nature, this will naturally be a source of collaboration with the other Themes of the ARIC team.

Our main objective in terms of cryptographic functionality will be to determine the extent to which lattices can help securing cloud services. For example, is there a way for users to delegate computations on their outsourced dataset while minimizing what the server eventually learns about their data? Can servers compute on encrypted data in an efficiently verifiable manner? Can users retrieve their files and query remote databases anonymously provided they hold appropriate credentials? Lattice-based cryptography is the only approach so far that has allowed to make progress into those directions. We will investigate the practicality of the current constructions, the extension of their properties, and the design of more powerful primitives, such as functional encryption (allowing the recipient to learn only a function of the plaintext message). To achieve these goals, we will in particular focus on cryptographic multilinear maps.

This research axis of ARIC is gaining strength thanks to the recruitment of Benoit Libert. We will be particularly interested in the practical and operational impacts, and for this reason we envision a collaboration with an industrial partner.

3.1.3. Application domains

- Diophantine equations. Lattice reduction algorithms can be used to solve diophantine equations, and in particular to find simultaneous rational approximations to real numbers. We plan to investigate the interplay between this algorithmic task, the task of finding integer relations between real numbers, and lattice reduction. A related question is to devise LLL-reduction algorithms that exploit specific shapes of input bases. This will be done within the ANR DynA3S project.
- Communications. We will continue our collaboration with Cong Ling on the use of lattices in communications. We plan to work on the wiretap channel over a fading channel (modeling cell phone communications in a fast moving environment). The current approaches rely on ideal lattices, and we hope to be able to find new approaches thanks to our expertise on them due to their use in lattice-based cryptography. We will also tackle the problem of sampling vectors from Gaussian distributions with lattice support, for a very small standard deviation parameter. This would significantly improve current schemes for communication schemes based on lattices, as well as several cryptographic primitives.
- Cryptanalysis of variants of RSA. Lattices have been used extensively to break variants of the RSA encryption scheme, via Coppersmith's method to find small roots of polynomials. We plan to work with Nadia Heninger (U. of Pennsylvania) on improving these attacks, to make them more practical.

This is an excellent test case for testing the practicality of LLL-type algorithm. Nadia Heninger has a strong experience in large scale cryptanalysis based on Coppersmith's method (http://smartfacts.cr.yp.to/)

3.2. Efficient approximation methods

3.2.1. Computer algebra generation of certified approximations.

We plan to focus on the generation of certified and efficient approximations for solutions of linear differential equations. These functions cover many classical mathematical functions and many more can be built by combining them. One classical target area is the numerical evaluation of elementary or special functions. This is currently performed by code specifically handcrafted for each function. The computation of approximations and the error analysis are major steps of this process that we want to automate, in order to reduce the probability of errors, to allow one to implement "rare functions", to quickly adapt a function library to a new context: new processor, new requirements – either in terms of speed or accuracy.

In order to significantly extend the current range of functions under consideration, several methods originating from approximation theory have to be considered (divergent asymptotic expansions; Chebyshev or generalized Fourier expansions; Padé approximants; fixed point iterations for integral operators). We have done preliminary work on some of them. Our plan is to revisit them all from the points of view of effectivity, computational complexity (exploiting linear differential equations to obtain efficient algorithms), as well as in their ability to produce provable error bounds. This work is to constitute a major progress towards the automatic generation of code for moderate or arbitrary precision evaluation with good efficiency. Other useful, if not critical, applications are certified quadrature, the determination of certified trajectories of spatial objects and many more important questions in optimal control theory.

3.2.2. Digital Signal Processing.

As computer arithmeticians, a wide and important target for us is the design of efficient and certified linear filters in digital signal processing (DSP). Actually, following the advent of Matlab as the major tool for filter design, the DSP experts now systematically delegate to Matlab all the part of the design related to numerical issues. And yet, various key Matlab routines are neither optimized, nor certified. Therefore, there is a lot of room for enhancing numerous DSP numerical implementations and there exist several promising approaches to do so.

The first important challenge that we want to address is the development and the implementation of optimal methods for rounding the coefficients involved in the design of the filter. If done in a naive way, this rounding may lead to a significant loss of performance. We will study in particular FIR and IIR filters.

3.2.3. Table Maker's Dilemma (TMD).

There is a clear demand for hardest-to-round cases, and several computer manufacturers recently contacted us to obtain new cases. These hardest-to-tound cases are a precious help for building libraries of correctly rounded mathematical functions. The current code, based on Lefèvre algorithm, will be rewritten and formal proofs will be done. We plan to use uniform polynomial approximation and diophantine techniques in order to tackle the case of the IEEE quad precision and analytic number theory techniques (exponential sums estimates) for counting the hardest-to-round cases.

3.3. High-performance reliable kernels

The main theme here is the study of fundamental operations ("kernels") on a hierarchy of symbolic or numeric data types spanning integers, floating-point numbers, polynomials, power series, as well as matrices of all these. Fundamental operations include basic arithmetic (e.g., how to multiply or how to invert) common to all such data, as well as more specific ones (change of representation/conversions, GCDs, determinants, etc.). For such operations, which are ubiquitous and at the very core of computing (be it numerical, symbolic, or hybrid numeric-symbolic), our goal is to ensure both high-performance and reliability.

3.3.1. Algorithmic design and analysis of symbolic or numerical algorithms.

On the symbolic side, we have so far obtained fast algorithms for basic operations on both polynomial matrices and structured matrices, but in a rather independent way. Both types turn out to have much in common, but this is sometimes not reflected by the complexities obtained, especially for applications in cryptology and coding theory. Our long term goal in this area is thus to explore these connections further, to provide a more unified treatment and bridge these complexity gaps, and to produce associated efficient implementations. A first step towards this goal will be the design and implementation of enhanced algorithms for various generalizations of Hermite-Padé approximation; in the context of list decoding, this should in particular make it possible to improve over the structured-matrix approach, which is so far the fastest known.

On the numerical side, we will continue to revisit and improve the classical error bounds of numerical analysis in the light of all the subtleties of IEEE floating-point arithmetic. These aspects will be developed jointly with the "symbolic floating-point" approach presented in the next paragraph. A complementary approach will also be studied, based on the estimation (possibly via automatic differentiation) of condition numbers in order to identify inputs leading to large backward errors. Finally, concerning interval arithmetic, a thorough analysis of the accuracy of several representations, such as mid-rad, is also to be done.

3.3.2. Symbolic floating-point arithmetic.

Our work on the analysis of algorithms in floating-point arithmetic leads us to manipulate floating-point data in their greatest generality, that is, as symbolic expressions in the base and the precision. A long-term goal here is to develop theorems as well as efficient data structures and algorithms for handling such quantities by computer rather than by hand as we do now. This is a completely new direction, whose main outcome will be a "symbolic floating-point toolbox" distributed in computer algebra systems like Sage and or Maple. In particular, such a toolbox will provide a way to check automatically the certificates of optimality we have obtained on the error bounds of various numerical algorithms. A PhD student has started on this subject in September 2014.

3.3.3. High-performance multiple precision arithmetic libraries.

Many numerical problems require higher precision than the conventional floating-point (single, double) formats. One solution is to use multiple precision libraries such as GNU MPFR, which allow the manipulation of very high precision numbers, but their generality (they are able to handle numbers with millions of digits), is a quite heavy alternative when high performance is needed. Our objective is to design a multiple precision arithmetic library that would allow to tackle problems where a precision of a few hundred bits is sufficient, but which have strong performance requirements. Applications include the process of long-term iteration of chaotic dynamical systems ranging from the classical Henon map to calculations of planetary orbits. The designed algorithms will be formally proved. We are in close contact with Warwick Tucker (Uppsala University, Sweden) and Mioara Joldes (LAAS, Toulouse) on this topic. A PhD student funded by a Région Rhône-Alpes grant has started on this topic in September 2014.

3.3.4. Interactions between arithmetics.

We will work on the interplay between floating-point and integer arithmetics, and especially on how to make the best use of both integer and floating-point basic operations when designing floating-point numerical kernels for embedded devices. This will be done in the context of the Metalibm ANR project and of our collaboration with STMicroelectronics. In addition, our work on the IEEE 1788 standard leads naturally to the development of associated reference libraries for interval arithmetic. A first direction will be to implement IEEE 1788 interval arithmetic using the fixed-precision hardware available for IEEE 754-2008 floating-point arithmetic. Another one will be to provide efficient support for multiple-precision intervals, in mid-rad representation and by developing MPFR-based code-generation tools aimed at handling families of functions.

3.3.5. Adequation algorithms/architectures.

So far, we have investigated how specific instructions like the fused multiply-add (FMA) impact the accuracy of computations, and have proposed several highly accurate FMA-based algorithms. The FMA being available

on several recent architectures, we now want to understand its impact on such algorithms in terms of practical performances. This should be a medium term project, leading to FMA-based algorithms with best speed/accuracy/robustness tradeoff. On the other hand (and on the long term), a major issue is how to exploit the various levels of parallelism of recent and upcoming architectures to ensure simultaneously high performance and reliability. A first direction will be to focus on SIMD parallelism, offered by instruction sets via vector instructions. This kind of parallelism should be key for small numerical kernels like elementary functions, complex arithmetic, or low-dimensional matrix computations. A second direction will be at the multi-core processor level, especially for larger numerical or algebraic problems (and in conjunction with SIMD parallelism when handling sub-problems of small enough dimension). Finally, we will work on aspects of automatic adaptation (auto-tuning) to such architectural features, not only for speed, but also for accuracy. This could be done via the design and implementation of heuristics capable of inserting more accurate codes, based for example on error-free transforms, whenever needed.

CARAMEL Project-Team

3. Research Program

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 algorithms known for factoring integers or for computing discrete logarithms in finite fields have — at best — a subexponential complexity, whereas the best attack known 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 (National Security Agency, USA) or the BSI (Bundesamt für Sicherheit in der Informationstechnik, Germany), 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 (UOV, 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 [32], 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), but the recent progress in discrete logarithms in finite fields will probably slow down its large deployment.

Despite the rise of elliptic curve cryptography and the variety of more or less mature 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 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.

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 support an increasing number of 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 intended hardware platforms, 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 other goals, since they all rely more or less on integer or polynomial arithmetic.

CASCADE Project-Team

3. Research Program

3.1. Randomness in Cryptography

Randomness is a key ingredient for cryptography. Random bits are necessary not only for generating cryptographic keys, but are also often an part of steps of cryptographic algorithms. In some cases, probabilistic protocols make it possible to perform tasks that are impossible deterministically. In other cases, probabilistic algorithms are faster, more space efficient or simpler than known deterministic algorithms. Cryptographers usually assume that parties have access to perfect randomness but in practice this assumption is often violated and a large body of research is concerned with obtaining such a sequence of random or pseudorandom bits.

One of the project-team research goals is to get a better understanding of the interplay between randomness and cryptography and to study the security of various cryptographic protocols at different levels (information-theoretic and computational security, number-theoretic assumptions, design and provable security of new and existing constructions).

Cryptographic literature usually pays no attention to the fact that in practice randomness is quite difficult to generate and that it should be considered as a resource like space and time. Moreover since the perfect randomness abstraction is not physically realizable, it is interesting to determine whether imperfect randomness is "good enough" for certain cryptographic algorithms and to design algorithms that are robust with respect to deviations of the random sources from true randomness.

The power of randomness in computation is a central problem in complexity theory and in cryptography. Cryptographers should definitely take these considerations into account when proposing new cryptographic schemes: there exist computational tasks that we only know how to perform efficiently using randomness but conversely it is sometimes possible to remove randomness from probabilistic algorithms to obtain efficient deterministic counterparts. Since these constructions may hinder the security of cryptographic schemes, it is of high interest to study the efficiency/security tradeoff provided by randomness in cryptography.

Quite often in practice, the random bits in cryptographic protocols are generated by a pseudorandom number generation process. When this is done, the security of the scheme of course depends in a crucial way on the quality of the random bits produced by the generator. Despite the importance, many protocols used in practice often leave unspecified what pseudorandom number generation to use. It is well-known that pseudorandom generators exist if and only if one-way functions exist and there exist efficient constructions based on various number-theoretic assumptions. Unfortunately, these constructions are too inefficient and many protocols used in practice rely on "ad-hoc" constructions. It is therefore interesting to propose more efficient constructions, to analyze the security of existing ones and of specific cryptographic constructions that use weak pseudorandom number generators.

The project-team undertakes research in these three aspects. The approach adopted is both theoretical and practical, since we provide security results in a mathematical frameworks (information theoretic or computational) with the aim to design protocols among the most efficient known.

3.2. Lattice Cryptography

The security of almost all public-key cryptographic protocols in use today relies on the presumed hardness of problems from number theory such as factoring and discrete log. This is somewhat problematic because these problems have very similar underlying structure, and its unforeseen exploit can render all currently used public key cryptography insecure. This structure was in fact exploited by Shor to construct efficient quantum algorithms that break all hardness assumptions from number theory that are currently in use. And so naturally, an important area of research is to build provably-secure protocols based on mathematical problems that are unrelated to factoring and discrete log. One of the most promising directions in this line of research is using lattice problems as a source of computational hardness —in particular since they also offer features that other alternative public-key cryptosystems (such as MQ-based, code-based or hash-based schemes) cannot provide.

At its very core, secure communication rests on two foundations: authenticity and secrecy. Authenticity assures the communicating parties that they are indeed communicating with each other and not with some potentially malicious outside party. Secrecy is necessary so that no one except the intended recipient of a message is able to deduce anything about its contents.

Lattice cryptography might find applications towards constructing practical schemes for resolving essential cryptographic problems —in particular, guaranteeing authenticity. On this front, our team is actively involved in pursuing the following two objectives:

1. Construct, implement, and standardize a practical public key digital signature scheme that is secure against quantum adversaries.
2. Construct, implement, and standardize a symmetric key authentication scheme that is secure against side channel attacks and is more efficient than the basic scheme using AES with masking.

Despite the great progress in constructing fairly practical lattice-based encryption and signature schemes, efficiency still remains a very large obstacle for advanced lattice primitives. While constructions of identity-based encryption schemes, group signature schemes, functional encryption schemes, and even fully-homomorphic encryption schemes are known, the implementations of these schemes are extremely inefficient.

Fully Homomorphic Encryption (FHE) is a very active research area. Let us just give one example illustrating the usefulness of computing on encrypted data: Consider an on-line patent database on which firms perform complex novelty queries before filing patents. With current technologies, the database owner might analyze the queries, infer the invention and apply for a patent before the genuine inventor. While such frauds were not reported so far, similar incidents happen during domain name registration. Several websites propose "registration services" preceded by "availability searches". These queries trigger the automated registration of the searched domain names which are then proposed for sale. Algorithms allowing arbitrary computations without disclosing their inputs (and/or their results) are hence of immediate usefulness.

In 2009, IBM announced the discovery of a FHE scheme by Craig Gentry. The security of this algorithm relies on worst-case problems over ideal lattices and on the hardness of the sparse subset sum problem. Gentry's construction is an ingenious combination of two ideas: a somewhat homomorphic scheme (capable of supporting many "logical or" operations but very few "ands") and a procedure that refreshes the homomorphically processed ciphertexts. Gentry's main conceptual achievement is a "bootstrapping" process in which the somewhat homomorphic scheme evaluates its own decryption circuit (self-reference) to refresh (recrypt) ciphertexts.

Unfortunately, it is safe to surmise that if the state of affairs remains as it is in the present, then despite all the theoretical efforts that went into their constructions, these schemes will never be used in practical applications.

Our team is looking at the foundations of these primitives with the hope of achieving a breakthrough that will allow them to be practical in the near future.

3.3. Security amidst Concurrency on the Internet

Cryptographic protocols that are secure when executed in isolation, can be completely insecure when multiple such instances are executed concurrently (as is unavoidable on the Internet) or when used as a part of a larger protocol. For instance, a man-in-the-middle attacker participating in two simultaneous executions of a cryptographic protocol might use messages from one of the executions in order to compromise the security of the second – Lowe's attack on the Needham-Schroeder authentication protocol and Bleichenbacher's attack on SSL work this way. Our research addresses security amidst concurrent executions in secure computation and key exchange protocols.

Secure computation allows several mutually distrustful parties to collaboratively compute a public function of their inputs, while providing the same security guarantees as if a trusted party had performed the computation. Potential applications for secure computation include anonymous voting as well as privacy-preserving auctions and data-mining. Our recent contributions on this topic include

1. new protocols for secure computation in a model where each party interacts only once, with a single centralized server; this model captures communication patterns that arise in many practical settings, such as that of Internet users on a website,

2. and efficient constructions of universally composable commitments and oblivious transfer protocols, which are the main building blocks for general secure computation.

In key exchange protocols, we are actively involved in designing new password-authenticated key exchange protocols, as well as the analysis of the widely-used SSL/TLS protocols.

3.4. Symmetric Key Cryptanalysis

Symmetric key cryptographic primitives play a very important role in secure communications. For example, block ciphers and stream ciphers are used to protect the privacy of cellular phone users from eavesdroppers, while MACs (message authentication codes) ensure that active attackers cannot interfere with cellular communication without being detected.

Since there is no method of formally proving that a complex modern symmetric key cipher is secure, there is no choice but to consider it secure if there are no known attacks against it. Thus, a symmetric key cipher should undergo an extensive cryptanalytic effort to evaluate its resistance against both well-known and new types of attacks. The goal of cryptanalytic is thus to ensure that only the strongest symmetric key cryptographic primitives are deployed and used in practice.

The team contributes to this field by proposing new cryptanalytic techniques and applying them to both new and existing secret key primitives, helping to understand their security.

CRYPT Team

3. Research Program

3.1. Public-Key Cryptanalysis

This project is interested in any public-key cryptanalysis, in the broad sense.

3.1.1. Mathematical Foundations

Historically, one useful side-effect of public-key cryptanalysis has been the introduction of advanced mathematical objects in cryptology, which were later used for cryptographic design. The most famous examples are elliptic curves (first introduced in cryptology to factor integer numbers), lattices (first introduced in cryptology to attack knapsack cryptosystems) and pairings over elliptic curves (first introduced in cryptology to attack the discrete logarithm problem over special elliptic curves). It is therefore interesting to develop the mathematics of public-key cryptanalysis. In particular, we would like to deepen our understanding of lattices by studying well-known mathematical aspects such as packing problems, transference theorems or random lattices.

3.1.2. Lattice Algorithms

Due to the strong interest surrounding lattice-based cryptography at the moment, our main focus is to attack lattice-based cryptosystems, particularly the most efficient ones (such as NTRU), and the ones providing new functionalities such as fully-homomorphic encryption or noisy multi-linear maps: recent cryptanalysis examples include [4], [5] for the latter, and [6] for the former. We want to assess the concrete security level of lattice-based cryptosystems, as has been done for cryptosystems based on integer factoring or discrete logarithms: this has been explored in [25], but needs to be developed. This requires to analyze and design the best algorithms for solving lattice problems, either exactly or approximately. In this area, much progress has been obtained the past few years (such as [26]), but we believe there is still more to come. We are working on new lattice computational records.

We are also interested in lattice-based cryptanalysis of non-lattice cryptosystems, by designing new attacks or improving old attacks. A well-known example is RSA for which the best attacks in certain settings are based on lattice techniques, following a seminal work by Coppersmith in 1996: recently [3], we improved the efficiency of some of these attacks on RSA, and we would like to extend this kind of results.

3.1.3. New Assumptions

In the past few years, new cryptographic functionalities (such as fully-homomorphic encryption, noisy multinear maps, indistinguishability obfuscation, etc.) have appeared, many of which being based on lattices. They usually introduce new algorithmic problems whose hardness is not well-understood. It is extremely important to study the hardness of these new assumptions, in order to evaluate the feasability of these new functionalities. Sometimes, the problem itself is not new, but the (aggressive) choices of parameters are: for instance, several implementations of fully-homorphic encryption used well-known lattice problems like LWE or BDD but with very large parameters which have not been studied much.

Currently, there are very few articles studying the concrete hardness of these new assumptions, especially compared to the articles using these new assumptions.

3.2. Secret-Key Cryptanalysis

Though secret-key cryptanalysis is the oldest form of cryptanalysis, there is regular progress in this area.

3.2.1. Hash Functions

In the past few years, the most important event has been the SHA-3 competition for a new hash function standard. This competition ended in 2012, with Keccak selected as the winner. We intend to study Keccak, together with the four other SHA-3 finalists. New cryptanalytical techniques designed to attack SHA-3 candidates are likely to be useful to attack other schemes. For instance, this was the case for the so-called rebound attack.

However, it is also interesting not to forget widespread hash functions: while it is now extremely easy to generate new MD5 collisions, a collision for SHA-1 has yet to be found, despite the existence of theoretical collision attacks faster than birthday attacks. Besides, there are still very few results on the SHA-2 standards family.

We may also be interested in related topics such as message authentication codes, especially those based on hash functions, which we explored in the past.

3.2.2. Symmetric Ciphers

Symmetric ciphers are widely deployed because of their high performances: a typical case is disk encryption and wireless communications.

We intend to study widespread block ciphers, such as the AES (now implemented in Intel processors) and Kasumi (used in UMTS) standards, as illustrated in recent publications [7], [9], [10], [9] of the team. Surprisingly, new attacks [24], [23] on the AES have appeared in the past few years, such as related-key attacks and single-key attacks. It is very important to find out if these attacks can be improved, even if they are very far from being practical. An interesting trend in block cipher cryptanalysis is to adapt recent attacks on hash functions: this is the reciprocal of the phenomenon of ten years ago, when Wang's MD5 collision attack was based on differential cryptanalysis.

Similarly to block ciphers, we intend to study widespread stream ciphers, such as RC4. The case of RC4 is particularly interesting due to the extreme simplicity of this cipher, and its deployment in numerous applications such as wireless Internet protocols. In the past few years, new attacks on RC4 based on various biases (such as [30]) have appeared, and several attacks on RC4 are used in WEP-attack tools.

GALAAD2 Team

3. Research Program

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 complex 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 contexts, 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. Research Program

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 raises 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 Project-Team

3. Research Program

3.1. Algorithmic Number Theory

Algorithmic Number Theory is concerned with replacing special cases with general algorithms to solve problems in number theory. In the Grace project, it appears in three main threads:

- fundamental algorithms for integers and polynomials (including primality and factorization);
- algorithms for finite fields (including discrete logarithms); and
- algorithms for algebraic curves.

Clearly, we use computer algebra in many ways. Research in cryptology has motivated a renewed interest in Algorithmic Number Theory in recent decades—but the fundamental problems still exist *per se*. Indeed, while algorithmic number theory application in cryptanalysis is epitomized by applying factorization to breaking RSA public key, many other problems, are relevant to various area of computer science. Roughly speaking, the problems of the cryptological world are of bounded size, whereas Algorithmic Number Theory is also concerned with asymptotic results.

3.2. Arithmetic Geometry: Curves and their Jacobians

Arithmetic Geometry is the meeting point of algebraic geometry and number theory: that is, the study of geometric objects defined over arithmetic number systems (such as the integers and finite fields). The fundamental objects for our applications in both coding theory and cryptology are curves and their Jacobians over finite fields.

An algebraic *plane curve*$\mathcal{X}$ over a field $\mathbf{K}$ is defined by an equation

$$\mathcal{X} : F_{\mathcal{X}}(x, y) = 0 \quad \text{where } F_{\mathcal{X}} \in \mathbf{K}[x, y].$$

(Not every curve is planar—we may have more variables, and more defining equations—but from an algorithmic point of view, we can always reduce to the plane setting.) The *genus* $g_{\mathcal{X}}$ of $\mathcal{X}$ is a non-negative integer classifying the essential geometric complexity of $\mathcal{X}$; it depends on the degree of $F_{\mathcal{X}}$ and on the number of singularities of $\mathcal{X}$. The simplest curves with nontrivial Jacobians are curves of genus 1, known as *elliptic curves*; they are typically defined by equations of the form $y^2 = x^3 + Ax + B$. Elliptic curves are particularly important given their central role in public-key cryptography over the past two decades. Curves of higher genus are important in both cryptography and coding theory.

The curve $\mathcal{X}$ is associated in a functorial way with an algebraic group $J_{\mathcal{X}}$, called the *Jacobian* of $\mathcal{X}$. The group $J_{\mathcal{X}}$ has a geometric structure: its elements correspond to points on a $g_{\mathcal{X}}$-dimensional projective algebraic group variety. Typically, we do not compute with the equations defining this projective variety: there are too many of them, in too many variables, for this to be convenient. Instead, we use fast algorithms based on the representation in terms of classes of formal sums of points on $\mathcal{X}$.

3.3. Curve-Based cryptology

Jacobians of curves are excellent candidates for cryptographic groups when constructing efficient instances of public-key cryptosystems. Diffie–Hellman key exchange is an instructive example.

Suppose Alice and Bob want to establish a secure communication channel. Essentially, this means establishing a common secret *key*, which they will then use for encryption and decryption. Some decades ago, they would have exchanged this key in person, or through some trusted intermediary; in the modern, networked world, this is typically impossible, and in any case completely unscalable. Alice and Bob may be anonymous parties who want to do e-business, for example, in which case they cannot securely meet, and they have no way to be sure of each other's identities. Diffie–Hellman key exchange solves this problem. First, Alice and Bob publicly agree on a cryptographic group G with a generator P (of order N); then Alice secretly chooses an integer a from $[1..N]$, and sends aP to Bob. In the meantime, Bob secretly chooses an integer b from $[1..N]$, and sends bP to Alice. Alice then computes $a(bP)$, while Bob computes $b(aP)$; both have now computed abP, which becomes their shared secret key. The security of this key depends on the difficulty of computing abP given P, aP, and bP; this is the Computational Diffie–Hellman Problem (CDHP). In practice, the CDHP corresponds to the Discrete Logarithm Problem (DLP), which is to determine a given P and aP.

This simple protocol has been in use, with only minor modifications, since the 1970s. The challenge is to create examples of groups G with a relatively compact representation and an efficiently computable group law, and such that the DLP in G is hard (ideally approaching the exponential difficulty of the DLP in an abstract group). The Pohlig–Hellman reduction shows that the DLP in G is essentially only as hard as the DLP in its largest prime-order subgroup. We therefore look for compact and efficient groups of prime order.

The classic example of a group suitable for the Diffie–Hellman protocol is the multiplicative group of a finite field $\mathbf{F}_q$. There are two problems that render its usage somewhat less than ideal. First, it has too much structure: we have a subexponential Index Calculus attack on the DLP in this group, so while it is very hard, the DLP falls a long way short of the exponential difficulty of the DLP in an abstract group. Second, there is only one such group for each q: its subgroup treillis depends only on the factorization of $q-1$, and requiring $q-1$ to have a large prime factor eliminates many convenient choices of q.

This is where Jacobians of algebraic curves come into their own. First, elliptic curves and Jacobians of genus 2 curves do not have a subexponential index calculus algorithm: in particular, from the point of view of the DLP, a generic elliptic curve is currently *as strong as* a generic group of the same size. Second, they provide some diversity: we have many degrees of freedom in choosing curves over a fixed $\mathbf{F}_q$, with a consequent diversity of possible cryptographic group orders. Furthermore, an attack which leaves one curve vulnerable may not necessarily apply to other curves. Third, viewing a Jacobian as a geometric object rather than a pure group allows us to take advantage of a number of special features of Jacobians. These features include efficiently computable pairings, geometric transformations for optimised group laws, and the availability of efficiently computable non-integer endomorphisms for accelerated encryption and decryption.

3.4. Algebraic Coding Theory

Coding Theory studies originated with the idea of using redundancy in messages to protect against noise and errors. The last decade of the 20th century has seen the success of so-called iterative decoding methods, which enable us to get very close to the Shannon capacity. The capacity of a given channel is the best achievable transmission *rate* for reliable transmission. The consensus in the community is that this capacity is more easily reached with these iterative and probabilistic methods than with algebraic codes (such as Reed–Solomon codes).

However, algebraic coding is useful in settings other than the Shannon context. Indeed, the Shannon setting is a random case setting, and promises only a vanishing error probability. In contrast, the algebraic Hamming approach is a worst case approach: under combinatorial restrictions on the noise, the noise can be adversarial, with strictly zero errors.

These considerations are renewed by the topic of *list decoding* after the breakthrough of Guruswami and Sudan at the end of the nineties. List decoding relaxes the uniqueness requirement of decoding, allowing a small list of candidates to be returned instead of a single codeword. List decoding can reach a capacity close to the Shannon capacity, with zero failure, with small lists, in the adversarial case. The method of Guruswami and Sudan enabled list decoding of most of the main algebraic codes: Reed–Solomon codes and

Algebraic–Geometry (AG) codes and new related constructions "capacity-achieving list decodable codes". These results open the way to applications again adversarial channels, which correspond to worst case settings in the classical computer science language.

Another avenue of our studies is AG codes over various geometric objects. Although Reed–Solomon codes are the best possible codes for a given alphabet, they are very limited in their length, which cannot exceed the size of the alphabet. AG codes circumvent this limitation, using the theory of algebraic curves over finite fields to construct long codes over a fixed alphabet. The striking result of Tsfasman–Vladut–Zink showed that codes better than random codes can be built this way, for medium to large alphabets. Disregarding the asymptotic aspects and considering only finite length, AG codes can be used either for longer codes with the same alphabet, or for codes with the same length with a smaller alphabet (and thus faster underlying arithmetic).

From a broader point of view, wherever Reed–Solomon codes are used, we can substitute AG codes with some benefits: either beating random constructions, or beating Reed–Solomon codes which are of bounded length for a given alphabet.

Another area of Algebraic Coding Theory with which we are more recently concerned is the one of Locally Decodable Codes. After having been first theoretically introduced, those codes now begin to find practical applications, most notably in cloud-based remote storage systems.

LFANT Project-Team

3. Research Program

3.1. Number fields, class groups and other invariants

Participants: Bill Allombert, Athanasios Angelakis, Karim Belabas, Julio Brau Avila, Jean-Paul Cerri, Henri Cohen, Jean-Marc Couveignes, Andreas Enge, Pınar Kılıçer, 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 [5]. 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 [35] 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 Avila, Jean-Marc Couveignes, Andreas Enge, Hamish Ivey-Law, Nicolas Mascot, Enea Milio, Damien Robert.

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, Hamish Ivey-Law, Chloë Martindale, 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 [37], for more background material, [36]. 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. Research Program

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 are 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, Jules Svartz, Louise Huot, Thibaut Verron.

Efficient algorithms F_4/F_5[0] 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.

[0] 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, Simone Naldi, Ivan Bannwarth.

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.

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 these 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 algebra 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 basis is a key step toward the resolution of difficult problems in this domain [0]. Using existing resolution methods is simply not enough to solve relevant problems. The main algorithmic look 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 [0] 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 Urvoy de Portzamparc, Rina Zeitoun, Jérémy Berthomieu.

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*.

[0] 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
[0] 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. Research Program

3.1. Scientific foundations

Our research work is mainly devoted to the design and analysis of cryptographic algorithms, either in the classical or in the quantum setting. 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, quantum physics... Most of our work mixes 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.

SPECFUN Project-Team

3. Research Program

3.1. Studying special functions by computer algebra

Computer algebra manipulates symbolic representations of exact mathematical objects in a computer, in order to perform computations and operations like simplifying expressions and solving equations for "closed-form expressions". The manipulations are often fundamentally of algebraic nature, even when the ultimate goal is analytic. The issue of efficiency is a particular one in computer algebra, owing to the extreme swell of the intermediate values during calculations.

Our view on the domain is that research on the algorithmic manipulation of special functions is anchored between two paradigms:

- adopting linear differential equations as the right data structure for special functions,
- designing efficient algorithms in a complexity-driven way.

It aims at four kinds of algorithmic goals:

- algorithms combining functions,
- functional equations solving,
- multi-precision numerical evaluations,
- guessing heuristics.

This interacts with three domains of research:

- computer algebra, meant as the search for quasi-optimal algorithms for exact algebraic objects,
- symbolic analysis/algebraic analysis;
- experimental mathematics (combinatorics, mathematical physics, ...).

This view is made explicit in the present section.

3.1.1. Equations as a data structure

Numerous special functions satisfy linear differential and/or recurrence equations. Under a mild technical condition, the existence of such equations induces a finiteness property that makes the main properties of the functions decidable. We thus speak of *D-finite functions*. For example, 60 % of the chapters in the handbook [20] describe D-finite functions. In addition, the class is closed under a rich set of algebraic operations. This makes linear functional equations just the right data structure to encode and manipulate special functions. The power of this representation was observed in the early 1990s [72], leading to the design of many algorithms in computer algebra. Both on the theoretical and algorithmic sides, the study of D-finite functions shares much with neighbouring mathematical domains: differential algebra, D-module theory, differential Galois theory, as well as their counterparts for recurrence equations.

3.1.2. Algorithms combining functions

Differential/recurrence equations that define special functions can be recombined [72] to define: additions and products of special functions; compositions of special functions; integrals and sums involving special functions. Zeilberger's fast algorithm for obtaining recurrences satisfied by parametrised binomial sums was developed in the early 1990s already [73]. It is the basis of all modern definite summation and integration algorithms. The theory was made fully rigorous and algorithmic in later works, mostly by a group in RISC (Linz, Austria) and by members of the team [61], [69], [38], [36], [37], [56]. The past ÉPI Algorithms contributed several implementations (*gfun* [64], *Mgfun* [38]).

3.1.3. Solving functional equations

Encoding special functions as defining linear functional equations postpones some of the difficulty of the problems to a delayed solving of equations. But at the same time, solving (for special classes of functions) is a sub-task of many algorithms on special functions, especially so when solving in terms of polynomial or rational functions. A lot of work has been done in this direction in the 1990s; more intensively since the 2000s, solving differential and recurrence equations in terms of special functions has also been investigated.

3.1.4. Multi-precision numerical evaluation

A major conceptual and algorithmic difference exists for numerical calculations between data structures that fit on a machine word and data structures of arbitrary length, that is, *multi-precision* arithmetic. When multi-precision floating-point numbers became available, early works on the evaluation of special functions were just promising that "most" digits in the output were correct, and performed by heuristically increasing precision during intermediate calculations, without intended rigour. The original theory has evolved in a twofold way since the 1990s: by making computable all constants hidden in asymptotic approximations, it became possible to guarantee a *prescribed* absolute precision; by employing state-of-the-art algorithms on polynomials, matrices, etc, it became possible to have evaluation algorithms in a time complexity that is linear in the output size, with a constant that is not more than a few units. On the implementation side, several original works exist, one of which (*NumGfun* [60]) is used in our DDMF.

3.1.5. Guessing heuristics

"Differential approximation", or "Guessing", is an operation to get an ODE likely to be satisfied by a given approximate series expansion of an unknown function. This has been used at least since the 1970s and is a key stone in spectacular applications in experimental mathematics [34]. All this is based on subtle algorithms for Hermite–Padé approximants [24]. Moreover, guessing can at times be complemented by proven quantitative results that turn the heuristics into an algorithm [32]. This is a promising algorithmic approach that deserves more attention than it has received so far.

3.1.6. Complexity-driven design of algorithms

The main concern of computer algebra has long been to prove the feasibility of a given problem, that is, to show the existence of an algorithmic solution for it. However, with the advent of faster and faster computers, complexity results have ceased to be of theoretical interest only. Nowadays, a large track of works in computer algebra is interested in developing fast algorithms, with time complexity as close as possible to linear in their output size. After most of the more pervasive objects like integers, polynomials, and matrices have been endowed with fast algorithms for the main operations on them [43], the community, including ourselves, started to turn its attention to differential and recurrence objects in the 2000s. The subject is still not as developed as in the commutative case, and a major challenge remains to understand the combinatorics behind summation and integration. On the methodological side, several paradigms occur repeatedly in fast algorithms: "divide and conquer" to balance calculations, "evaluation and interpolation" to avoid intermediate swell of data, etc. [29].

3.2. Trusted computer-algebra calculations

3.2.1. Encyclopedias

Handbooks collecting mathematical properties aim at serving as reference, therefore trusted, documents. The decision of several authors or maintainers of such knowledge bases to move from paper books [20], [22], [65] to websites and wikis [0] allows for a more collaborative effort in proof reading. Another step toward further confidence is to manage to generate the content of an encyclopedia by computer-algebra programs, as is the case with the Wolfram Functions Site [0] or DDMF [0]. Yet, due to the lingering doubts about computer-algebra systems, some encyclopedias propose both cross-checking by different systems and handwritten companion paper proofs of their content [0]. As of today, there is no encyclopedia certified with formal proofs.

[0] for instance http://dlmf.nist.gov/ for special functions or http://oeis.org/ for integer sequences
[0] http://functions.wolfram.com/
[0] http://ddmf.msr-inria.inria.fr/1.9.1/ddmf

3.2.2. Computer algebra and symbolic logic

Several attempts have been made in order to extend existing computer-algebra systems with symbolic manipulations of logical formulas. Yet, these works are more about extending the expressivity of computer-algebra systems than about improving the standards of correctness and semantics of the systems. Conversely, several projects have addressed the communication of a proof system with a computer-algebra system, resulting in an increased automation available in the proof system, to the price of the uncertainty of the computations performed by this oracle.

3.2.3. Certifying systems for computer algebra

More ambitious projects have tried to design a new computer-algebra system providing an environment where the user could both program efficiently and elaborate formal and machine-checked proofs of correctness, by calling a general-purpose proof assistant like the Coq system. This approach requires a huge manpower and a daunting effort in order to re-implement a complete computer-algebra system, as well as the libraries of formal mathematics required by such formal proofs.

3.2.4. Semantics for computer algebra

The move to machine-checked proofs of the mathematical correctness of the output of computer-algebra implementations demands a prior clarification about the often implicit assumptions on which the presumably correctly implemented algorithms rely. Interestingly, this preliminary work, which could be considered as independent from a formal certification project, is seldom precise or even available in the literature.

3.2.5. Formal proofs for symbolic components of computer-algebra systems

A number of authors have investigated ways to organize the communication of a chosen computer-algebra system with a chosen proof assistant in order to certify specific components of the computer-algebra systems, experimenting various combinations of systems and various formats for mathematical exchanges. Another line of research consists in the implementation and certification of computer-algebra algorithms inside the logic [68], [48], [57] or as a proof-automation strategy. Normalization algorithms are of special interest when they allow to check results possibly obtained by an external computer-algebra oracle [41]. A discussion about the systematic separation of the search for a solution and the checking of the solution is already clearly outlined in [54].

3.2.6. Formal proofs for numerical components of computer-algebra systems

Significant progress has been made in the certification of numerical applications by formal proofs. Libraries formalizing and implementing floating-point arithmetic as well as large numbers and arbitrary-precision arithmetic are available. These libraries are used to certify floating-point programs, implementations of mathematical functions and for applications like hybrid systems.

3.3. Machine-checked proofs of formalized mathematics

To be checked by a machine, a proof needs to be expressed in a constrained, relatively simple formal language. Proof assistants provide facilities to write proofs in such languages. But, as merely writing, even in a formal language, does not constitute a formal proof just per se, proof assistants also provide a proof checker: a small and well-understood piece of software in charge of verifying the correctness of arbitrarily large proofs. The gap between the low-level formal language a machine can check and the sophistication of an average page of mathematics is conspicuous and unavoidable. Proof assistants try to bridge this gap by offering facilities, like notations or automation, to support convenient formalization methodologies. Indeed, many aspects, from the logical foundation to the user interface, play an important role in the feasibility of formalized mathematics inside a proof assistant.

[0] http://129.81.170.14/~vhm/Table.html

3.3.1. Logical foundations and proof assistants

While many logical foundations for mathematics have been proposed, studied, and implemented, type theory is the one that has been more successfully employed to formalize mathematics, to the notable exception of the Mizar system [58], which is based on set theory. In particular, the calculus of construction (CoC) [39] and its extension with inductive types (CIC) [40], have been studied for more than 20 years and been implemented by several independent tools (like Lego, Matita, and Agda). Its reference implementation, Coq [66], has been used for several large-scale formalizations projects (formal certification of a compiler back-end; four-color theorem). Improving the type theory underlying the Coq system remains an active area of research. Other systems based on different type theories do exist and, whilst being more oriented toward software verification, have been also used to verify results of mainstream mathematics (prime-number theorem; Kepler conjecture).

3.3.2. Computations in formal proofs

The most distinguishing feature of CoC is that computation is promoted to the status of rigorous logical argument. Moreover, in its extension CIC, we can recognize the key ingredients of a functional programming language like inductive types, pattern matching, and recursive functions. Indeed, one can program effectively inside tools based on CIC like Coq. This possibility has paved the way to many effective formalization techniques that were essential to the most impressive formalizations made in CIC.

Another milestone in the promotion of the computations-as-proofs feature of Coq has been the integration of compilation techniques in the system to speed up evaluation. Coq can now run realistic programs in the logic, and hence easily incorporates calculations into proofs that demand heavy computational steps.

Because of their different choice for the underlying logic, other proof assistants have to simulate computations outside the formal system, and indeed fewer attempts to formalize mathematical proofs involving heavy calculations have been made in these tools. The only notable exception, which was finished in 2014, the Kepler conjecture, required a significant work to optimize the rewriting engine that simulates evaluation in Isabelle/HOL.

3.3.3. Large-scale computations for proofs inside the Coq system

Programs run and proved correct inside the logic are especially useful for the conception of automated decision procedures. To this end, inductive types are used as an internal language for the description of mathematical objects by their syntax, thus enabling programs to reason and compute by case analysis and recursion on symbolic expressions.

The output of complex and optimized programs external to the proof assistant can also be stamped with a formal proof of correctness when their result is easier to *check* than to *find*. In that case one can benefit from their efficiency without compromising the level of confidence on their output at the price of writing and certify a checker inside the logic. This approach, which has been successfully used in various contexts, is very relevant to the present research project.

3.3.4. Relevant contributions from the Mathematical Component libraries

Representing abstract algebra in a proof assistant has been studied for long. The libraries developed by the MathComp project for the proof of the Odd Order Theorem provide a rather comprehensive hierarchy of structures; however, they originally feature a large number of instances of structures that they need to organize. On the methodological side, this hierarchy is an incarnation of an original work [42] based on various mechanisms, primarily type inference, typically employed in the area of programming languages. A large amount of information that is implicit in handwritten proofs, and that must become explicit at formalization time, can be systematically recovered following this methodology.

Small-scale reflection [45] is another methodology promoted by the MathComp project. Its ultimate goal is to ease formal proofs by systematically dealing with as many bureaucratic steps as possible, by automated computation. For instance, as opposed to the style advocated by Coq's standard library, decidable predicates are systematically represented using computable boolean functions: comparison on integers is expressed as

program, and to state that $a \leq b$ one compares the output of this program run on a and b with $true$. In many cases, for example when a and b are values, one can prove or disprove the inequality by pure computation.

The MathComp library was consistently designed after uniform principles of software engineering. These principles range from simple ones, like naming conventions, to more advanced ones, like generic programming, resulting in a robust and reusable collection of formal mathematical components. This large body of formalized mathematics covers a broad panel of algebraic theories, including of course advanced topics of finite group theory, but also linear algebra, commutative algebra, Galois theory, and representation theory. We refer the interested reader to the online documentation of these libraries [67], which represent about 150,000 lines of code and include roughly 4,000 definitions and 13,000 theorems.

Topics not addressed by these libraries and that might be relevant to the present project include real analysis and differential equations. The most advanced work of formalization on these domains is available in the HOL-Light system [50], [51], [52], although some existing developments of interest [27], [59] are also available for Coq. Another aspect of the MathComp libraries that needs improvement, owing to the size of the data we manipulate, is the connection with efficient data structures and implementations, which only starts to be explored.

3.3.5. User interaction with the proof assistant

The user of a proof assistant describes the proof he wants to formalize in the system using a textual language. Depending on the peculiarities of the formal system and the applicative domain, different proof languages have been developed. Some proof assistants promote the use of a declarative language, when the Coq and Matita systems are more oriented toward a procedural style.

The development of the large, consistent body of MathComp libraries has prompted the need to design an alternative and coherent language extension for the Coq proof assistant [47], [46], enforcing the robustness of proof scripts to the numerous changes induced by code refactoring and enhancing the support for the methodology of small-scale reflection.

The development of large libraries is quite a novelty for the Coq system. In particular any long-term development process requires the iteration of many refactoring steps and very little support is provided by most proof assistants, with the notable exception of Mizar [63]. For the Coq system, this is an active area of research.

VEGAS Project-Team (section vide)

ALF Project-Team

3. Research Program

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- core 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 whose 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 [37] 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 [38]). Such simulations are CPU consuming and simulations cannot be run on a complete application. Sampling representative slices of the application was proposed [4] and popularized by the Simpoint [48] 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 [40]), accurate evaluation of the WCET of an algorithm on a complex uniprocessor system is a difficult problem. Accurately modelling data cache behavior [3] 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) [45].

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 [52].

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

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 [10], [12]. 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 [39]. 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 [46], but the most promising solutions integrate hardware support for software thread-level speculation [50].

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* [9] through techniques such as clock gating or throttling and *activity migration* [49][5].

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 [6], 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 [42] 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 [43] 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 [2]. Moreover, this bytecode representation can be compiled to native code with performance similar to static compilation [1]. 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 maintenance since 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 information 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 only high 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. 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) [45]. 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 [44] 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 [53], therefore hardware-based or compiler methods such as [51] 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.

ATEAMS Project-Team

3. Research Program

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, transform it and generate it. Each analysis or transformation begins with fact extraction. After that we may analyze specific software systems or large bodies of software systems. Our goal is to improve software systems by understanding and resolving the causes of software complexity. The approach is captured in the EASY acronym: Extract Analyze SYnthesize. The first step is to extract facts from source code. These facts are then enriched and refined in an analysis phase. Finally the result is synthesized in the form of transformed or generated source code, a metrics report, a visualization or some other output artifact.

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 are exploring is the use of syntax-related program patterns for fact extraction. An early proposal for such a pattern-based approach consisted of extending a fixed base language (either C or PL/1 variant) 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 closure) and the performance problems encountered, this approach has not seen wider use.

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 existing (legacy) programming languages, which is highly relevant for experimental research and validation.

Extracted facts are often refined, enriched and queried in the analysis phase. We propose to use a relational formalization of the facts. That is, facts are represented as sets of tuples, which can then be queried using relational algebra operators (e.g., domain, transitive closure, projection, composition etc.). This relational representation facilitates dealing with graphs, which are commonly needed during program analysis, for instance when processing control-flow or data-flow graphs. The Rascal language integrates a relational sub-language by providing comprehensions over different kinds of data types, in combination with powerful pattern matching and built-in primitives for computing (transitive/reflexive) closures and fixpoint computations (equation solving).

3.2.1. Goals

The main goal is to replace labour-intensive manual programming of fact extractors by automatic generation based on concise and formal specification. 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, yet flexible enough to be customized to the analysis case at hand. 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. Refactoring and Transformation

The second 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.

3.3.1. 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.

3.4. 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.

3.4.1. 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. As an end-result we envision Rascal to be a one-stop shop for source code analysis, transformation, generation and visualization.

3.5. Domain-specific Languages

Our final goal is centered around Domain-specific languages (DSLs), which are software languages tailored to a specific problem domain. DSLs can provide orders of magnitude improvement in terms of software quality and productivity. However, the implementation of DSLs is challenging and requires not only thorough knowledge of the problem domain (e.g., finance, digital forensics, insurance, auditing etc.), but also knowledge of language implementation (e.g., parsing, compilation, type checking etc.). Tools for language implementation have been around since the archetypical parser generator YACC. However, many of such tools are characterized by high learning curves, lack of integration of language implementation facets, and lead to implementations that are hard to maintain. This line of research focuses on two topics: improve the practice and experience of DSL implementation, and evaluate the success of DSLs in industrial practice.

Language workbenches [4] are integrated environments to facilitate the development of all aspects of DSLs. This includes IDE support (e.g., syntax coloring, outlining, reference resolving etc.) for the defined languages. Rascal can be seen as a language workbench that focuses on flexibility, programmability and modularity. DSL implementation is, in essence, an instance of source code analysis and transformation. As a result, Rascal's features for fact extraction, analysis, tree traversal and synthesis are an excellent fit for this area. An important aspect in this line of research is bringing the IDE closer to the source code. This will involve investigation of heterogeneous representations of source code, by integrating graphical, tabular or forms-based user interface elements. As a result, we propose Rascal as a feature-rich workbench for model-driven software development.

The second component of this research is concerned with evaluating DSLs in industrial contexts. This means that DSLs constructed using Rascal will be applied in real-life environments so that expected improvements in quality, performance, or productivity can be observed. We already have experience with this in the domain of digital forensics, computational auditing and games.

3.5.1. Goals

The goal of this research topic is to improve the practice of DSL-based software development through language design and tool support. A primary focus is to extend the IDE support provided by Rascal, and to facilitate incremental, and iterative design of DSLs. The latter is supported by new (meta-)language constructs for extending existing language implementations. This will require research into extensible programming and composition of compilers, interpreters and type checkers. Finally, a DSL is never an island: it will have to integrate with (third-party) source code, such as host language, libraries, runtime systems etc. This leads to the vision of multi-lingual programming environments [15].

CAIRN Project-Team

3. Research Program

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 [0].

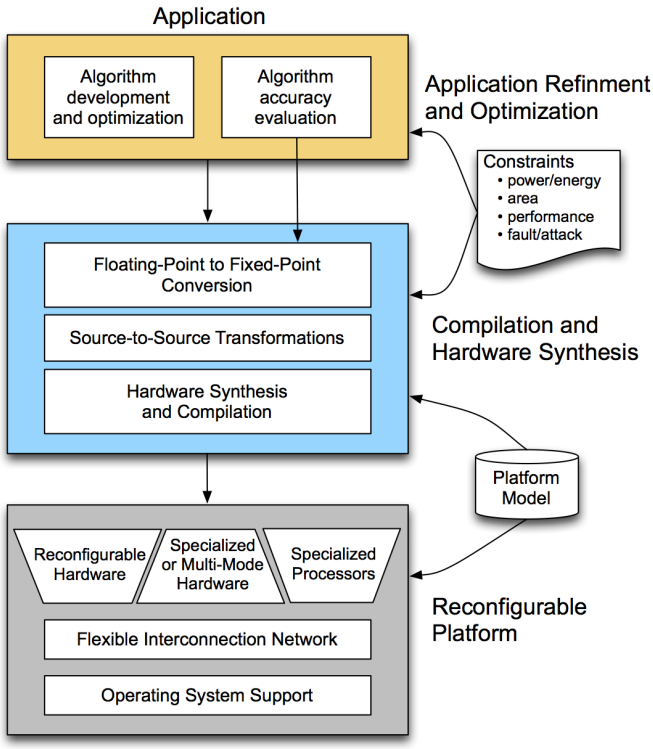

Figure 1. CAIRN*'s general design flow and related research themes*

Figure 1 shows the global design flow 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).

[0] 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 [113], [97], [78], [114]). 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 [93][12], CRIP [81], Adres [105] or others [116]. These works have led to commercial products such as the Extreme Processor Platform (XPP) [82] from PACT or Montium [0] 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 [96], or of *field-configurable elements* such as the Xirisc processor [103] 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) [104]: communication, memory management, task scheduling [92], [85][1] and task placement. 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 [111]. 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 CAIRNwe 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.

[0] http://www.recoresystems.com/

- 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 [108], [84], [115], [118]. 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 [99], [108], [107]. 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 processors by extending their instruction set to speed-up computationally-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 [86]. 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 [101], [106] 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 [100], [89]. 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 [90], [112]. 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 [94], [102] and have shown their efficiency in terms of energy consumption [91]. 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 [80]. 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 [87], 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. Research Program

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 [28]. 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).

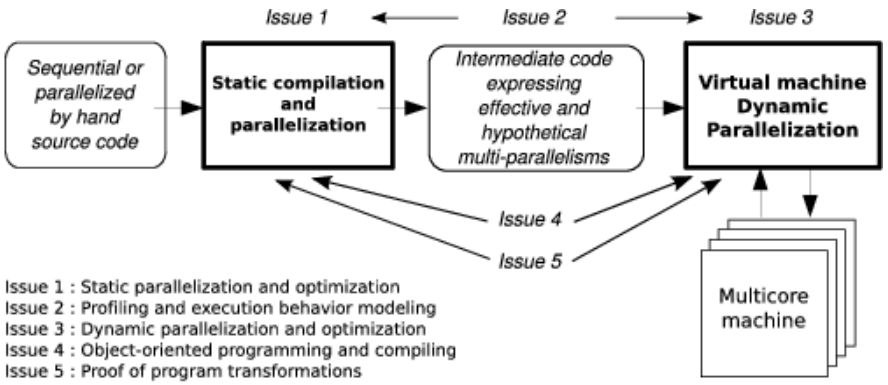

Figure 1. Automatic parallelizing steps for multicore architectures

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, Jean-François Dollinger, Aravind Sukumaran-Rajam, Juan Manuel Martinez Caamaño.

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 [27]. 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: adapting parallelization to multicore architecture and expressing many potential parallelisms.

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.

3.4. Dynamic parallelization and optimization, virtual machine

Participants: Aravind Sukumaran-Rajam, Juan Manuel Martinez Caamaño, Jean-François Dollinger, Alexandra Jimborean, Philippe Clauss, Vincent Loechner, Alain Ketterlin.

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.

3.5. Proof of program transformations for multicores

Participants: Éric Violard, Julien Narboux, Nicolas Magaud.

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 [29]. Proving transformations involving program which are not data-race free will constitute a longer term research goal.

COMPSYS Project-Team

3. Research Program

3.1. Architecture and compilation trends

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 however 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 (or even low-power GPUs). Such a combination DSP+FPGA 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 multicores 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 (and the associated analysis) in this transformation chain, restricting to compilation (transforming a program to a program) for embedded

processors and programmable accelerators, 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 (in the last 10 years now). 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 and its many "streaming" variants). In all aspects, the frontier between software and hardware is vanishing. For example, in terms of architecture, customized processors (with processor extension as first 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, and at both front-end and back-end levels, the first one acting more at the granularity of memories, transfers, and multiple cores, the second one more at the granularity of registers, system calls, and single core. Both levels must be considered as they interact with each other. Front-end optimizations must be aware of what back-end optimizations will do, as single core performance remain the basis for good parallel performances. Some front-end optimizations even act directly on back-end features, for example register tiling considered as a source-level transformation. Also, from a conceptual point of view, the polyhedral techniques developed by Compsys are actually the symbolic front-end counterpart, for structured loops, of back-end analysis and optimizations of unstructured programs (through control-flow graphs), such as dependence analysis, scheduling, lifetime analysis, register allocation, etc. A strength of Compsys so far was to juggle with both aspects, one more on graph theory with SSA-type optimizations, the other with polyhedra representing loops, and to exploit the correspondence between both. This has still to be exploited, for applying polyhedral techniques to more irregular programs.

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.

3.1.1. Compilation and languages issues in the context of embedded processors, "embedded systems", and programmable accelerators

Compilation is an old activity, in particular back-end code optimizations. The development of embedded systems was one of the reasons for the revival of compilation activities as a research topic. 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 focused on 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), (quasi) general-purpose processors (DSP for multimedia applications), and now hardware accelerators (dedicated platforms – such as those developed by Thales or the CEA –, or more general-purpose accelerators such as GPUs or even multicores, even if these are closer to small HPC platforms than truly embedded systems). Our cooperation with STMicroelectronics, until 2012, focused on investigating the compilation for specialized processors, such as the ST100 (DSP processor) and the ST200 (VLIW DSP processor) family. Even for this restricted class of processors, the diversity is large, and 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, generate many open problems. Our goal was to contribute to their understanding and their solutions.

An important concept to cope with the diversity of platforms is the concept of *virtualization*, which is a key for more portability, more simplicity, more reliability, and of course more security. This concept – implemented at

low level through binary translation and just-in-time (JIT) compilation [0] – consists in hiding the architecture-dependent features as long as possible during the compilation process. It has been used for a while for servers such as HotSpot, a bit more recently for workstations, and now for embedded computing. The same needs drive the development of intermediate languages such as OpenCL to, not necessarily hide, but at least make more uniform, the different facets of the underlying architectures. The challenge is then to design and compile high-productivity and high-performance languages [0] (coping with parallelism and heterogeneity) that can be ported to such intermediate languages, or to architecture-dependent runtime systems. The offloading of computation kernels, through source-to-source compilation, targeting back-end C dialects, has the same goals: to automate application porting to the variety of accelerators.

For JIT compilation, the compactness of the information representation, and thus its pertinence, is an important criterion for such late compilation phases. Indeed, the intermediate representation (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 one of the reasons why static single assignment (SSA), a sparse compact representation of liveness information, became popular in embedded compilation. 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. In this context, our aim has been, 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 in register allocation, opcode selection, code placement, when removing the SSA multiplexer functions (known as ϕ functions). These optimizations are usually done in the last phases of the compiler, using an assembly-level intermediate representation. As mentioned in Sections 2.3 and 2.4 , we made a lot of progress in this area in our past collaborations with STMicroelectronics (see also previous activity reports). Through the Sceptre and Mediacom projects, we first revisited, in the light of SSA, some 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 we exploited 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.

Our research directions are currently more focused on programmable accelerators, such as GPU and multi-cores, but still considering *static* compilation and without forgetting the link between high-level (in general at source-code level) and low-level (i.e., at assembly-code level) optimizations. They concern program analysis (of both sequential and parallel specifications), program optimizations (for memory hierarchies, parallelism, streaming, etc.), and also the link with applications and between compilers and users (programmers). Polyhedral techniques play an important role in these directions, even if control-flow-based techniques remain in the background and may come back at any time in the foreground. This is also the case for high-level synthesis, as exposed in the next section.

3.1.2. Context of high-level synthesis and FPGA platforms

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

[0] *Aggressive compilation* consists in allowing more time to implement more complete and costly solutions: the compiled program is loaded in permanent memory (ROM, flash, etc.) and its compilation time is less relevant than the execution time, size, and energy consumption of the produced code, which can have a critical impact on the cost and quality of the final product. Hence, the application is cross-compiled, i.e., compiled on a powerful platform distinct from the target processor. *Just-in-time compilation*, on the other hand, corresponds to compiling applets on demand on the target processor. For compatibility and compactness, the source languages are CIL or Java bytecode. The code can be uploaded or sold separately on a flash memory. Compilation is performed at load time and even dynamically during execution. The optimization heuristics, constrained by time and limited resources, are far from being aggressive. They must be fast but smart enough.

[0] For examples of such languages, see the keynotes event we organized in 2013: http://labexcompilation.ens-lyon.fr/hpc-languages.

and time-to-market minimization are paramount. When Compsys started, we were convinced that our expertise in compilation and automatic parallelization could 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 were proposed 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, Vivado HLS. 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, and communication through FIFOs. 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, even if, over the years, they become more and more mature. But it remains 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 or to design new tools from scratch.

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, need to be revisited, to pay more attention to locality, process streaming, and low-level pipelining, which are of paramount importance in hardware. Besides, classical methods mostly 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 resource, locality,

and pipelining 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, or in an incrementally-extended model.

Our research activities on polyhedral code analysis and optimizations directly target these HLS challenges. But they are not limited to the automatic generation of hardware as can be seen from our different contributions on X10, OpenStream, parametric tiling, etc. The same underlying concepts also arise when optimizing codes for GPUs and multicores. In this context of polyhedral analysis and optimizations, we will focus on three 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, as well as for hardware accelerators;
- developing concepts and techniques in a more global view of high-level synthesis and high-level parallel programming, starting from specification languages down to hardware implementation;
- developing more general code analysis so as to extract more information from codes as well as to extend the programs that can be handled.

3.2. Code analysis, code transformations, code optimizations

Embedded systems generated new problems in code analysis and optimization both for optimizing embedded software (compilation) and hardware (HLS). We now give a bit more details on some general challenges for program analysis, optimizations, and transformations, induced by this context, and on our methodology, in particular our development and use of polyhedral optimizations and its extensions.

3.2.1. Processes, scheduling, mapping, communications, etc.

Before mapping an application to an architecture, 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 processors/accelerators.

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 items flow from an input channel to the output. Here, the data are 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, for such applications. 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 (memory based, see for example CRP below). It is an interesting fact that several researchers have investigated translation from process networks [22] and to process networks [27], [28]. Streaming languages such as StreamIt and OpenStream have also been developed.

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 can be 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. Recent developments on the theory of polynomials (see Section 6.7) may offer a solution to this problem. One can also 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 and prove that the determinacy property still holds. 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 [24] or variants. 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. Tiling mechanisms can also be used to define atomic processes that can be pipelined as we proposed initially for FPGA [17].

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, techniques for data allocation in scratch-pad memory, or techniques for folding multi-dimensional arrays [21]. 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 (same questions for analyzing explicitly-parallel programs)? How does one take into account latency or throughput constraints, bandwidth constraints for input and output channels, memory hierarchies? 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 or C-level compilers) or to generate directly VHDL or lower-level C-dialects such as OpenCL. 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. In particular, the design of cost models remain a fundamental challenge.

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 naive implementation may destroy the benefits of high-level optimization. There are two possibilities here as suggested before; one may target another high-level synthesis or compilation tool, or one may target directly VHDL or low-level code. Each approach has its advantages and drawbacks. However, both situations require that the input program respects some strong constraints on the code shape, array accesses, memory accesses, communication protocols, etc. Furthermore, to get the compilers do what the user wants requires a lot of program tuning, i.e., of program rewriting or of program annotations. What can be automated in this rewriting process? Semi-automated?

In other words, we still need to address scheduling, memory, communication, and code generation issues, in the light of the developments of new languages and architectures, pushing the limits of such an automation.

3.2.2. Beyond static control programs

With the advent of parallelism in supercomputers, the bulk of research in code transformation resulted in (semi-)automatic parallelization, with many techniques (analysis, scheduling, code generation, etc.) based on the description and manipulation of nested loops with polyhedra. Compsys has always taken an active part in the development of these so-called “polyhedral techniques”. Historically, these analysis were (wrongly) understood to be limited to static control programs.

Actually, the polyhedral model is neither a programming language nor an execution model rather an intermediate representation. As such, it can be generated from imperative sequential languages like C or Fortran, streaming languages like CRP, or equational languages like Alpha. While the structure of the model is the same in all three cases, it may enjoy different properties, e.g., a schedule always exists in the first case, not in the two others. The import of the polyhedral model is that many questions relative to the analysis of a program and the applicability of transformations can be answered precisely and efficiently by applying well-known mathematical results to the model.

For irregular programs, the basic idea is to construct a polyhedral over-approximation, i.e., a program which has more operations, a larger memory footprint, and more dependences than the original. One can then parallelize the approximated program using polyhedral tools, and then return to the original, either by introducing guards, or by insuring that approximations are harmless. This technique is the standard way of dealing with approximated dependences. We already started to study the impact of approximations in our kernel offloading technique, for optimizing remote communications [4]. It is clear however that this method will apply only to mildly non-polyhedral programs. The restriction to arrays as the only data structure is still present. Its advantage is that it will be able to subsume in a coherent framework many disparate tricks: the extraction of SCoPs, induction variable detection, the omission of non-affine subscripts, or the conversion of control dependences into data dependences. The link with the techniques developed in the PIPS compiler (based on array region analysis) is strong and will have to be explored.

Such over-approximations can be found by mean of abstract interpretation, a general framework to develop static analysis on real-life programs. However, they were designed mainly for verification purposes, thus precision was the main issue before scalability. Although many efforts were made in designing specialized analyses (pointers, data structures, arrays), these approaches still suffer from a lack of experimental evidence concerning their applicability for code optimization. Following our experience and work on termination analysis (that connects the work on back-end CFG-like and front-end polyhedral-like optimizations), and our work on range analysis of numerical variables and on the memory footprint on real-world C programs [9], our objective is to bridge the gap between abstract interpretation and compilation, by designing cheaper analyses that scale well, mainly based on compact representations derived from variants of static single assignment (SSA). We will focus on complex control, and complex data structures (pointers, lists) that still suffer from complexity issues in the area of optimisation.

Another possibility is to rely on application specific knowledge to guide compiler decisions. As it is impossible for a compiler alone to fully exploit such pieces of information. A possible approach to better utilize such knowledge is to put the programmers “in the loop”. Expert parallel programmers often have a good idea about coarse-grain parallelism and locality that they want to use for an application. On the other hand, fine-grain parallelism (e.g., ILP, SIMD) is tedious and specific to each underlying architecture, and is best left to the compiler. Furthermore, approximations will have opportunities to be refined using programmer knowledge. The key challenge is to create a programming environment where compiler techniques and programmer knowledge can be combined effectively. One of the difficulties is to design a usable interface between the compiler and the programmer.

3.3. Mathematical tools

All compilers have to deal with *sets* and relations. In classical compilers, these sets are finite: the set of statements of a program, the set of its variables, its abstract syntax tree (AST), its control-flow graph (CFG), and many others. It is only in the first phase of compilation, parsing, that one has to deal with infinite objects, regular and context-free languages, and those are represented by finite grammars, and are processed by a symbolic algorithm, `yacc` or one of its clones.

When tackling parallel programs and parallel compilation, it was soon realized that this position was no longer tenable. Since it makes no sense to ask whether a statement can be executed in parallel with itself, one has to consider sets of operations, which may be so large as to forbid an extensive representation, or even be infinite. The same is true for dependence sets, for memory cells, for communication sets, and for many other objects

a parallel compiler has to consider. The representation is to be *symbolic*, and all necessary algorithms have to be promoted to symbolic versions.

Such symbolic representations have to be efficient – the formula representing a set has to be much smaller than the set itself – and effective – the operations one needs, union, intersection, emptiness tests and many others – have to be feasible and fast. As a parenthesis, note that progress in algorithm design has blurred the distinction between polynomially-solvable and NP-complete problems, and between decidable and undecidable questions. For instance SAT, SMT, and ILP software tools solve efficiently many NP-complete problems, and the Z3 tool is able to "solve" many instances of the undecidable Hilbert's 10th problem.

Since the times of Pip and of the Polylib, Compsys has been active in the implementation of basic mathematical tools for program analysis and synthesis. Pip is still developed by Paul Feautrier and Cédric Bastoul, while the Polylib is now taken care of by the Inria Camus project, which introduced Ehrhart polynomials. These tools are still in use world-wide and they also have been reimplemented many times with (sometimes slight) improvements, e.g., as part of the Parma Polylib, of Sven Verdoolaege's Isl and Barvinok libraries, or of the Jollylib of Reservoir Labs. Other groups also made a lot of efforts towards the democratization of the use of polyhedral techniques, in particular the Alchemy Inria project, with Cloog and the development of Graphite in GCC, and Sadayappan's group in the USA, with the development of U. Bondhugula's Pluto prototype compiler. The same effort is made through the PPCG prototype compiler (for GPU) and Pencil (directives-based language on top of PPCG).

After 2009, Compsys continued to focus on the introduction of concepts and techniques to extend the polytope model, with a shift toward tools that may prepare the future. For instance, PoCo and C2fsm are able to parse general programs, not just SCoPs (static control programs), while the efficient handling of Boolean affine formulas [23] is a prerequisite for the construction of non-convex approximations. Euclidean lattices provide an efficient abstraction for the representation of spatial phenomena, and the construction of *critical lattices* as embedded in the tool Cl@k is a first step towards memory optimization in stream languages and may be useful in other situations. Our work on Chuba introduced a new element-wise array reuse analysis and the possibility of handling approximations. Our work on the analysis of while loops is both an extension of the polytope model itself (i.e., beyond SCoPs) and of its applications, here links with program termination and worst-case execution time (WCET) tools.

A recent example of the same approach is the proposal by Paul Feautrier to use polynomials for program analysis and optimization [5]. The associated tools are based on Handelman and Schweighofer theorems, the polynomial analogue of Farkas lemma. While this is definitely work in progress, with many unsolved questions, it has the potential of greatly enlarging the set of tractable programs.

As a last remark, observe that a common motif of these development is the transformation of finite algorithms into symbolic algorithms, able to solve very large or even infinite instances. For instance, PIP is a symbolic extension of the Simplex; our work on memory allocation is a symbolic extension of the familiar register allocation problem; loop scheduling extends DAG scheduling. Many other algorithms await their symbolic transformation: a case in point is resource-constrained scheduling.

DREAMPAL Team

3. Research Program

3.1. New Models for New Technologies

Over the past 25 years there have been several hardware-architecture generations dedicated to massively parallel computing. We have contributed to them in the past, and shall continue doing so in the Dreampal project. The three generations, chronologically ordered, are:

- Supercomputers from the 80s and 90s, based on massively parallel architectures that are more or less distributed (from the Cray T3D or Connection Machine CM2 to GRID 5000). Computer scientists have proposed methods and tools for mapping sequential algorithms to those parallel architectures in order to extract maximum power from them. We have contributed in this area in the past: http://www.lifl.fr/west/team.html.
- Parallelism pervades the chips! A new challenge appears: hardware/software co-design, in order to obtain performance gains by designing algorithms together with the parallel architectures of chips adapted to the algorithms. During the previous decade many studies, including ours in the Inria DaRT team, were dedicated to this type of co-design. DaRT has contributed to the development of the OMG MARTE standard (http://www.omgmarte.org) and to its implementation on several parallel platforms. Gaspard2, our implementation of this concept, was identified as one of the key software tools developed at Inria: http://www.inria.fr/en/centre/lille/research/platforms-and-flagship-software/flagship-software.
- The new challenge of the 2010s is, in our opinion, the integration of dynamic reconfiguration and massive parallelism. New circuits with high-density integration and supporting dynamic hardware reconfiguration have been proposed. In such architectures one can dynamically change the architecture while an algorithm is running on it. The Dynamic Partial Reconfiguration (DPR) feature offered by recent FPGA boards even allows, in theory, to generate optimized hardware at runtime, by adding, removing, and replacing components on a by-need basis. This integration of dynamic reconfiguration and massive parallelism induces a new degree of complexity, which we, as computer scientists, need to understand and deal with order to make possible the design of applications running on such architectures. This is the main challenge that we address in the Dreampal project. We note that we adress these problems as computer scientists; we do, however, collaborate with electronics specialists in order to benefit from their expertise in 3-D FPGAs.

Excerpt from the HiPEAC vision 2011/12

> *"The advent of 3D stacking enables higher levels of integration and reduced costs for off-chip communications. The overall complexity is managed due to the separation in different dies, independently designed."*

FPGAs (Field Programmable Gate Arrays) are configurable circuits that have emerged as a privileged target platform for intensive signal processing applications. FPGAs take advantage of the latest technological developments in circuits. For example, the Virtex7 from Xilinx offers a 28-nanometer integration, which is only one or two generations behind the latest general-purpose processors. 3D-Stacked Integrated Circuits (3D SICs) consist of two or more conventional 2D circuits stacked on the top of each other and built into the same IC. Recently, 3D SICs have been released by Xilinx for the Virtex 7 FPGA family. 3D integration will vastly increase the integration capabilities of FPGA circuits. The convergence of massive parallelism and dynamic reconfiguration in inevitable: we believe it is one of the main challenges in computing for the current decade.

By incorporating the configuration and/or data/program memory on the top of the FPGA fabric, with fast and numerous connections between memory and elementary logic blocks (∼10000 connections between dies), it will be possible to obtain dynamically reconfigurable computing platforms with a very high reconfiguration rate. Such a rate was not possible before, due to the serial nature of the interface between the configuration memory and the FPGA fabric itself. The FPGA technology also enables massively parallel architectures due to the large number of programmable logic fabrics available on the chip. For instance, Xilinx demonstrated 3600 8-bit picoBlaze softcore processors running simultaneously on the Virtex-7 2000T FPGA. For specific applications, picoBlaze can be replaced by specialized hardware accelerators or other IPs (Intellectual Property) components. This opens the possibility of creating massively parallel IP-based machines.

3.2. Multi-softcore on 3D FPGA

From the 2010 Xilinx white paper on FPGAs:

> *"Unlike a processor, in which architecture of the ALU is fixed and designed in a general-purpose manner to execute various operations, the CLBs (configurable logic blocks) can be programmed with just the operations needed by the application... The FPGA architecture provides the flexibility to create a massive array of application-specific ALUs..The new solution enables high-bandwidth connectivity between multiple die by providing a much greater number of connections... enabling the integration of massive quantities of interconnect logic resources within a single package"*

Softcore processors are processors implemented using hardware synthesis. Proprietary solutions include PicoBlaze, MicroBlaze, Nios, and Nios II; open-source solutions include Leon, OpenRisk, and FC16. The choice is wide and many new solutions emerge, including multi-softcore implementations on FPGAs. An alternative to softcores are hardware accelerators on FPGAs, which are dedicated circuits that are an order of magnitude faster than softcores. Between these two approaches, there are other various approaches that connect IPs to softcores, in which, the processor's machine-code language is extended, and IP invocations become new instructions. We envisage a new class of softcores (we call them reflective softcores [0]), where almost everything is implemented in IPs; only the control flow is assigned to the softcore itself. The partial dynamic reconfiguration of next-generation FPGAs makes such dynamic IP management possible in practice. We believe that efficient reflective softcores on the new 3D-FPGAs should be as small as possible: low-performance generic hardware components (ALU, registers, memory, I/O...) should be replaced by dedicated high-performance IPs.

We are developing a sofcore processor called HoMade (http://www.lifl.fr/~dekeyser/Homade) following these ideas.

In the multi-reflective softcores that we develop, some softcores will be slaves and others will be masters. Massively parallel dynamically reconfigurable architectures of softcores can thus be envisaged. This requires, additionally, a parallel management of the partial dynamic reconfiguration system. This can be done, for example, on a given subset of softcores: a massively parallel reconfiguration will replace the current replication of a given IP with the replication of a new IP. Thanks to the new 3D-FPGAs this task can be performed efficiently and in parallel using the large number of 3D communication links (Through-Silicon-Vias). Our roadmap for HoMade is to evolve towards this multi-reflective softcore model.

3.3. When Hardware Meets Software

HIPEAC vision 2011/12: *"The number of cores and instruction set extensions increases with every new generation, requiring changes in the software to effectively exploit the new features."*

[0] Hereafter, by reflective system, we mean a system that is able to modify its own structure and behaviour while it is running. A reflective softcore thus dynamically adds, removes, and replaces IPs in the application running on it, and is able to dynamically modify its own program memory, thereby dynamically altering the program it is executing.

When the new massively parallel dynamically reconfigurable architectures become reality users will need languages for programming software applications on them. The languages will be themselves dynamic and parallel, in order to reflect and to fully exploit the dynamicity and parallelism of the architectures. Thus, developers will be able to invoke reconfiguration and call parallel instructions in their programs. This expressiveness comes with a cost, however, because new classes of bugs can be induced by the interaction between dynamic reconfiguration and parallelism; for example, deadlocks due to waiting for output from an IP that does not exist any more due to a reconfiguration. The detection and elimination of such bugs before deployment is paramount for cost-effectiveness and safety reasons.

Thus, we shall build an environment for developing software on parallel, dynamically reconfigurable architectures that will include languages and adequate formal analyses and verification tools for them, in addition to more traditional tools (emulators, compilers, etc). To this end we shall be using formal-semantics frameworks associated with easy-to-use formal verification tools in order to formally define our languages of interest and allow users to formally verify their programs. The K semantic framework (http://k-framework.org), developed jointly by Univs. Urbana Champaign, USA, and Iasi, Romania) is one such framework, which is mature enough (it has allowed defining a formal semantics of the largest subset of the C language to date, as well as many other languages from essentially all programming paradigms) and is familiar to us from previous work. In K, one can rapidly prototype a language definition and try several versions of the syntax and semantics of instructions. This is important in our project, where the proposed programming languages (in particular, the HoMade assembly language) will go through several versions before being stabilized. Moreover, once a language is defined in K one gets an interpreter of the language and one gains access to formal verification tools for free. We are also developing new analysis verification tools for K (in collaboration with the K team), which will be adapted and used in the Dreampal project.

GCG Team

3. Research Program

3.1. Foundations

It has been ten years now since Intel bumped on the energy wall. Parallelism is now ubiquitous, not only restricted to expensive servers dedicated to some regular scientific computation. Also, the panel of possible mainstream architectures became extremely diverse. The use of byte-codes (e.g. nVIDIA PTX) along with Just-In-Time (JIT) compilation allowed fast evolution of designs. Quite recently, silicon companies understood that this heterogeneity should be integrated into the same chip (e.g. ARM big.LITTLE, nVIDIA Tegra K1); also re-configurable architectures (from FPGA to CGRA) are becoming present in such design as specialization is clearly useful to increase performance with less increase in energy consumption. Even cache-size, crossbar will be dynamically re-configurable; distributed DVFS being now mainstream... Postponing the decision of where and how (depending on the context) to execute part of an application, involves the use of late/adaptive compilation so as to avoid code size blowing. This observation is amplified by the fact that application behavior gets more and more dominated by data-characteristics. This is precisely what motivated more than fifteen years ago the development of dynamic compiler optimization technology. Many transformations, decisions, code-generation phases done by a compiler are now critically required to be postponed at run-time when the information is becoming available. But, this is not to mention the need of auto-tuning and adaptive compilation that imposes itself to address the increasing complexity (and hard to model) of each individual core.

The research direction of GCG is motivated by the perspective of optimizing (sometimes complex and irregular) micro-kernels for a single core (SIMD/VLIW). It starts from the observation that despite the clear motivation for JIT/dynamic compilation, despite its clear maturity, we lost the battle of performance portability: such technologies are not as optimizing as we pretended it would be. The reason for this defeat is that there is no perfect place to analyze, optimize, transform. On one hand “ JIT-ing” source-level code would usually be too slow, while on the other hand byte-code close to machine-level lost high-level semantics. Apart from spending its time to retrieve somehow obvious information, the JIT-compiler has to deal with limited resources, with realistic time constraints. Thus the need for being hybrid, in other words combine static and dynamic compilation/analysis techniques using rich intermediate languages.

Hybrid compilation consists in combining in any possible ways static analysis with profiling and run-time tests, but also ahead-of-time with run-time code optimization. This leads GCG to put efforts on researches on hybrid compilation frameworks but also on compiler architecture design. This last is to address the difficult problem of information telescoping (maintain of information of different type) and the problem of code size.

Current projects include:

- characterization of applications (I/O complexity) and profiling feedback using trace analyses;
- combined scheduling and memory allocation for irregular applications;
- extension of the polyhedral model using hybrid analysis and compilation;
- design, promotion and development of an hybrid and extensible byte-code, Tirex;
- design of a run-time handling communications, scheduling and placement for distributed memory parallel architectures.

PAREO Project-Team

3. Research Program

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 and the transition rules describing the behaviors of tree automata. Rewriting is used in semantics in order to describe the meaning of programming languages [22] as well as in program transformations like, for example, re-engineering of Cobol programs [31]. 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 [18], a theorem prover [20] or a constraint solver [21]. 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 [23], [14], [12]. We have introduced and studied the notion of *strategy* [13], 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 [28], [25], [19] 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 [16]. 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*.

POSTALE Team

3. Research Program

3.1. Architectures and program optimization

In this research topic, we focus on optimizing resources in a systematic way for the programmer by addressing fundamental issues like optimizing communication and data layout, generating automatically optimized codes via Domain Specific Languages (DSL), and auto-tuning of computer systems.

3.1.1. Optimization techniques for data and energy

3.1.1.1. Scientific context

Among the main challenges encountered in the race towards performance for supercomputers are energy (consumption, power and heat dissipation) and the memory/communication wall. This research topic addresses more specialized code analysis and optimization techniques as well as algorithmic changes in order to meet these two criteria, both from an expert - meaning handmade code transformations - or automatic - meaning compile time or run time - point of view.

Memory/communication wall means that processor elementary clock cycle decreases more rapidly over years than data transfer whether vertically between memory-ies and CPU (memory access) or horizontally between processors (data transfer). Moreover current architectures include complex memory features such as deep memory hierarchies, shared caches between cores, data alignment constraints, distributed memories etc. As a result data communication and data layout are becoming the bottleneck to performance and most program transformations aim at organizing them carefully and possibly avoiding or minimizing them. Energy consumption is also a limitation for today's processor performance. Then the options are either to design processors that consume less energy or, at the software level, to design energy-saving compilers and algorithms.

In general, the memory and energy walls are tackled with the same kind of program transformations that consist of avoiding as much as possible data communication [158] but considering these issues separately offers a different perspective. In this research axis, we focus on data/memory and energy/power optimization that include handmade or automatic compiler, code and algorithm optimizations. The resulting tools are expected to be integrated in other Postale topics related to auto-tuning [93], code generation [83] or communication-avoiding algorithms [51], [112].

3.1.1.2. Activity description and recent achievements

3.1.1.2.1. Optimization for data:

Program data transformation - data layout, data transfers. Postale has been addressing these issues in the past ANR PetaQCD project described in [63], [64] and in the PhD thesis of Michael Kruse [113]. The latter describes handmade data layout optimizations for optimizing a 4D stencil computation taking into account the BlueGene Q features. It also presents the Molly software based on the LLVM (Low Level Virtual Machine) Polly optimizing compiler that automatically generates code for MPI data transfers (see Figure 1 that shows an example of code generating a decomposition of a stencil computation into 4 subdomains and how data are exchanged between subdomains).

Data layout is still a critical point that Postale will address. The DSL [83] approach allows us to consider data layout globally, providing then an opportunity to study aggressive layouts without transformation penalty. We will also seize this opportunity to investigate the data layout problem as a new dimension of the CollectiveMind [93] optimization topic.

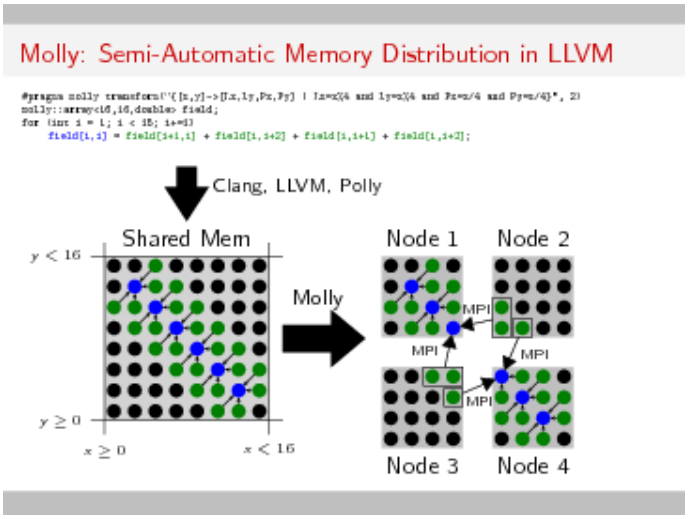

Figure 1. Automatic generation of subdomains using the Molly software.

Algorithm transformation - automating communication avoiding algorithms. This part is related to the Postale work on numerical algorithms. It originates from a research grant application elaborated with the former PetaQCD [64] team and the Inria Alpine project-team. One essential research direction consists of providing a set of high level optimizations that are generally out of reach from a traditional compiler approach. Among these optimizations, we consider communication-avoiding transformations and address the current open question of integrating these transformations in the polyhedral model in order to make them available in most software environments. Communication-avoiding algorithms improve parallelism and decrease communication requirements by ignoring some of dependency constraints at the frontiers of subdomains. Integrating communication-avoiding transformations is challenging first because these transformations change code semantics, which is unsual in program transformations, second because the validity of these transformations relies on numerical properties of the underlying transformed algorithms. This requires both compiler and algorithm skills since these transformations have important impact on the numerical stability and convergence of algorithms. Tools for the automatic generation of these transformed algorithms have two kinds of application. First, they accelerate the fastidious task of reprogramming for testing numerical properties. They may even be incorporated in an iterative tool for systematically evaluating these properties. Second, if these transformations are formalized we can consider generating different versions on line at run time, to adapt automatically algorithms to run time values [65]. In particular we plan to address s-steps algorithms [133] in iterative methods as these program transformations are similar to loop unrolling and ghosting (inverse of loop peeling). These are aggressive transformations and special preconditioning is needed in order to ensure convergence.

3.1.1.2.2. Optimizing energy:

In this topic there are two main research directions. The first one is about reversible computing based on the Landauer's conjecture that heat dissipation is produced by information erasing. The second one is on actual measurements of energy/power of program execution and on understanding which application features are the most likely to save or consume energy.

Regarding **reversible computing**, the Landauer's hypothesis - still in discussion among physicists - says that erasing one bit of information dissipates energy, independently from hardware. This implies that energy saving algorithms should avoid as much as possible erasing information: it should be possible to recover values of variables at any time in program execution. In a previous work we have analyzed the impact of

making computing DAG (Directed Acyclic Graphs) reversible [61]. We have also used reversible computing in register allocation by enabling value rematerialization also by reverse computing [62]. We are now working on characterizing algorithms by the amount of input and output data that have to be added to make algorithms reversible. We also plan to analyze mixed precision numerical algorithms [50] from this perspective.

Another research direction concerns **energy and power profiling and optimizing**. Understanding and monitoring precise energetic behavior of current programs is still a not easy task for the programmer or the compiler. One can measure it with wattmeters, or perform processor simulations or use hardware counters or sensors, or approximate it by the number of data that are communicated [159]. Especially on supercomputers or cloud framework it might be impossible to get this information. Besides making experiments on energy and power profiling [128], this research axis also includes the analysis of programming features that are the key parameters for saving energy. The ultimate goal is to have a cost model that describes the program energetic behavior of programs for the programmer or compiler being able to control it. One obvious key parameter is the count of memory accesses but one can also think of regularity features such as constant strides memory access, whether the code is statically or dynamically controlled, regularity/predictability conditional branchs. We have already performed this kind of analysis in the context of value prediction techniques where we designed entropy based criteria for estimating the predictibility of the sequence of values of some variables [129].

3.1.1.3. Research tracks for the 4 next years

Short term objectives are related to handmade or semi-automatic profiling and optimization of current scientific or image processing challenging applications. This gives a very good insight and expertise over state of the art applications and architectures. This know-how can be exploited under the form of libraries. This includes performance profiling, analysis of the energetic behavior of applications, and finding hot spots and focus optimization on these parts. This also implies to implement new numerical algorithms such as the communication-avoiding algorithms. Mid term objectives are to go forward to the automatization or semi-automatization of these techniques. Long term objectives are to understand the precise relationship between physics and computation both in programs as in reversible computing and in algorithms like in algorithmic thermodynamics [60]. The path is to define a notion of energetic complexity, which we intend to do it with the Galac team at Laboratoire de Recherche en Informatique.

3.1.2. Generative programming for new parallel architectures

3.1.2.1. Scientific context

Design, development and maintenance of high-performance scientific code is becoming one of the main issue of scientific computing. As hardware is becoming more complex and programming tools and models are proposed to satisfy constantly evolving applications, gathering expertise in both any scientific field and parallel programming is a daunting task. The natural conclusion is then to provide software design tools such that non-experts in computer science are able to produce non-trivial yet efficient codes on modern hardware architectures at their disposal. These tools can be divided in two types:

- **Compilers**. Compilers can be designed to either automatically derive parallel version of sequential codes or to support specific annotations to do so. Various successful examples include ISPC [137], SPADE [167] or GCC and its support for polyhedral compilation [140]. By offloading these tasks to compilers, the performance of the resulting codes is free of any overhead and the amount of user input is minimized. However, the scope and applicability of these techniques are fragile and can be hindered by complex code flow, inadequate data types or the use of high level languages features.
- **Libraries**. The unability of compilers to handle complex semantic is often mitigated by the design of libraries. Libraries can expose an arbitrary high level of abstraction through abstract data types and functions operating on them. User code is then expressed as a combination of function calls over instances of these data types. Different level of abstraction for parallel systems are available ranging from linear algebra [42], [109], image processing [70] to graph algorithms [153]. The main limitation of this approach is the lack of inter-procedural optimizations and the inherent divergence in API among vendors and targeted systems.

One emerging solution is to combine aspects of both solutions by designing systems which are able to provide abstraction and performance. One such approach is the design and development of **Domain Specific Languages** (or DSL) and more precisely, **Domain Specific Embedded Languages** (DSEL). DSLs [154] are non-general purpose, declarative language that simplify development by allowing users to express "the problem to solve" instead of "how to solve it". Actual code generation is then left to a proper compiler, interpreter or code generator that use high-level abstraction analysis and potential knowledge about target hardware to ensure performance. SCALA – and more precisely the FORGE tool [156] – is one of the most successful attempt at applying such techniques to parallel programming. DSELs differ from regular DSLs in the fact that they exist as a subset of an existing general purpose language. Often implemented as **Active Libraries** [166], they perform high-level optimizations based on a semantic analysis of the code before any real compilation process.

3.1.2.2. Activity description and recent achievements

In this research, we investigate the impact and applicability of software design methods based on DSELs to parallel programming and we study the portability and forward scalability of such programs. To do so, we investigate **Generative Programming** [76] applied to parallel programming.
Generative Programming is based on the hypothesis that any complex software system can be split into a list of interchangeable components (with clearly identified tasks) and a series of generators that combine components by following rules derived from an a priori domain specific analysis. In particular, we want to show that integrating the architectural support as another generative component of the set of tools leads to a better performance and an easier development on embedded or custom architecture targets (see Figure 2).

The application of Generative Programming allows us to build active libraries that can be easily re-targeted, optimized and deployed on a large selection of hardware systems. This is done by decoupling the abstract description of the DSEL from the description of hardware systems and the generation of hardware agnostic software components.
Current applications of this methodology include:

- BOOST.SIMD [84] is a C++ library for portable SIMD computations. It uses architecture aware generative programming to generate zero-overhead SIMD code on a large selection of platforms (from SSE to AVX2, Xeon Phi, PowerPC and ARM). Its interface is made so it is totally integrated into modern C++ design strategy based on the use of generic code and calls to the standard template libraries. In most cases, BOOST.SIMD delivers performance on the par with hand written SIMD code or with autovectorizers.
- NT^2 [83], [89] is a C++ library which implements a DSEL similar to MATLAB while providing automatic parallelization on SIMD systems, multicores and GPGPUs. NT^2 uses the high level of abstraction brought by the MATLAB API to detect, analyze and generate efficient loop nests taking care of every level of parallel hardware available. NT^2 eases the design of scientific computing application prototypes while delivering a significant percentage of the peak performance.

Our work uses a methodogy similar to SCALA [134], and more specifically, the DeLITE [157] toolset. Both approach rely on extracting high level, domain specific information from user code to optimize HPC applications. If our approach tries to maximize the use of compile-time optimization, DeLITE uses a runtime approach due to its reliance on the JAVA language.
In terms of libraries, various existing Scientific Computing library in C++ are actually available. The three most used are Armadillo [152], which shares a MATLAB-like API with our work, Blaze [69] which supports a similar cost based system for optimizing code and Eigen [100]. Our main feature compared to these solutions is the fact that hardware support is built-in the library core instead of beign tacked on the existing library, thus allowing us to support a larger amount of hardware.

3.1.2.3. Research tracks for the 4 next years

At short term, research and development on BOOST.SIMD and NT^2 will explore the applicability of our code generation methodology on distributed system, accelerators and heterogeneous systems. Large system support like Blue Gene/Q and other similar super-computer setup has been started.

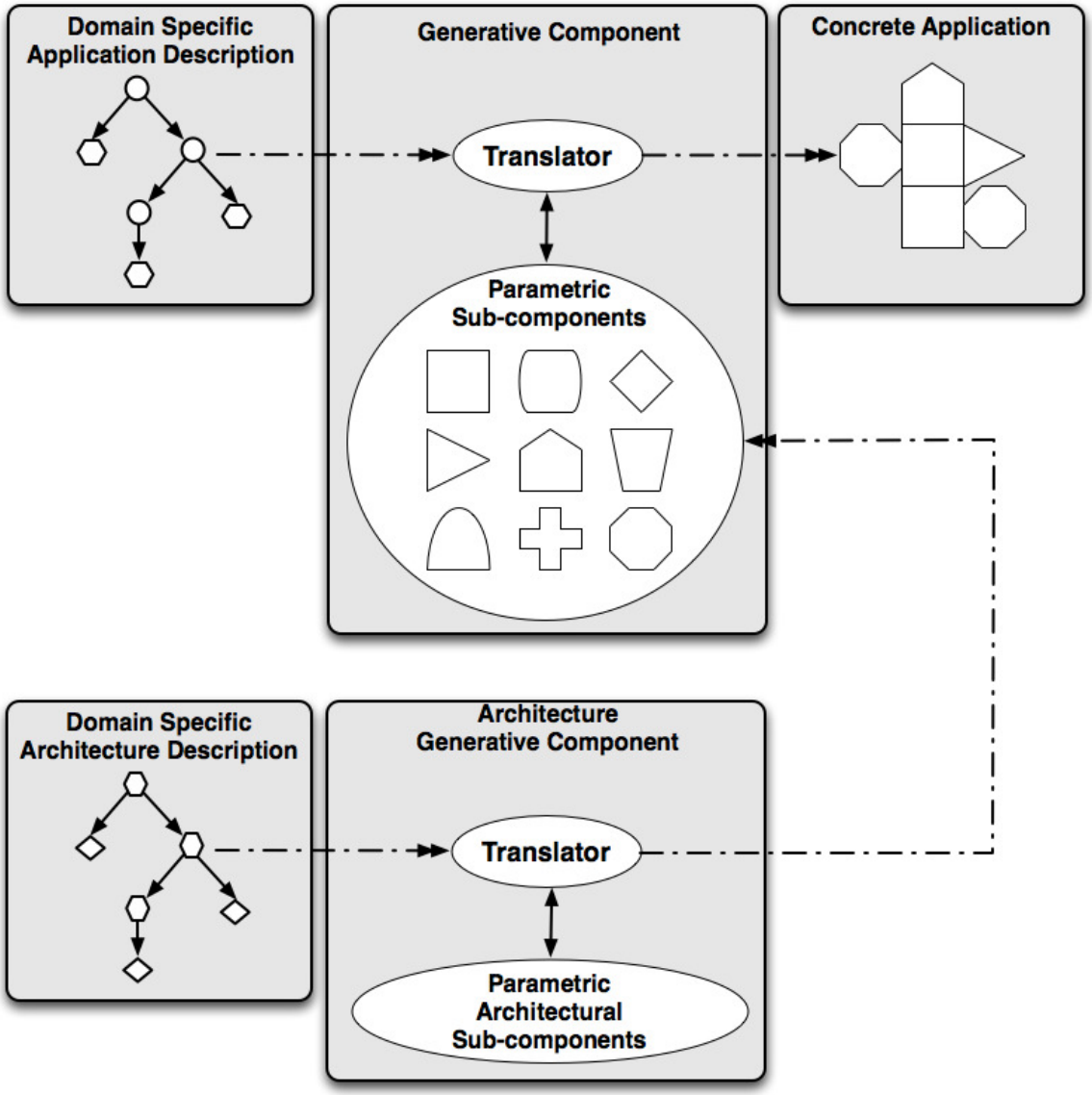

Figure 2. Principles of Architecture Aware Generative Programming

Another axis of research is to apply generative programing to other scientific domain and to propose other domain specific tools using efficient code generators. Such a work has been started to explore the impact of generative programming on the design of portable linear algebra algorithms with an going PhD thesis on automatic generation of linear algebra software.
A mid-term objective is to bridge the gap with the Data Analytics community in order to both extract new expertise on how to make Big Data related issues scalable on modern HPC hardware and to provide tools for Data Analytics practitioners based on this collaboration.
On a larger scope, the implication of our methodology on language design will be explored. First by proposing evolution to C++ (as for example with our SIMD proposal [85]) so that generative programming can become a first class citizen in the language itself. Second by exploring how this methodology can be extended to other languages [99] or to other runtime systems including Cloud computing systems and JIT support. Application to other performance metric like power consumption is also planned [171].

3.1.3. Systematizing and automating program optimization

3.1.3.1. Scientific context

Delivering faster, more power efficient and reliable computer systems is vital for our society to continue innovation in science and technology. However, program optimization and hardware co-design became excessively time consuming, costly and error prone due to an enormous number of available design and optimization choices, and complex interactions between all software and hardware components. Worse, multiple characteristics have to be always balanced at the same time including execution time, power consumption, code size, memory utilization, compilation time, communication costs and reliability using a growing number of incompatible tools and techniques with many ad-hoc and intuition based heuristics. As a result, nearly peak performance of the new systems is often achieved only for a few previously optimized and not necessarily representative benchmarks while leaving most of the real user applications severely underperforming. Therefore, users are often forced to resort to a tedious and often non-systematic optimization of their programs for each new architecture. This, in turn, leads to an enormous waste of time, expensive computing resources and energy, dramatically increases development costs and time-to-market for new products and slows down innovation [41], [39], [46], [80].

3.1.3.2. Activity description and recent achievements

For the european project MILEPOST (2006-2009) [40], we, for the first time to our knowledge, attempted to address above challenges in practice with several academic and industrial partners including IBM, CAPS, ARC (now Synopsys) and the University of Edinburgh by combining automatic program optimization and tuning, machine learning and a public repository of experimental results. As a part of the project, we established a non-profit cTuning association (cTuning.org) that persuaded the community to voluntarily support our open source tools and repository while sharing benchmarks, data sets, tools and machine learning models even after the project. This approach, highly prised by the European Commission, Inria and the international community, helped us to substitute and automatically learn best compiler optimization heuristics by crowdsourcing auto-tuning (processing a large amount of performance statistics or "big data" collected from many users to classify application and build predictive models) [40], [91], [92]. However, it also exposed even more fundamental challenges including:

- Lack of common, large and diverse benchmarks and data sets needed to build statistically meaningful predictive models;
- Lack of common experimental methodology and unified ways to preserve, systematize and share our growing optimization knowledge and research material from the community including benchmarks, data sets, tools, tuning plugins, predictive models and optimization results;
- Problem with continuously changing, “black box” and complex software and hardware stack with many hardwired and hidden optimization choices and heuristics not well suited for auto-tuning and machine learning;
- Difficulty to reproduce performance results from the cTuning.org database submitted by the community due to a lack of full software and hardware dependencies;

- Difficulty to validate related auto-tuning and machine learning techniques from existing publications due to a lack of culture of sharing research artifacts with full experiment specifications along with publications in computer engineering.

As a result, we spent a considerable amount of our "research" time on re-engineering existing tools or developing new ones to support auto-tuning and learning. At the same time, we were trying to somehow assemble large and diverse experimental sets to make our research and experimentation on machine learning and data mining statistically meaningful. We spent even more time when struggling to reproduce existing machine learning-based optimization techniques from numerous publications. Worse, when we were ready to deliver auto-tuning solutions at the end of such tedious developments, experimentation and validation, we were already receiving new versions of compilers, third-party tools, libraries, operating systems and architectures. As a consequence, our developments and results were already potentially outdated even before being released while optimization problems considerably evolved.

We believe that these are major reasons why so many promising research techniques, tools and data sets for auto-tuning and machine learning in computer engineering have a life span of a PhD project, grant funding or publication preparation, and often vanish shortly after. Furthermore, we witness diminishing attractiveness of computer engineering often seen by students as "hacking" rather than systematic science. Many recent long-term research visions acknowledge these problems for computer engineering and many research groups search for "holy grail" auto-tuning solutions but no widely adopted solution has been found yet [39], [80].

3.1.3.3. Research tracks for the 4 next years

In this project, we will be evaluating the first, to our knowledge, alternative, orthogonal, interdisciplinary, community-based and big-data driven approach to address above problems. We are developing a knowledge management system for computer engineering (possibly based on GPL-licensed cTuning and BSD-licensed Collective Mind) to preserve and share through the Internet the whole experimental (optimization) setups with all related artifacts and exposed meta-description in a unified way including behavior characteristics (execution time, code size, compilation time, power consumption, reliability, costs), semantic and dynamic features, design and optimization choices, and a system state together with all software and hardware dependencies besides just performance data. Such approach allows community to consider analysis, design and optimization of computer systems as a unified, formalized and big data problem while taking advantage of mature R&D methodologies from physics, biology and AI.

During this project, we will gradually structure, systematize, describe and share all research material in computer engineering including tools, benchmarks, data sets, search strategies and machine learning models. Researchers can later take advantage of shared components to collaboratively prototype, evaluate and improve various auto-tuning techniques while reusing all shared artifacts just like LEGO™pieces, and applying machine learning and data mining techniques to find meaningful relations between all shared material. It can also help crowdsource long tuning and learning process including classification and model building among many participants.

At the same time, any unexpected program behavior or model mispredictions can now be exposed to the community through unified web-services for collaborative analysis, explanation and solving. This, in turn, enables reproducibility of experimental results naturally and as a side effect rather than being enforced - interdisciplinary community needs to gradually find and add missing software and hardware dependencies to the Collective Mind (fixing processor frequency, pinning code to specific cores to avoid contentions) or improve analysis and predictive models (statistical normality tests for multiple experiments) whenever abnormal behavior is detected.

We hope that our approach will eventually help the community collaboratively evaluate and derive the most effective optimization strategies. It should also eventually help the community collaboratively learn complex behavior of all existing computer systems using top-down methodology originating from physics. At the same time, continuously collected and systematized knowledge ("big data") should allow community make quick and scientifically motivated advice about how to design and optimize the future heterogeneous HPC systems (particularly on our way towards extreme scale computing) as conceptually shown in Figure 3 .

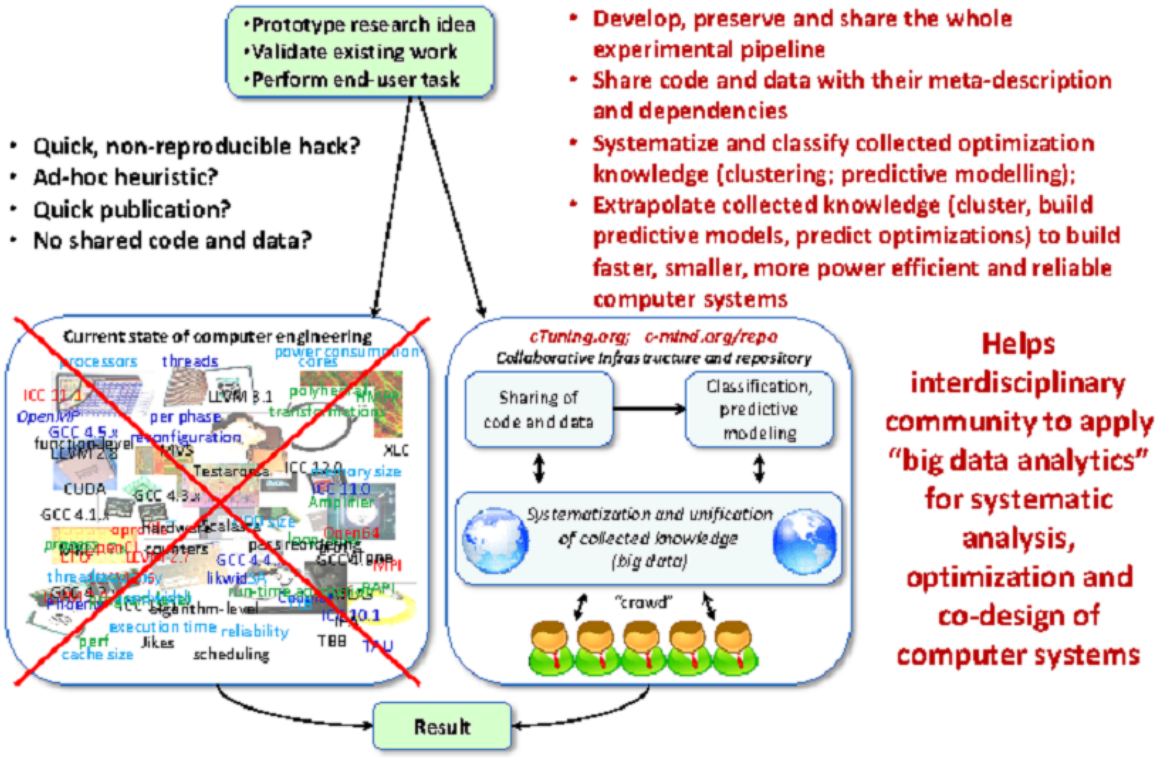

Figure 3. Considering program optimization and run-time adaptation as a "big data problem"

Similar systematization, formalization and big data analytics already revolutionized biology, machine learning, robotics, AI, and other important scientific fields in the past decade. Our approach also started revolutionizing computer engineering making it more a science rather than non-systematic hacking. It helps us effectively deal with the rising complexity of computer systems while focusing on improving classification and predictive models of computer systems' behavior, and collaboratively find missing features (possibly using new deep learning algorithms and even unsupervised learning [106], [126]) to improve optimization predictions, rather than constantly reinventing techniques for each new program, architecture and environment.

Our approach is strongly supported by a recent Vinton G. Cerf's vision for computer engineering [73] as well as our existing technology, repository of knowledge and experience, and a growing community [91], [92], [93]. Even more importantly, our approach already helped to promote reproducible research and initiate a new publication model in computer engineering supported by ACM SIGPLAN where all experimental results and related research artifacts with their meta-description and dependencies are continuously shared along with publications to be validated and improved by the community [90].

3.2. High-level HPC libraries and applications

In this research topic, we focus on developing optimized algorithms and software for high-performance scientific computing and image processing.

3.2.1. Taking advantage of heterogeneous parallel architectures

3.2.1.1. Activity description

In recent years and as observed in the latest trends from the Top 500 list [0], heterogeneous computing combining manycore systems with accelerators such as Graphics Processing Units (GPU) or Intel Xeon Phi coprocessors has become a *de facto* standard in high performance computing. At the same time, data movements between memory hierarchies and/or between processors have become a major bottleneck for most numerical algorithms. The main goal of this topic is to investigate new approaches to develop linear algebra algorithms and software for heterogeneous architectures [56], [164], with also the objective of contributing to public domain numerical linear algebra libraries (e.g., MAGMA [0]).
Our activity in the field consists of designing algorithms that minimize the cost of communication and optimize data locality in numerical linear algebra solvers. When combining different architectures, these algorithms should be properly "hybridized". This means that the workload should be balanced throughout the execution, and the work scheduling/mapping should ensure matching of architectural features to algorithmic requirements.

In our effort to minimize communication, an example concerns the solution of general linear systems (via LU factorization) where the main objective is to reduce the communication overhead due to pivoting. We developed several algorithms to achieve this objective for hybrid CPU/GPU platforms. In one of them the panel factorization is performed using a communication-avoiding pivoting heuristic [97] while the update of the trailing submatrix is performed by the GPU [51]. In another algorithm, we use a random preconditioning (see also Section 3.2.2) of the original matrix to avoid pivoting [54]. Performance comparisons and tests on accuracy showed that these solvers are effective on current hybrid multicore-GPU parallel machines. These hybrid solvers will be integrated in a next release of the MAGMA library.

Another issue is related to the impact of non-uniform memory accesses (NUMA) on the solution of HPC applications. For dense linear systems, we illustrated how an appropriate placement of the threads and memory on a NUMA architecture can improve the performance of the panel factorization and consequently accelerate the global LU factorization [148], when compared to the hybrid multicore/GPU LU algorithm as it is implemented in the public domain library MAGMA.

[0] http://www.top500.org/
[0] Matrix Algebra on GPU and Multicore Architectures, http://icl.cs.utk.edu/magma/

3.2.1.2. Research tracks for the 4 next years

3.2.1.2.1. Towards automatic generation of dense linear solvers:

In an ongoing research, we investigate a generic description of the linear system to be solved in order to exploit numerical and structural properties of matrices to get fast and accurate solutions with respect to a specific type of problem. Information about targeted architectures and resources available will be also taken into account so that the most appropriate routines are used or generated. An application of this generative approach is the possibility of prototyping new algorithms or new implementations of existing algorithms for various hardware.

A track for generating efficient code is to develop new functionalities in the C++ library NT^2 [89] which is developed in the Postale team. This approach will enable us to generate optimized code that support current processor facilities (OpenMP and TBB support for multicores, SIMD extensions...) and accelerators (GPU, Intel Xeon Phi) starting from an API (Application Programming Interface) similar to Matlab. By analyzing the properties of the linear algebra domain that can be extracted from numerical libraries and combining them with architectural features, we have started to apply the generic approach mentioned in Section 3.1.2 to solve dense linear systems on various architectures including CPU and GPU. As an application, we plan to develop a new software that can run either on CPU or GPU to solve least squares problems based on semi-normal equations in mixed precision [50] since, to our knowledge, such a solver cannot be found in current public domain libraries (Sca)LAPACK [43], [68], PLASMA [165] and MAGMA [52]. This solver aims at attaining a performance that corresponds to what state-of-the-art codes achieve using mixed precision algorithms.

3.2.1.2.2. Communication avoiding algorithms for heterogeneous platforms:

In previous work, we focused on the LU decomposition with respect to two directions that are numerical stability and communication issue. This research work has lead to the development of a new algorithm for the LU decomposition, referred to as LU_PRRP: LU with panel rank revealing pivoting [112]. This algorithm uses a new pivoting strategy based on strong rank revealing QR factorization [98]. We also design a communication avoiding version of LU_PRRP, referred to as CALU_PRRP, which aims at overcoming the communication bottleneck during the panel factorization if we consider a parallel version of LU_PRRP. Thus CALU_PRRP is asymptotically optimal in terms of both bandwidth and latency. Moreover, it is more stable than the communication avoiding LU factorization based on Gaussian elimination with partial pivoting in terms of growth factor upper bound [78].

Due to the huge number and the heterogeneity of computing units in future exascale platforms, it is crucial for numerical algorithms to exhibit more parallelism and pipelining. It is thus important to study the critical paths of these algorithms, the task decomposition and the task granularity as well as the scheduling techniques in order to take advantage of the potential of the available platforms. Our goal here is to adapt our new algorithm CALU_PRRP to be scalable and efficient on heterogeneous platforms making use of the available accelerators and coprocessors similarly to what was achieved in [51].

3.2.1.2.3. Application to numerical fluid mechanics:

In an ongoing PhD thesis [168], [169], we apply hybrid programming techniques to develop a solver for the incompressible Navier-Stokes equations with constant coefficients, discretized by the finite difference method. In this application, we focus on solving large sparse linear systems coming from the discretization of Helmholtz and Poisson equations using direct methods that represent the major part of the computational time for solving the Navier-Stokes equations which describe a large class of fluid flows. In the future, our effort in the field will concern how to apply hybrid programming techniques to solvers based on iterative methods. A major task will consist of developing efficient kernels and choosing appropriate preconditioners. An important aspect is also the use of advanced scheduling techniques to minimize the number of synchronizations during the execution. The algorithms developed during this research activity will be validated on physical data provided by the physicists either form the academic world (e.g., LIMSI/University Paris-Sud [0] or industrial partners (e.g., EDF, ONERA). This research is currently performed in the framework of the CALIFHA project [0] and will be continued in an industrial contract with EDF R&D (starting October 2014).

[0] Laboratoire d'Informatique pour la Mécanique et les Sciences de l'Ingénieur, http://www.limsi.fr/

[0] CALculations of Incompressible Fluids on Heterogeneous, funded by Région Île-de-France and Digitéo (http://www.digiteo.fr)

3.2.2. Randomized algorithms in HPC applications

Activity description

Randomized algorithms are becoming very attractive in high-performance computing applications since they are able to outperform deterministic methods while still providing accurate results. Recent advances in the field include for instance random sampling algorithms [47], low-rank matrix approximation [130], or general matrix decompositions [101].

Our research in this domain consists of developing fast algorithms for linear algebra solvers which are at the heart of many HPC physical applications. In recent works, we designed randomized algorithms [54], [66] based on random butterfly transformations (RBT) [135] that can be applied to accelerate the solution of general or symmetric indefinite (dense) linear systems for multicore [49] or distributed architectures [48]. These randomized solvers have the advantage of reducing the amount of communication in dense factorizations by removing completely the pivoting phase which inhibits performance in Gaussian Elimination.

We also studied methods and software to assess the numerical quality of the solution computed in HPC applications. The objective is to compute quantities that provide us with information about the numerical quality of the computed solution in an acceptable time, at least significantly cheaper than the cost for the solution itself (typically a statistical estimation should require $\mathcal{O}(n^2)$ flops while the solution of a linear system involves at least $\mathcal{O}(n^3)$ flops, where n is the problem size). In particular, we recently applied in [58] statistical techniques based on the small sample theory [111] to estimate the condition number of linear system/linear least squares solvers [45], [53], [57]. This approach reduces significantly the number of arithmetic operations in estimating condition numbers. Whether designing fast solvers or error analysis tools, our ultimate goal is to integrate the resulting software into HPC libraries so that these routines will be available for physicists. The targeted architectures are multicore systems possibly accelerated with GPUs or Intel Xeon Phi coprocessors.

This research activity benefits from the Inria associate-team program, through the **associate-team R-LAS**[0], created in 2014 between Inria Saclay/Postale team and University of Tennessee (Innovative Computing Laboratory) in the area of randomized algorithms and software for numerical linear algebra. This project is funded from 2014 to 2016 and is lead jointly by Marc Baboulin (Inria/University Paris-Sud) and Jack Dongarra (University of Tennessee).

Research tracks for the 4 next years

3.2.2.1. Extension of random butterfly transformations to sparse matrices:

We recently illustrated how randomization via RBT can accelerate the solution of dense linear systems on multicore architectures possibly accelerated by GPUs. We recently started to extend this method to sparse linear systems arising from the discretization of partial differential equations in many physical applications. However, a major difficulty comes from the possible fill-in introduced by RBT. One of our first task consists of performing experiments on a collection of sparse matrices to evaluate the fill-in depending on the number of recursions in the algorithm. In a recent work [59], we investigated the possibility of using another form of RBT (one-side RBT instead of two-sided) in order to minimize the fill-in and we obtain promising preliminary results (Figure 4 shows that the fill-in is significantly reduced when using one-side RBT).

Another track of research is related to iterative methods for solving large sparse linear systems, and more particularly preconditioned Krylov subspace methods implemented in the solver ARMS (Algebraic Recursive Multilevel Solver (pARMS for its parallel distributed version). In this solver, our goal is to find the last level of preconditioning and then replace the original ILU factorization by our RBT preprocessing. A PhD thesis (supervised by Marc Baboulin) started in October 2014 on using randomization techniques like RBT for sparse linear systems.

[0] Randomized Linear Algebra Software, https://www.lri.fr/~baboulin/presentation_r-las.html/

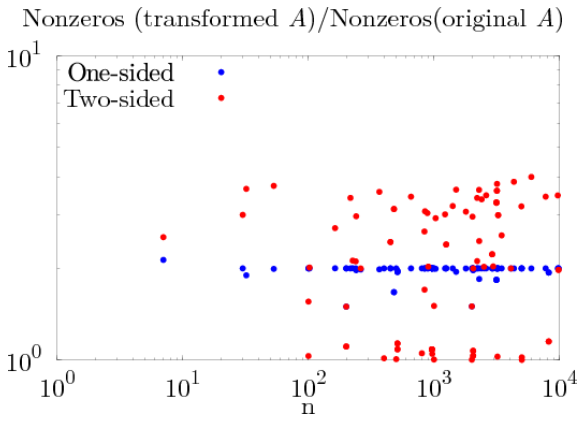

Figure 4. Evaluation of fill-in for one-sided RBT (90 matrices sorted by size).

3.2.2.2. Randomized algorithms on large clusters of multicore:

A major challenge for the randomized algorithms that we develop is to be able to solve very large problems arising in real-world physical simulations. As a matter of fact, large-scale linear algebra solvers from standard parallel distributed libraries like ScaLAPACK often suffer from expensive inter-node communication costs. An important requirement is to be able to schedule these algorithms dynamically on highly distributed and heterogeneous parallel systems [110]. In particular we point out that even though randomizing linear systems removes the communication due to pivoting, applying recursive butterflies also requires communication, especially if we use multiple nodes to perform the randomization. Our objective is to minimize this communication in the tiled agorithms and to use a runtime that enforces a strict data locality scheduling strategy [48]. A state of the art of possible runtime systems and how they can be combined with our randomized solvers will be established. Regarding the application of such solver, a collaboration with Pr Tetsuya Sakurai (University of Tsukuba, Japan) and Pr Jose Roman (Universitat Politècnica de València, Spain) will start in December 2014 to apply RBT to large linear systems encountered in contour integral eigensolver (CISS) [108]. Optimal tuning of the code will be obtained using holistic approach developed in the Postale team [93].

3.2.2.3. Extension of statistical estimation techniques to eigenvalue and singular value problems:

The extension of statistical condition estimation techniques can be carried out for eigenvalue/singular value calculations associated with nonsymmetric and symmetric matrices arising in, for example, optimization problems. In all cases, numerical sensitivity of the model parameters is of utmost concern and will guide the choice of estimation techniques. The important class of componentwise relative perturbations can be easily handled for a general matrix [111]. A significant outcome of the research will be the creation of high-quality open-source implementations of the algorithms developed in the project, similarly to the equivalent work for least squares problems [55]. To maximize its dissemination and impact, the software will be designed to be extensible, portable, and customizable.

3.2.2.4. Random orthogonal matrices:

Random orthogonal matrices have a wide variety of applications. They are used in the generation of various kinds of random matrices and random matrix polynomials [67], [77], [79], [105]. They are also used in some finance and statistics applications. For example the random orthogonal matrix (ROM) simulation [127] method uses random orthogonal matrices to generate multivariate random samples with the same mean and covariance as an observed sample.

The natural distribution over the space of orthogonal matrices is the Haar distribution. One way to generate a random orthogonal matrix from the Haar distribution is to generate a random matrix A with elements from the standard normal distribution and compute its QR factorization $A = QR$, where R is chosen to have nonnegative diagonal elements; the orthogonal factor Q is then the required matrix [104].

Stewart [155] developed a more efficient algorithm that directly generates an $n \times n$ orthogonal matrix from the Haar distribution as a product of Householder transformations built from Householder vectors of dimensions $1, 2, \cdots, n-1$ chosen from the standard normal distribution. Our objective is to design an algorithm that significantly reduces the computational cost of Stewart's algorithm by relaxing the property that Q is exactly Haar distributed. We also aim at extending the use of random orthogonal matrices to other randomized algorithms.

3.2.3. *Embedded high-performance systems & computer vision*

Scientific context

High-performance embedded systems & computer vision address the design of efficient algorithms for parallel architectures that deal with image processing and computer vision. Such systems must enforce realtime execution constraint (typically 25 frames per second) and power consumption constraint. If no COTS (*Component On The Shelf*) architecture (e.g., SIMD multicore processor, GPU, Intel Xeon Phi, DSP) satisfy the constraints, then we have to develop a specialized one.

A more and more important aspect when designing an embedded system is the tradeoff between speed (and power consumption) and numerical accuracy (and stability). Such a tradeoff leads to 16-bit computation (and storage) and to the design of less accurate algorithms. For example, the final accuracy for stabilizing an image is $10-1$ pixel, which is far from the maximum accuracy of (10^{-7}) available using the 32-bit IEEE format.

3.2.3.1. Activity description and recent achievements

Concerning image processing, our efforts concern the redesign of data-dependent algorithms for parallel architectures. A representative example of such an algorithm is the connected-component labeling (CCL) algorithm [147] which is used in industrial or medical imaging and classical computer vision like optical character recognition. As far as we know our algorithm (*Light Speed Labeling*) [71], [72] still outperforms other existing CCL algorithms [96], [103], [160] (the first versions of our algorithm appeared in 2009 [119], [120]).

Concerning computer vision (smart camera, autonomous robot, aerial drone), we developed in collaboration with LIMSI [0] two applications that run in realtime on embedded parallel systems [121], [146] with some accuracy tradeoffs. The first one is based on mean shift tracking [94], [95] and the second one relies on covariance matching and tracking [143], [144], [145].

These applications are used in video-surveillance: they perform motion detection [118], motion analysis [161], [162], motion estimation and multi-target tracking. Depending on the image nature and size, some algorithmic transforms (integral image, cumulative differential sum) can be applied and combined with hybrid arithmetic (16-bit / 32-bit / 64-bit). Finally, to increase the algorithm robustness color, space optimization is also used [122].

Usually one tries to convert 64-bit computations into 32-bit. But sometimes 16-bit floating point arithmetic is sufficient. As 16-bit numbers are now normalized by IEEE (754-2008) and are available in COTS processors like GPU and GPP (AVX2 for storage in memory and conversion into 32-bit numbers), we can run such kind of code on COTS processors or we can design specialized architectures like FPGA (*Field-Programmable gate array*) and ASIC (*Application-specific integrated circuit*) to be more efficient. This approach is complementary to that of [131] which converts 32-bit floating point signal processing operators into fixed-point ones.

[0] Laboratoire d'Informatique pour la Mécanique et les Sciences de l'Ingénieur

By extension to computer vision, we also address *interactive sensing HPC applications*. One CEA thesis funded by CEA and co-supervised by Lionel Lacassagne addresses the parallelization of Non Destructive Testing applications on COTS processors (super-charged workstation with GPUs and Intel Xeon Phi manycore processor). This PhD thesis deals with irregular computations with sparse-addressing and load-balacing problems. It also deals with floating point accuracy, by finding roots of polynomials using Newton and Laguerre algorithms. Depending on the configuration, 64-bit is required, but sometime 32-bit computations are sufficient with respect to the physics. As the second application focuses on interactive sensing, one has to add a second level of tradeoff for physical sampling accuracy and the sensor displacement [123], [124], [125], [141], [142].
In order to achieve realtime execution on the targeted architectures, we develop *High Level Transforms* (HLT) that are algorithmic transforms for memory layout and function re-organization. We show on a representative algorithm [102] in the image processing area that a fully parallelized code (SIMD+OpenMP) can be accelerated by a factor $\times 80$ on a multicore processor [115]. A CIFRE thesis (defended in 2014) funded by ST Microelectronics and supervised by Lionel Lacassagne has led to the design of very efficient implementations into an ASIC thanks to HLT. We show that the power consumption can be reduced by a factor 10 [170], [171].
All these applications have led to the development of software libraries for image processing that are currently under registration at APP (Agence de Protection des Programmes): `myNRC 2.0`[0] and `covTrack`[0].

3.2.3.2. Future: system, image & arithmetic

Concerning image processing we are designing new versions of CCL algorithms. One version is for parallel architectures where graph merging and efficient transitive closure is a major issue for load balancing. For embedded systems, *time prediction* is as important as execution time, so a specialized version targets embedded processors like ARM processors and Texas Instrument VLIW DSP C6x.

We also plan to design algorithms that should be less data-sensitive (the execution time depends on the nature of the image: a structured image can be processed quickly whereas an unstructured image will require more time). These algorithms will be used in even more data-dependent algorithms like *hysteresis thresholding* for image binarization, *split-and-merge* [44], [114] for realtime image segmentation using the Horowitz-Pavlidis quad-tree decomposition [107]. Such an algorithm could be useful for accelerating image decomposition like *Fast Level Set Transform* algorithm [132].

Concerning Computer vision we will study 16-bit floating point arithmetic for image processing applications and linear algebra operators. Concerning image processing, we will focus on iterative algorithms like optical flow computation (for motion estimation and image stabilization). We will compare the efficiency (accuracy and speed) of 16-bit floating point [86], [117], [116], [139] with fixed-point arithmetic. Concerning linear algebra, we will study efficient implementation for very small matrix inversion (from 6×6 up to 16×16) for our covariance-tracking algorithm.

According to Nvidia (see Figure 5), the computation rate (Gflop/s) for ZGEMM (complex matrix-matrix multiplication with 64-bit precision – for small value of N – is linearly proportional to N. That means that, for a 6×6 matrix, we achieve around 6 Gflop/s on a Tesla M2090 (400 Gflop/s peak power). This represents 1.5 % of the peak power. For that reason, designing efficient parallel codes for embedded systems [74], [81], [82] is different and may be more complex than designing codes for classical HPC systems. Our `covTrack` software requires many hundreds of 6×6 matrix-matrix multiplications every frame.
Last point is to develop tools that help to automatically distribute or parallelize a code on an architecture code parallelization/distribution dealing with scientific computing [83], MPI [87] or image applications on the Cell processor [75], [88], [138], [149], [150], [151], [163].

[0] smart memory allocator and management for 2D and 3D image processing
[0] agile realtime multi-target tracking algorithm, co-developped with Michèle Gouiffès at LIMSI

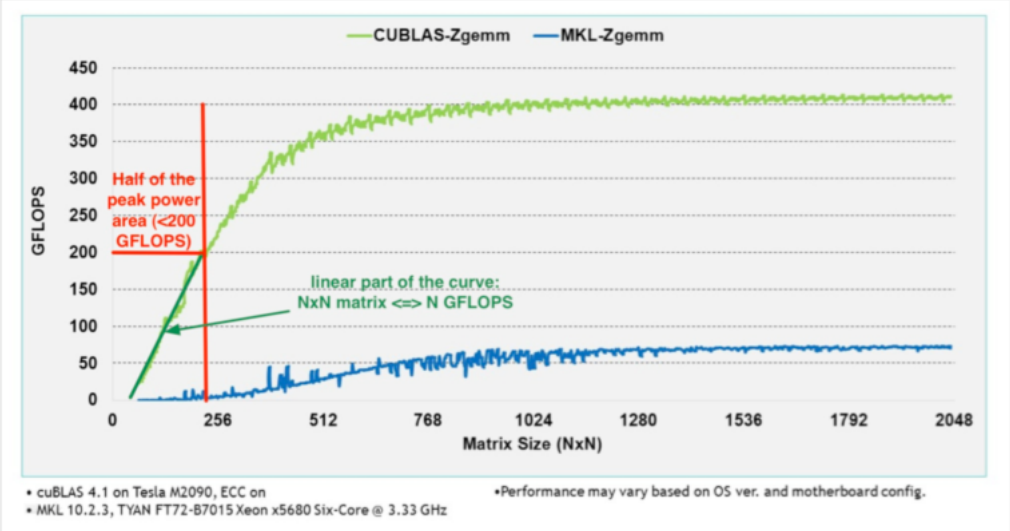

Figure 5. Nvidia cuBLAS performance versus Intel MKL: both have poor performance for small N

TASC Project-Team

3. Research Program

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://sofdem.github.io/gccat/). 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 431 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 described with first order logic.

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 combine the expressive power of global constraints with local search frameworks.

3.2.4. Solvers

- *Discrete solver* 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 initially 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). Since 2008 the main developments are done by Charles Prud'homme and Xavier Lorca. 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.).
- *Continuous solver* Since 2009 year, due to the hiring of Gilles 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).
- *Constraint programming and verification* Constraint Programming has already had several applications to verification problems. It also has many common ideas with Abstract Interpretation, a theory of approximation of the semantics of programs. In both cases, we are interested in a particular set (solutions in CP, program traces in semantics), which is hard or impossible to compute, and this set is replaced by an over-approximation (consistent domains / abstract domains). Previous works (internship of Julie Laniau, PhD of Marie Pelleau, collaboration with the Abstract Interpretation team at the ENS and Antoine Miné in particular) have exhibited some of these links, and identified some situations where the two fields, Abstract Interpretation and Constraint Programming, can complement each other. It is the case in real-time stream processing languages, where Abstract Interpretation techniques may not be precise enough when analyzing loops. With the PhD of Anicet Bart, we are currently working on using CP techniques to find loop invariants for the Faust language, a functional sound processing language.

This work around the design and the development of solvers thus forms the fourth direction of basic research of the project.

AOSTE Project-Team

3. Research Program

3.1. Models of Computation and Communication (MoCCs)

Participants: Julien Deantoni, Robert de Simone, Frédéric Mallet, 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 these 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: 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, the result of such compositional law is an algorithm graph, "architecture-aware", corresponding to refinements of the initial algorithm graph, by computing spatial (distribution) and timing (scheduling) allocations of the operations onto the architecture graph resources. In that context "Adequation" refers to some search amongst the solution space of resulting algorithm graphs, labelled by timing characteristics, for one algorithm graph 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: uniprocessor 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 the case of uniprocessor real-time scheduling, besides the classical deadline constraint, often equal to a period, we take into consideration dependences beetween tasks and several, latencies. The latter are complex related “end-to-end” constraints. Dealing with multiple real-time constraints raises the complexity of the scheduling problems. Moreover, because the preemption leads, at least, 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 of the preemption cost, 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, and we focus on the exact cost of the preemption. We have integrated this cost in the schedulability conditions that we propose, 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 cost of preemption is the most difficult part because it varies according to the instance (job) of each task.

In the case of multiprocessor real-time scheduling, we chose at the beginning the partitioned approach, rather than the global approach, since the latter allows task migrations whose cost is prohibitive for current commercial processors. The partitioned approach enables us to reuse the results obtained in the uniprocessor case in order to derive solutions for the multiprocessor case. We consider also the semi-partitioned approach which allows only some migrations in order to minimize the overhead they involve. In addition to satisfy the multiple real-time constraints mentioned in the uniprocessor case, we have to minimize the total execution time (makespan) since we deal with automatic control applications involving feedback loops. Furthermore, the domain of embedded systems leads to solving minimization resources problems. Since these optimization problems are NP-hard 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.

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).

The arrival of complex hardware responding to the increasing demand for computing power in next generation systems exacerbates the limitations of the current worst-case real-time reasoning. Our solution to overcome these limitations is based on the fact that worst-case situations may have a extremely low probability of appearance within one hour of functioning (10^{-45}), compared to the certification requirements for instance (10^{-9} for the highest level of certification in avionics). Thus we model and analyze the real-time systems using probabilistic models and we propose results that are fundamental for the probabilistic worst-case reasoning over a given time window.

CONVECS Project-Team

3. Research Program

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 [35] 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 analyzes 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 interoperability 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 analyzes.

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 a 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 make it possible 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 catalog of interoperable components serving as building blocks for new analysis features. A long-term goal of this approach is to provide an increasingly large catalog 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 catalog. 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.

HYCOMES Team

3. Research Program

3.1. Hybrid Systems Modeling

Systems industries today make extensive use of mathematical modeling tools to design computer controlled physical systems. This class of tools addresses the modeling of physical systems with models that are simpler than usual scientific computing problems by using only Ordinary Differential Equations (ODE) and Difference Equations but not Partial Differential Equations (PDE). This family of tools first emerged in the 1980's with SystemBuild by MatrixX (now distributed by National Instruments) followed soon by Simulink by Mathworks, with an impressive subsequent development.

In the early 90's control scientists from the University of Lund (Sweden) realized that the above approach did not support component based modeling of physical systems with reuse [0]. For instance, it was not easy to draw an electrical or hydraulic circuit by assembling component models of the various devices. The development of the Omola language by Hilding Elmqvist was a first attempt to bridge this gap by supporting some form of Differential Algebraic Equations (DAE) in the models. Modelica quickly emerged from this first attempt and became in the 2000's a major international concerted effort with the Modelica Consortium [0]. A wider set of tools, both industrial and academic, now exists in this segment [0]. In the EDA sector, VHDL-AMS was developed as a standard [11].

Despite these tools are now widely used by a number of engineers, they raise a number of technical difficulties. The meaning of some programs, their mathematical semantics, can be tainted with uncertainty. A main source of difficulty lies in the failure to properly handle the discrete and the continuous parts of systems, and their interaction. How the propagation of mode changes and resets should be handled? How to avoid artifacts due to the use of a global ODE solver causing unwanted coupling between seemingly non interacting subsystems? Also, the mixed use of an equational style for the continuous dynamics with an imperative style for the mode changes and resets is a source of difficulty when handling parallel composition. It is therefore not uncommon that tools return complex warnings for programs with many different suggested hints for fixing them. Yet, these "pathological" programs can still be executed, if wanted so, giving surprising results — See for instance the Simulink examples in [6], [3] and [14].

Indeed this area suffers from the same difficulties that led to the development of the theory of synchronous languages as an effort to fix obscure compilation schemes for discrete time equation based languages in the 1980's. Our vision is that hybrid systems modeling tools deserve similar efforts in theory as synchronous languages did for the programming of embedded systems.

3.2. Background on non-standard analysis

Non-Standard analysis plays a central role in our research on hybrid systems modeling [3], [6], [15], [14]. The following text provides a brief summary of this theory and gives some hints on its usefulness in the context of hybrid systems modeling. This presentation is based on our paper [3], a chapter of Simon Bliudze's PhD thesis [21], and a recent presentation of non-standard analysis, not axiomatic in style, due to the mathematician Lindström [41].

[0] http://www.lccc.lth.se/media/LCCC2012/WorkshopSeptember/slides/Astrom.pdf
[0] https://www.modelica.org/
[0] SimScape by Mathworks, Amesim by LMS International, now Siemens PLM, and more.

Non-standard numbers allowed us to reconsider the semantics of hybrid systems and propose a radical alternative to the *super-dense time semantics* developed by Edward Lee and his team as part of the Ptolemy II project, where cascades of successive instants can occur in zero time by using $\mathbb{R}_+ \times \mathbb{N}$ as a time index. In the non-standard semantics, the time index is defined as a set $\mathbb{T} = \{n\partial \mid n \in {}^*\mathbb{N}\}$, where ∂ is an *infinitesimal* and ${}^*\mathbb{N}$ is the set of *non-standard integers*. Remark that 1/ $\mathbb{T}$ is dense in $\mathbb{R}_+$, making it "continuous", and 2/ every $t \in \mathbb{T}$ has a predecessor in $\mathbb{T}$ and a successor in $\mathbb{T}$, making it "discrete". Although it is not effective from a computability point of view, the *non-standard semantics* provides a framework that is familiar to the computer scientist and at the same time efficient as a symbolic abstraction. This makes it an excellent candidate for the development of provably correct compilation schemes and type systems for hybrid systems modeling languages.

Non-standard analysis was proposed by Abraham Robinson in the 1960s to allow the explicit manipulation of "infinitesimals" in analysis [48], [35], [10]. Robinson's approach is axiomatic; he proposes adding three new axioms to the basic Zermelo-Fraenkel (ZFC) framework. There has been much debate in the mathematical community as to whether it is worth considering non-standard analysis instead of staying with the traditional one. We do not enter this debate. The important thing for us is that non-standard analysis allows the use of the non-standard discretization of continuous dynamics "as if" it was operational.

Not surprisingly, such an idea is quite ancient. Iwasaki et al. [37] first proposed using non-standard analysis to discuss the nature of time in hybrid systems. Bliudze and Krob [22], [21] have also used non-standard analysis as a mathematical support for defining a system theory for hybrid systems. They discuss in detail the notion of "system" and investigate computability issues. The formalization they propose closely follows that of Turing machines, with a memory tape and a control mechanism.

The introduction to non-standard analysis in [21] is very pleasant and we take the liberty to borrow it. This presentation was originally due to Lindstrøm, see [41]. Its interest is that it does not require any fancy axiomatic material but only makes use of the axiom of choice — actually a weaker form of it. The proposed construction bears some resemblance to the construction of $\mathbb{R}$ as the set of equivalence classes of Cauchy sequences in $\mathbb{Q}$ modulo the equivalence relation $(u_n) \approx (v_n)$ iff $\lim_{n\to\infty}(u_n - v_n) = 0$.

3.2.1. Motivation and intuitive introduction

We begin with an intuitive introduction to the construction of the non-standard reals. The goal is to augment $\mathbb{R} \cup \{\pm\infty\}$ by adding, to each x in the set, a set of elements that are "infinitesimally close" to it. We will call the resulting set ${}^*\mathbb{R}$. Another requirement is that all operations and relations defined on $\mathbb{R}$ should extend to ${}^*\mathbb{R}$.

A first idea is to represent such additional numbers as convergent sequences of reals. For example, elements infinitesimally close to the real number zero are the sequences $u_n = 1/n$, $v_n = 1/\sqrt{n}$ and $w_n = 1/n^2$. Observe that the above three sequences can be ordered: $v_n > u_n > w_n > 0$ where 0 denotes the constant zero sequence. Of course, infinitely large elements (close to $+\infty$) can also be considered, e.g., sequences $x_u = n$, $y_n = \sqrt{n}$, and $z_n = n^2$.

Unfortunately, this way of defining ${}^*\mathbb{R}$ does not yield a total order since two sequences converging to zero cannot always be compared: if u_n and u'_n are two such sequences, the three sets $\{n \mid u_n > u'_n\}$, $\{n \mid u_n = u'_n\}$, and $\{n \mid u_n < u'_n\}$ may even all be infinitely large. The beautiful idea of Lindstrøm is to enforce that *exactly one of the above sets is important and the other two can be neglected*. This is achieved by fixing once and for all a finitely additive positive measure μ over the set $\mathbb{N}$ of integers with the following properties: [0]

1. $\mu : 2^{\mathbb{N}} \to \{0, 1\}$;
2. $\mu(X) = 0$ whenever X is finite;
3. $\mu(\mathbb{N}) = 1$.

[0] The existence of such a measure is non trivial and is explained later.

Now, once μ is fixed, one can compare any two sequences: for the above case, exactly one of the three sets must have μ-measure 1 and the others must have μ-measure 0. Thus, say that $u > u', u = u'$, or $u < u'$, if $\mu(\{n \mid u_n > u'_n\} = 1)$, $\mu(\{n \mid u_n = u'_n\}) = 1$, or $\mu(\{n \mid u_n < u'_n\}) = 1$, respectively. Indeed, the same trick works for many other relations and operations on non-standard real numbers, as we shall see. We now proceed with a more formal presentation.

3.2.2. Construction of non-standard domains

For I an arbitrary set, a *filter*$\mathcal{F}$ over I is a family of subsets of I such that:

1. the empty set does not belong to $\mathcal{F}$,
2. $P, Q \in \mathcal{F}$ implies $P \cap Q \in \mathcal{F}$, and
3. $P \in \mathcal{F}$ and $P \subset Q \subseteq I$ implies $Q \in \mathcal{F}$.

Consequently, $\mathcal{F}$ cannot contain both a set P and its complement P^c. A filter that contains one of the two for any subset $P \subseteq I$ is called an *ultra-filter*. At this point we recall Zorn's lemma, known to be equivalent to the axiom of choice:

Lemma 1 (Zorn's lemma) *Any partially ordered set $(X, \leq)$ such that any chain in X possesses an upper bound has a maximal element.*

A filter $\mathcal{F}$ over I is an ultra-filter if and only if it is maximal with respect to set inclusion. By Zorn's lemma, any filter $\mathcal{F}$ over I can be extended to an ultra-filter over I. Now, if I is infinite, the family of sets $\mathcal{F} = \{P \subseteq I \mid P^c \text{ is finite}\}$ is a *free* filter, meaning it contains no finite set. It can thus be extended to a free ultra-filter over I:

Lemma 2 Any infinite set has a free ultra-filter.

Every free ultra-filter $\mathcal{F}$ over I uniquely defines, by setting $\mu(P) = 1$ if $P \in \mathcal{F}$ and otherwise 0, a finitely additive measure [0]$\mu : 2^I \mapsto \{0, 1\}$, which satisfies

$$\mu(I) = 1 \text{ and, if } P \text{ is finite, then } \mu(P) = 0.$$

Now, fix an infinite set I and a finitely additive measure μ over I as above. Let $\mathbb{X}$ be a set and consider the Cartesian product $\mathbb{X}^I = (x_i)_{i \in I}$. Define $(x_i) \approx (x'_i)$ iff $\mu\{i \in I \mid x_i \neq x'_i\} = 0$. Relation $\approx$ is an equivalence relation whose equivalence classes are denoted by $[x_i]$ and we define:

$$^*\mathbb{X} = \mathbb{X}^I / \approx \tag{1}$$

$\mathbb{X}$ is naturally embedded into $^*\mathbb{X}$ by mapping every $x \in \mathbb{X}$ to the constant tuple such that $x_i = x$ for every $i \in I$. Any algebraic structure over $\mathbb{X}$ (group, ring, field) carries over to $^*\mathbb{X}$ by almost point-wise extension. In particular, if $[x_i] \neq 0$, meaning that $\mu\{i \mid x_i = 0\} = 0$ we can define its inverse $[x_i]^{-1}$ by taking $y_i = x_i^{-1}$ if $x_i \neq 0$ and $y_i = 0$ otherwise. This construction yields $\mu\{i \mid y_i x_i = 1\} = 1$, whence $[y_i][x_i] = 1$ in $^*\mathbb{X}$. The existence of an inverse for any non-zero element of a ring is indeed stated by the formula: $\forall x \, (x = 0 \vee \exists y \, (xy = 1))$. More generally:

Lemma 3 (Transfer Principle) Every first order formula is true over $^*\mathbb{X}$ iff it is true over $\mathbb{X}$.

The above general construction can simply be applied to $\mathbb{X} = \mathbb{R}$ and $I = \mathbb{N}$. The result is denoted $^*\mathbb{R}$; it is a field according to the transfer principle. By the same principle, $^*\mathbb{R}$ is totally ordered by $[u_n] \leq [v_n]$ iff $\mu\{n \mid u_n > v_n\} = 0$. We claim that, for any finite $[x_n] \in {}^*\mathbb{R}$, there exists a unique $st([x_n])$, call it the *standard part* of $[x_n]$, such that

$$st([x_n]) \in \mathbb{R} \text{ and } st([x_n]) \approx [x_n]\,. \tag{2}$$

[0]Observe that, as a consequence, μ cannot be sigma-additive (in contrast to probability measures or Radon measures) in that it is *not* true that $\mu(\bigcup_n A_n) = \sum_n \mu(A_n)$ holds for an infinite denumerable sequence A_n of pairwise disjoint subsets of $\mathbb{N}$.

To prove this, let $x = \sup\{u \in \mathbb{R} \mid [u] \leq [x_n]\}$, where $[u]$ denotes the constant sequence equal to u. Since $[x_n]$ is finite, x exists and we only need to show that $[x_n] - x$ is infinitesimal. If not, then there exists $y \in \mathbb{R}, y > 0$ such that $y < |x - [x_n]|$, that is, either $x < [x_n] - [y]$ or $x > [x_n] + [y]$, which both contradict the definition of x. The uniqueness of x is clear, thus we can define $st([x_n]) = x$. Infinite non-standard reals have no standard part in $\mathbb{R}$.

It is also of interest to apply the general construction (1) to $\mathbb{X} = I = \mathbb{N}$, which results in the set ${}^*\mathbb{N}$ of *non-standard natural numbers*. The non-standard set ${}^*\mathbb{N}$ differs from $\mathbb{N}$ by the addition of *infinite natural numbers*, which are equivalence classes of sequences of integers whose essential limit is $+\infty$.

3.3. Contract-Based Design, Interfaces Theories, and Requirements Engineering

System companies such as automotive and aeronautic companies are facing significant difficulties due to the exponentially raising complexity of their products coupled with increasingly tight demands on functionality, correctness, and time-to-market. The cost of being late to market or of imperfections in the products is staggering as witnessed by the recent recalls and delivery delays that many major car and airplane manufacturers had to bear in the recent years. The specific root causes of these design problems are complex and relate to a number of issues ranging from design processes and relationships with different departments of the same company and with suppliers, to incomplete requirement specification and testing.

We believe the most promising means to address the challenges in systems engineering is to employ structured and formal design methodologies that seamlessly and coherently combine the various viewpoints of the design space (behavior, space, time, energy, reliability, ...), that provide the appropriate abstractions to manage the inherent complexity, and that can provide correct-by-construction implementations. The following technology issues must be addressed when developing new approaches to the design of complex systems:

- The overall design flows for heterogeneous systems and the associated use of models across traditional boundaries are not well developed and understood. Relationships between different teams inside a same company, or between different stake-holders in the supplier chain, are not well supported by solid technical descriptions for the mutual obligations.
- System requirements capture and analysis is in large part a heuristic process, where the informal text and natural language-based techniques in use today are facing significant challenges. Formal requirements engineering is in its infancy: mathematical models, formal analysis techniques and links to system implementation must be developed.
- Dealing with variability, uncertainty, and life-cycle issues, such as extensibility of a product family, are not well-addressed using available systems engineering methodologies and tools.

The challenge is to address the entire process and not to consider only local solutions of methodology, tools, and models that ease part of the design.

Contract-based design has been proposed as a new approach to the system design problem that is rigorous and effective in dealing with the problems and challenges described before, and that, at the same time, does not require a radical change in the way industrial designers carry out their task as it cuts across design flows of different type. Indeed, contracts can be used almost everywhere and at nearly all stages of system design, from early requirements capture, to embedded computing infrastructure and detailed design involving circuits and other hardware. Contracts explicitly handle pairs of properties, respectively representing the assumptions on the environment and the guarantees of the system under these assumptions. Intuitively, a contract is a pair $C = (A, G)$ of assumptions and guarantees characterizing in a formal way 1) under which context the design is assumed to operate, and 2) what its obligations are. Assume/Guarantee reasoning has been known for a long time, and has been used mostly as verification mean for the design of software [45]. However, contract based design with explicit assumptions is a philosophy that should be followed all along the design, with all kinds of models, whenever necessary. Here, specifications are not limited to profiles, types, or taxonomy of data, but also describe the functions, performances of various kinds (time and energy), and reliability. This amounts to enrich a component's interface with, on one hand, formal specifications of the behavior of the environment in

which the component may be instantiated and, on the other hand, of the expected behavior of the component itself. The consideration of rich interfaces is still in its infancy. So far, academic researchers have addressed the mathematics and algorithmics of interfaces theories and contract-based reasoning. To make them a technique of choice for system engineers, we must develop:

- Mathematical foundations for interfaces and requirements engineering that enable the design of frameworks and tools;
- A system engineering framework and associated methodologies and tool sets that focus on system requirements modeling, contract specification, and verification at multiple abstraction layers.

A detailed bibliography on contract and interface theories for embedded system design can be found in [4]. In a nutshell, contract and interface theories fall into two main categories:

Assume/guarantee contracts. By explicitly relying on the notions of assumptions and guarantees, A/G-contracts are intuitive, which makes them appealing for the engineer. In A/G-contracts, assumptions and guarantees are just properties regarding the behavior of a component and of its environment. The typical case is when these properties are formal languages or sets of traces, which includes the class of safety properties [38], [29], [44], [13], [30]. Contract theories were initially developed as specification formalisms able to refuse some inputs from the environment [36]. A/G-contracts were advocated by the SPEEDS project [16]. They were further experimented in the framework of the CESAR project [31], with the additional consideration of *weak* and *strong* assumptions. This is still a very active research topic, with several recent contributions dealing with the timed [20] and probabilistic [25], [26] viewpoints in system design, and even mixed-analog circuit design [46].

Automata theoretic interfaces. Interfaces combine assumptions and guarantees in a single, automata theoretic specification. Most interface theories are based on Lynch Input/Output Automata [43], [42]. Interface Automata [51], [50], [52], [27] focus primarily on parallel composition and compatibility: Two interfaces can be composed and are compatible if there is at least one environment where they can work together. The idea is that the resulting composition exposes as an interface the needed information to ensure that incompatible pairs of states cannot be reached. This can be achieved by using the possibility, for an Interface Automaton, to refuse selected inputs from the environment in a given state, which amounts to the implicit assumption that the environment will never produce any of the refused inputs, when the interface is in this state. Modal Interfaces [5] inherit from both Interface Automata and the originally unrelated notion of Modal Transition System [40], [12], [23], [39]. Modal Interfaces are strictly more expressive than Interface Automata by decoupling the I/O orientation of an event and its deontic modalities (mandatory, allowed or forbidden). Informally, a *must* transition is available in every component that realizes the modal interface, while a *may* transition needs not be. Research on interface theories is still very active. For instance, timed [53], [17], [19], [33], [32], [18], probabilistic [25], [34] and energy-aware [28] interface theories have been proposed recently.

Requirements Engineering is one of the major concerns in large systems industries today, particularly so in sectors where certification prevails [49]. DOORS projects collecting requirements are poorly structured and cannot be considered a formal modeling framework today. They are nothing more than an informal documentation enriched with hyperlinks. As examples, medium size sub-systems may have a few thousands requirements and the Rafale fighter aircraft has above 250,000 of them. For the Boeing 787, requirements were not stable while subcontractors performed the development of the fly-by-wire and of the landing gear subsystems.

We see Contract-Based Design and Interfaces Theories as innovative tools in support of Requirements Engineering. The Software Engineering community has extensively covered several aspects of Requirements Engineering, in particular:

- the development and use of large and rich *ontologies*; and
- the use of Model Driven Engineering technology for the structural aspects of requirements and resulting hyperlinks (to tests, documentation, PLM, architecture, and so on).

Behavioral models and properties, however, are not properly encompassed by the above approaches. This is the cause of a remaining gap between this phase of systems design and later phases where formal model based methods involving behavior have become prevalent—see the success of Matlab/Simulink/Scade technologies. We believe that our work on contract based design and interface theories is best suited to bridge this gap.

MUTANT Project-Team

3. Research Program

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 [29] with impacts both on signal processing using machine learning techniques and their application in real-world scenarios.

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 [30] and subsequents [34], 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 two consecutive lines:

- **Development of a Timed and Synchronous DSL for Real Time Musician-Computer Interaction:** The design of relevant time models and dedicated temporal interactions mechanisms are integrated in the ongoing and continuous development of the *Antescofo* language. The new tools are validated in the production of new musical pieces and other musical applications. This work is performed in strong coupling with composers and performers. The PhD works of José Echeveste (computer science) and Julia Blondeau (composer) take place in this context.
- **Formal Methods:** Failure during an artistic performance should be avoided. This naturally leads to the use of formal methods, like static analysis, verification or test generation, to ensure formally that *Antescofo* programs will behave as expected on stage. The checked properties may also provide some assistance to the composer especially in the context of "non deterministic score" in an interactive framework. The PhD of Clément Poncelet is devoted to these problems.

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 [36]. 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 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" [33].

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. Research Program

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 [30], 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, [00] an ML extension of the synchronous language LUSTRE with higher-order features, dedicated type systems (clock calculus as a type system [30], [41], initialization analysis [42] and causality analysis [44]). 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 [40], [39], and modular code generation [31], [26].

In 2000, Marc Pouzet started to collaborate with the SCADE team of Esterel-Technologies on the design of a new version of SCADE. [0] 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. [0]

In parallel, the goal of REACTIVEML[0] 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 [55], [53] and is ongoing. The language extends OCAML[0] with Esterel-like synchronous primitives — synchronous composition, broadcast communication, pre-emption/suspension — applying the solution of Boussinot [27] to solve causality issues.

[0] http://www.di.ens.fr/~pouzet/lucid-synchrone
[0] The name is a reference to Lustre which stands for "Lucid Synchrone et Temps réel".
[0] http://www.esterel-technologies.com/products/scade-suite/
[0] http://www.3ds.com/products/catia/portfolio/dymola/overview/
[0] http://rml.lri.fr/
[0] 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 [56].

REACTIVEML has been used for simulating routing protocols in ad-hoc networks [52] and large scale sensor networks [67]. 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 [66]. 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 [32], [33]. 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 [73]. It was algorithmically expensive and limited to periodic systems. In [37], 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 [54], as was a formalization of the theory in COQ [38]. 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 [57].

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 [61] (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). Further directions of research and experimentation have been and are being followed through the theses of Léonard Gérard, Adrien Guatto and Nhat Minh Lê.

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 [45]. 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 [51], [50], [68], [46] 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 [34], [47], [64], [63]. 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 [35], [34], we ported these techniques to GCC [62], [70], [69] and LLVM [49], 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 [64], [63], [65], [23], [60]. 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). A new framework has been designed, centered on the Integer Set Library (isl, http://freecode.com/projects/isl) and implemented through multiple compiler interfaces (Graphite in GCC, Polly in LLVM) and a source-to-source research compiler (PPCG) [72], [36], [48], [71]. This new framework underlies our collaborative research activities in the CARP and COPCAMS European projects, as well as emerging transfer projects through the TETRACOM European coordination action and bilateral industry contracts in preparation.

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 [58], [59]. 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.

SPADES Team

3. Research Program

3.1. Introduction

The SPADES research program is organized around three main themes, *Components and contracts*, *Real-time multicore programming*, and *Language-based fault tolerance*, that seek to answer the three key questions identified in Section 2.1 . We plan to do so by developing and/or building on programming languages and techniques based on formal methods and formal semantics (hence the use of *"sound programming"* in the project-team title). In particular, we seek to support design where correctness is obtained by construction, relying on proven tools and verified constructs, with programming languages and programming abstractions designed with verification in mind.

3.2. Components and contracts

Component-based construction has long been advocated as a key approach to the "correct-by-construction" design of complex embedded systems [53]. Witness component-based toolsets such as UC Berkeley's Ptolemy [44], Verimag's BIP [30], or the modular architecture frameworks used, for instance, in the automotive industry (AUTOSAR) [25]. For building large, complex systems, a key feature of component-based construction is the ability to associate with components a set of *contracts*, which can be understood as rich behavioral types that can be composed and verified to guarantee a component assemblage will meet desired properties. The goal in this theme is to study the formal foundations of the component-based construction of embedded systems, to develop component and contract theories dealing with real-time, reliability and fault-tolerance aspects of components, and to develop proof-assistant-based tools for the computer-aided design and verification of component-based systems.

Formal models for component-based design are an active area of research (see *e.g.*, [26], [27]). However, we are still missing a comprehensive formal model and its associated behavioral theory able to deal *at the same time* with different forms of composition, dynamic component structures, and quantitative constraints (such as timing, fault-tolerance, or energy consumption). Notions of contracts and interface theories have been proposed to support modular and compositional design of correct-by-construction embedded systems (see *e.g.*, [32], [33] and the references therein), but having a comprehensive theory of contracts that deals with all the above aspects is still an open question [58]. In particular, it is not clear how to accomodate different forms of composition, reliability and fault-tolerance aspects, or to deal with evolving component structures in a theory of contracts.

Dealing in the same component theory with heterogeneous forms of composition, different quantitative aspects, and dynamic configurations, requires to consider together the three elements that comprise a component model: behavior, structure and types. *Behavior* refers to behavioral (interaction and execution) models that characterize the behavior of components and component assemblages (*e.g.*, transition systems and their multiple variants – timed, stochastic, etc.). *Structure* refers to the organization of component assemblages or configurations, and the composition operators they involve. *Types* refer to properties or contracts that can be attached to components and component interfaces to facilitate separate development and ensure the correctness of component configurations with respect to certain properties. Taking into account dynamicity requires to establish an explicit link between behavior and structure, as well as to consider higher-order systems, both of which have a direct impact on types.

We plan to develop our component theory by progressing on two fronts: component calculi, and semantical framework. The work on typed component calculi aims to elicit process calculi that capture the main insights of component-based design and programming and that can serve as a bridge towards actual architecture description and programming language developments. The work on the semantical framework should, in the longer term, provide abstract mathematical models for the more operational and linguistic analysis afforded by component calculi. Our work on component theory will find its application in the development of a Coq-based toolchain for the certified design and construction of dependable embedded systems, which constitutes our third main objective for this axis.

3.3. Real-time multicore programming

Programming real-time systems (i.e. systems whose correct behavior depends on meeting timing constraints) requires appropriate languages (as exemplified by the family of synchronous languages [31]), but also the support of efficient scheduling policies, execution time and schedulability analyses to guarantee real-time constraints (*e.g.*, deadlines) while making the most effective use of available (processing, memory, or networking) resources. Schedulability analysis involves analyzing the worst-case behavior of real-time tasks under a given scheduling algorithm and is crucial to guarantee that time constraints are met in any possible execution of the system. Reactive programming and real-time scheduling and schedulability for multiprocessor systems are old subjects, but they are nowhere as mature as their uniprocessor counterparts, and still feature a number of open research questions [29], [41], in particular in relation with mixed criticality systems. The main goal in this theme is to address several of these open questions.

We intend to focus on two issues: multicriteria scheduling on multiprocessors, and schedulability analysis for real-time multiprocessor systems. Beyond real-time aspects, multiprocessor environments, and multicore ones in particular, are subject to several constraints *in conjunction*, typically involving real-time, reliability and energy-efficiency constraints, making the scheduling problem more complex for both the offline and the online cases. Schedulability analysis for multiprocessor systems, in particular for systems with mixed criticality tasks, is still very much an open research area.

Distributed reactive programming is rightly singled out as a major open issue in the recent, but heavily biased (it essentially ignores recent research in synchronous and dataflow programming), survey by Bainomugisha et al. [29]. For our part, we intend to focus on two questions: devising synchronous programming languages for distributed systems and precision-timed architectures, and devising dataflow languages for multiprocessors supporting dynamicity and parametricity while enjoying effective analyses for meeting real-time, resource and energy constraints in conjunction.

3.4. Language-based fault tolerance

Tolerating faults is a clear and present necessity in networked embedded systems. At the hardware level, modern multicore architectures are manufactured using inherently unreliable technologies [36], [48]. The evolution of embedded systems towards increasingly distributed architectures highlighted in the introductory section means that dealing with partial failures, as in Web-based distributed systems, becomes an important issue. While fault-tolerance is an old and much researched topic, several important questions remain open: automation of fault-tolerance provision, composable abstractions for fault-tolerance, fault diagnosis, and fault isolation.

The first question is related to the old question of “system structure for fault-tolerance” as originally discussed by Randell for software fault tolerance [65], and concerns in part our ability to clearly separate fault-tolerance aspects from the design and programming of purely “functional” aspects of an application. The classical arguments in favor of a clear separation of fault-tolerance concerns from application code revolve around reduced code and maintenance complexity [42]. The second question concerns the definition of appropriate abstractions for the modular construction of fault-tolerant embedded systems. The current set of techniques available for building such systems spans a wide range, including exception handling facilities, transaction management schemes, rollback/recovery schemes, and replication protocols. Unfortunately, these different

techniques do not necessarily compose well – for instance, combining exception handling and transactions is non trivial, witness the flurry of recent work on the topic, see *e.g.*, [52] and the references therein –, they have no common semantical basis, and they suffer from limited programming language support. The third question concerns the identification of causes for faulty behavior in component-based assemblages. It is directly related to the much researched area of fault diagnosis, fault detection and isolation [54].

We intend to address these questions by leveraging programming language techniques (programming constructs, formal semantics, static analyses, program transformations) with the goal to achieve provable fault-tolerance, i.e. the construction of systems whose fault-tolerance can be formally ensured using verification tools and proof assistants. We aim in this axis to address some of the issues raised by the above open questions by using aspect-oriented programming techniques and program transformations to automate the inclusion of fault-tolerance in systems (software as well as hardware), by exploiting reversible programming models to investigate composable recovery abstractions, and by leveraging causality analyses to study fault-ascription in component-based systems. Compared to the huge literature on fault-tolerance in general, in particular in the systems area (see *e.g.*, [49] for an interesting but not so recent survey), we find by comparison much less work exploiting formal language techniques and tools to achieve or support fault-tolerance. The works reported in [34], [37], [39], [46], [55], [64], [69] provide a representative sample of recent such works.

A common theme in this axis is the use and exploitation of causality information. Causality, i.e., the logical dependence of an effect on a cause, has long been studied in disciplines such as philosophy [60], natural sciences, law [61], and statistics [62], but it has only recently emerged as an important focus of research in computer science. The analysis of logical causality has applications in many areas of computer science. For instance, tracking and analyzing logical causality between events in the execution of a concurrent system is required to ensure reversibility [57], to allow the diagnosis of faults in a complex concurrent system [50], or to enforce accountability [56], that is, designing systems in such a way that it can be determined without ambiguity whether a required safety or security property has been violated, and why. More generally, the goal of fault-tolerance can be understood as being to prevent certain causal chains from occurring by designing systems such that each causal chain either has its premises outside of the fault model (*e.g.*, by introducing redundancy [49]), or is broken (*e.g.*, by limiting fault propagation [66]).

TEA Project-Team

3. Research Program

3.1. State of the Art

System design based on the “synchronous paradigm” has focused the attention of many academic and industrial actors on abstracting non-functional implementation details from system design. This design abstraction focuses on the logic of interaction in reactive programs rather than their timed behaviour, allowing to secure functional correctness while remaining an intuitive programming model for embedded systems.

Maintaining the “synchronous hypothesis” on software at runtime, however, demands a quasi-synchronous model of execution (hardware or middleware) in order to be effectively implemented [0]. Strong software constraints to ensure functional correctness imply strong runtime restrictions and simple hardware. If we look at recent features found in synchronous programming languages such as Quartz [0], Lucid [0] departing from the simpler semantics of Esterel [0] and Lustre [0], we observe that all try to cope in a way or another with the availability of more general execution architectures: clock domains [0], pipelining [0], streaming [0]. Unfortunately, attempts to scale the simple "typed programming language" approach of the 90’s [0] to the above purpose hit inherent computational complexity limits. For example, a periodic clock operation like $0^{(1920*(1080-480))}\{0^{1200}1^{720}\}^{480}$ in Lucy-n (0^n means n zeros) yields an exponentially larger term [0]. This explains why team TEA opts for focusing on the semantics of time and concurrency in system design and on implementing the implied design methodologies using program analysis and abstract interpretation.

By contrast with a synchronous hypothesis, the polychronous MoCC implemented in the specification language Signal, available in the Eclipse project POP [0] and in the CCSL standard [0], is inherently capable of describing circuits and systems with multiple clocks.

The Eclipse project POP provides a tooled infrastructure to refine high-level specifications into real-time streaming applications or locally synchronous and globally asynchronous systems, through a series of model analysis and synthesis libraries. These tool-supported refinement and transformation techniques can assist the system engineer from the earliest design stages of requirement specification to the latest stages of synthesis, scheduling and deployment. These characteristics make polychrony much closer to the required semantic for compositional, refinement-based, architecture-driven, system design.

3.2. Modelling Time

The elegant abstraction offered by the "synchronous hypothesis" [0] has translated in famous leitmotivs like *"computation takes no time"* and *"communication is instantaneous"* and contributed to the impact and commercial success of Esterel Studio [0] and SCADE [0].

[0] *A protocol for loosely time-triggered architectures*. A. Benveniste et al. Embedded Software Conference. ACM, 2002
[0] The Averest System http://www.averest.org.
[0] Lucid synchrone http://www.di.ens.fr/~pouzet/lucid-synchrone.
[0] *The Esterel synchronous programming language*. G. Berry, G. Gonthier. Science of Computer Programming, v. 19(2). Elsevier, 1992.
[0] *The synchronous data flow programming language Lustre*. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D. Proceedings of the IEEE v. 79(9), 1991.
[0] *A formal semantics of clock refinement in imperative synchronous languages*. Gemünde, M., Brandt, J., Schneider, K. Application of Concurrency to System Design. IEEE Press, 2010.
[0] *Parallelism with futures in Lustre*. Cohen, A., Gérard, L., Pouzet, M. Embedded Software Conference. ACM, 2012.
[0] *N-synchronous Kahn networks*. Cohen, A., et al. Principles of Programming Languages. ACM, 2006.
[0] A. Benveniste et al. *The Synchronous Languages Twelve Years Later*. Proceedings of the IEEE v. 91(1), 2003.
[0] http://www.di.ens.fr/~guatto/slides_parkas_14_05_12.pdf, page 15.
[0] *Polychrony on POLARSYS (POP)*, an Eclipse project in the POLARSYS Industry Working Group, 2013. https://www.POLARSYS.org/projects/POLARSYS.pop
[0] *Clock Constraints in UML/MARTE CCSL*. C. André, F. Mallet. Technical Report RR-6540. Inria, 2008. http://hal.inria.fr/inria-00280941
[0] *The synchronous languages 12 years later*. A. Benveniste, et al. Proceedings of IEEE, 91(1), 2003.
[0] *Esterel Studio*, Sinfora. http://www.synfora.com/products/esterelStudio.html.

Meanwhile, proposals and standards have appeared to push the technical boundaries of synchronous concurrency, in order to address a larger spectrum of concerns related to modern, heterogeneous, many-core architectures. The challenge becomes more largely about representing time in system design, alongside with many, so called, non-functional properties: cost, power, heat, speed, throughput.

One reference for the purpose of modelling timed hardware behaviour is PSL [0]. PSL is a formal specification language based on Kleene algebras that was originally designed to model regular hardware signal traces. The duality between automata and this formalism also makes it suitable to express requirements, formal properties and abstraction of program behaviours. It is widely used for modelling and verification of hardware systems.

A more recent reference of broader spectrum is CCSL [0], the clock constraints specification language of UML Marte. CCSL's core specification formalism is based on the Signal MoCC, it is synchronous and multi-clocked. Yet, CCSL supports extensions to model multi-rate, multi-periodic systems, that are adequate to represent hardware clocks, as well as asynchronous and continuous extensions (although largely unexploited in the related work). Another well-developed model is that of Ptolemy [0], which represents time as a first-class citizen alongside data carried by streams in the modelled system. It relates to the notion of PRET [0], (precision time machine) to support real-time simulation.

In the meantime, and from a totally different perspective, type theory has made considerable advances since the avent of effect systems [0] to formally represent formal properties alongside with values. Hybrid types [0] (linked to interface and contract theories), refinement types [0], value-dependant types, allow formal program properties, logical or temporal, to flow alongside with data-types during program analysis and verification. While a combination of all the above is yet unexplored, it offers an exciting venue for contributing in either/both of these fields with new theoretical developments on modelling time using principles of type theory.

3.3. Modelling Architectures

An architectural model represents components in a distributed system as boxes with well-defined interfaces, connections between ports on component interfaces, and specifies component properties that can be used in analytical reasoning about the model. Models are hierarchically organised, so that each box can contain another system with its own set of boxes and connections between them. An architecture description language for embedded systems, for which timing and resource availability form an important part of the requirements, must in addition describe resources of the system platform, such as processors, memories, communication links, etc. Several architectural modelling languages for embedded systems have emerged in recent years, including the SAE AADL [0], SysML [0], UML MARTE [0].

An architectural specification serves several important purposes. First, it breaks down a system model into manageable components to establish clear interfaces between components. In this way, complexity becomes manageable by hiding details that are not relevant at a given level of abstraction. Clear, formally defined, component interfaces allow us to avoid integration problems at the implementation phase. Connections between components, which specify how components affect each other, help propagate the effects of a change in one component to the linked components.

[0] *Scade System*, ANSYS. http://www.esterel-technologies.com/products/scade-system
[0] *IEEE Standard for Property Specification Language*. IEEE, 2005. http://dx.doi.org/10.1109/IEEESTD.2005.97780.
[0] *CCSL: specifying clock constraints with UML/MARTE*, OMG, 2008. http://www.omgmarte.org/node/66.
[0] *Ptolemy*, UC Berkeley. http://ptolemy.eecs.berkeley.edu.
[0] *Precision Timed Computation in Cyber-Physical Systems*. E. A. Lee and S. A. Edwards, 2007. http://ptolemy.eecs.berkeley.edu/publications/papers/07/PRET.
[0] *Polymorphic effect systems*. J. M. Lucassen , D. K. Gifford. Principles of Programming Languages. ACM, 1988.
[0] *Hybrid type checking*. K.W. Knowles and C. Flanagan. ACM Transactions on Programming languages and systems, 32(2). ACM, 2010
[0] *Abstract Refinement Types*. N. Vazou, P. Rondon, and R. Jhala. European Symposium on Programming. Springer, 2013.
[0] *Architecture Analysis and Design Language*, AS-5506. SAE, 2004. http://standards.sae.org/as5506b
[0] *System Modelling Language*. OMG, 2007. http://www.omg.org/spec/SysML
[0] *UML Profile for MARTE*. OMG, 2009. http://www.omg.org/spec/MARTE

Most importantly, an architectural model is a repository to share knowledge about the system being designed. This knowledge can be represented as requirements, design artefacts, component implementations, held together by a structural backbone. Such a repository enables automatic generation of analytical models for different aspects of the system, such as timing, reliability, security, performance, energy, etc. Since all the models are generated from the same source, the consistency of assumptions w.r.t. guarantees, of abstractions w.r.t. refinements, used for different analyses becomes easier, and can be properly ensured in a design methodology based on formal verification and synthesis methods.

Related works in this aim, and closer in spirit to our approach (to focus on modelling time) are domain-specific languages such as Prelude [0] to model the real-time characteristics of embedded software architectures. Conversely, standard architecture description languages could be based on algebraic modelling tools, such as interface theories with the ECDAR tool [0].

3.4. Time Scheduling

Cyber-physical systems are reactive systems whose correctness not only depends on a deterministic behavior but also on timing predictability. The timing parameters of a CPS are requirements that arise from the system's specification (e.g. minimum throughput, maximum latency, deadlines) or timing properties of the physical and cyber-parts that restrict the CPS implementation. The design of a CPS must ensure that these timing requirements will be met, even in the worst-case scenario, through the different components and their timing properties.

3.4.1. Scheduling theory

Real-time scheduling theory provides tools for predicting the timing behaviour of a CPS which consists of many interacting software and hardware components. Expressing parallelism among software components is a crucial aspect of the design process of a CPS. It allows for efficient partition and exploitation of available resources. In the real-time scheduling theory literature, many models of computation have been proposed to express such parallelism, for instance:

- Set of independent periodic, sporadic, or aperiodic tasks where each real-time task is generally characterised with some timing parameters: deadline, period, first start time, jitter, etc. The periodic and sporadic task models [0] are very well studied task models since they allow to analytically reason about the timing behaviour of tasks. More expressive task models [0] such as the multi-frame and the recurring real-time task models have also emerged.
- Task graph models [0] where precedence constraints among real-time tasks may exist.
- Data-flow graph models such as synchronous data-flow (SDF [0]) and cyclo-static dataflow (CSDF [0]. IEEE, 1996.) models where the set of tasks (also called actors) communicate with each other through FIFO channels. When it fires, an actor consumes a predefined number of tokens from its inputs and produces a predefined number of tokens on its outputs. The scheduling problem is hence more complex since data dependencies must be satisfied.

[0] *The Prelude language*. LIFL and ONERA, 2012. http://www.lifl.fr/~forget/prelude.html
[0] *PyECDAR, timed games for timed specifications*. Inria, 2013. https://project.inria.fr/pyecdar
[0] *Scheduling algorithms for multiprogramming in a hard-real-time environment*. C. L. Liu and J. W. Layland. Journal of the ACM 20(1), 1973.
[0] *The digraph real-time task model*. M. Stigge, P. Ekberg, N. Guan, and W. Yi. Real-Time and Embedded Technology and Applications Symposium. IEEE, 2011.
[0] *Task graph scheduling using timed automata*. Y. Abdeddaïm, A. Kerbaa, and O. Maler. International Symposium on Parallel and Distributed Processing. IEEE, 2003.
[0] *Synchronous data-flow*. E. A. Lee and D. G. Messerschmitt. Proceedings of the IEEE, 1987.
[0] *Cycle-static dataflow*. G. Blisen, M. Engels, R. Lauwereins, and J. Peperstraete. Transactions on Signal Processing

The literature about real-time scheduling of sets of independent real-time tasks [0] provides very mature schedulability tests regarding many scheduling strategies, preemptive or non-preemptive scheduling, uniprocessor or multiprocessor scheduling, etc. Historically, real-time systems where scheduled by cyclic executives (i.e. static scheduling). However, since this approach produces rigid and difficult to maintain systems and handles only periodic tasks, the research community has proposed many dynamic scheduling algorithms, which can be classified as fixed-priority scheduling (e.g. rate-monotonic scheduling, deadline monotonic scheduling) and dynamic priority scheduling (e.g. earliest-deadline first scheduling, least laxity scheduling). Multiprocessor scheduling can be further classified as partitioned scheduling (each task is allocated to a processor and no migration is allowed), global scheduling (a single job can migrate to and execute on different processors), or hybrid.

Scheduling of data-flow graphs has also been extensively studied in the past decades. Static-periodic scheduling is the main scheduling approach, which consists in infinitely repeating a firing sequence of actors. This problem has been addressed with respect to many performance criteria: throughput maximisation [0], latency minimisation [0], buffer minimisation [0], code size minimisation [0], etc. Recently, real-time dynamic scheduling (fixed-priority and earliest-deadline first scheduling) of data-flow graphs has been addressed where actors are mapped to periodic real-time tasks and existing schedulability tests are adapted to synthesise the timing characteristics of actors [00]

3.5. Virtual Prototyping

Virtual Prototyping is the technology of developing realistic simulators from models of a system under design; that is, an emulated device that captures most, if not all, of the required properties of the real system, based on its specifications. A virtual prototype should be run and tested like the real device. Ideally, the real application software would be run on the virtual prototyping platform and produce the same results as the real device with the same sequence of outputs and reported performance measurements. This may be true to some extent only. Some trade-offs have often to be made between the accuracy of the virtual prototype, and time to develop accurate models.

A virtual prototyping platform must include operating system or hardware emulation technology since the hardware functions must be simulated at least to a minimum extent in order to run the software and evaluate the design alternatives. The hardware simulation engine is a key component of a virtual prototyping platform, which makes it possible to run the application software and produce output that can be analysed by other tools. Because electronic design tools (EDAs) simulate the hardware in every detail, it is possible to verify that the circuit operates properly and also to measure how many clock cycles will be required to achieve an operation. But because they simulate very low-level operations, simulation is much too slow to be usable for virtual prototyping. The authors of the FAST system [0] and SocLib project reports [0] speed-ups with a factor of several

[0] *A survey of hard real-time scheduling for multiprocessor systems*. R. I. Davis and A. Burns. *ACM Computing Survey* 43(4), 2011.

[0] *Throughput analysis of synchronous data-flow graphs*. Ghamarian, A.H. et al. Application of Concurrency to System Design. IEEE, 2006

[0] *Latency minimization for synchronous data flow graphs*. A. H. Ghamarian, et al. Conference on Digital System Design Architectures, Methods and Tools. Euromicro, 2007.

[0] *Minimal memory schedules for data-flow networks*. M. Cubric and P. Panangaden. International Conference on Concurrency Theory. Springer, 1993.

[0] *Looped schedules for dataflow descriptions of multirate signal processing algorithms*. S. S. Bhattacharyya and E. A. Lee. Journal of Formal Methods in System Design. Kluwer, 1994.

[0] *Affine data-flow graphs for the synthesis of hard real-time applications*. A. Bouakaz, J.-P. Talpin, and J. Vitek. International Conference on Application of Concurrency to System Design. IEEE Press, 2012.

[0] *Hard-real-time scheduling of data-dependent tasks in embedded streaming applications*. M. Bamakhrama and T. Stefanov. International Conference on Embedded Software. ACM, 2011.

[0] *The fast methodology for high-speed SOC simulation.* D. Chiou, et al. International conference on Computer-aided design. IEEE, 2007.

[0] *Using binary translation in event driven simulation for fast and flexible MPSOC simulation.* M. Gligor, N. Fournel, and F. Pétrot. In CODES+ISSS, IEEE, 2009.

hundreds in a comparison between their cycle accurate simulator and their virtual prototyping framework. A factor of the order of 100 times faster than EDA tools is required for virtual prototyping.

In order to speed-up simulation time, the virtual prototype must trade-off with something. Depending upon the application designers goals, one may be interested in trading some loss of accuracy in exchange for simulation speed, which leads to constructing simulation models that focus on some design aspects and provide abstraction of others. A simulation model can provide an abstraction of the simulated hardware in three directions:

- *Computation abstraction*. A hardware component computes a high level function by carrying out a series of small steps executed by composing logical gates. In a virtual prototyping environment, it is often possible to compute the high level function directly by using the available computing resources on the simulation host machine, thus abstracting the hardware function.
- *Communication abstraction*. Hardware components communicate together using some wiring, and some protocol to transmit the data. Simulation of the communication and the particular protocol may be irrelevant for the purpose of virtual prototyping: communication can be abstracted into higher level data transmission functions.
- *Timing Abstraction*. In a cycle accurate simulator, there are multiple simulation tasks, and each task makes some progress on each clock cycle, but this is slowing down the simulation. In a virtual prototyping experiment, one may not need to so precise timing information: coarser time abstractions can be defined allowing for faster simulation.

The cornerstone of a virtual prototyping platform is the component that simulates the processor(s) of the platform, and its associated peripherals. Such simulation can be *static* or *dynamic*.

3.6. Research Objectives

The challenges addressed by team TEA support the claim that sound Cyber-Physical System design (including embedded, reactive, and concurrent systems altogether) should consider (logical, formal) time modelling as a central aspect.

In this aim, architectural specifications found in software engineering are a natural focal point to start from. Architecture descriptions organise a system model into manageable components, establish clear interfaces between them, and help correct integration of these components during system design.

The definition of a formal design methodology to support the heterogeneous modelling of time in architecture descriptions demands the elaboration of sound mathematical foundations and the development of formal calculi methods to instrument them that constitute the research program of team TEA.

3.6.1. Objective n. 1 – Semantics and specification of time in system design

Time systems. To mitigate and generalise algebraic representations of time, we propose to introduce the paradigm of "time system" (type systems to represent time). Just as a type system abstracts data carried along operations in a program, a time system abstracts the causal interaction of that program module or hardware element with its environment, its pre and post conditions, its assumptions and guarantees, either logical or numerical. Instances of the concept of time system we envision are the clock calculi found in data-flow synchronous languages like Signal, Lustre and its different incarnations. All are bound to a particular model of time.

To gain generality and compositionality, we wish to proceed from recent developments on hybrid types [0] (linked to interface and contract theories), refinement types [0], value-dependant type [0] theories, to formally define a time system.

[0] *Hybrid type checking*. K.W. Knowles and C. Flanagan. ACM Transactions on Programming languages and systems, 32(2). ACM, 2010

[0] *Abstract Refinement Types*. N. Vazou, P. Rondon, and R. Jhala. European Symposium on Programming. Springer, 2013.

[0] *Secure distributed programming with value-dependent types*. N. Swamy, et al. International Conference on Functional Programming. Springer, 2011.

The principle of these type systems is to allow data-types inferred in the program with properties, possibly temporal, pertaining, for instance, to the algebraic domain on their value, or any algebraic property related to its computation: effect, memory usage [0], pre-post condition, value-range, cost, speed, time.

In the quest of an appropriate algebra for time, we are studying both the CCSL and PSL standards and, more generally, Kleene algebras [0] which offer greater expressivity in the prospect of timed specification as well as refinement checking and verification [00].

Being grounded on type and domain theories, a time system can naturally be equipped with program analysis techniques based on type inference (for data-type inference) or abstract interpretation (for program properties inference) [0]. We intend to use and learn from existing open-source implementations in this field of research [0] in order to prototype our solution.

Relating time systems. Just as a time system formally represents the timed behaviour of a given component, timing relations (abstraction and refinement) represent interaction among components. Logically, their specification should be the role of a module system, and verifying their conformance that of a module checking algorithm.

Scalability and compositionality dictate the use of assume-guarantee reasoning, as found in interface automata and contract algebra, in order to facilitate composition by behavioural sub-typing, in the spirit of the (static) contract-based formalism proposed by Passerone et al. [00].

To further elaborate a formal verification approach, we will additionally consider notions of refinement calculi based on temporal logic [0], in order to possibly extend our interface and contract theories with liveness properties. The definition of a module/interface for timed architectures should hence proceed directly from the definition of its time system, using mostly existing theoretical results on the matter of module systems, interface and contract theories.

Conformance of time relations. Verification problems encompassing heterogeneously timed specifications are common and of great variety: checking correctness between abstract and concrete time models relates to desynchronisation (from synchrony to asynchrony) and scheduling analysis (from synchrony to hardware). More generally, they can be perceived from heterogeneous timing viewpoints (e.g. mapping a synchronous-time software on a real-time middleware or hardware).

This perspective demands capabilities not only to inject time models one into the other (by abstract interpretation, using refinement calculi), to compare time abstractions one another (using simulation, refinement, bisimulation, equivalence relations) but also to prove more specific properties (synchronisation, determinism, endochrony).

---

[0] *Region-based memory management*. Tofte, M., Talpin, J.-P. Information and Computation, 132(2). Academic Press, 1997.

[0] *Automated reasoning in Kleene algebra*. P. Höfner and G. Struth. Conference on Automated Reasoning. Springer, 2007.

[0] *Algebraic Verification Method for SEREs Properties via Groebner Bases Approaches*. N. Zhou, J. Wu, X. Gao. Journal of Applied Mathematics. Hindawi, 2013

[0] *From monadic logic to PSL*. M. Y. Vardi. Pillars of Computer Science, 2008.

[0] *Timed polyhedra analysis for synchronous languages*. Besson, F., Jensen, T., Talpin, J.-P. Static Analysis Symposium. Springer, 1999.

[0] The Microsoft F* project, https://research.microsoft.com/en-us/projects/fstar.

[0] *A contract-based formalism for the specification of heterogeneous systems*. L. Benvenistu, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, C. Sofronis. Forum on design languages, 2008

[0] *Moving from Specifications to Contracts in Component-Based Design.* S. Bauer, A. David, R. Hennicker, K. Larsen, A. Legay, U. Nyman, A. Wasowski. Fundamental Aspects in Software Engineering. Springer, 2012

[0] *Refinement Calculus: A Systematic Introduction*. R.J. Back, J. von Wright. Springer, 1998.

In the spirit of our recent work developing an abstract scheduling theory, we want to develop a method of abstract interpretation [0] to reason about the abstraction and refinement of heterogeneous timed specifications in the aim of checking their conformance. A source of inspiration in that prospect is the notion of contract abstraction [0]. To this end, we plan to use SAT-SMT solving techniques to check conformance of abstracted time constraints, in a way which we previously experienced with the automated code generation validation of Polychrony [000].

To check conformance between heterogeneously timed specifications, we will consider variants of the abstract interpretation framework proposed by Bertrane et al. [0] to inject properties from one time domain into another, be it continuous [0] or discrete [0].

This will for instance enable the possibility of verifying cross-domain properties, e.g. cost v.s. power v.s. performance v.s. software mapping. This will allow to formalise intuitions such as that this typical inter-domain constraint: the cost of a system has an impact on the system's controlability; and allow to formally explain why: lower cost means hardware with lower performances, which means longer WCRTs, which means longer end-to-end latency, which may result in a response-time longer than controllability limits. This particular topic (which we could call cross-domain conformance checking) has not been studied in the related literature (on contract-based design, for instance), and could be based on both abstraction techniques, e.g. linear abstractions, or morphisms between domains or even discrete relations, e.g. a simple catalog or "price list" relating price and performance for a data-base of hardware components.

3.6.2. Objective n. 2 – A standard for modelling time in system design

A second objective, to be developed in parallel and synergy to objective n. 1, is the definition of an architecture-specific specification formalism, that would serve as semantic foundation, structure and repository for tooling a component-based design methodology with semantic analysis, to synthesise component interfaces, and formal methods, to verify specified requirements.

In project TEA, it will take form by the definition and tooling of a time annex for the AADL standard, based on the theory developed in objective n. 1. The aim of the AADL time annex is to formalise the logical and physical timing properties of architecture models and represent them as constraints expressed using regular grammars (like in PSL), or using the process calculus of CCSL.

This is an objective reminiscent and in direct application of the principle of time system (objective n.1). We not only want to model time in the heterogeneous logical and physical constituents in an AADL specification, but relate them, and verify the correctness of their composition.

Our aim is to start from the modelling standards AADL and CCSL to define a standard for time in system design. Our contribution will be formalised by a timing annex for the AADL and tools collaboratively developed to support its use. Our first milestone in this prospect is a report [0] of recommendations accepted by the AADL committee. Our next step, the submission of a time annex by team TEA at the SAE consortium, will

[0] *La vérification de programmes par interprétation abstraite*. P. Cousot. Séminaire au Collège de France, 2008.

[0] *Compositional contract abstraction for system design*. A. Benveniste, D. Nickovic, T. Henzinger.

[0] *Efficient deadlock detection for polychronous data-flow specifications*. C. Ngo, J.-P. Talpin, T. Gautier. Electronic System Level Synthesis Conference (ESLSYN'14). IEEE, 2014.

[0] *Formal verification of synchronous data-flow program transformations toward certified compilation*. V.-C. Ngo, J.-P. Talpin, Gautier, P. Le Guernic, L. Besnard. Frontiers of Computer Systems. Springer, 2013.

[0] *Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction*. P. Feautrier, A. Gamatié and L. Gonnord. Journal of Computing, 1(4). Computer Society of India, 2012.

[0] *Temporal Abstract Domains*. J. Bertrane. International Conference on Engineering of Complex Computer Systems. IEEE, 2011

[0] *Abstract Interpretation of the Physical Inputs of Embedded Programs*. O. Bouissou, M. Martel. Verification, Model Checking, and Abstract Interpretation. LNCS 4905, Springer, 2008

[0] *Proving the Properties of Communicating Imperfectly-Clocked Synchronous Systems*. J. Bertrane. Static Analysis Symposium. Springer, 2006

[0] *"Logically timed specifications in the AADL – Recommendations to the SAE committee on AADL*. L. Besnard, E. Borde, P. Dissaux, T. Gautier, P. Le Guernic, J.-P. Talpin, H. Yu. Inria Technical Report n.446, 2014.

employ the principles exposed in objective n.1 in order to formally define a modular and scalable specification formalism to specify heterogeneous timing constraints in the AADL.

Then, the specification of timing relations between AADL objects will be made explicit by contracts. Together with these contracts, we will then formally define abstraction and refinement relation in order to inject properties assumed by one component into the time model guaranteed by another, and vice versa. Lastly, conformance-checking abstracted contracts will be supported by state-of-the-art verification tools. This all will define a design methodology for time in the AADL, and our very last step will be to tool this methodology and provide a reference implementation.

3.6.3. Objective n. 3 – Applications to real-time scheduling

As a prime application of formal methods for interacting time models, scheduling thousands of program blocks or modules found on modern embedded architecture poses a challenging problem. It simply defies known bounds of complexity theory in the field. It is an issue that requires a particular address, because it would find direct industrial impact in present collaborative projects in which we are involved.

One recent milestone in the prospect of large-scale scheduling is the development of abstract affine scheduling [0]. It consists, first, of approximating threads communication patterns in Safety-Critical Java using cyclo-static data-flow graphs and affine functions. Then, it uses state of the art ILP techniques to find optimal schedules and concretise them as real-time schedules for Safety Critical Java programs [00]

To develop the underlying theory of this promising research topic, we first need to deepen the theoretical foundation to establish links between scheduling analysis and abstraction interpretation [0].

The theory of time system developed in objective n.1 offers the ideal framework to pursue this development. It amounts to representing scheduling constraints, inferred from programs, as types. It allows to formalise the target time model of the scheduler (the architecture, its middle-ware, its real-time system) and defines the basic concepts to verify assumptions made in one with promises offered by the other: contract verification or, in this case, synthesis. Objective n.3 is hence defined as a direct application of objective n.1.

3.6.4. Objective n. 4 – Applications to virtual prototyping

A solution usually adopted to handle time in virtual prototyping is to manage hierarchical time scales, use component abstractions where possible to gain performance, use refinement to gain accuracy where needed. Localised time abstraction may not only yield faster simulation, but facilitate also verification and synthesis (e.g. synchronous abstractions of physically distributed systems). Such an approach requires computations and communications to be harmoniously discretised and abstracted from originally heterogeneous viewpoints onto a structuring, articulating, pivot model, for concerted reasoning about time and scheduling of events in a way that ensures global system specification correctness.

Just as model checking usually employs goal-directed abstraction techniques, in order to approximate parts of the model that are not in the path of the property to check, we plan to equivalently define, possibly semi-automate, abstraction techniques to approximate the time model of system components that do not directly influence timing properties to evaluate.

In the short term these component models could be based on libraries of predefined models of different levels of abstractions. Such abstractions are common in large programming workbench for hardware modelling, such as SystemC, but less so, because of the engineering required, for virtual prototyping platforms. Additionally, the level of abstraction required to simulate components could simply (and best) be specified manually by annotating the architecture specification.

[0] *Buffer minimization in earliest-deadline first scheduling of dataflow graphs*. A. Bouakaz and J.-P. Talpin. Conference on Languages, Compilers and Tools for Embedded Systems. ACM, June 2013.

[0] *Affine data-flow graphs for the synthesis of hard real-time applications*. A. Bouakaz, J.-P. Talpin, and J. Vitek. Application of Concurrency to System Design. IEEE Press, June 2012.

[0] *Design of Safety-Critical Java Level 1 Applications Using Affine Abstract Clocks*. A. Bouakaz and J.-P. Talpin. International Workshop on Software and Compilers for Embedded Systems. ACM, June 2013.

[0] *Abstraction-Refinement for Priority-Driven Scheduling of Static Dataflow Graphs*. Submitted for publication, 2014.

The approach of team TEA provides an additional ingredient in the form of rich component interfaces. It therefore dictates to further investigate the combined use of conventional virtual prototyping libraries, defined as executable abstractions of real hardware, with executable component simulators synthesised from rich interface specifications (using, e.g., conventional compiling techniques used for synchronous programs).

Just as virtual integration consists of synthesising the verification model of an architecture specification, virtual prototyping can be seen as synthesising an executable simulator from a model in, e.g., the spirit of the A-350 DMS case study that was realised by team ESPRESSO in the frame of Artemisia project CESAR [0].

[0] *System-level co-simulation of integrated avionics using polychrony*. Yu, H., Ma, Y., Glouche, Y., Talpin, J.-P., Besnard, L., Gautier, T., Le Guernic, P., Toom, A., and Laurent, O. ACM Symposium on Applied Computing. ACM, 2011.

ANTIQUE Team

3. Research Program

3.1. Semantics

Semantics plays a central role in verification since it always serves as a basis to express the properties of interest, that need to be verified, but also additional properties, required to prove the properties of interest, or which may make the design of static analysis easier.

For instance, if we aim for a static analysis that should prove the absence of runtime error in some class of programs, the concrete semantics should define properly what error states and non error states are, and how program executions step from a state to the next one. In the case of a language like C, this includes the behavior of floating point operations as defined in the IEEE 754 standard. When considering parallel programs, this includes a model of the scheduler, and a formalization of the memory model.

In addition to the properties that are required to express the proof of the property of interest, it may also be desirable that semantics describe program behaviors in a finer manner, so as to make static analyses easier to design. For instance, it is well known that, when a state property (such as the absence of runtime error) is valid, it can be established using only a state invariant (i.e., an invariant that ignores the order in which states are visited during program executions). Yet searching for trace invariants (i.e., that take into account some properties of program execution history) may make the static analysis significantly easier, as it will allow it to make finer case splits, directed by the history of program executions. To allow for such powerful static analyses, we often resort to a *non standard semantics*, which incorporates properties that would normally be left out of the concrete semantics.

3.2. Abstract interpretation and static analysis

Once a reference semantics has been fixed and a property of interest has been formalized, the definition of a static analysis requires the choice of an *abstraction*. The abstraction ties a set of *abstract predicates* to the concrete ones, which they denote. This relation is often expressed with a *concretization function* that maps each abstract element to the concrete property it stands for. Obviously, a well chosen abstraction should allow expressing the property of interest, as well as all the intermediate properties that are required in order to prove it (otherwise, the analysis would have no chance to achieve a successful verification). It should also lend itself to an efficient implementation, with efficient data-structures and algorithms for the representation and the manipulation of abstract predicates. A great number of abstractions have been proposed for all kinds of concrete data types, yet the search for new abstractions is a very important topic in static analysis, so as to target novel kinds of properties, to design more efficient or more precise static analyses.

Once an abstraction is chosen, a set of *sound abstract transformers* can be derived from the concrete semantics and that account for individual program steps, in the abstract level and without forgetting any concrete behavior. A static analysis follows as a result of this step by step approximation of the concrete semantics, when the abstract transformers are all computable. This process defines an *abstract interpretation* [40]. The case of loops requires a bit more work as the concrete semantics typically relies on a fixpoint that may not be computable in finitely many iterations. To achieve a terminating analysis we then use *widening operators* [40], which over-approximates the concrete union and ensure termination.

A static analysis defined that way always terminates and produces sound over-approximations of the programs behaviors. Yet, these results may not be precise enough for verification. This is where the art of static analysis design comes into play through, among others:

- the use of more precise, yet still efficient enough abstract domains;
- the combination of application specific abstract domains;
- the careful choice of abstract transformers and widening operators.

3.3. Applications of the notion of abstraction in semantics

In the previous subsections, we sketched the steps in the design of a static analyzer to infer some family of properties, which should be implementable, and efficient enough to succeed in verifying non trivial systems.

Yet, the same principles can also be applied successfully to other goals. In particular, the abstract interpretation framework should be viewed a very general tool to *compare different semantics*, not necessarily with the goal of deriving a static analyzer. Such comparisons may be used in order to prove two semantics equivalent (i.e., one is an abstraction of the other and vice versa), or that a first semantics is strictly more expressive than another one (i.e., the latter can be viewed an abstraction of the former, where the abstraction actually makes some information redundant, which cannot be recovered). A classical example of such comparison is the classification of semantics of transition systems [38], which provides a better understanding of program semantics in general. For instance, this approach can be applied to get a better understanding of the semantics of a programming language, but also to select which concrete semantics should be used as a foundation for a static analysis, or to prove the correctness of a program transformation, compilation or optimization.

3.4. The analysis of biological models

One of our application domains, the analysis of biological models, is not a classical target of static analysis because it aims at analyzing models instead of programs. Yet, the analysis of biological models is closely intertwined with the other application fields of our group. Firstly, abstract interpretation provides a formal understanding of the abstraction process which is inherent to the modeling process. Abstract interpretation is also used to better understand the systematic approaches which are used in the systems biology field to capture the properties of models, until getting formal, fully automatic, and scalable methods. Secondly, abstract interpretation is used to offer various semantics with different grains of abstraction, and, thus, new methods to apprehend the overall behavior of the models. Conversely, some of the methods and abstractions which are developed for biological models are inspired by the analysis of concurrent systems and by security analysis. Lastly, the analysis of biological models raises issues about differential systems, stochastic systems, and hybrid systems. Any breakthrough in these directions will likely be very important to address the important challenge of the certification of critical systems in interaction with their physical environment.

CELTIQUE Project-Team

3. Research Program

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 [72].
- 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 [54], 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** [52], [75]. 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 [53]. 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 [5].

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) [45] 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 [73], [74] 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.,* [61], [60] have explored the algorithmic design space of contexts for benchmarks of significant size. Latter on, Milanova *et. al.,* [67] 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 [65] have extended the empiric evaluation of object-sensitivity using a BDD implementation allowing to cope with benchmarks otherwise out-of-scope. Besson and Jensen [49] 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 [66] and the race analysis of Naik, Aiken [68] and Whaley [69] 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 [57] 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 [64]. The proposed technique has been implemented in a tool called NIT [63]. Null pointer detection is also done by bug-detection tools such as FindBugs [63]. 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 [62], [44], linear

logic approaches to implicit computational complexity [46], cost model for Java byte code [40] based on size relation inference, and WCET computation by abstract interpretation based loop bound interval analysis techniques [55].

We have proposed an original approach for designing static analyses computing program costs: inspired from a probabilistic approach [76], 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 [51]. An example of cache miss analysis has been developed within this framework [79].

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 [0] 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.

[0] 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 ?

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], [48]. 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 [50]. **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) [70] 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" [71]. Many solutions have been proposed to reduce the size of the trusted computing base. In the *foundational proof carrying code* of Appel and Felty [43], [42], 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 [71] 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” 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.* [41] 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 [39] 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 [7].

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 [58], [59]. We defined a simple and efficient algorithm [56] 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 [5]. This checker is now used to certify all analysis results produced by the regular completion tool as well as the optimised version of [47].

DEDUCTEAM Exploratory Action

3. Research Program

3.1. From proof-checking to Interoperability

A new turn with Deduction modulo was taken when the idea of reasoning modulo an arbitrary equivalence relation was applied to typed λ-calculi with dependent types, that permits to express proofs as algorithms, using the Brouwer-Heyting-Kolmogorov interpretation and the Curry-de Bruijn-Howard correspondence [32]. It was shown in 2007, that extending the simplest λ-calculus with dependent types, the $\lambda\Pi$-calculus, with an equivalence relation, led to a calculus we called the $\lambda\Pi$-calculus modulo, that permitted to simulate many other λ-calculi, such as the Calculus of Constructions, designed to express proofs in specific theories.

This led to the development of a general proof-checker based on the $\lambda\Pi$-calculus modulo [3], that could be used to verify proofs coming from different proof systems, such as Coq [30], HOL [39], etc. To emphasize this versatility of our proof-system, we called it Dedukti —"to deduce" in Esperanto. This system is currently developed together with companion systems, Coqine, Holide, Focalide, and Zenonide, that permits to translate proofs from Coq, HOL, Focalize, and Zenon, to Dedukti. Other tools, such as Zenon Modulo, directly output proofs that can be checked by Dedukti.

Dedukti proofs can also be exported to other systems, in particular to the MMT format [47].

A thesis, which is at the root of our research effort, and which was already formulated by the team of the Logical Framework [38] is that proof-checkers should be theory independent. This is for instance expressed in the title of our invited talk at Icalp 2012: *A theory independent Curry-De Bruijn-Howard correspondence*.

Using a single prover to check proofs coming from different provers naturally led to investigate how these proofs could interact one with another. This issue is of prime importance because developments in proof systems are getting bigger and, unlike other communities in computer science, the proof-checking community has given little effort in the direction of standardization and interoperability. On a longer term we believe that, for each proof, we should be able to identify the systems in which it can be expressed.

3.2. Automated theorem proving

Deduction modulo has originally been proposed to solve a problem in automated theorem proving and some of the early work in this area focused on the design of an automated theorem proving method called *Resolution modulo*, but this method was so complex that it was never implemented. This method was simplified in 2010 [5] and it could then be implemented. This implementation that builds on the iProver effort [46] is called iProver modulo.

iProver modulo gave surprisingly good results [4], so that we use it now to search for proofs in many areas: in the theory of classes—also known as *B* set theory—, on finite structures, etc. Similar ideas have also been implemented for the tableau method with in particular several extensions of the *Zenon* automated theorem prover. More precisely, two extensions have been realized: the first one is called *Super Zenon* [13] [35] and is an extension to superdeduction (which is a variant of Deduction modulo), and the second one is called *Zenon Modulo* [33], [34] and is an extension to Deduction modulo. Both extensions have been extensively tested over first order problems (of the TPTP library), and also provide good results in terms of number of proved problems. In particular, these tools provide good performances in set theory, so that *Super Zenon* has been successfully applied to verify *B* proof rules of *Atelier B* (work in collaboration with *Siemens*). Similarly, we plan to apply *Zenon Modulo* in the framework of the *BWare* project to verify *B* proof obligations coming from the modeling of industrial applications.

More generally, we believe that proof-checking and automated theorem proving have a lot to learn from each other, because a proof is both a static linguistic object justifying the truth of a proposition and a dynamic process of proving this proposition.

3.3. Models of computation

The idea of Deduction modulo is that computation plays a major role in the foundations of mathematics. This led us to investigate the role played by computation in other sciences, in particular in physics. Some of this work can be seen as a continuation of Gandy's [36] on the fact that the physical Church-Turing thesis is a consequence of three principles of physics, two well-known: the homogeneity of space and time, and the existence of a bound on the velocity of information, and one more speculative: the existence of a bound on the density of information.

This led us to develop physically oriented models of computations.

ESTASYS Exploratory Action

3. Research Program

3.1. Systems of Systems, Heterogeneous Systems, Dynamicity, Statistical Model Checking

Formal methods rely on the notion of *transition system* (TS): an abstract machine that characterises a system's *complete* behaviour. This machine consists of a complete set of states (each representing full knowledge of the system at a given moment) and transitions between states, which may be labelled with labels chosen from some set of actions. This definition makes it necessary to have advanced knowledge of all the possible states of the system – to have a statically configured system. The algorithms used by formal methods perform an exhaustive exploration of the state space of the TS, so such methods suffer from the so-called *state-space explosion problem*. As a consequence, there are many real systems that are beyond the scope of such techniques. Despite this, over the last thirty years it has been shown that, when combined with heuristics such as partial order reductions or abstraction, **formal approaches are powerful enough to verify industrial-scale systems**.

The first wave of techniques was deployed to verify whether a certain set of (problem) states can be reached ('reachability'). Later, extensions of TS, such as *hybrid systems* and *stochastic automata*, were proposed to cope with new problems (e.g., energy consumption) or to reason on distributed real-time embedded components (possibly heterogeneous). It was quickly observed that the complexity of assessing correctness of such extended models arises not exclusively from the fact that they are large, but also because they introduce *undecidability*. As a concrete example, the reachability problem is already undecidable for any real-time system whose time evolution is described by a non-constant derivative equation.

This motivated the development of more efficient techniques that approximate the answer to the original problem or approximate the problem. Of these, perhaps the most successful quantitative technique is *Statistical Model Checking* , that can be seen as a trade-off between testing and formal verification. The core idea of SMC is to generate a number of *simulations* of the system and verify whether they satisfy a given property expressed in temporal logics, which can be done by using *runtime verification approaches*. The results are then used together with algorithms from the statistical area in order to decide whether the system satisfies the property with some probability. SMC resembles classical simulation-based techniques used in industry, but uses a formal model of systems and requirements. This not only gives a rigorous meaning to industrial practices, but also makes available more than twenty years of research in the area of *runtime verification*. Last but not least, **the use of statistical algorithms allows us to approximate undecidable problems**. Recent successful applications of SMC can be found in systems biology, security protocols and avionics. In particular, SMC was used to discover inconsistent requirements of an EADS airplane communication system.

3.1.1. Systems of Systems (SoS)

The advent of service-oriented and cloud architectures is leading to generations of computer systems that exhibit a new type of complexity: such systems are no longer statically configured, but comprise components that are systems in their own right, able to discover, select and bind on-the-fly to other components that can deliver services that they require. These complex systems, referred to as *Systems of Systems* (SoS), can change over time as each component creates and modifies the network over which it needs to operate: as they execute, the components create a network of their own and use it to fulfil their goals.

The Internet, made up of an unsupervised and rapidly growing, dynamically configured set of computers and physical connections, is an obvious illustration of the potential complexity of dynamic networks of interactions. Another example is the so-called "Flash Crash" in the U.S. equity market: on May 6, 2010, a block sale of 4.1 billion dollars of futures contracts executed on behalf of a fund-management company triggered a complex pattern of interactions between the high-frequency algorithmic trading systems (algos) that buy and sell blocks of financial instruments and made the Dow Jones Industrial Average drop more

than 600 points, representing the disappearance of 800 billion dollars of market value. This example is an illustration of the faulty divergence of SoS behaviour, where the system starts to misbehave and dynamically creates new components that follow the same pattern and make the problem worse. Examples of this include when a SoS detects high energy use and invokes a new component to reduce the energy, thus consuming *more* energy. **Until now, such divergence has been mostly handled by humans that eventually observe the faulty behaviour and manually intervene to stop it. This human-based solution is not always successful and clearly unsatisfactory, since it acts retrospectively, when the system has already failed.**

3.1.2. Grand Challenge and Breakthroughs of ESTASYS

SoS are an efficient means of achieving high performance and are thus becoming ubiquitous. Society's increasing reliance on SoS demands that they are reliable, but tools to guarantee this at the design stage do not exist. Most conventional formal analysis techniques, even those dedicated to adaptive systems, fail when applied to SoS because they are designed to reason on systems whose state space can be predicted in advance. **The grand challenge addressed by ESTASYS is the fundamental overhaul of formal methods techniques in the design of SoS life cycle**.

It is clear that SMC can be applied to the verification of complex systems. Unfortunately, SMC cannot yet be applied to SoS, because existing techniques are designed to capture the behaviour of statically configured systems, or systems whose dynamical configuration arises from permutations of known components. ESTASYS defines new abstract computational models and extend the state of the art of SMC to include SoS.

ESTASYS proposes a new formal methodology to support an evolutionary adaptive and iterative SoS life-cycle. *We foresee the following breakthroughs:*

1. Our ground-breaking computational model addresses the complex dynamic nature of SoS. The model is based on new interface theories that take into account behaviours of possibly unknown components and thus abstract what is unknown.
2. Cutting edge algorithms coming from the area of statistics and learning are exploited to make predictions about autonomous systems making local decisions. For example, **statistical abstraction** abstracts the behaviour of unknown environments; interleaving analysis and runtime monitoring of deployed systems to continuously update distributions embedded in the interfaces.
3. New statistical algorithms for SMC that scale efficiently and handle undecidability impacts the formal analysis of complex systems.
4. Our results are implemented in a professional toolset, ESTASYS-PLASMA, that is constructed in close collaboration with our industrial partners. This ensures relevance to industry and potentially high impact in the marketplace.

3.1.3. Methodology and Organization

ESTASYS's main challenge is to lay the foundation of a novel rigorous software construction methodology for SoS, based on simulation, statistics and industrial practices. ESTASYS establishes theories and empirical evidence for the introduction of formal verification-based approaches in the rigorous design of SoS.

ESTASYS addresses essential research questions for the introduction of formal techniques to support the SoS life-cycle. SoS occur in multiple disciplines and therefore there is a need for a common language. In particular, notions such as **autonomous decisions and dynamicity** must be standardized and well understood by those that will apply our methodology. Additionally, **characterizing the topological structure** of a SoS is essential for the study of component interactions and data exchanges. The complexity of SoS requires the development of a **sound formal semantic foundation** to support deployment of formal methods. We thus identify a minimal computational model that characterize SoS, on which classes of properties of interest can be defined. The project investigates new simulation-based approaches, combined with other domains (statistics, learning, ...), to verify such properties on the new computational model. Finally, ESTASYS identifies under which conditions the new techniques can be used, to take decisions during design and evolution time, leading to a fully integrated development cycle.

ESTASYS focuses on both the static and dynamic properties of SoS. ESTASYS establishes models for each component and investigates the connection and dynamical interactions between them. ESTASYS's activities are organize in six main tasks: tasks 1, 2 and 3 are responsible for breakthrough 1; task 4 is responsible for breakthrough 2; task 5 is responsible for breakthrough 3; task 6 is responsible for breakthrough 4.

Task 1. Characterizing SoS. Examples of SoS found in various areas, such as health care, smart buildings and energy grids, are analysed and used to standardize notions of autonomous decisions and dynamicity. We also study and classify SoS-related problems, such as faulty behaviour divergence. Our objective is to derive in Task 2 formal models that abstract the above classification.

Task 2. Formal Modeling of SoS. Classical theories do not provide for SoS, hence we require new formal models for SoS that take into account (i) dynamicity and emergent behaviours, (ii) autonomous decisions of components, and (iii) architectural constraints, including information regarding the viability of the hardware. In particular, we devise new logics tailored to the specific needs of SoS. Such logics, dynamic by nature, includes extended notions of quantification, such as energy, and considers hardware constraints and distributions of system configurations. Task 2 includes modelling the various components running within the SoS and their (dynamical) interactions. This requires the definition of a new type of interface able to work with heterogeneous components and to abstract the behaviour of unknown resources. Interfaces act as an abstraction for the internal behaviour of each component and encodes the dynamical constraints of the SoS. They are used to (i) model and define the authorised interactions between the components, (ii) reason on dynamical aspects and (iii) abstract unknown behaviour.

Task 3. Statistical abstraction interleaving design and deployment. Abstraction techniques are necessary to reduce the complexity of SoS and to model uncertainty. Specifically, **statistical abstractions** of the observed runtime behaviour of components is used to quantify, e.g., the probability that a number of new components satisfying some constraints is started at a given execution point. Runtime verification monitors the executions of the deployed system to create distributions embedded in the interfaces developed in Task 1. When a deployed system is available, ESTASYS interleaves simulation, analysis and runtime monitoring, using real behaviour to update the statistical abstractions, and eventually replace some of those abstractions by concrete ESTASYS-Interface models. The ESTASYS methodology adopts a Bayesian approach: (i) an initial, plausible distribution is 'guessed', based on whatever is known; (ii) the system is simulated using the current approximated distribution; (iii) the behaviour of the simulated system becomes the new approximation; (iv) the process is iterated as necessary. While learning-based simulation approaches, such as model fitting, can be used to learn the abstraction by conducting simulations from a finite set of initial components, we have to provide clear evidence that a global property holds on the system if it holds on its corresponding statistical abstraction. The task requires strong competences in statistics.

Task 4. Developing Efficient Simulation and Monitoring Algorithms for SoS. The ground-breaking models developed in Task 2 requires efficient simulation and monitoring techniques. This necessitates the study of new algorithms for dynamically configured systems and monitoring approaches to reason on heterogeneous components and the new quantitative logics and interface paradigms developed in Task 2.
A major difficulty in developing monitoring techniques for SoS is that the components have their own goals and behave differently in different environments. Unnecessary high-level hypotheses on properties may drastically increase simulation time and should be avoided.

Task 5. Developing Efficient Statistical Techniques for SoS. SoS pose new challenges for statistical techniques, requiring the study of new SMC algorithms dedicated to SoS goals. In contrast to existing SMC algorithms that can only be applied to pure stochastic systems, SMC algorithms for SoS have to take into account the non-deterministic aspects of autonomous decisions made by neighbour components. We postulate that this can be done by extending very recent advances in reinforcement learning algorithms. Rare events play an important role in system reliability, so we include rare-event simulation algorithms, such as importance sampling and importance splitting, which can reduce variance and significantly increase simulation efficiency.

Task 6. Evaluating the impact of statistical and simulation-based techniques. Evidence of the success of ESTASYS is provided by the publishing of a complete experimental environment, ESTASYS-PLASMA, that supports the empirical validation of ESTASYS's theories. ESTASYS-PLASMA contains efficient implementations of the results discovered in Tasks 2-5, and will provide intuitive feedback mechanisms so that the engineer can use the results of the verification process to improve SoS design.

GALLIUM Project-Team

3. Research Program

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 [48]. Through collaborations, we also gained experience with several domain-specific languages based on a functional core, including distributed programming (JoCaml), XML processing (XDuce, CDuce), reactive functional programming, and hardware modeling.

3.2. Type systems

Type systems [65] 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 [59], [53], 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 [69], 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 [66]. Finally, the notion of "coercion polymorphism" proposed by Cretin and Rémy [28] combines and generalizes both parametric and subtyping polymorphism.

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.

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. Research Program

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. Secondly, 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). Therefore, 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. Research Program

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, Serge Haddad, Loig Jezequel, Hernán Ponce de Léon, 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 [63], [64], [68], 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 [47]

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 [64], [49], 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 [60], [73]). 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 [46], [52]). 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 [78], [79]. Automatizing such procedures for the supervision and management of distributed and locally monitored asynchronous systems is a long-term goal to which *MExICo* hopes to contribute.

3.1.3. Contextual nets

Participant: Stefan Schwoon.

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 [62].

However, Petri nets are inadequate to model concurrent read accesses to the same resource. Such situations often arise naturally, 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. An abstract algorithm for contextual unfoldings was first given in [48]. In recent work, we further studied this subject from a theoretical and practical perspective, allowing us to develop concrete, efficient data structures and algorithms and a tool (Cunf) that improves upon existing state of the art. This work led to the PhD thesis of César Rodríguez.

Contexutal unfoldings deal well with two sources of state-space explosion: concurrency and shared resources. Recently, we proposed an improved data structure, called *contextual merged processes* (CMP) to deal with a third source of state-space explosion, i.e. sequences of choices. The work on CMP [81] is currently at an abstract level. In the short term, we want to put this work into practice, requiring some theoretical groundwork, as well as programming and experimentation.

Another well-known approach to verifying concurrent systems is *partial-order reduction*, exemplified by the tool SPIN. Although it is known that both partial-order reduction and unfoldings have their respective strengths and weaknesses, we are not aware of any conclusive comparison between the two techniques. Spin comes with a high-level modeling language having an explicit notion of processes, communication channels, and variables. Indeed, the reduction techniques implemented in Spin exploit the specific properties of these features. On the other side, while there exist highly efficient tools for unfoldings, Petri nets are a relatively general low-level

formalism, so these techniques do not exploit properties of higher language features. Our work on contextual unfoldings and CMPs represents a first step to make unfoldings exploit richer models. In the long run, we wish raise the unfolding technique to a suitable high-level modelling language and develop appropriate tool support.

3.1.4. Verification of Concurrent Recursive Programs

Participants: Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, Stefan Schwoon.

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.

3.1.5. Dynamic and parameterized concurrent systems

Participants: Benedikt Bollig, Paul Gastin.

In the past few years, our research has focused on concurrent systems where the architecture, which provides a set of processes and links between them, is *static* and *fixed in advance*. However, the assumption that the set of processes is fixed somehow seems to hinder the application of formal methods in practice. It is not appropriate in areas such as mobile computing or ad-hoc networks. In concurrent programming, it is actually perfectly natural to design a program, and claim its correctness, independently of the number of processes that participate in its execution. There are, essentially, two kinds of systems that fall into this category. When the process architecture is static but unknown, it is a parameter of the system; we then call a system *parameterized*. When, on the other hand, the process architecure is generated at runtime (i.e., process creation is a communication primitive), we say that a system is *dynamic*. Though parameterized and dynamic systems have received increasing interest in recent years, there is, by now, no canonical approach to modeling and verifying such systems. Our research program aims at the development of *a theory of parameterized and dynamic concurrent systems.* More precisely, our goal is a *unifying* theory that lays algebraic, logical, and automata-theoretic foundations to support and facilitate the study of parameterized and dynamic concurrent systems. Such theories indeed exist in non-parameterized settings where the number of processes and the way they are connected are fixed in advance. However, parameterized and dynamic systems lack such foundations and often restict to very particular models with specialized verification techniques.

3.1.6. Testing

Participants: Benedikt Bollig, Paul Gastin, Stefan Haar, Hernán Ponce de Léon.

3.1.6.1. 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.

3.1.6.2. 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 [70], [53] 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.

3.1.6.3. Near Future

Completion of asynchronous testing in the setting without any big-step synchronization, and an improved understanding of the relations and possible interconnections between local (i.e. distributed) and asynchronous (centralized) testing. This has been the objective of the *TECSTES* project (2011-2014), funded by a DIGITEO *DIM/LSC* grant, and which involved 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. We have developped a complete testing framework for concurrent systems, 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 as well as an algorithm to construct a test suite with such properties.

A mid-to long term goal (which may or may not be addressed by *MExICo* depending on the availability of staff for this subject) 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 [59] 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 [77], [72], [75], [65], [67].

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 [66], [57]. 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: 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.

Our work on direct modeling and handling of "grey boxes" via modal models (see [61]) was halted when Dorsaf El-Hog stopped her PhD work to leave academia, and has not resumed for lack of staff. However, it should be noted that semi-transparent system management in a larger sense remains an active field for the team, witness in particular our work on diagnosis and testing.

3.3. Management of Quantitative Behavior

Participants: 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, Claudine Picaronny.

3.3.2.1. Non-sequential probabilistic processes

Practical fault diagnosis requires to select explanations of *maximal likelihood*. For partial-order based diagnosis, this leads therefore to the question what the probability of a given partially ordered execution is. In Benveniste et al. [51], [44], 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 [69], 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.

In another line of research [55] we have studied the likelihood of occurrence of non-sequential runs under random durations in a stochastic Petri net setting. It remains to better understand the properties of the probability measures thus obtained, to relate them with the models in logical time, and exploit them e.g. in *diagnosis*.

3.3.2.2. Distributed Markov Decision Processes

Participant: Serge Haddad.

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 have been cooperating on this subject with the SUMO team at Inria Rennes; in the joint work [45]; there, we strive to synthesize for a given MDP a control so as to guarantee a specific stationary behavior, rather than - as is usually done - so as to maximize some reward.

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 [50].
- 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 [58].
- 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 [83] .

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), see [9]; (2) we want to adapt methods for discrete-event systems that require some theoretical developments in the stochastic framework and, (3) we plan 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 [74] [56]. 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. Implementation of Real-Time Concurrent Systems

Participants: Thomas Chatain, Stefan Haar, Serge Haddad.

This was one of the tasks of the ANR ImpRo.

Formal models for real-time systems, like timed automata and time Petri nets, have been extensively studied and have proved their interest for the verification of real-time systems. On the other hand, the question of using these models as specifications for designing real-time systems raises some difficulties. One of those comes from the fact that the real-time constraints introduce some artifacts and because of them some syntactically correct models have a formal semantics that is clearly unrealistic. One famous situation is the case of Zeno executions, where the formal semantics allows the system to do infinitely many actions in finite time. But there are other problems, and some of them are related to the distributed nature of the system. These are the ones we address here.

One approach to implementability problems is to formalize either syntactical or behavioral requirements about what should be considered as a reasonable model, and reject other models. Another approach is to adapt the formal semantics such that only realistic behaviors are considered.

These techniques are preliminaries for dealing with the problem of implementability of models. Indeed implementing a model may be possible at the cost of some transformation, which make it suitable for the target device. By the way these transformations may be of interest for the designer who can now use high-level features in a model of a system or protocol, and rely on the transformation to make it implementable.

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.

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 [43]. 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 [54]. That paper, however, does not give definite answers to all the aforementioned problems. It identifies directions for future research that we will be tackling.

PARSIFAL Project-Team

3. Research Program

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 [64], and normalization in different logics can justify the design of new and different functional programming languages [38].
- *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 [70], and different proof search strategies can be used to justify the design of new and different logic programming languages [68].

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 [39] 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 [76] 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 [72] 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 [73], 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. Inductive and co-inductive reasoning

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 is based on earlier work by McDowell, Miller, and Tiu [66] [65] [71] [77], and on more recent work by Gacek, Miller, and Nadathur [3] [52]. The Parsifal team, along with our colleagues in Minneapolis, Canberra, Singapore, and Cachen, have been building two tools that exploit the novel features of this logic. These two 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.

We have used these systems to provide formalize reasoning of a number of complex formal systems, ranging from programming languages to the λ-calculus and π-calculus.

During 2014, the Abella system has been extended with a number of new features. A number of new significant examples have been implemented in Abella and an extensive tutorial for it has been written [31].

3.3. Developing a foundational approach to defining proof evidence

The team is developing a framework for defining the semantics of proof evidence. With this framework, implementers of theorem provers can output proof evidence in a format of their choice: they will only need to be able to formally define that evidence's semantics. With such semantics provided, proof checkers can then check alleged proofs for correctness. Thus, anyone who needs to trust proofs from various provers can put their energies into designing trustworthy checkers that can execute the semantic specification.

In order to provide our framework with the flexibility that this ambitious plan requires, we have based our design on the most recent advances within the theory of proofs. For a number of years, various team members have been contributing to the design and theory of *focused proof systems* [40] [42] [44] [45] [55] [62] [63] and we have adopted such proof systems as the corner stone for our framework.

We have also been working for a number of years on the implementation of computational logic systems, involving, for example, both unification and backtracking search. As a result, we are also building an early and reference implementation of our semantic definitions.

3.4. Deep inference

Deep inference [57], [59] 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.

3.5. 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 [54] for linear logic but also in Robinson’s proof nets [74] 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., [5], [4] and [58]. 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 [5] 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. Research Program

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 [51] 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 [44], then by Howard and de Bruijn at the end of the 60's [54], [66], 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 [41] stemmed out – a formalism that is both a logic and a programming language and that is at the source of the Coq system [64].

3.1.2. Towards the calculus of constructions

The λ-calculus, defined by Church [40], 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 [37].

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 [60].

In 1985, Coquand and Huet [41], [42] in the Formel team of Inria-Rocquencourt explored an alternative approach based on Girard-Reynolds' system F [52], [63]. 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 called OCaml and 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 [43] 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 moved to the new Inria Saclay research center. 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 typed 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 exist – generally first-order ones.

3.4. Around and beyond the Curry-Howard correspondence

For two decades, the Curry-Howard correspondence has been limited to the intuitionistic case but since 1990, an important stimulus spurred on the community following Griffin's discovery that this correspondence was extensible to classical logic. The community then started to investigate unexplored potential connections between computer science and logic. One of these fields is the computational understanding of Gentzen's sequent calculus while another one is 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 [53] that some operators known as control operators were typable by the principle of double negation elimination ($\neg\neg A \Rightarrow A$), a principle that enables classical reasoning.

Control operators are used to jump from one location of a program to another. They were first considered in the 60's by Landin [58] and Reynolds [62] and started to be studied in an abstract way in the 80's by Felleisen *et al* [49], leading to Parigot's $\lambda\mu$-calculus [61], a reference calculus that is in close Curry-Howard correspondence with classical natural deduction. In this respect, control operators are fundamental pieces to establish a 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, for which two kinds of interpretations 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 rules. This line of work led in 2000 to 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 [57] and Krivine's abstract machine for call-by-name evaluation [56], [55]. 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 [50]: 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.

SUMO Project-Team

3. Research Program

3.1. Model expressivity and quantitative verification

The overall objective of this axis is to combine the quantitative aspects of models with a distributed/modular setting, while maintaining the tractability of verification and management objectives.

There is first an issue of modeling, to nicely weave time, costs and probabilities with concurrency and/or asynchronism. Several approaches are quite natural, as time(d) Petri nets, networks of timed automata, communicating synchronously or through FIFO, etc. But numerous bottlenecks remain. For example, so far, no probabilistic model nicely fits the notion of concurrency: there is no clean way to express that two components are stochastically independent between two rendez-vous.

Second, the models we want to manipulate should allow for quantitative verification. This covers two aspects: either the verification question is itself quantitative (compute an optimal scheduling policy) or boolean (decide whether the probability is greater than a threshold). Our goal is to explore the frontier between decidable and undecidable problems, or more pragmatically tractable and untractable problems. Of course, there is a tradeoff between the expressivity and the tractability of a model. Models that incorporate distributed aspects, probabilities, time, etc, are typically untractable. In such a case, abstraction or approximation techniques are a work around that we will explore.

In more details, our research program on this axis covers the following topics:

- the verification of distributed timed systems,
- the verification of large scale probabilistic (dynamic) systems, with a focus on approximation techniques for such systems,
- the evaluation of the opacity/diagnosability degree of stochastic systems,
- the design of modular testing methods for large scale modular systems.

3.2. Management of large distributed systems

The generic terms of "supervision" or "management" of distributed systems cover problems like control (and controller synthesis), diagnosis, sensor placement, planning, optimization, (state) estimation, parameter identification, testing, etc. These questions have both an offline and an online facet. The literature is abundant for discrete event systems (DES), even in the distributed case, and for some quantitative aspects of DES in the centralized case (for example partially observed Markov decision processes (POMDP), probabilistic diagnosis/diagnosers, (max,+) approaches to timed automata). And there is a strong trend driving formal methods approaches towards quantitative models and questions like the most likely diagnosis, control for best average reward or for best QoS, optimal sensor placement, computing the probability of failure (un)detection, estimating the average impact of some failure or of a decision, etc. This second research axis focuses on these issues, and aims at developing new concepts and tools to master some already existing large scale systems, as telecommunication networks, cloud infrastructures, web-services, etc. (see the Application Domains section).

The objective being to address large systems, our work will be driven by two considerations: how to take advantage of the modularity of systems, and how to best approximate/abstract too complex systems by more tractable ones. We mention below main topics we will focus on:

- Approximate management methods. We will explore the relevance of ideas developed for large scale stochastic systems, as turbo-algorithms for example, in the setting of modular dynamic systems.
- Self-modeling, which consists in managing large scale systems that are known by their building rules, but which specific managed instance is only discovered at runtime, and on the fly. The model of the managed system is built on-line, following the needs of the management algorithms.

- Distributed control. We will tackle issues related to asynchronous communications between local controllers, and abstraction techniques to address large systems.
- Test and enforcement. We will tackle coverage issues for the test of large systems, and the test and enforcement of properties for timed models, or for systems handling data.

3.3. Data driven systems

The term data-driven systems refers to systems the behavior of which depends both on explicit workflows (scheduling and durations of tasks, calls to possibly distant services,...) and on the data processed by the system (stored data, parameters of a request, results of a request,...). This family of systems covers workflows that convey data (business processes or information systems), transactional systems (web stores), large databases managed with rules (banking systems), collaborative environments (health systems), etc. These systems are distributed, modular, and open: they integrate components and sub-services distributed over the web and accept requests from clients. Our objective is to provide validation and supervision tools for such systems. To achieve this goal, we have to solve several challenging tasks:

- provide realistic models, and sound automated abstraction techniques, to reason on models that are reasonable abstractions of real implemented systems designed in low-level languages (for instance BPEL (Business Process Execution Language)). These models should be able to encompass modularity, distribution, in a context where workflows and data aspects are tightly connected.
- provide tractable solutions for validation of models. Important questions that are frequently addressed (for instance safety properties or coverability) should remain decidable on our models, but also with a decent complexity.
- address design of data driven systems in a declarative way: declarative models are another way to handle data-driven systems. Rather than defining the explicit workflows and their effects on data, rule-based models state how actions are enacted in terms of the shape (pattern matching) or value of the current data. Such declarative models are well accepted in business processes (Companies such as IBM use their own model of business rules [53] to interact with their clients). Our approach is to design collaborative activities in terms of distributed structured documents, that can be seen as communicating rewriting systems. This modeling paradigm also includes models such as distributed Active XML [48], [51]. We think that distributed rewriting rules or attributed grammars can provide a practical but yet formal framework for maintenance, by providing a solution to update mandatory documentation during the lifetime of an artifact.
- address QoS management in large reconfigurable systems:

Data driven distributed systems such as web services often have constraints in terms of QoS. This calls for an analysis of quantitative features, and for reconfiguration techniques to meet QoS contracts. We will build from the experience in our team on QoS contracts composition [54] and planning [47], [49] to propose optimization and reconfiguration schemes.

TEMPO Team

3. Research Program

3.1. Cyber Physical Systems

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.

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 and 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, in a co-simulation environment such as [12], [13]. 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 simulation environment, which uses a single simulation loop thanks to Transaction Level Modeling (TLM) [3] 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 [2]. Another technique to implement a fast ISS is *dynamic translation* [8], [4] which has been favored by many implementors [18], [19], [20], [14] in the past decade.

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. A challenge in the development of high performance simulators is to maintain simultaneously fast speed and simulation accuracy. In the TEMPO 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 TEMPO simulation is to extract information from the simulated applications in order to prove system properties. One can use model based tools to generate tests that can be run on the simulator to check whether the test fails or not on the real application. The project is considering an approach as illustrated in Figure 1

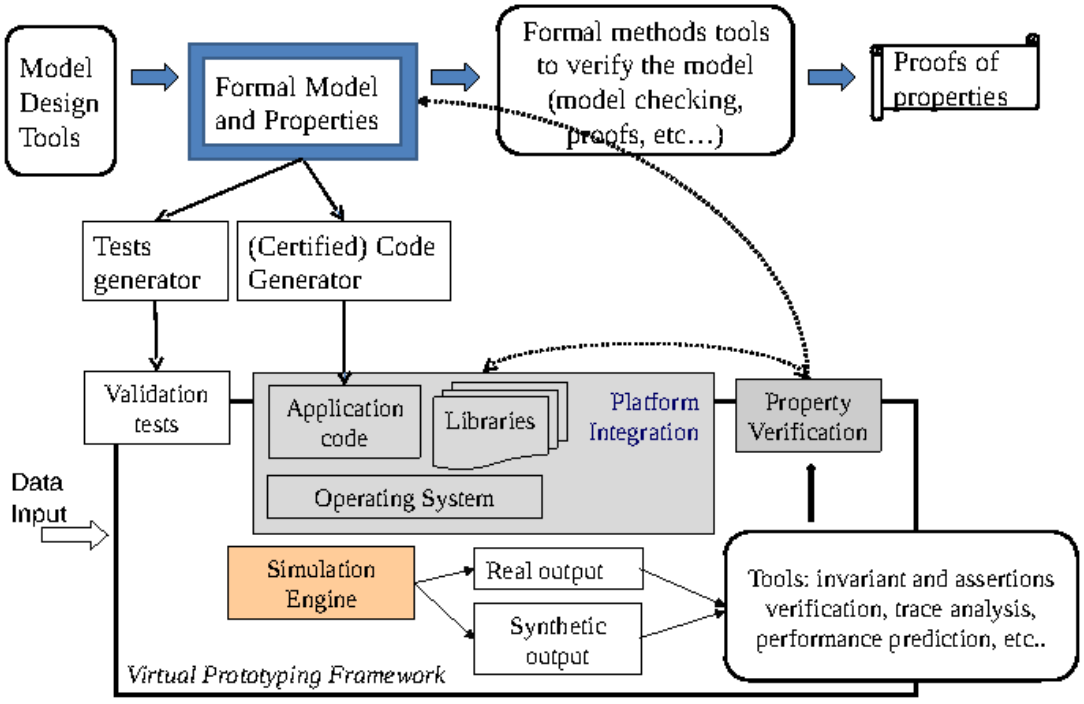

Figure 1. Proposed development methodology

Thus, it is also a goal of TEMPO activities to use such formal methods tools to detect failures, either by generating tests, or by using formal methods tools to analyze the results of simulation sessions.

3.2. Verification of Embedded Systems Properties

Since last decade, we have witnessed rapid development in embedded system domain. More and more state-of-the-art embedded systems adopt the heterogeneous multi-processor platform rather than the platform with single core. To achieve better quality and performance, the design paradigm shift from simple control system to complex heterogeneous Cyber-Physical Systems (CPS) is gaining more interests. Increasing complexity coupled with time-to-market pressure create a critical need to validate heterogeneous embedded system designs. The functional validation is thus widely acknowledged as a major bottleneck in embedded system design. To guarantee the reliability of heterogeneous embedded systems, up to 70% of the overall design time and resources are spent on functional validation.

From the verification point of view, the major objective of this project is to reduce the overall validation efforts in the top-down design flow of embedded system design using the high-level specifications. In this project, we plan to address the following three major problems:

- **Formal modeling of high-level specifications**. We want to investigate how to model heterogeneous systems with multiple models of computation (MoC) and how to extract the formal models from system-level specifications to enable automated analysis.
- **Efficient validation of system-level specifications with minimum effort**. The idea here is to investigate the automated directed test generation from high-level specification validation and explore various approaches and techniques to further reduce the directed test generation time (eliminate redudant tests).
- **Consistency checking between different abstraction layers**. We also want to explore the possibility of reusing high-level validation efforts for low-level implementation validation as well as to check the consistency between different abstraction layers.

In conclusion, this project targets to improve the effectiveness and efficiency of functional validation of heterogeneous embedded systems. We believe that our approaches can not only enhance the reliability of heterogeneous embedded systems, but also reduce the time-to-market.

TOCCATA Project-Team

3. Research Program

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, we propose a new axis of research: to develop certified tools, i.e. 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 [6], including both interactive proving and automated solving [10]. 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 [64], [5]. This motivated our research on the formal verification of numerical programs.

Moreover, we 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. Formally Verified Programs,
2. Certified Tools,
3. Numerical Programs.

3.2. Formally Verified Programs

Formal program verification is a research theme that builds upon our expertise in the development of methods and tools for proving programs, from source codes annotated with specifications to proofs. In the past, 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 correct-by-construction 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 verified 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 Verified 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 also requires designing 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 verified 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. [98]). The well-known proposals for solving it include *Separation logic* [121], *implicit dynamic frames* [118], and *considerate reasoning* [120]. 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 [122] or a system of theory cloning in our system *Why3* [2]; 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 [72], [96], [51].

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 generating *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 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 [7].

A key challenge also lies in the 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 [83], [84]. 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 verified components will require higher-order specifications. A few solutions have been proposed, essentially based on encoding of higher-order goals into first-order goals [96].

Generally speaking, there are several theories, interesting for program verification, that we would like to add as built-in decision procedures in an SMT context. First, although there already exist decision procedures for variants of bit-vectors, they 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 [114]. 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 [70]. We now have 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 follow is the design of 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* [110] 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

Concerning formal verification of 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 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 [1], to cryptographic programs [50]. We plan to use proof techniques on applications related to databases. We also have plans to support low-level programs such as assembly code [86], [113] 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 [0] [76]) or abstract interpretation techniques (project Cafein, started in 2013) and also for discharging proof obligations coming from other systems like *Atelier B* [109] (project BWare).

3.3. Certified Tools

One of our goals is to guarantee the soundness of the tools we develop. In fact, 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 analyzers 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 [78] formalizing term rewriting systems, the Alea library [1] for the formalization of randomized algorithms, several libraries formalizing floating-point numbers (Floats [60], Gappalib [107], and now Flocq [6] which unifies the formers). Second we conducted the development of a certified decision procedure [103] that corresponds to a core part of *Alt-Ergo*. Third we developed, still in *Coq*, certified verification condition generators, in a first step [94] for a language similar to *Why*, and in a second step [93] for C annotated in ACSL [56], based on the operational semantics formalized in the CompCert certified compiler project [102].

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 for more freedom. For example, it should allow one to use a bit of side-effects when the underlying algorithm justifies 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 verified libraries of programs, but also libraries of specifications (e.g. theories of sets, maps, etc.).

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.

[0] http://cubicle.lri.fr/

A first attempt in considering such an issue was done in earlier work [109]. 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 is an important topic of the 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* [103] 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. However, for very complex decision procedures, developing a certified proof search might be too ambitious. Instead, the idea is to ask *Gappa* to produce Coq proofs on a per-goal basis, so as to check *a posteriori* the soundness of its result on the given instance. More generally, we can have *Gappa* produce traces of its execution that can later be processed by a certified trace checker. This approach was used in the past for certified proofs of termination of rewriting systems [79], and it was also used internally in CompCert for several passes of compilation [102].

3.3.4. Certified VC Generation

The other kind of tools that we would like to certify are the VC generators. This is a continuation of the work on developing in *Coq* a certified VC generator for C code annotated in ACSL [93]. We develop such a generator in *Why3* instead of *Coq* [105]. As before, this builds 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* [99], and a generator for unstructured programs like assembly, that would operate on an arbitrary control-flow graph.

On a longer term, we wish 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 [73] 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 cannot be generated through the evaluation of *Coq* definitions. 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 [53].

3.4. 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 [67], [123], [62], [117], [66], [61], [108], [106] as well as the web galleries of certified programs at our Web page [0], the Hisseo project [0], S. Boldo's page [0], and industrial case studies in the U3CAT ANR project. The ability to express such complex properties comes from models developed in *Coq* [6]. 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 [5]. However, the proofs can get difficult even on short programs. To build these proofs, some automation is needed. It can be obtained by combining SMT solvers and *Gappa* [64], [82], [46], [10]. 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 [68], [113].

[0] http://toccata.lri.fr/gallery/index.en.html
[0] http://hisseo.saclay.inria.fr/
[0] http://www.lri.fr/~sboldo/research.html

The directions of research concerning floating-point programs that we 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 ongoing development of the Flocq library continues 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 advanced algorithms to compute efficiently with floats, which operate at the bit-level. The formalization of real numbers is being 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 appear to loose their effectiveness when floating-point arithmetic is involved [77]. 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

Our goal is to handle floating-point programs that are related to continuous quantities. This includes numerical analysis programs we have already worked on [63], [62], [4]. 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 other programs of this kind. 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 nevertheless 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, 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 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 3.2.2 , directly as a certified program in *Coq* or *Why3*.

VERIDIS Project-Team

3. Research Program

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 designed for developing concurrent and distributed systems and algorithms that are firmly grounded on precise mathematical and semantical abstractions. Our common objective is to advance the state of the art in interactive and automatic deduction techniques, and their combinations, resulting in powerful tools for the computer-aided verification of distributed systems and protocols. Our techniques and tools support sound methods for the development of trustworthy distributed systems that scale to algorithms relevant for practical applications.

VeriDis members from Saarbrücken are developing SPASS [10], one of the leading automated theorem provers for first-order logic based on the superposition calculus [46]. Recent extensions to the system include the integration of dedicated reasoning procedures for specific theories, such as linear arithmetic [56], [45], 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 [57] and automated reasoning mechanisms these induce. Finally, members of the group design effective quantifier elimination methods and decision procedures for algebraic theories, supported by their efficient implementation in the Redlog system [4].

In a complementary approach to automated deduction, VeriDis members from Nancy develop veriT [1], an SMT (Satisfiability Modulo Theories [48]) 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$^+$ [52] specifications. Our prover relies on a declarative proof language, and we contribute several automatic backends [3].

3.2. Formal Methods for Developing Algorithms and Systems

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], [9], and in applying them to concrete use cases. In particular, the concept of *refinement* [44], [47], [54] in state-based modeling formalisms is central to our approach. Its basic idea is to present an algorithm or implementation through 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 establish precise proof obligations that can be discharged to a high degree 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.

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 modern system paradigms, including ad-hoc and overlay networks or peer-to-peer systems, and they must be integrated for concrete applications.

CARTE Project-Team

3. Research Program

3.1. Computer Virology

From a historical point of view, the first official virus appeared in 1983 on Vax-PDP 11. At the same time, a series of papers was published which always remains a reference in computer virology: Thompson [71], Cohen [39] and Adleman [28]. The literature which explains and discusses practical issues is quite extensive [44], [46]. 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 [59] 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, a worm is an autonous self-replicating malware and so falls 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 [43], [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 [69]with continuous time.

3.3. Rewriting

The rewriting paradigm is now widely used for specifying, modelizing, programming and proving. It allows one 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 [66].

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 [30], [42], [70].

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 [41]. 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 proposed an approach using rewriting techniques to abstract program behaviors for detecting suspicious or malicious programs [31], [32].

CASSIS Project-Team

3. Research Program

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 constraintsto 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.

COMETE Project-Team

3. Research Program

3.1. Probability and information theory

Participants: Nicolas Bordenabe, Konstantinos Chatzikokolakis, Thomas Given-Wilson, Yusuke Kawamoto, 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 is able to exploit such information.

The recent tendency is to use an information theoretic approach to model the problem and define the leakage in a quantitative way. The idea is to consider the 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 as a 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). Most of the proposals in the literature use Shannon entropy, which is the most established notion of entropy in information theory. We, however, consider also other notions, in particular Rényi min-entropy, which seems to be more appropriate for security in common scenarios like one-try attacks.

3.2. Expressiveness of Concurrent Formalisms

Participants: Catuscia Palamidessi, Luis Pino, Frank Valencia.

We study computational models and languages for distributed, 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, also taking into account the issue of (efficient) implementability.

3.3. Concurrent constraint programming

Participants: Michell Guzman, Yamil Salim Perchy, Luis Pino, Frank Valencia.

Concurrent constraint programming (ccp) is a well established process calculus for modeling systems where agents interact by posting and asking information in a store, much like in users interact in *social networks*. 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 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. **(a)** 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. **(b)** The extension of ccp with constructs to capture emergent systems such as those in social networks and cloud computing.

3.4. Model checking

Participants: Konstantinos Chatzikokolakis, Catuscia Palamidessi.

Model checking addresses the problem of establishing whether a given specification satisfies a certain property. We are interested in developing model-checking techniques for verifying concurrent systems of the kind explained above. In particular, we focus on security and privacy, i.e., on the problem of proving that a given system satisfies the intended security or privacy properties. Since the properties we are interested in have a probabilistic nature, we use probabilistic automata to model the protocols. A challenging problem is represented by the fact that the interplay between nondeterminism and probability, which in security presents subtleties that cannot be handled with the traditional notion of a scheduler,

DICE Team

3. Research Program

3.1. Introduction

Our aim is to address both

- challenges in the field of information technology, as well as
- trans-disciplinary issues emerging from the global impact of the digital revolution.

We believe that addressing both directions at the same time is an efficient way to be relevant in each of them.

We focus on intermediation platforms, which are becoming dominant systems in the Web industries. Intermediation platforms are systems which offer services to their users, which are well tuned for their expectation, thanks to the knowledge the platform has accumulated on usage. Search engines, social networks are examples of intermediation platforms. They ensure a gatekeeping function, always in direct contact to their users, providing them with the most relevant information or contact. Their economic model relies on a biface economy, with two types of users, one subsidizing the other. Their impact goes beyond the Web, and they disrupt step by step all sectors of the economy, transportation, Press, education, to name a few.

So far as IT is concerned, we focus on the technologies used for intermediation, which are at the basis of the largest online systems. For the transdisciplinary questions, we focus mostly on the new equilibrium that is resulting from the evolution of power balances due mostly to intermediation platforms.

3.2. Intermediation technologies

DICE focuses on intermediation platforms because of the central role they play in the new economy.

Intermediation platforms connect users to one another, or users to services with a very high accuracy. They rely on innovations both technological and social, which were unthinkable only ten years ago, when Facebook started. They allow communication and interaction between billions of users, gathered in the same digital space, both producers and consumers of data and services. State-of-the-art intermediation platforms include Facebook, Google, Twitter, GitHub, as well as Wikipedia, StackOverflow or Quora. These systems share a common design and their market penetration follows the same pattern. They are built around an initial minimal viable product based on a somehow naive low-tech implementation, which evolves after a few years of improvement to Web giants. Their domination now contributes to standardize the web industry, that means in particular:

- Gatekeeping, a direct relation with users together with services satisfying users' needs;
- Continuous data flows mapped to users' profiles;
- Search engines associating, in a relevant manner, producers, consumers and services.

These common characteristics lead to new software architectural standards, which are shared by all these systems, and used in the peripheral services developed in the ecosystem around their API:

- Authentication systems: openId, OAuth, ...
- Object graphs: opengraph, follower/followee scheme, ...
- DataFlow engines: Twitter storm, Google millwheel, ...
- Databases: noSql, keyValues stores, ...
- Web Browsers: javascript, dart, MEAN (Mongo, Express, Angular, Node),...

These architectural components impact the whole digital world. DICE targets systems that use standard architecture services but preserve some aspects we consider as disruptive ones: *data concentration, data symmetry* and *computational subsidiarity*. Our current research activity includes the following directions:

- Peer-to-peer design for preserving users' primary data;
- Third parties based organic systems providing subsidiary data computation hosted at peer sites;
- In-Browser applications that impact mobile device and demonstrate instantaneous usability;
- Flow-based computing enabling a stream based exchange of information between peers at runtime.

3.3. Economy of the digital world

The digital revolution is impacting all sectors of our societies and organizations, education, energy, transportation, health, to name a few. This revolution results in the phenomena of Schumpeter's *creative destruction*, with the disappearance of traditional sectors and the creation of new ones. Our societies, which did not anticipate the depth of the changes, have to struggle to adapt to the pace of the development of the industry. Legal reforms in various important sectors including taxation are at stake. Some countries, more reactive than others, are clearly pulling the changes, exploiting the benefits for businesses and the capacity to generate information and value, while others are trying to catch up with the global trends.

Data form the bricks of the information society, and their flows between users and services constitute the blood of the industry. We focus in DICE on the strategic role of data in this revolution, and in particular on the systems that harvest the data and concentrate it.

We are also interested in the global political impact of this revolution, which deeply changes the relations between governments and citizens. If the privacy is the focus of considerable attention, together with the state surveillance, in Europe in particular, it is only one aspect of the new knowledge made available. Social media produce considerable knowledge not only on individuals, but on populations as well, their economic fate, their political orientation, etc. On the other hand, open data from governments allow citizens to monitor the action of their governments, as well as to contribute to it. The digital revolution, with the capacity to access information in ways unthinkable in the recent past, modifies completely the balance of powers between citizens, states and corporations.

We investigate the digital world, and more precisely the power relations, from an interdisciplinary perspective. We simultaneously quantify power relations by studying data flows and the rise of intermediation platforms and produce an economical, political and ethical analysis of this new state of affairs. Namely, we show that areas such as the US or China dominate the digital world when others, such as Europe, do not succeed in proposing widely used intermediation platforms. This situation generates several conflicts between countries and companies and prevents weak countries from promoting their values and policies.

A new trend is emerging in the humanities, around in particular the digital studies, which promote the cooperation between computer scientists and specialists of social sciences. Among them, the Berkman center for Internet and Society in Harvard, the Medialab at MIT, or the Web Science Institute in the UK have gained strong visibility. They address positive as well as negative externalities of IT for societies, that is the new potentials offered as well as their risks. The Center for Information Technology Research in the Interest of Society in Berkeley also addresses fundamental political impacts on democracy, which can be enhanced by open data as well as another philosophy of political power as currently implemented in the State of California for instance. The Open Data Institute in the UK is also a leading center for political issues in Europe. France should catch up on these research trends, at the intersection of different scientific fields.

PRIVATICS Project-Team (section vide)

PROSECCO Project-Team

3. Research Program

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 (including several in 2014) 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 [86]; it has motivated a serious research effort on the formal analysis of cryptographic protocols, starting with [84] 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* [81]. 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 [79], [71], correspondences (including authentication) [80], and observational equivalences [78]. 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 [75], JFK [72], and Web Services Security [77]. 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 [82], results which were extended by formal analysis work using a Dolev-Yao style model of the standard [83]. 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 and F*

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# [75]. 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 [73] 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 [76]. F* [87] 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.

A different approach is to directly verify cryptographic applications in the computational model by typing. A recent work [85] 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.

An alternative to direct computation proofs is to identify the cryptographic assumptions under which symbolic proofs, which are typically easier to derive automatically, can be mapped to computational proofs. This line of research is sometimes called computational soundness and the extent of its applicability to real-world cryptographic protocols is an active area of investigation.

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. To this end, we have developed two security-oriented type systems for JavaScript, Defensive JavaScript [74],[65] and TS* [62], and used them to guarantee security properties for a number of JavaScript applications. 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 F* to verify their correctness. We also propose to translate verified F* web applications to JavaScript via a verified compiler that preserves the semantics of F* programs in JavaScript.