Magistère Informatique de Grenoble | MIG

Conférences de fin de stage

Le magistère est une formation complémetaire ayant pour objectif d'initier des étudiants au monde de la recherche en informatique

JEUDI 29 AOÛT

14:00	L3	**FARHAT Amine**	Procedural stylisation of 3D animations - MAVERICK (INRIA)
14:30	L3	**ANTHOINE Gaspard**	Proof of retrievability based on linear algebra - CASC (LJK)
15:35	L3	**BOULANGER Louis**	Rust for Performance: Implementing a parallel PMA - DataMove (LIG/INRIA)
16:05	L3	**JOZEAU Ludovic**	Tools for debugging lock-free programs - PACSS (VERIMAG)

VENDREDI 30 AOÛT

09:15	M2	**GRAND Maxence**	Action Model Learning with System Interaction - MARVIN (LIG)
09:55	M2	**CALKA Maxime**	Biomechanical Modelisation of the Human Tongue: Subject-Specific Generation and Model Order Reduction - (GMCAO) TIMC-IMAG
10:40	M2	**BRIGNON Enzo**	Toward automated test generation and fault tolerance analysis for Programmable Logic Controllers - SLS (TIMA)
13:30	M2	**BARROIS Florian**	Blockchain versus PKI : setting up and analysis of two architectures for securing an industrial system - LSOSP (CEA)
14:10	M1	**DEFAUW Nils**	Joint analysis in frequency and in pattern of large parallel application traces - Polaris (LIG)
14:55	M1	**BENAISSA Manal**	Development of tracing tools and trace analysis for OpenMP - Polaris (LIG)

Grand Amphithéatre de l'IMAG

Procedural stylisation of 3D Animations

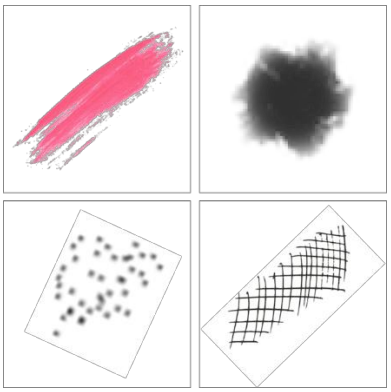
Amine FARHAT – L3 INFO

Under the supervision of **Romain VERGNE** and **Joelle THOLLOT**
Maverick Team - **INRIA**

When it comes to generating an image, there are two schools of thought in the computer graphics field:

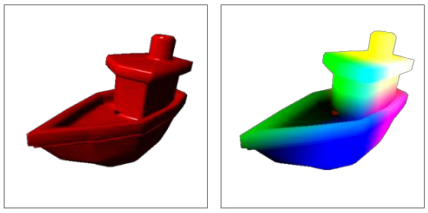
- Realistic rendering: which aims at producing images almost indistinguishable from real pictures.
- Expressive rendering: which focuses more on creating stylized images, giving them the looks of sketches or paintings, for example.

While there are countless different ways to engineer a stylized picture procedurally, the method we worked on relies solely on the reproduction of a 2D scene using an atomic building block – referred to as a "**splat**" – which consists of a simple flat image.[11]



[1] A few examples of splats

The way in which the algorithm will decide on the size, the shape, the color, or the order of appearance of a splat is determined by a collection of spatial data, gathered from a number of 2D images – labeled **G-Buffers** – generated alongside the original picture.



[2] An input image and its corresponding position buffer

Thus, it becomes possible to procedurally generate, on a 3D space, an aggregate of **anchor points**. These points will later be used to position the splats on the 2D screen.

As such, our method maintains the spatial coherency of the input image, while giving enough control to reproduce a considerable selection of different art styles, by variation of the parameters previously cited.

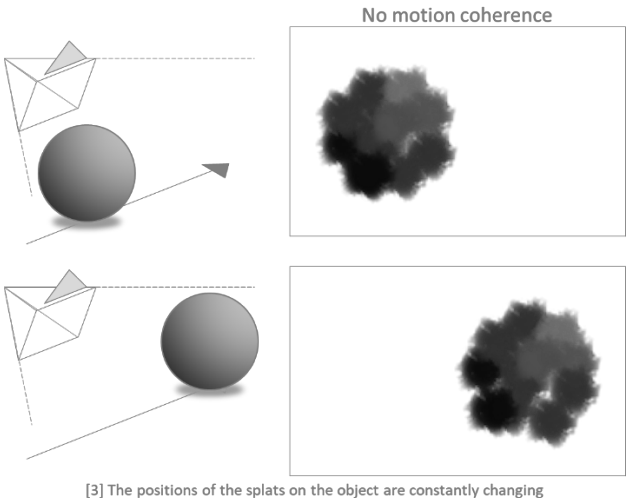
The essence of the problem truly manifests when processing animated scenes; moving objects in a 3D space requires to maintain some degree of **movement coherence**.[12]

My contribution mainly focused on tackling the issue of movement coherence for complex scenes, i.e. scenes composed of different objects with distinct motions.

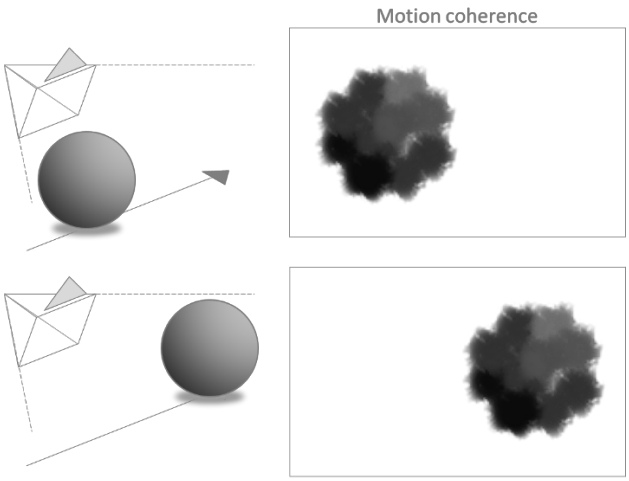
Since the anchor points are generated according to the absolute position of the camera, an object moving relatively to the origin of the scene will appear to be trembling: since the anchor points don't follow the object, they seem frozen in place, and simply appear and disappear as the object moves around. This phenomenon was dubbed "**popping**".

Thus, the solution I proposed was to factor in the **transformation matrices** of each object on the screen. These four by four matrices are commonly used in computer graphics to store and process data on the position, rotation and scale of a 3D coordinate.

By doing so, the aspect of the object remains visually the same. The points will appear to be anchored on its surface as it moves and deforms.



[3] The positions of the splats on the object are constantly changing



[4] The positions of the splats remain bound to the object as it moves

This minor improvement on the stylization process allowed us to generate short animations with a more visually pleasing aesthetic.

SOURCES

[11] : Romain Vergne, David Vanderhaeghe, Jiazhou Chen, Pascal Barla, Xavier Granier, and Christophe Schlick. Im-plicit Brushes for Stylized Line-based Rendering.Computer Graphics Forum, 30(2):513–522, April 2011.

[12] : Pierre Bénard, Adrien Bousseau, and Joëlle Thollot. State-of-the-Art Report on Temporal Coherence for StylizedAnimations.Computer Graphics Forum, 30(8):2367–2386, December 2011.

Procedural stylisation of 3D Animations

Farhat Amine
Joelle Thollot - Romain Vergne (MAVERICK - INRIA)

Introduction

When it comes to generating an image, there are two schools of thought in the computer graphics field:

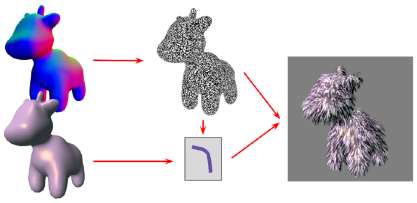
- Realistic rendering: which aims at producing images almost indistinguishable from real pictures.
- Expressive rendering: which focuses more on creating stylized images, giving them the looks of sketches or paintings, for example.

In the animation industry, it is often preffered to rely on realistic rendering for computer generated sceneries. This tendency is mainly motivated by efficiency and economic concerns, since hand-drawn animation tend to be costly and time-consuming.

For this reason alone, it is easy to understand the many applications of a technology capable of generating complex animations in a hand-painted style, for example.

Method

Our reasarch efforts were mostly focused on finding a way to generate stylised images from an input 3D scene, by the rendition of multiple 2D images - labeled splats



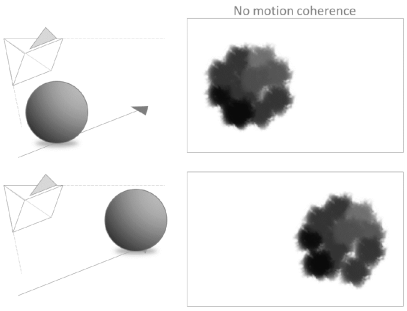
This method allows us to reproduce a wide varitety of artstyles, while maintaining the 3D aspect of the objects we are aiming to stylize.

Johannes Schmid, Martin Sebastian Senn, Markus Gross, and Robert W. Sumner. Overcoat: An implicit canvasfor 3d painting.ACM Trans. Graph., 30(4):28:1–28:10, July 2011.

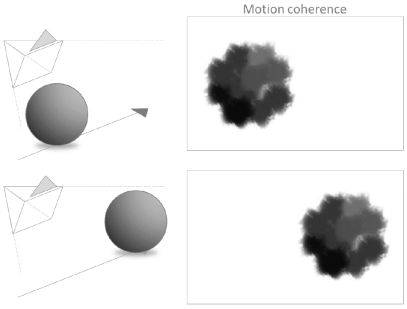
Problem

The main problem arises when we use the previously cited method to stylize a sequence of images, supposed to represent an object in motion. Since the algorithm has virtually no way of differentiating two distinct objects in space, or interpolating a movement from two frames, the output image presents an important amout of anomalies. This problem was labeled temporal coherency by Pierre Benard in his reseach on stylized animations.

Pierre Bénard, Adrien Bousseau, Joëlle Thollot. State-of-the-Art Report on Temporal Coher-ence for Stylized Animations. Computer Graphics Forum, Wiley, 2011, 30



[3] The positions of the splats on the object are constantly changing



[4] The positions of the splats remain bound to the object as it moves

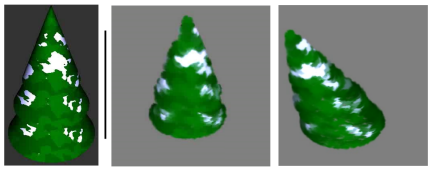
Solution

The issues described in the previous section arise from the way the anchor points are created for each frame.

Since the anchor points are generated according to the absolute position of the camera, an object moving relatively to the origin of the scene will appear to be trembling: since the anchor points don't follow the object, they seem frozen in place, and simply appear and disappear as the object moves around.

Thus, the solution I proposed was to factor in the transformation matrices of each object on the screen. These four by four matrices are commonly used in computer graphics to store and process data on the position, rotation and scale of a 3D coordinate.
By doing so, the aspect of the object remains visually the same. The points will appear to be anchored on its surface as it moves and deforms.

Moreover, since each objects displays information on its velocity and its orientation, it is possible to distort the aforementioned splats themselves to accomodate and better translate its movements.



Conclusion

At the end of my internship, I was able to bring a minor yet useful contribution to the stylisation of complex animations.

However, this added layer of complexity forces us to consider additional possibilities in the way the anchor points are generated.

For instance, as an object increases in size, it occupies more space in the screen, thus requiring more anchor points to cover its whole visible surface. The addition of anchor points can be handled in many different ways; however, it is quite tricky to determine the best way to do it.

In the end, we are only trying to flesh out a number of tools that would allow for a wide variety of styles and options.

Proof of retrievability based on linear algebra

Gaspard Anthoine under the supervision of Clément Pernet

Team CASC,Laboratoire Jean Kuntzmann, University of Grenoble-Alpes

Introduction. In the context of remote storage on insecure resources, evidence of recoverability provides the user with guarantees that the remote server owns the data. The internship focuses on a new protocol that effectively solves this problem in practice by using probabilistic certification techniques in linear algebra. We want to store a large amount of data on a server and be able to ensure through audits that the server keeps the data without the server having to send it all to us to check. This will be called "Provable Data Possession" (PDP) (1).
If the various audits make it possible to recover part of the data and if multiplying the audits ultimately makes it possible to recover the entire data, we will then speak of "Proof of Retrievability" (PoR).

A new solution. It is possible to see the data as a large M matrix with coefficients in a finite field. We can then use probabilistic linear algebra methods such as the Freivalds algorithm (2). We can initialize our challenge like in table 1.

to … [unfinished sentence]

[avoid notation in abstract if you don't have place to explain it]

Table 1. Matrix-form database probabilistic Initialisation

Server	Communications	Client
	$\xleftarrow{M}$	$u \xleftarrow{\$} S^n \subseteq \mathbb{F}_q^n$
		$v \leftarrow Mu$

As we can see in this version, the complexity on the client side is about $O(m)$. We can now perform the audit as in the table 2.

Table 2. Matrix-form database probabilistic audit

Server	Communications	Client
	$\xleftarrow{x}$	$x \xleftarrow{\$} S^m \subseteq \mathbb{F}_q^m$
$\mathbb{F}_q^n \ni y^\intercal \leftarrow x^\intercal M$	$\xrightarrow{y}$	
		$y^\intercal u \stackrel{?}{=} x^\intercal v$

Optimization of the finite field size

The probability of success will depend on the size of the body of the matrix elements, a larger body will have a greater probability of detecting a change in the matrix M but the cost of multiplication $x^\intercal M$ will be more important in practice. Part of my internship is to find an optimal finite size for the calculation of audits. We will take into account the fact that to reduce the bandwidth cost of audits (sending the vector y), we want to limit as much as possible the size in number of columns of the matrix (ideally $log_2(N)$ with N the total size of the data in bits).
The calculation of the optimal size is all the more complicated because it avoids the client having to store the vector v which is the size of the number of lines in the matrix ($O(m)$). We store it on the server and have the server calculate the product $x^\intercal v$. Only to avoid the server to easily forge a response that would pass the audit, the v vector must be encrypted. We will therefore use a semi-homorphic encryption that will allow algebraic operations to be performed directly on the encrypted vector. Carrying out operations on ciphertexts is much more expensive than operations on plaintext, so it is interesting to try to limit them by reducing the number of lines in the matrix. Indeed, either $M \in \mathbb{F}_q^{m \times n}$ and $\gamma = \lfloor log_2(q) \rfloor$ the number of bits contained in a matrix coefficient. We have $N = \gamma mn$ and therefore $m = \frac{N}{\gamma n}$ with in the ideal case to reduce the cost of communications $n = log_2(N)$. We see here the two parameters that will allow to vary the number of columns of the matrix: γ the number of bits that a coefficient can contain and n the number of columns of the matrix. It will therefore be necessary to find an optimal combination of these two parameters in order to optimize the audit in practice while keeping in mind that we want n to be small. Keeping γ under 22 bits will allow to use FFLAS-FFPACK (3).

Doing updates

The protocol must also allow to do updates. We can imagine to simply update the vector w with a new value a, $\Delta w_i \leftarrow E((a - m_{i,j})u_j)$. But this is leaking information at every update : $\Delta w_i^{\frac{1}{a-m_{i,j}}} = \mathcal{E}(u_j)$, after $\Theta(n)$ updates we have obtained $\mathcal{E}(u)$. This will permit to fake a δ when doing audit. To avoid the attack we've seen above, we are using two different encryptions and two new random values at each update, $c_i = c_i \cdot \Delta c_i$.

- $v = Mu + t + b$, $w = E_1(v), c = E_2(b)$
- We take a random μ and a random u'_j, we calculate $\Delta w_i = \mathcal{E}_1(au'_j - m_{i,j}u_j + \mu)$, $\Delta u_j = \mathcal{E}_1(u'_j - u_j)$
- $\beta = x^\intercal \odot c$
- send also $\Delta c_i = E_2(\mu)$ and store it in a new vector

When doing an audit the server will compute $\beta = x^\intercal \odot c$. The client will have to check $\mathcal{D}_1(\delta) - \mathcal{D}_2(\beta) \stackrel{?}{=} y^\intercal u + x^\intercal t$.

Conclusion

The protype enables us to see the importance of different parameters for the speed of an update. Using FFLAS-FFPACK (3) seems to be the fastest. The proofs of security are still to be done.

Bibliography

1. Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. pages 598–609, 01 2007. doi: 10.1145/1315245.1315318.
2. Rūsiņš Freivalds. Fast probabilistic algorithms. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *LNCS*, pages 57–69, Olomouc, Czechoslovakia, September 1979. Springer-Verlag. doi: 10.1007/3-540-09526-8_5.
3. Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages. *ACM Trans. on Mathematical Software (TOMS)*, 35(3):1–42, 2008. ISSN 0098-3500. doi: 10.1145/1391989.1391992.

Proof of retrievability based on linear algebra

Gaspard Anthoine under the supervision of Clément Pernet
Team CASC, Laboratoire Jean Kuntzmann, University of Grenoble-Alpes

Problem

While storing sensitive data on distant storage we want to be sure that the server will keep the data without having to send them back entirely. First introduction of this problem was in Ateniese and al. [1] and provides what is called a Proof of Data possession protocol (PDP). The protocol consists of two phases, an initialisation where you send the data at the end and an audit phase where you will challenge the server to verify that it is storing the data.

Semi-homomorphic encryption

Homomorphic Encryption Let $\mathcal{E}(m) : E \to F$ be an encryption function m. Let $\mathcal{D}(m) : F \to E$ be de decryption function associated.

- $\mathcal{D}(\mathcal{E}(m_1) +_F \mathcal{E}(m_2)) = m_1 +_E m_2$
- $\mathcal{D}(\mathcal{E}(m_1) \times_F \mathcal{E}(m_2)) = m_1 \times_E m_2$

The most promising fully homomorphic encryption is based on lattice by Gentry et al. [4] is too expensive to be used in practice.

Semi-homomorphic Encryption

- $\mathcal{D}(\mathcal{E}(m_1) +_F \mathcal{E}(m_2)) = m_1 +_E m_2$
- $\mathcal{D}(\mathcal{E}(m) \cdot x) = m \times x$ avec x en clair

Different encryption respecting this operation exists for example Paillier [5].

A new protocol based on probabilistic linear algebra

We made the arbitrary choice to see the data as a matrix M : $M \in \mathbb{F}_q^{m \times n}$. Either N the size of data in bits, $N = \gamma mn$ with $\gamma = \lfloor log_2(q) \rfloor$ the number of bits in a coefficient. It is then possible to use some probabilistic algorithm like Freivald's one [3]. The idea will be to do the initialisation as in tab 1.

Server	Communications	Client
	$\xleftarrow{M}$	$u \xleftarrow{\$} S^n \subseteq \mathbb{F}_q^n$
		$v \leftarrow Mu$

Tab. 1: Simple initialisation

You can then do a simple audit with complexity $O(m)$ client side.

Server	Communications	Client
	$\xleftarrow{x}$	$x \xleftarrow{\$} S^m \subseteq \mathbb{F}_q^m$
$\mathbb{F}_q^n \ni y^\intercal \leftarrow x^\intercal M$	$\xrightarrow{y}$	
		$y^\intercal u \stackrel{?}{==} x^\intercal v$

Tab. 2: Simple audit

Storing v on the server

Because we want to limit complexity client side we are going to store the vector v server side we will use semi-homorphic encryption to do operations on the server. As the server is doing both δ and $y^\intercal$ and because we want to avoid replay attack we should use a τ stored client side.

	Server	Communications	Client
Setup		$N = mn \log_2(q)$	$\tau \xleftarrow{\$} S \subseteq \mathbb{F}_q^*$
			$u \xleftarrow{\$} S^n \subseteq \mathbb{F}_q^n$
			form $t = [\tau, \tau^2, \dots, \tau^m]^\intercal$
		$\xleftarrow{M,w}$	$v = Mu + t$, $w = E(v)$
			discard M, v, w
Audit	form $x = [r, r^2, \dots, r^m]^\intercal$	$\xleftarrow{r}$	$r \xleftarrow{\$} S \subseteq \mathbb{F}_q^*$
	$\delta = x^\intercal \odot w$	$\xrightarrow{\delta}$	
	$y^\intercal = x^\intercal M$	$\xrightarrow{y}$	$D(\delta) \stackrel{?}{==} y^\intercal u + x^\intercal t$

Tab. 3: audit with masked encryption

$x^\intercal t$ will be fast because it's simply a geometric progression, $x^\intercal t = \frac{1-(r\tau)^m}{1-r\tau}$ and complexity $O(\log(m))$.

Optimisation of finite field size

Size of the finite field will influence the speed of computation. Using field smaller than 22 bits will allow to use FFLAS-FFPACK [2].

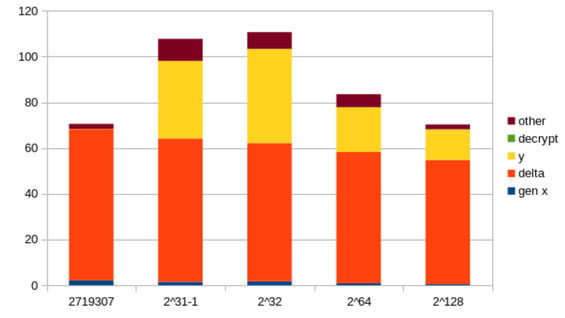


Fig. 1: Benchmark for 1 gigabit of data

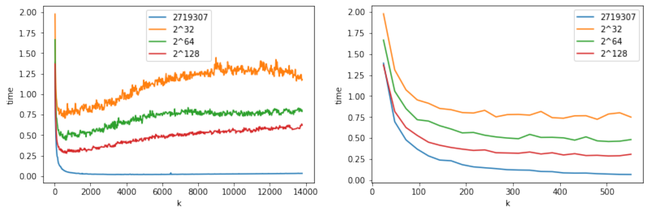


Fig. 2: different values for k, between $\log(n)$ and $\log^3(n)$, between $\log(n)$ and $\log^2(n)$

Simple updates is problematic

We can imagine to simply update the vector w at every update like in table 4. But this is leaking information at every update : $\Delta w_i^{\frac{1}{a-m_{i,j}}} = \mathcal{E}(u_j)$, after $\Theta(n)$ updates we have obtained $\mathcal{E}(u)$. This will permit to fake a δ when doing audit.

Server	Communications	Client
	$\xleftarrow{i,j}$	$1 \le i \le m$, $1 \le j \le n$
block $N_k \ni m_{i,j}$		
$b_1 \dots, b_\delta$, each in $\{0,1\}^\lambda$	$\xrightarrow{N_k; b_1, \dots, b_\delta}$	
		$H(\dots H(i\|\|j\|\|N_k\|\|b_1)\dots) \stackrel{?}{==} r_0$
		Extract $m_{i,j} \in N_k$
update block $N_k \ni a$	$\xleftarrow{a, \Delta w_i}$	$\Delta w_i \leftarrow E((a - m_{i,j})u_j)$
$w_i \leftarrow w_i \cdot \Delta w_i$		

Tab. 4: Bad update

Use of two encryptions

To avoid the attack we've seen above, we are using two different encryptions and two new random values at each update.

- $v = Mu + t + b$, $w = E_1(v)$, $c = E_2(b)$
- We take a random μ and a random u'_j, we calculate $\Delta w_i = \mathcal{E}_1(au'_j - m_{i,j}u_j + \mu)$, $\Delta u_j = \mathcal{E}_1(u'_j - u_j)$
- $\beta = x^\intercal \odot c$
- send also $w = E_2(\mu)$

When doing an audit the server will compute $\beta = x^\intercal \odot c$. The client will have to check $\mathcal{D}_1(\delta) - \mathcal{D}_2(\beta) \stackrel{?}{==} y^\intercal u + x^\intercal t$.

References

[1] Giuseppe Ateniese et al. "Provable Data Possession at Untrusted Stores". In: Jan. 2007, pp. 598–609. DOI: 10.1145/1315245.1315318.

[2] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. "Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK Packages". In: *ACM Trans. on Mathematical Software (TOMS)* 35.3 (2008), pp. 1–42. ISSN: 0098-3500. DOI: 10.1145/1391989.1391992.

[3] Rūsiņš Freivalds. "Fast Probabilistic Algorithms". In: *Mathematical Foundations of Computer Science 1979*. Ed. by J. Bečvář. Vol. 74. LNCS. Olomouc, Czechoslovakia: Springer-Verlag, Sept. 1979, pp. 57–69. DOI: 10.1007/3-540-09526-8_5.

[4] Craig Gentry, Amit Sahai, and Brent Waters. *Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based*. Cryptology ePrint Archive, Report 2013/340. https://eprint.iacr.org/2013/340. 2013.

[5] Pascal Paillier. "Public-key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'99. Prague, Czech Republic: Springer-Verlag, 1999, pp. 223–238. ISBN: 3-540-65889-0. URL: http://dl.acm.org/citation.cfm?id=1756123.1756146.

Rust for parallel programming : Packed-memory Array implementation

Louis Boulanger

July 2019

1 Introduction

Parallel programming is challenging and requires a lot of preparation and care: a whole class of problems arise when working in parallel. This is why programmers have searched ways to improve the quality of parallel programming for decades. Especially, they have strived to make programs more fault-proof, while keeping the advantages of parallelism, such as increased speed of execution.

When adapting existing programs or structures in parallel, one must keep in mind these problems and choose the correct tools. ~~This is why~~ we chose to use the Rust programming language for implementing the Packed-memory Array [1] structure in parallel programming.

[which ones ?]

[insufficient justification]

2 Packed-memory Array

We implemented the Packed-memory Array (**PMA**) structure described in [1]. The Packed-memory Array is designed to keep elements sorted, while allowing efficient insertions by keeping *gaps* in the underlying array, and using a tree-like organisation to ensure density bounds on each segment.

In the PMA, the array is divided into *segments* of fixed size. Segments are grouped into *windows*, which act like subtrees. For each level of the implicit tree structure, we define *density bounds*, which determine how many gaps are allowed in a window. For most of the tests and benchmarks we performed, we used $\rho_0 = 0.08$, $\tau_0 = 0.92$, $\rho_h = 0.3$, $\tau_h = 0.7$ (these values are given in [1] and yield the best performances).

[undefined notation]

3 Rust parallel implementation

To implement the PMA in Rust with parallelism, we used a modified version of the parallelism library of rust, rayon, called **rayon-adaptive** (created and maintained by Frédéric Wagner). It uses adaptive algorithms for scheduling, yielding better performances.

The current implementation of the PMA supports insertions of one, or many elements (*bulk insertions*) ; the latter is where parallelism is important. The algorithm for inserting many elements into the PMA resembles the following :

if *the PMA would be ~~too~~ full* **then**
  double the PMA size and merge into the elements;
else
  if *the target window would be ~~too~~ full* **then**
    merge the window and the input;
  else
    divide the input in two (parts) with a pivot;
    recursively insert in the left and right subwindows; (them)
  end
end

Algorithm 1: Bulk insertion algorithm

The merge operation is the true parallel component of the PMA: by specifying how a merge operation could be divided into two, smaller merge operations, we allow the scheduler to efficiently share them across multiple processors, without losing the sorted aspect.

4 Conclusion

This implementation is sadly not yet complete: it lacks the remove operation, and could benefit from more optimisation. But it's a proof of concept and shows how using an efficient language like Rust enables programmers to easily write parallel programs, and how it could facilitate parallelism in general algorithms.

References

[1] Marie Durand, Bruno Raffin, and François Faure. A Packed Memory Array to Keep Moving Particles Sorted. In Jan Bender, Arjan Kuijper, Di-

eter W. Fellner, and Eric Guérin, editors, *VRIPHYS - Ninth Workshop on Virtual Reality Interactions and Physical Simulations*, Ninth Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS), pages 69–77, Darmstadt, Germany, December 2012. Jan Bender and Arjan Kuijper and Dieter W. Fellner and Eric Guérin, The Eurographics Association.

Tools for debugging lock-free programs

Ludovic Jozeau
Supervised by: David Monniaux and Michaël Périn.

1 Threads synchronization

Multithreaded programs can achieve better performance than single threaded ones. The counterpart is that it is harder to develop programs with multiple threads running at the same time because we must take into account the possible different executions and correctly synchronize threads to make them communicate.

1.1 Locks

The Traditional way to synchronize threads is using locks [mutex, 2019]. This is relatively simple to do, we lock before the critical section, then we unlock after and the mutual exclusion is set up. Locks are expensive and counterproductive on high contention (when many threads lock and unlock the same lock very often).

1.2 Lock-free program

[atomic data =?]

Another way to synchronize threads exists with no locks. In this case we use atomic[atomic, 2019] data and actions to check the state of the system and make a decision: publish to other threads or redo the action if other threads have made updates. Taking advantage of atomic built-in processor primitives can speed up multithreaded programs but it is even more difficult to use correctly than locks.

2 Problems in lock-free programs

Lock-free methods brings specific problems such as:

- Data race: when two threads access a data and at least one is a write.
- Memory reordering[Howells *et al.*, 2018]: memory instruction (load/store) can be reordered leading to unexpected results.
- ABA' problem[Dechev *et al.*, 2010]: typical problem when using lock-free technics.
- Race condition: when an operation result change depending on threads interleaving.
- Live lock: when the state of two threads are changing influencing each other but none progressing.

3 Methodology

The purpose was to see what tools are available to debug lock-free programs then check what they cover to find out what is missing or how we could improve lock-free problem detection.

First we took 5 tools to debug multithreaded programs. Dynamic ones, which work when executing the program: DRD, Helgrind, ThreadSanitizer, Intel Inspector. And a static one, working on the source code to inspect all the possible executions: CppMem.

We then tested all the tools on 21 simple examples with 3 identified problems: data races, memory reordering and the ABA problem. These examples included correct code to see if there was any false positive.

As dynamic tools can give different results on different configuration or runs, the tests were launched several times and with 3 different compiler: GCC, Clang and Clang with libc++.

4 Results

Most tools have poor support of lock-free programming. CppMem can shows us what is going on in lock-free programs but has serious limitations, it support a tiny subset of C++ and process small programs. ThreadSanitizer can work well in lock-free context but has not been seriously tested[tsan, 2019]. The ABA problem is rarely and only detected by ThreadSanitizer. Finally other tools does not support lock-free primitives.

All the work is available in 1200 lines of markdown here: `https://github.com/FederAndInk/lock_free_debug_tools_comparison`

References

[atomic, 2019] C++ reference for atomic. `https://en.cppreference.com/w/cpp/header/atomic`, 2019.

[Dechev *et al.*, 2010] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10, pages 185–192, Washington, DC, USA, 2010. IEEE Computer Society.

[Howells *et al.*, 2018] David Howells, Paul E McKenney, Will Deacon, and Peter Zijlstra. Linux kernel memory barriers. https://www.mjmwired.net/kernel/Documentation/memory-barriers.txt, 2018.

[mutex, 2019] C++ reference for mutex. https://en.cppreference.com/w/cpp/thread/mutex, 2019.

[tsan, 2019] Threadsanitizer faq. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual#faq, 2019.

Tools for debugging lock-free programs

Ludovic Jozeau supervized by David Monniaux and Michaël Périn
Verimag – PACSS - UGA

What is lock-free programming ?

Multithreaded program

```
7   int x = 0;
8
9   int main()
10  {
11    std::thread t1([] { //
12      x = 1999999999;
13    });
14    std::thread t2([] { //
15      print("x = {}\n", x);
16    });
```

Lock-free technics:
- Atomics
- Memory barriers
- Copy and swap

Mutual exclusion with locks

```
int x = 0;

int main()
{
  std::mutex  m;
  std::thread t1([&m] {
    m.lock();
    x = 1999999999;
    m.unlock();
  });
  std::thread t2([&m] {
    m.lock();
    print("x = {}\n", x);
    m.unlock();
});
```

Mutual exclusion with **lock-free** technics using atomic

```
std::atomic<int> x{0};

int main()
{
  std::thread t1([] { //
    x.store(1999999999);
  });
  std::thread t2([] { //
    print("x = {}\n", x.load())
  });
```

- Atomics are limited to certain types and operations
- Lock-free is harder to use but perform better than locks

Methodology for evaluating tools

5 tools tested
- DRD
- Helgrind
- ThreadSanitizer
- Intel Inspector
- CppMem

Over 21 in/correct programs
Covering 3 lock-free problems:
- Data race
- Memory reordering
- ABA’ problem

With 3 compilers
Under different conditions
With serveral runs for dynamic tools

Experiment report

- 1200 lines of markdown report
- 140 lines of code in 21 programs

All documented here:

https://github.com/FederAndInk/lock_free_debug_tools_comparison

ThreadSanitizer result:

Correct sample	Gcc	Clang	Clang libc++
Simple data race fix	✔	✔	✔
Simple data race fix relaxed	✔	✔	✔
Notification sequentially consistent	✔	✔	✔
Notification acquire release	✔	✔	✔
ABA' fixed	✔	✔	✔
store/load sequentially consistent	✔	✔	✔
store/load acquire release	✔	✔	✔
store/load relaxed	✔	✔	✔

Helgrind result:

Correct sample	Gcc	Clang	Clang libc++
Simple data race fix	~	✔	✔
Simple data race fix relaxed	~	~	~
Notification sequentially consistent	~	~	~
Notification acquire release	~	~	~
ABA' fixed	✔	✔	✔
store/load sequentially consistent	✔	✔	✔
store/load acquire release	✔	✔	✔
store/load relaxed	✔	✔	✔

✔: The tool has correctly detected the error

~: Out of scope, the tool report false positive

Conclusion

- ThreadSanitizer: not widely tested with lock-free
- CppMem: only for a tiny subset of C++ on tiny programs
- Other tools are not meant for lock-free

Action Model Learning with System Interaction

Maxence Grand
Supervised by : Damien Pellier and Humbert Fiorino
Université Grenoble Alpes
Laboratoire Informatique de Grenoble, Team MARVIN

Abstract

[A bit too specific for an abstract. Give informations on the context of usage of AMLSI. The aim of the abstract is too quickly checks if the paper worth a further reading.]

This paper presents the AMLSI (**A**ction **M**odel **L**earning with **S**ystem **I**nteraction) algorithm. This algorithm learn action model from positive and negative samples generated from interaction with an oracle. This algorithm is divided in three steps. At the first step, AMLSI learn a finite state automata corresponding to the regular grammar of samples. Then, AMLSI use an inductive algorithm to learn the first version of operators from the grammar learned previously. Finally, AMLSI use a refinement algorithm and a tabu search to refine operator's preconditions and effects. This algorithm deals with noisy and partial observations for intermediate and final states and it is able to learn negative preconditiions and static relations in preconditions.

Keywords PDDL, STRIPS, Automated Planning, Grammar Induction, Informant Learning, Action Model Learning

1 Introduction

In an automated planning systems, an autonomous agent achieve goals by producing sequences of actions, called plans, from a given action model [3]. Automated planning systems can be used to automate existing systems or tasks, such as an industrial process. To automate these systems we have to build an action model. These action models are symbolic representations of the different actions necessary to solve tasks. A typical way to describe the action models is to use an action languages such as the **P**lanning **D**omain **D**efinition **L**anguage (PDDL) [8].
The traditional way to build an action model is to ask a system expert to analyze the different tasks and manually build the action model using a language such as PDDL. However, it is very difficult and tedious to manually create these action models, even for experts. The problem is compounded if the system expert is not a planning expert, because the system expert have to analyze the domain and ask planning experts to build the action model using its analysis. [no CR here]
That's why, researchers have started to explore learning algorithm to reduce the human effort to build action models by learning from observed examples or plan traces. These algorithm take as input several example plans and try to learn an action model. The general way to collect example plans is to generate goal oriented plan solution. [How ?] Goal oriented plan solutions are generally expensive because a planner is needed to generate a large number of correct plans to be used by the leaner. To do this one must also have a pre-existing action model and a planner.
In this paper we propose the algorithm AMLSI (**A**ction **M**odel **L**earning with **S**ystem **I**nteraction), an algorithm using informant learning to learn action model. Unlike the majority of action model learning algorithms, the AMLSI algorithm itself generates its learning data. The AMLSI algorithm generates action sequences to learn an action model that copies an existing system. Action sequences are generated by querying an oracle and interacting with the system to be copied. This algorithm is based on informant learning described in [5] and regular grammar induction. This algorithm is divided in three steps. At the first step, AMLSI learns a finite state automaton corresponding to the regular grammar of training samples. Then, AMLSI use an inductive algorithm to learn the first version of operators from the grammar learned previously. Finally, AMLSI use a refinement algorithm and a tabu search to refine operator's preconditions and effects. This algorithm deals with noisy and partial observations for intermediate and final states.
The rest of the paper is organized as follows. We will first discuss some related works in the field of action model learning. Next, we will give the definition of our problem. Then, we will discuss about informant learning and about our method to generate training samples. Then we describe the detailed steps of our AMLSI algorithm. We then evaluate our algorithm in four planning domains to learn the action models and show some desirable properties of these learned action models. Finally, we will conclude paper and discuss our future works.

1.1 Related Work

There exist various approaches for the acquisition of action models. The early works such as EXPO [4] improve action models incrementally after observing some problem during plan execution. Another approach, such as Observer [15] learns from expert traces and subsequent simulations. Frequently, the acquisition problem consists of finding the action model from examples of plans. In other words, the problem is to learn a correct state transition function according to observed sequences of actions and states (see section 1.2). Some approach deals with incomplete information in intermediate states. ARMS algorithm [16] gathers knowledge on the statistical distribution of frequent sets of actions in the example plans. It then forms a weighted propositional satisfiability (weighted SAT) problem and resolves it using a weighted MAX-SAT solver. ARMS operates in two phases, where it first applies a frequent set mining algorithm to find the frequent subsets of plans that share a common set of parameters. It then applies a SAT algorithm for finding a consistent assignment of preconditions and effects. Then, Louga algorithm [6] uses a genetic algorithm to learn action effects and an ad-hoc algorithm to learn action preconditions. Then, some approach deals with noisy observations. For instance, in the IRALe approach [13], an autonomous agent uses an online active algorithm to explore an environment and learn incrementally the action model with noisy observations. Finally, some approach deals with both partial information and noise in observations. For instance, LSO-NIO approach [9] use a classifier with the kernel tricks method to learn action model. It decomposes the problem into first learning a transition function between states in the form of a set of classifiers, and then deriving explicit STRIPS rules from the classifiers' parameters. Finally, Plan-Milner algorithm [14] uses a classification algorithm, based on inductive rule learning techniques, to learn action models with discrete numerical values from incomplete and noisy observations.

1.2 Problem Statement

We work with classical STRIPS [2] planning that deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal condition. World states are modeled as sets of predicates that are true or false in those states and actions are changing validity of certain predicates. For instance, take q a state such as :

$$\begin{aligned} q = \quad & (at - robby\ r_1) \wedge \neg(at - robby\ r_2) \wedge (at\ b\ r_1) \wedge \\ & \neg(at\ b\ r_2) \wedge (free\ left) \wedge (free\ right) \wedge \\ & \neg(carry\ b\ right) \wedge \neg(carry\ b\ left) \end{aligned}$$

In natural language q means : "Robby and the ball b are in the room r_1, and both grippers, *right* and $left$ are free". And, if we perform the action $pick(b\ r_1\ right)$ the new state q' is :

$$\begin{aligned} q' = \quad & (at - robby\ r_1) \wedge \neg(at - robby\ r_2) \wedge \neg(at\ b\ r_1) \wedge \\ & \neg(at\ b\ r_2) \wedge (free\ left) \wedge \neg(free\ right) \wedge \\ & (carry\ b\ right) \wedge \neg(carry\ b\ left) \end{aligned}$$

Formally, let P be a set of all predicates modeling properties of world states. Then a state $q \in P$ is a set of predicates that are true or false in that state. Each action a is described by four sets of predicates $(\rho_a^+, \rho_a^-, \epsilon_a^+, \epsilon_a^-)$, where $\rho_a^+, \rho_a^-, \epsilon_a^+, \epsilon_a^- \in P$, $\rho_a^+ \cap \rho_a^- = \emptyset$ and $\epsilon_a^+ \cap \epsilon_a^- = \emptyset$. Let ρ_a^+, ρ_a^- describe positive and negative preconditions of action a, these are, predicates that must be true and false right before the action a. Action a is applicable to state q iff ρ_a^+, ρ_a^- are checked in state q. For instance, take action $pick(b\ r_1\ right)$:

$$\begin{aligned} \rho^+_{pick(b\ r_1\ right)} &= (at\ b\ r_1) \wedge (at - robby\ r_1) \wedge (free\ left) \\ \rho^-_{pick(b\ r_1\ right)} &= \neg(carry\ b\ right) \end{aligned}$$

The state q check both $\rho^+_{pick(b\ r_1\ right)}$ and $\rho^-_{pick(b\ r_1\ right)}$ so the action $pick(b\ r_1\ right)$ is feasible in state q. Let $\epsilon_a^+, \epsilon_a^-$ describe positive (add list) and negative (del list) effects of action a, that are, predicates that must be true and false after the execution of the action a. We denote $q' = \delta(q, a)$ a transition function that returns q' the state q where $\epsilon_a^+, \epsilon_a^-$ have been applied : $q' = \delta(q, a) = \{q \cup \epsilon_a^+\} \setminus \epsilon_a^-$. For instance, take action $pick(b\ r_1\ right)$:

$$\begin{aligned} \epsilon^+_{pick(b\ r_1\ right)} &= (carry\ b\ left) \\ \epsilon^-_{pick(b\ r_1\ right)} &= \neg(at\ b\ r_1) \wedge \neg(free\ right) \end{aligned}$$

After execution of the action $pick(b\ r_1\ right)$ in state q, the state $q' = \delta(q, a)$ is :

$$\begin{aligned} q' = \quad & (at - robby\ r_1) \wedge \neg(at - robby\ r_2) \wedge \neg(at\ b\ r_1) \wedge \\ & \neg(at\ b\ r_2) \wedge (free\ left) \wedge \neg(free\ right) \wedge \\ & (carry\ b\ right) \wedge \neg(carry\ b\ left) \end{aligned}$$

$\delta(q, a)$ is defined iff the action a is feasible in state q. Our problem is to learn a planning domain consisting of a set of operators and a set of predicates. Operators can be seen as a parameterized action. Each operator has a set of argument and specifies preconditions and effects as predicates over these arguments. For instance, we can describe the operator *pick* with the PDDL language as follow :

```
(:action pick
:parameters (?obj - ball
             ?room - room
             ?gripper - gripper)
:precondition (and
               (at ?obj ?room)
               (at-robby ?room)
               (free ?gripper)
               (not (carry ?obj ?gripper)))
:effect (and
         (carry ?obj ?gripper)
         (not (at ?obj ?room))
         (not (free ?gripper))
         ))
```

Actions are obtained by substituting constant for the arguments. The planning domain model is then specified by the set of predicates and the set of operators. The PDDL is the most used language for modeling planning domains; we used syntax of this language in our examples.
As we see it in section 2, observations are kept directly by the agent, so observations for intermediate state can be partial and noisy. A partial intermediate state is a state q, where some predicates are missing. For instance, a partial state q could be :

$$\begin{aligned} q = \quad & (at - robby\ r_1) \wedge missed[(at - robby\ r_2)] \wedge \\ & (at\ b\ r_1) \wedge \neg(at\ b\ r_2) \wedge missed[(free\ left)] \wedge \\ & (free\ right) \wedge \neg(carry\ b\ right) \wedge \\ & \neg(carry\ b\ left) \end{aligned}$$

where $missed[(free\ left)]$ and $missed[(at - robby\ r_2)]$ indicates that information about predicates $(free\ left)$ and $(at - robby\ r_2)$ are missing. Finally, observations can be noisy, i.e, there may be mistakes for some observed predicates' values. For instance, if we add noise to the partial state q, we could have :

$$\begin{aligned} q = \quad & (at - robby\ r_1) \wedge missed[(at - robby\ r_2)] \wedge \\ & (at\ b\ r_1) \wedge \neg(at\ b\ r_2) \wedge missed[(free\ left)] \wedge \\ & (free\ right) \wedge (\textbf{carry b right}) \wedge \\ & (\textbf{carry b left}) \end{aligned}$$

In our approach, we assume two types of input information. First, there is a partial planning domain model consisting of a set of predicates and a set of operators with arguments type but without the description of preconditions and effects. The second type of input is a set of sequences of actions, where each sequences consists of the initial state and a valid sequence of actions and observations. All sequences share the same initial state. We assume that the initial state is completely known, i.e, the initial state is noiseless and not partial. As for the ARMS algorithm, we assume that :

1. We can only remove information that are always present : $\forall p \in P, \neg p \in \epsilon_a^- \implies p \in \rho_a^+$
2. We can't add information that are always present : $\forall p \in P, p \in \rho_a^+ \implies p \notin \epsilon_a^+$

As we learn negative precondition too, we assume that we can't remove information that are never present : $\forall p \in P, \neg p \in \rho_a^- \implies \neg p \notin \epsilon_a^-$

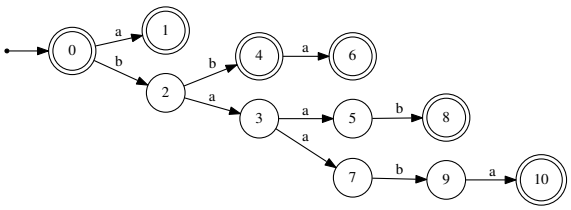
2 Informant learning

2.1 Regular grammar Induction

The aim of regular grammar induction is to learn regular grammar. A regular grammar is a grammar that can be represented as a deterministic finite automaton. A deterministic finite automaton is a popular mathematical abstraction used widely in computer science, mostly to design digital logic or computer programs. It is a behavioral model that consists of a finite number or states, transitions between those states, and actions, which execute the state transitions. Deterministic finite automata can also produce some output, using the defined output alphabet, but for the purpose of this paper we shall consider only the automata that produce binary output, true or false, based on whether the state of the finite-state automaton after processing the input is from the set of accepting states. We can define formally a deterministic finite automaton [7] as follow : A deterministic finite automaton is a quintuple $< \Sigma, Q, q_0, \delta, F >$, where :

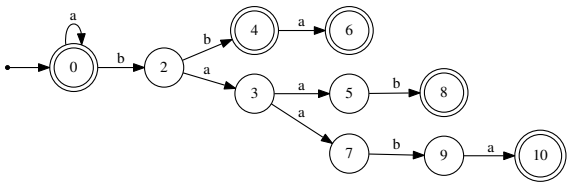
- Σ is the input alphabet (a finite nonempty set of symbols)
- Q is a finite nonempty set of states,
- q_0 is the initial state, an element of Q,
- δ is the state transition function; $\delta : Q \times \Sigma \to Q$,
- F is the set of final states, a subset of Q.

Among the regular grammar Induction algorithms, we are interested in algorithms allowing to learn a deterministic automaton using positive and negative sample because they are able to identify the class of the regular languages in the limit [5] in a polynomial time. And we are particularly interested in the RPNI [10] algorithm which has two advantages : (1) it has a polynomial complexity, (2) it is optimal. The algorithm 1 describes RPNI.
RPNI takes as input I_+ and I_-. I_+ (resp I_-) is the positive (resp negative) sample, it is a set of positive (resp negative) examples. RPNI returns the smallest automaton A accepting all positive examples and rejecting all negative examples.
RPNI starts by constructing the prefix acceptor tree $PTA(I_+)$ (line 2). $PTA(I_+)$ is a deterministic finite automaton that accepts only all positive examples. For instance, suppose that $I_+ = \{\lambda[1], a, bb, bba, baab, baaaba\}$ and $I- = \{b, ab, aba\}$, then $PTA(I_+)$:

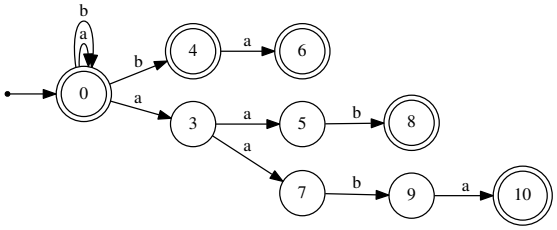


Then RPNI tries all states merges and retains only merges compatible with I_- (line 3-14). RPNI begins with the merge between states 0 and 1 :

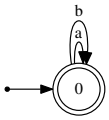
[1] λ is the empty word, some literature can use ϵ to designate the empty word



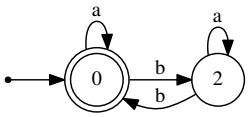
The new automaton is compatible with I_- (line 8-11), so RPNI retains this merge. Then it tries the merge between states 0 and 2 :



The new automaton is non-deterministic, so RPNI merges states responsible for the non-determinism (line



7).
This automaton is not compatible with I_- so the merge is not retained. Finally RPNI tries all other states merges and found the following automaton :



2.2 Action model learning and regular grammar induction

To learn our domain we learn an intermediate representation of our planning domain. This intermediate representation is a regular grammar. We use a grammar induction algorithm (see section 2.1) to learn this grammar.
As we have seen in section 1.2, an instantiated action model can be represented by a state-transition system. Indeed, an instantiated action model have a set of state Q and a transition function $\delta : Q \times \Sigma \rightarrow Q$, where Σ is a set of possibles actions. To represent a domain as a regular grammar we need an initial state, so the regular

Algorithm 1: RPNI

input : I_+, I_- : Negative and positive samples
output : A : automaton

1 $\pi \leftarrow \{\{0\},\{1\},\ldots,\{N-1\}\}$;
2 $A \leftarrow PTA(I_+)$;
3 **for** $i \leftarrow 1 : |\pi| - 1$ **do**
4 **for** $j \leftarrow 0 : i - 1$ **do**
5 $\pi' \leftarrow \pi/\{B_j, B_i\} \cup \{B_i \cup B_j\}$;
6 $A/\pi' \leftarrow derive(A/\pi')$;
7 $A/\pi'' \leftarrow deterministic\_merge(A/\pi'')$;
8 **if** $compatible(A/\pi'', I_-)$ **then**
9 $A \leftarrow A/\pi''$;
10 $\pi \leftarrow \pi''$;
11 *break*;
12 **end**
13 **end**
14 **end**
15 **return** A, π

grammar will be the language that accepts all action sequences starting by the given initial state.
For instance, figure 1 represents a regular grammar for the domain Gripper where the initial state is :

$$(at - robby\ r_1) \wedge \neg(at - robby\ r_2) \wedge (at\ b\ r_1) \wedge \neg(at\ b\ r_2) \wedge (free\ grip) \wedge \neg(carry\ b\ grip)$$

with $q_0 = 0$, $Q = \{0,1,2,3,4,5\}$, $F = \{0,1,2,3,4,5\}$ and $\Sigma = \{move(r_1\ r_2), move(r_2\ r_1), pick(b\ r_1\ grip), pick(b\ r_2\ grip), drop(b\ r_1\ grip), drop(b\ r_2\ grip)\}$.
There are many interests to such a method. Learning regular grammars is a well-defined area where there are several algorithms with good results. Moreover, this method makes it possible to learn any type of domains as long as it is possible for us to represent it in the form of a regular grammar. Indeed, this method is based on informant learning which is one of the approaches to language learning formalized by [5]. In informant learning, an informant introduces a learning system to both positive and negative examples of the language to be learned, and the learner system is also informed of the validity of each example. It allows to identify complex grammars in the limit. We are not in a linguistic setting, but this property is something we can reuse. Indeed, although a planning problem is not a language, it is possible for us, from an initial state, to represent our domain as a regular grammar, and this type of grammar is identifiable in the limit with informant learning, so it is possible for us to learn this automaton in the limit.

2.3 Samples generation

Figure 2 shows an overview of our method to generate positive and negative samples. From a given initial state q and an empty action sequence s, the autonomous agent will ask the oracle about the feasibility of an action a, chosen randomly, in the state q. In the case where

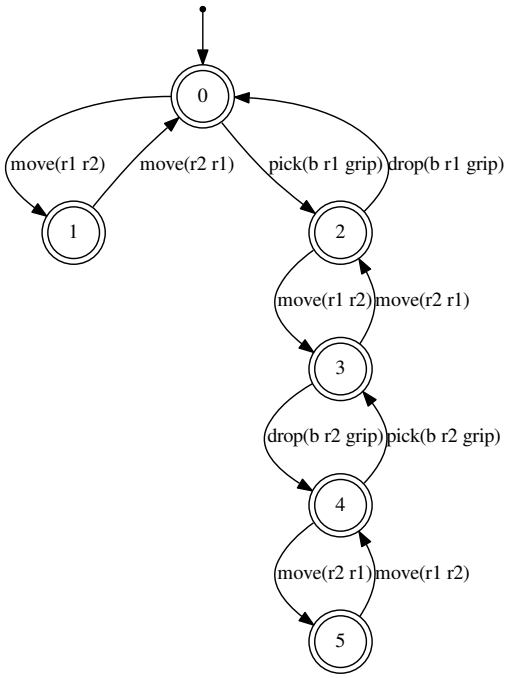


Figure 1: Regular grammar representing Gripper

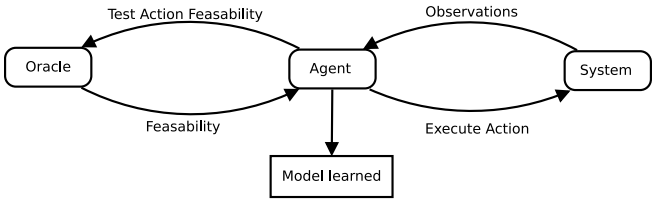


Figure 2: Samples generation

the action a is feasible, the autonomous agent adds the action a at the end of the sequence s. Then, the system that the autonomous agent tries to copy executes the action a, and the agent updates the state q by observing the system. In the case where the action a is not feasible, the agent adds the sequence $s' = s \cup \{a\}$ to the negative sample I_-. The agent restart this step until the size of the sequence s is N, where N is an integer randomly selected between 10 and 20. Once the sequence s built, the agent restart all the process since the beginning. We can assume that the oracle does not make error about the feasibility of actions. Nevertheless, we can not assume that observations are perfect. Indeed, since it is the agent that retrieves the information on the states directly using these observations through sensors, it is possible that there are errors. These errors can be due to a bad calibration of the sensors (noise) or to the fact that some properties are no longer observable (partial information). Nevertheless, we can assume that the static information are always completely observe without noise because these information are never modified.

3 The AMLSI algorithm

The algorithm is divided into three steps. The first step is the grammar induction step. The purpose of this first step is to learn an intermediate representation of a planning problem. We use a regular grammar as an intermediate representation and we use the RPNI-R (see section 3.1) algorithm to learn this grammar. Then the second step of our algorithm is the operators induction step. At this stage we infer the operators' preconditions and effects from the regular grammar learned previously. Finally, the third and final step of the AMLSI algorithm is a refinement step. This step is aimed at refining the operators' preconditions and effect using the regular grammar learned and a tabu search.
Before to learn the regular grammar we perform a pre-process step. We decompose each positive examples. Indeed, for each feasible sequences of actions, each sub-sequences is feasible, so the correct grammar have to accept all sub-sequences of each sequences. So we add each sub-sequences to the positive sample I_+. For instance, suppose we have the positive example $move(r_1\ r_2); move(r_2\ r_1); pick(b\ r_1\ left) \in I_+$, we have to add sub-sequences $move(r_1\ r_2); move(r_2\ r_1)$ and $move(r_1\ r_2)$ to the positive sample I_+.

3.1 Grammar Induction

We propose the algorithm RPNI-R to learn our grammar. RPNI-R is an alternative version of the algorithm RPNI taking into account pairwise constraints.
Pairwise constraints are a set of rules giving impossible transition sequences. If a pair (a_1, a_2) is present in our set of constraints, that indicates that there is no possible sequence where action a_i is followed by action a_j. These pairwise constraints make it possible to simulate negative examples and to ensure that in our automaton, all the pairs of actions present, are pairs of actions actually present in the domain. These constraints are based on the fact that for an action to be feasible a certain number of resources must be released (add list) and others must be captured (del list). For instance, for the domain gripper, the action $move(r_1\ r_2)$ will never be followed by the action $pick(b\ r_1\ grip)$, because we need that the predicate $(at - robby\ r_1)$ be evaluated to true to execute the action $pick(b\ r_1\ grip)$, and after the execution of the action $move(r_1\ r_2)$, the predicate $(at - robby\ r_1)$ is always evaluated to false, so the action $pick(b\ r_1\ grip)$ can't follow the action $move(r_1\ r_2)$. Formally, we can describe these pairwise constraints as follows :

$$(a_1, a_2) \in R \Rightarrow$$
$$a_1 \in \Sigma, a_2 \in \Sigma\ \nexists q_1 \in Q, q_2 \in Q, q_3 \in Q\ s.t.$$
$$\delta(q1, a_1) \rightarrow q_2 \wedge \delta(q2, a_2) \rightarrow q_3$$

where R is the set of pairewise constraints.
To compute these constraints we use our positive sample I_+. We consider that only the pairs present in our

positive examples are possible pairs. So all the pairs that are not present in I_+, belong to the set R :

$$\forall (a_i, a_j) \in \Sigma^2 : \begin{cases} (a_i, a_j) \notin R & if\ \exists x \in I_+\ s.t. \\ & x = u; a_i; a_j; v \\ (a_i, a_j) \in R & Otherwise \end{cases}$$

The RPNI-R algorithm ensure that the automaton learn is compatible with both I_- and R. Let's take the algorithm 1 describing RPNI. After the merge between state i and j, RPNI keeps π'' iff $compatible(A/\pi'', I_-)$ returns true, i.e, iff A/π'' rejects all the negative examples. RPNI-R retains π'' iff A/π'' rejects all the negative examples and rejects all pairs $(a_l, a_k) \in R$.

3.2 Operator induction

Mapping

Once the automaton has been learned, we need to know which states of the automaton correspond to which observation for each action. For that we play in the learned automaton all positive examples and record the pairs "state of the automaton, action" with the pairs "observation, action". There are two different mapping : the mapping ante μ_A and the mapping post μ_P. $\mu_A(q, a)$ (resp $\mu_P(q, a)$) gives the set of observations observed before (resp after) the execution of the action a in state q. Once mappings computed, we compute the reduced mapping. We denote the reduce mapping $\mu'_A = \bigcap \mu_A$ (resp $\mu'_P = \bigcap \mu_P$). The reduce mapping contains the common pattern of all observations. For instance, $\mu'_A(0, pick(b\ r_1\ grip))$ evaluates the predicate $(at\ b\ r_1)$ to *true* (resp *false*) iff $(at\ b\ r_1)$ is always evaluated to *true* (resp *false*) in $\mu_A(0, pick(b\ r_1\ grip))$.

Preconditions induction

The algorithm 2 describes our method to learn operators' preconditions. To learn preconditions of an operator o, we have to find the common pattern of all states where an action a instantiating o is feasible. Suppose we wanted to learn the precondition of $pick(?obj\ ?room\ ?gripper)$. Take $a = pick(b\ r_1\ grip)$, we start by searching all states where a is feasible (line 5), take $q = 0$. Let us assume that :

$$\mu'_A(q, pick(b\ r_1\ grip)) = \\ (at - robby\ r_1) \wedge \neg(at - robby\ r_2) \wedge (at\ b\ r_1) \wedge \\ \neg(at\ b\ r_2) \wedge (free\ grip) \wedge \neg(carry\ b\ grip)$$

At the line 6 we compute the observation q' that is the observation $\mu'_A(q, pick(b\ r_1\ grip))$ without the useless information, i.e, the observation where all predicates incompatible with $pick(b\ r_1\ grip)$ have been removed :

$$q' = reduce(\mu'_A(q, pick(b\ r_1\ grip)), \Phi_{pick(b\ r_1\ grip)})$$

where $\Phi_{pick(b\ r_1\ grip)}$ is the set of parameters of the action $pick(b\ r_1\ grip)$. In our example,

$$q' = \quad (at - robby\ r_1) \wedge (at\ b\ r_1) \wedge (free\ grip) \\ \wedge \neg(carry\ b\ grip)$$

Algorithm 2: Preconditions induction

input : A : an automaton, μ'_A : mapping o : an operator
output : ρ_o

```
1  Ξ ← {};
2  for a ∈ o do
3      τ ← {};
4      for q ∈ A.Q do
5          if a ∈ actions(q) then
6              q' ← reduce(μ_A[q, a], Φ_a);
7              if q' ≠ ∅ then
8                  τ ← τ ∪ {q'};
9              end
10         end
11     end
12     ρ_a ← ⋂(τ);
13     Ξ ← Ξ ∪ {generalize(ρ_a)};
14 end
15 ρ_o ← ⋂(Ξ);
16 return ρ_o
```

We can remove incompatible predicates thanks to the STRIPS scope assumption [2]. Then, we redo these different steps to find all observations q' for all states q where the action $pick(b\ r_1\ grip)$ is feasible (line 5-11). We denote τ the set of all observations q'. We can now compute $\rho_{pick(b\ r_1\ grip)} = \bigcap \tau$ the common pattern of all observations q' (line 12). Then, we generalize $\rho_{pick(b\ r_1\ grip)}$ by replacing all constants in the precondition by the arguments of the operator $pick(?obj\ ?room\ ?gripper)$ (line 13). Once generalized preconditions of all instances of the operator $pick(?obj\ ?room - ?gripper)$ computed, we recover the common pattern of all these generalized preconditions to compute $\rho_{pick(?obj\ ?room\ ?gripper)}$[2]

Effects induction

Effects induction is analogous to preconditions induction, with the difference that, instead of recovering the common pattern to all states where the operator is feasible, we recover the common pattern to all gap between states before/after executions of the operator.

3.3 Refinement

The method previously seen is sufficient if and only if three hypotheses are satisfied :

1. We learned the perfect automaton
2. We have complete observations
3. We have noiseless observations

In the general case, none of these assumptions are satisfied. That's why we add a refinement step. The figure 3 gives an overview of the refinement step.

[2] $\rho_{pick(?obj\ ?room\ ?gripper)}$ contains both positive preconditions ρ^+ and negative preconditions ρ^-

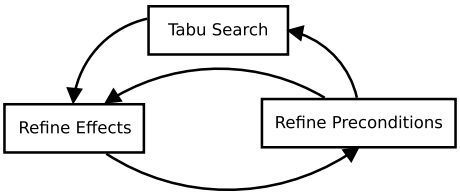


Figure 3: Refinement

Effect refinement

We begin by refining the effects. This step will ensure that the induced action model is able to regenerate the previously learned automaton. We use the mapping μ'_A to verify that for each transition : $\delta(q,a) \rightarrow q', \delta(q',a') \rightarrow q''$,, the effects of the action a and $\mu(q,a)'$ generate the preconditions of action a', if it is not the case, we add the effects ensuring that the next preconditions are satisfied. Formally, we add effects to verify that the state $q'' = \mu(q,a)' \cup \epsilon^+(a) \setminus \epsilon^-(a)$ satisfies $\rho_{a'}$. For instance, suppose we have the following transitions : $\delta(q, move(r_1\ r_2)) \leftarrow q'$ and $\delta(q', pick(b\ r_2\ grip)) \leftarrow q''$. Now suppose we have $\neg(at-robby\ r_2) \in \mu'_A(q, move(r_1\ r_2))$, $(at-robby\ r_2) \in \rho^+_{pick(b\ r_2\ grip}$ and $(at-robby\ r_2) \notin \epsilon^+_{move(r_1\ r_2)}$. We need to have $(at-robby\ r_2) \in \epsilon^+_{move(r_1\ r_2)}$ for that transitions $\delta(q, move(r_1\ r_2))$ and $\delta(q', pick(b\ r_2\ grip))$ are feasible, so we had $(at-robby\ ?to)$ to $\epsilon^+_{move(?from\ ?to)}$.

Precondition refinement

Then we can refine preconditions. This step ensure the assumption seen in section 1.2 : we can only remove information that are always present : $\forall p \in P, \neg p \in \epsilon^-_a \implies p \in \rho^+_a$. So for each negative effects of each operators, we add the corresponding positive precondition. For instance, suppose we have $\neg(at-robby\ ?from) \in \epsilon^-_{move(?from\ ?to)}$ so we must have $(at-robby\ ?from) \in \rho^+_{move(?from\ ?to)}$ after refinement.

Since effects refinement depends on the preconditions and preconditions refinement depends on the effects, we repeat these two stages until convergence.

Tabu search

Then we perform a tabu search to improve our model independently of our automaton. This search will allow us to find the information lost by the mapping because of the noise and the fact that the observations are partial. To reduce the search space, we keep only the candidates who respect the assumptions seen in section 1.2. Once this research is done, we repeat all the stages of refining until convergence.
The fitness function use to evaluate a candidate D is :

$$J(D|I_+, I_-) = \begin{array}{l} J_\rho(D|I_+) + J_\epsilon(D|I_+) + \\ J^+(D|I_-) + J^-(D|I_-) \end{array}$$

where

- $J_\rho(D|I_+) = \sum_{x_+ \in I_+} \sum_{(q,a) \in x_+} Accept(\rho_a, q) - Reject(\rho_a, q)$ compute the fitness score for preconditions. $Accept(\rho_a, q)$ count the number of positive and negative preconditions accepted by the observed state q. And $Reject(\rho_a, q)$ count the number of positive and negative preconditions rejected by the observed state q.
- $J_\epsilon(D|I_+) = \sum_{x_+ \in I_+} \sum_{(q,a,q') \in x_+} Equal(q'', q') - Different(q'', q')$ compute the fitness score for effects. Where $q'' = \{q \cup \epsilon^+_a\} \setminus \epsilon^-_a$. $Equal(q'', q')$ count the number of positive and negative predicates present in both state q'' and q'. And $Different(q'', q')$ count the number of positive (resp negative) predicates in q'' which are negative (resp positive) in the observed state q'.
- $J^+(D|I_+) = \sum_{x_+ \in I_+} \mathbb{1}_{Accept(D, x_+)}$ where $\mathbb{1}_{Accept(D,x_+)} = 1$ if and only if the candidate domain D can generate the positive example x_+.
- $J^-(D|I_-) = \sum_{x_- \in I_-} \mathbb{1}_{Accept(D, x_-[1:N-1]) \wedge Reject(D, x_-)}$ where N is the size of the negative example x_- and $\mathbb{1}_{Accept(D,x_-[1:N-1]) \wedge Reject(D,x_-)} = 1$ if and only if the candidate domain D can't generate the negative example x_- but can generate the sub-sequence $x_-[1:N-1]$. This fitness score takes into account how the negative sample is built (see section 2.3).

Fitness functions $J_\rho(D|I_+)$ and $J_\epsilon(D|I_+)$ measure the candidate's ability to explain the observation states. While fitness functions $J^+(D|I_-)$ and $J^-(D|I_-)$ measure the candidate's ability to regenerate the grammar. Finally, functions $J_\rho(D|I_+)$ and $J_\epsilon(D|I_+)$ are not enough to learn the domain because our observations are noisy. It is therefore possible that the optimal domain D^* is not the domain maximizing functions $J_\rho(D|I_+)$ and $J_\epsilon(D|I_+)$.

4 Evaluation

4.1 Experimental setup

AMLSI was tested using a collection domains from the International Planning Competition (IPC)[3]. The objective of these experiments is to demonstrate that AMLSI is able to learn action models with high levels of missing information in observations and some levels of noise. These domains are Blocksworld, Hanoi, N-Puzzle, Peg Solitaire and Gripper. The details of the domains used can be seen in table 1. We test our algorithm with a positive test sample E_+ and a negative test sample E_-. We use also a set of 20 problems. Incomplete observations were simulated by randomly selecting a fraction (0%, 25%, 50% or 100%) of fluent (including negations)

[3]Our implementation and the experimental setup can be found at https://gitlab.com/grandmaxIm2ag/amlsi

	Blocksworld	Gripper	Hanoi	N-Puzzle	Peg Solitaire
#*Operators*	4	3	4	1	3
#*Predicates*	5	4	7	3	5
$\lvert I_+ \rvert$	30	30	30	30	30
$\lvert I_- \rvert$	2399	1169	2989	3330	3474
$\lvert x_+ \rvert$	14.89	15.10	14.58	15.20	6.77
$\lvert x_- \rvert$	8.33	8.29	8.08	8.38	5.33
$\lvert E_+ \rvert$	100	100	100	100	100
$\lvert E_- \rvert$	80508	39575	110693	112279	22742
$\lvert e_+ \rvert$	50.79	51.03	51.03	51.03	6.50
$\lvert e_- \rvert$	33.39	33.71	33.71	33.72	4.6
Reference *Acc*	100.0	100.0	100.0	100.0	95.0

Table 1: Benchmark Domains Characteristics (from top to bottom): domain's number of operators, domain's number of predicates, average number of examples in the positive sample, average number of examples in the negative sample, average size of positive examples, average size of negative examples, average number of examples in the positive test sample, average number of examples in the negative test sample, average size of positive test examples, average size of negative test examples, accuracy of reference domain.

from the system to observe after each action. Sensor noise was simulated similarly by flipping the value of observed predicate in the state with probability 0%, 1%, 2%, 5%, 10% and 20%. We use an alternative version of the method use to generate training samples (see section 2.3) to generate positive and negative test samples. In this version, we test each action at each step to keep all negative information. The system to be copied and the oracle are simulated by the java library pddl4j [11] by encapsulating in a black-box the reference IPC domains. To determine an error bound on our results, we test each domain with three different initial states manually built over five runs. To allow the reproducibility of the experiment, we use five seeds randomly draw for each run. Finally, the tabu search for the refinement is computed over 200 iterations.

Metrics

We test our algorithm over three metrics. The first metric is the error rate for preconditions : $E_\rho = \sum_{e \in E^+} \frac{\sum_{(q,a) \in e} error(\rho_a, q)}{\sum_{a \in e} |\rho_a|}$ where $error(\rho_a, q) = |\{p \in \rho_a^+ \wedge \neg p \in q\}| + |\{\neg p \in \rho_a^- \wedge p \in q\}|$. This metrics, proposed by [16], compute the rate of preconditions that were not satisfied in the test sample E_+. Then we use the F-Score [12]. F-Score $= \frac{2.P.R}{P+R}$ where $R = \frac{|\{s \in E^+ \mid accept(D,e)\}|}{N^+}$, and $P = \frac{|\{e \in E^+ \mid accept(D,e)\}|}{|\{e \in E^+ \mid accept(A,e)\} \cup \{e \in E^- \mid accept(D,e)\}|}$. F-Score is initially a metric used for pattern recognition and binary classification. Nevertheless, it can be used to evaluate the quality of a learned grammar. Indeed, a grammar is equivalent to a binary classification system labeled with $\{1, 0\}$. For grammars we can assume that the sequences belonging to the grammar are data labeled with 1, and non-grammar sequences are data labeled with 0. The score will therefore be able to test the ability of the domain to regenerate the grammar of the reference domain. Finally we use the accuracy $Acc = \frac{N}{N^*}$. This metric, proposed by [17], compute the ability of the learned domain to resolve different planning problem, where N is the number of solved problem, and N^* the total number of problem.

Baseline

Comparisons with other approaches in the literature are difficult due to differences in the learning settings. First of all we use a type of input that is not used much in the literature. Indeed, we use both feasible action sequences, and unfeasible action sequences, while most algorithms use only feasible action sequences. In addition, we use a random walk with an oracle to generate our positive examples (see section 2.3), whereas most of the algorithms of the literature like ARMS, SLAF, Plan-Milner etc use goal-directed plans or only one positive sequence like LSO-NIO, IRALe or LOCM [1]. In addition, we are able to learn our domain with partial and noisy observations. Several algorithms like LOUGA and ARMS learn domains with partial observations but they are not robust to noise. For ARMS, this is due to the fact that the learning is done using logic induction, and for LOUGA, this is due to the fact that LOUGA uses the intermediate information to reduce the search space, and the noise could remove from the search space the optimal solution. The IRALe algorithm is able to learn a domain with noisy observations, but it needs that the observations are not partial. LSO-NIO and Plan-Milner can deals with both noisy and partial observations, but the type of learned domain differs. AMLSI learn negative preconditions and static relations. But Plan-Milner and LSO-NIO learn only positive precondition and can't learn static relation. That's why we do not use baseline in this evaluation.

4.2 Results

We begin by focusing on three particular cases :

1. High level of intermediate information and low level of noise (table 2).
2. Low level of intermediate information and high level of noise (table 3).
3. No intermediate information (table 4).

First of all, the table 2 shows the results of our experiment with a high level of intermediate information (100% of observable fluent) and a low level of noise (1%). We note that for all planning domains, each metrics reaches its optimal value (0% for Error Rate 100% for F-Score and 100% or 95% for Accuracy). Then, the table 3 shows the results of our experiment with a low level of intermediate information (25% of observable fluent) and a high level of noise (20%). We note that for all planning domains, except for Blocksworld and Peg Solitaire, E_ρ and Accuracy reach their optimal values. Nevertheless, we also note that for some domain the F-Score have bad value. For instance, for the domain Gripper, we have F-Score = 80.44%. This is due

	Blocksworld	Gripper	Hanoi	N-Puzzle	Peg Solitaire
E_ρ	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
F-Score	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
Acc	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	95.0 ± 0.0

Table 2: Action model learning results on four domains with a high level of intermediate information (100% of observable fluent) and a low level of noise (1%). Performance is measured in terms of error rates E_ρ (lower is better) and F-Score and accuracy Acc (higher is better). We reported the average and standard deviation computed over five runs and three different initial states.

	Blocksworld	Gripper	Hanoi	N-Puzzle	Peg Solitaire
E_ρ	0.61 ± 2.29	0.0 ± 0	0.0 ± 0.0	0.0 ± 0.0	0.83 ± 1.8
F-Score	93.21 ± 25.38	80.44 ± 40.48	100.0 ± 0.0	100.0 ± 0.0	89.64 ± 24.92
Acc	93.50 ± 24.28	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	70.66 ± 42.65

Table 3: Action model learning results on four domains with a low level of intermediate information (25% of observable fluent) and a high level of noise (20%). Performance is measured in terms of error rates E_ρ (lower is better) and F-Score and accuracy Acc (higher is better). We reported the average and standard deviation computed over five runs and three different initial states.

to errors for learning of the operator move. The learned move operator is :

```
(:action move
        :parameters (?x1-room ?x2-room)
        :precondition (and)
        :effect (and
        (at-robby ?x1)
        (at-robby ?x2))
)
```

while the reference operator is :

```
(:action move
        :parameters (?x1-room ?x2-room)
        :precondition (and
        (at-robby ?x1)
        (not(at-robby ?x2)))
        :effect (and
        (not(at-robby ?x1))
        (at-robby ?x2))
)
```

We notice that there are errors for both preconditions and effects. We can note that although the are errors in preconditions, $E_\rho = 0\%$. This is because preconditions are missing and no preconditions have been added. The fact that preconditions are missing implies that unfeasible action sequences can be generated by the learned domain, which explains that F-Score $< 100\%$. In addition, a positive effect $(at - robby\ r_1)$ has been added and a negative effect $(not(at - robby\ r_1))$ is missing. This also has the consequences of allowing the learned domain to generate unfeasible sequences of actions. Nevertheless these errors do not prevent the resolution of the problems. Indeed, we have a high level of resolution $Acc > 70\%$ for all domains. This is because all feasible action sequences can be generated from the learned domain. In addition, the goals of planning problems take into account only the positive predicate, so the fact that a negative effect is missing does not prevent finding a state that satisfies the goal. Finally, the table 4 shows the

	Blocksworld	Gripper	Hanoi	N-Puzzle	Peg Solitaire
$E_\rho(\%)$	10.05 ± 10.07	0.0 ± 0.0	0.41 ± 1.61	0.0 ± 0.0	0.3 ± 0.43
F-Score (%)	35.55 ± 48.49	100.0 ± 0.0	5.39 ± 3.95	100.0 ± 0.0	81.06 ± 26.84
Acc (%)	47.45 ± 47.39	80.00 ± 29.27	68.33 ± 36.33	100.0 ± 0.0	64.66 ± 45.88

Table 4: Action model learning results on four domains with no intermediate information. Performance is measured in terms of error rates E_ρ (lower is better) and F-Score and accuracy Acc (higher is better). We reported the average and standard deviation computed over five runs and three different initial states.

results of learning when no intermediate information is observable. We notice that only the N-Puzzle domain has been learned. This comes from the fact that this domain has only one operator, so it is easier to learn this domain than others. We also note that for some domain $E_\rho > 0\%$. For instance, in the learned Hanoi domain, the positive precondition $(holding\ ?x_1)$ has been added for the $pick\_up$ operator. Finally we notice nevertheless that for some domain we keep a high resolution rate of problems, $Acc = 80\%$ for the domain Gripper for instance.

	Initial State 1	Initial State 2	Initial State 3
$E_\rho(\%)$	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
F-Score (%)	80.47 ± 43.2	100.0 ± 0.0	60.86 ± 53.59
Acc (%)	100 ± 0.0	100.00 ± 0.0	100.00 ± 0.0

Table 5: Comparison between initial states for domain Gripper. Performance is measured in terms of error rates E_ρ (lower is better) and F-Score and accuracy Acc (higher is better). We reported the average and standard deviation computed over five runs.

Then, the table 5 gives a comparison between each initial states for the domain Gripper. We can observe that there exists a high variability between initial states. For example, with the initial state 2 we have F-Score $=$ 100.0% while with the initial state 3 we have F-Score $=$ 60.83%. This gap can be explained by the quality of the learned automaton. Indeed, if initial states are different then grammars are different. It is therefore possible that the grammar generated from the initial state 3 is more complex (greater number of states) or that the grammar gives less information on the actions than the grammar generated from the initial state 2.
Then, we can observe in all experiments that the standard deviation can be high. For instance, in the table 3 for the domain Gripper, the standard deviation is 40.48 for the F-Score metric. This high standard deviation is due to the fact that we have "extreme" values for the F-Score. Indeed, we have F-Score $=$ 100.0% 12 times while we have F-Score $<$ 2.4% three times. This means

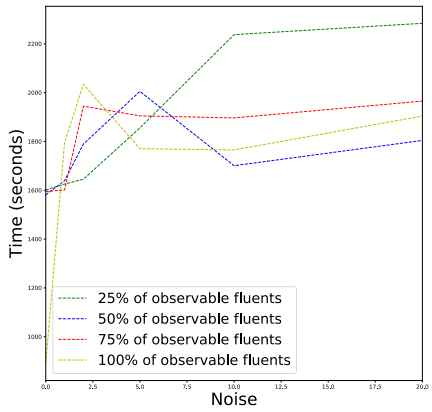


Figure 4: Time to learn Hanoi

that in the majority of cases we learn the optimal planning domain, but in a few cases we learn very different domains from the optimal domain.
Finally, we study the time to needed to learn planning domain. First of all, we notice that the learning of a planning domain is much faster when there is no noise, whatever the level of observability. Also, we notice that when the noise varies between 1% and 20%, the learning time is stable. Moreover, there is not a big difference between the different levels of observability, but AMLSI is often the faster when 75% or 100% of fluent are observable. Finally, we notice that the learning time is globally high. For example, for the Hanoi domain (see figure 4), when the noise level exceeds 0%, it always takes at least 1600 seconds to learn the domain. This high time is due to the refinement of the operators, and more particularly to the tabu research.

5 Conclusion and Future work

The results demonstrate that our approach successfully learns operators from noisy and incomplete observations when we have a high level of observable fluent whatever the noise level. Then our approach can't learn the optimal planning domain when there are no intermediate information in the majority of case. However, we are always able to resolve problem.
The quality of our learning will depend on the learned regular grammar. The learned grammar must be sufficiently expressive to be able to induce the operators. The expressiveness of our grammar depends on our initial state, so to learn the optimal domain, we need an initial state allowing a strong expressivity of the grammar. Nevertheless, the more expressive the grammar, the more difficult it is to learn. So to maximize the quality of learning the automaton we must use the least expressive initial state possible, and to maximize the quality of learning the domain of planning we must use an initial state as expressive as possible. For our experiment, the initial states have been built in such a way that the grammars are sufficiently expressive for learning the planning domain, without it being too expressive to allow an efficient learning of the automaton. However this is not realistic for a real-world application.
Then, our learning algorithm is passive, so we do not take advantage of the fact that we ask an oracle. With an active learner we will be able to benefit from the fact that we ask an oracle, to bias the drawing of the different actions to have examples allowing to learn the best automaton while minimizing the number of times when we ask the oracle.
Moreover, as we have seen in the section 4.2, the tabu search takes a lot of time to refine the operators, it might be interesting to use another method, less time consuming, to refine the operator without having a great loss of efficiency.
Finally, we learn only the STRIPS part of the PDDL language, it would be interesting to extend the PDDL part that our algorithm is able to learn, as disjunctive precondition, conditional effect, cost function etc.

Acknowledgements

This work was funded by the Circular project - IDEX.

Real Time Simulations with a Biomechanical Model of the Human Tongue using a Machine-Learning-based Model Order Reduction approach

Maxime Calka[a,b], Pascal Perrier[b], Jacques Ohayon[a], Yohan Payan[a]

[a]Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMC-IMAG, F-38000 Grenoble, France;
[b]Univ. Grenoble Alpes, CNRS, Grenoble INP, GIPSA-lab, F-38000 Grenoble, France

ARTICLE HISTORY
Compiled August 28, 2019

ABSTRACT
This paper focuses on real time simulations with a biomechanical model of the human tongue in order to create a Digital Twin model that can be integrated in the medical framework. As a long term application, this Digital Twin of the tongue could be used to predict functional consequences of tongue surgery.

The method proposed uses an "a posteriori" Model Order Reduction method that learns from a limited number of simulations (excitation/output) of the model to estimate new simulations of the human tongue.

In the preliminary results presented in this paper, the Reduced Order Model is shown to be able to estimate with a sub-millemetric accuracy the non-linear behavior of this organ in response to muscle activations.

KEYWORDS
Model Order Reduction, Digital Twin, Real Time Simulation

1. Introduction

1.1. Medical context

Nowadays, tongue cancer are still present with 4200 new cases in France in 2017 (Jéhannin-Ligier et al. (2017)). This type of cancer particularly affects people exposed to tobacco and alcohol consumption (Sturgis and Cinciripini (2007)).

The most common techniques to treat the patient is the exeresis of a part of the tongue. However such a surgery can have severe consequences on tongue mobility and tongue deformation capabilities. It can generate impairments for masticating, swallowing and speaking that are three basic biological functions in humans life. Impairments can reduce drastically the quality of life of the patients. For now, quantitatively predicting the functional consequences of this surgery is very complex for the clinician.

This project has for final objective to develop a planning system "in silico" that should predict the functional consequences in a quantitative way for real surgery. The software should be able:

- to generate in an automatic way patient-specific 3D finite element (FE) models of the tongue.
- to simulate in interactive time the anatomic changes induced by the surgery (e.g.

tumor resection with flap reconstruction). For this, we propose to use a Model Order Reduction (MOR) technique.
- to quantitatively predict the consequences of these anatomical changes on mastication, swallowing and speech.

Solving the first two phases of the planning system (namely the creation of a digital biophysical replica usable for the FE model analysis and the interactive time simulations of such a model) without simplify the physics is a challenge in some computational medicine applications.

A quasi-automatic generation of the patient-specific mesh for human tongue has already been proposed and evaluated in our group (Bijar et al. (2016)). However, because of the non-linear mechanical behavior of human tongue tissues, the simulations using any patient-specific FE model in the tongue can take very long time (about one hour to simulate a movement of some milliseconds) and is not usable in a clinical context. Indeed, the tongue is a complex organ with incompressible tissues and a non-linear visco-elastic behavior (large deformations and non-linear elasticity). In addition, to study the functional outcome of the surgery in terms of speech production, a key point of tongue simulation is to be able to model the tongue's trajectory over time and not just its final shape after muscles contractions. So, a transient analysis (solving the Lagrange equations) is required to take into account this temporal notion in the FE analysis. A difficulty during this FE analysis is that tongue gets a very fast movement during speech production(tens of milliseconds of transition) increasing the viscous effect of the modeled organ. The challenge of any MOR technique is therefore to account for this temporal evolution and to capture the non-linear and fast behavior of the tongue.

1.2. Related Work

From many years model order reduction (MOR) methods have received a growing interest to challenge the real time simulation problem in computational surgery (Cueto and Chinesta (2014)). These methods allow to obtain real-time *online* simulations by reducing the computational complexity in numerical simulations without simplifing the physics of the model. Such methods require a computationally intensive *offline* phase.

For now, the projection-based and collocation-based MOR methods are the most popular ones. The projection-based MOR are usually divided in two categories: the *a posteriori* methods such as the Proper Orthogonal Decompositions (Chatterjee (2000)) that create the Reduced Order Model (ROM) from a large set of simulations called "Snapshots" and the *a priori* methods such as the Proper Generalized Decomposition (PGD) (Chinesta et al. (2013)) that reduce the model during the problem solving process. Niroomandi et al. (2012) applied a POD MOR method for computational medicine in the case of non-linear quasi-static problem to simulate the palpation of the human cornea and solved tissues large deformations problems.

In Lauzeral et al. (2019) the ROM is created with a collocation-based MOR called Space Subspace Learning (SSL) (Borzacchiello et al. (2019)). In the same idea as our project, the problematic of this paper was to create a Digital Twin of the liver and to consider large deformation but only linear elasticity of the model. As for Niroomandi et al. (2012) the simulations were solved in a quasi-static way.

Contrary to the related work, our project has to model a non-linear mechanical behavior, namely the one of the tongue, and to solve the Lagrangian equations (tran-

sient analysis). In addition, our ROM is created thanks to an *a posteriori* machine learning-based technique (ML-based MOR). This method is evaluated to estimate the specific case of the human tongue to solve real-time simulation problem. Below, the MOR method and the biomechanical tongue model are detailed. Then, the experimental data are introduced to describe the learning and evaluation condition of the MOR method. In a last part, the result of the MOR method to estimate the displacement of the whole tongue are shown.

2. Materials & Methods

2.1. Model order reduction based on machine learning

In this paper, an ML-based MOR technique developed by ANSYS® called "DynamicROM" is used to estimate the non-linear behavior of the tongue in transient analysis. It is a "non-intrusive" and "a posteriori" ML-based method with a large dataset of snapshots generated off-line by the MAPDL® full-order solver. The whole process is describe in Fig. 1 and is made of two steps: the reduction of the number of output in a smaller number of "modes" using a Singular Value Decomposition (SVD) and the learning on the dataset to generate the ROM.

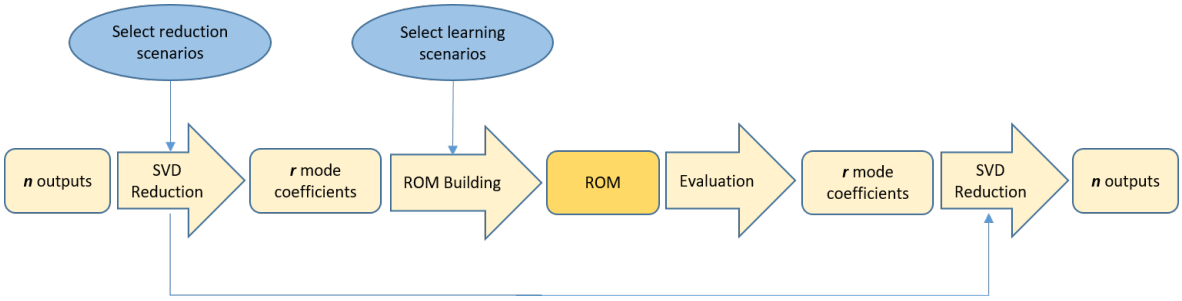


Figure 1. DynamicROM process

2.1.1. Learning phase

The DynamicROM algorithm predicts the output temporal matrix $X_{n\times n_t}$ of non-linear dynamic system (our tongue model) for a certain temporal excitation $B_{m\times n_t}$. To realize this prediction, the algorithm learns from a set of data called "scenario" S generated with the full-order solver that encapsulates an excitation and the corresponding output $S = (B, X)$. The goal of the learning phase is to solve the differential equation (eq.1) by founding the optimal non-linear function $\hat{f}$.

$$X'(t) = f(X(t), B(t)) \tag{1}$$
$$X(0) = X_0 \tag{2}$$

with,

- $X(t)$: vector of n output at time t
- $B(t)$: vector of m excitation at time t
- f: non-linear function correlating the output and the excitation

- $X'(t)$: velocity of each output at time t
- X_0: initial condition of the system

2.1.2. Reduction of spatial dimensionality

In some cases the number of outputs can be huge and considerably increase the computation time of the learning phase. A spatial dimension reduction allows to reduce this computation time by decreasing the n output points to learn in a less number of mode coefficient r. In the DynamicROM method, the dimension reduction method used is a Singular Value Decomposition (SVD) that factorizes the matrix X such that:

$$X_{n \times n_t} = U_{n \times n} \cdot \Sigma_{n \times n_t} \cdot V_{n_t \times n_t} \tag{3}$$

where, U and V are matrices respectively corresponding to left and right singular value of X.

This decomposition allows to do an approximation of X using only the r first left singular value.

$$X \hat{=} X_{r \times n_t} = U_{n \times r} \cdot \Sigma_{r \times r} \cdot V_{r \times n_t} \tag{4}$$

where, X_r is the optimal approximation of X with r modes.

2.1.3. 3D biomechanical model of the human tongue

The tongue model used for the simulations is described in Hermant et al. (2017). It is based on a Finite Element mesh including 7763 nodes and 8780 hexahedral elements. The constitutive equation used to model the mechanical properties of the tongue soft tissue is based on a Mooney-Rivelin formulation with two parameters $C_{10} = 192Pa$ and $C_{20} = 90Pa$. In addition, the viscoelasticity of the tongue is approximated with a Rayleigh model (Rayleigh coefficient $\alpha = 20$ and $\beta = 0.0$). The tongue being assumed quasi-incompressible, a Poisson ratio is fixed to $v = 0.4999$ and "no-displacement boundaries conditions" are defined on the external nodes in contact with the jaw and the floor cavity. Fig. 2 shows the mesh of the tongue model with the two muscles that will be activated in the simulation, namely the styloglossus and the genioglossus (pathways of the muscle fibers represented in blue).

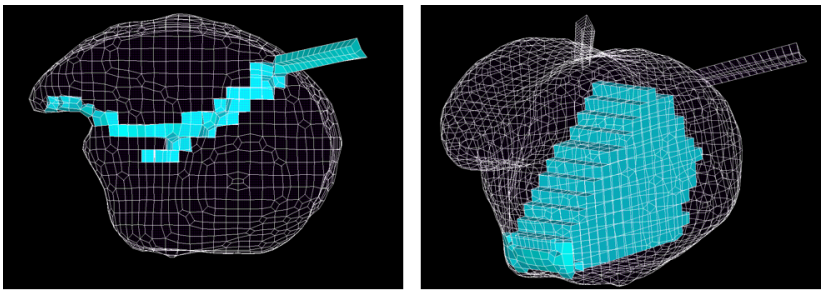


Figure 2. Tongue model used for simulations with muscles represented in blue.

Fig. 3 illustrates the complexity of estimating the non-linear behavior and kinematic of the tongue, with the plot of the displacement in the 3 axis of a point at the tip of the tongue during the genioglossus maximum activation. Except for the Y axis that shows a quite straight displacement, the two other axis show a complex non-linear kinematic.

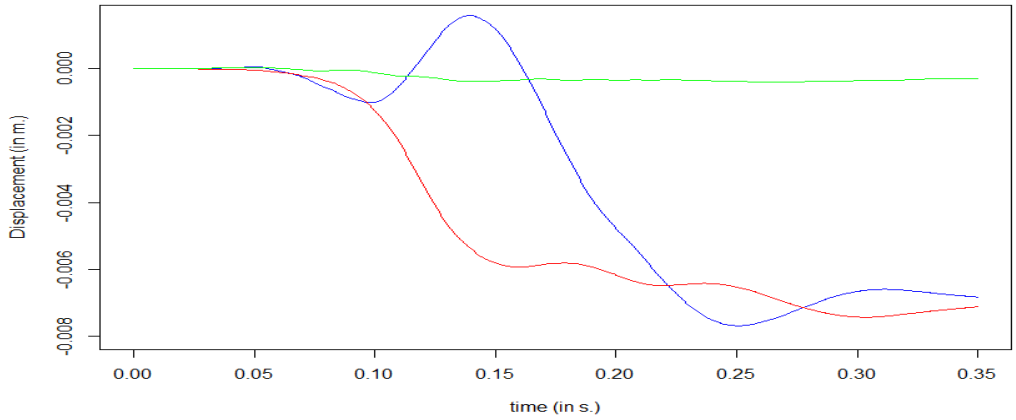


Figure 3. Tongue displacement after activation of the genioglossus. X: red, Y: green, Z: blue.

2.2. Experimental data

As said above, a scenario is a set of two types of data called "excitation" and "output". In the case of the tongue, an excitation $B(t)$ and an output $X(t)$ represent respectively a temporal muscular activation of the system and a temporal displacement of the tongue surface.

2.2.1. Excitation

In this paper, two types of excitation of the model are studied: the separate activation of the posterior genioglossus (GG-P) responsible for protrusion and elevation of the tongue, and the separate activation of the styloglossus (SG) allowing the tongue to be raised and retracted. For all excitation cases studied here the muscles activations consist of a linearly increasing phase and a stabilization phase (Fig. 4).

In the muscle model, the activation is defined by a stress σ which has a maximal value σ_{max}^{SG} normalized by an activation ratio of α varying between $[0; 1]$ such that $\sigma = \sigma_{max} \times \alpha\, Pa$.

In our study, for each muscle the activation ratio α varies in the interval $[0.2, 1.0]$. In addition, all the simulations have a total time $t_{total} = 0.35\, s.$ with a first linearly interpolated phase time of $t_{activation} \in [0.05, 0.11]$. These durations have been defined because they correspond to average times observed to generate a tongue movement between two speech sounds (a few tens of milliseconds of transition and a few hundred milliseconds of average movement duration). α and $t_{activation}$ are the two parameters that varies in the excitation scenarios.

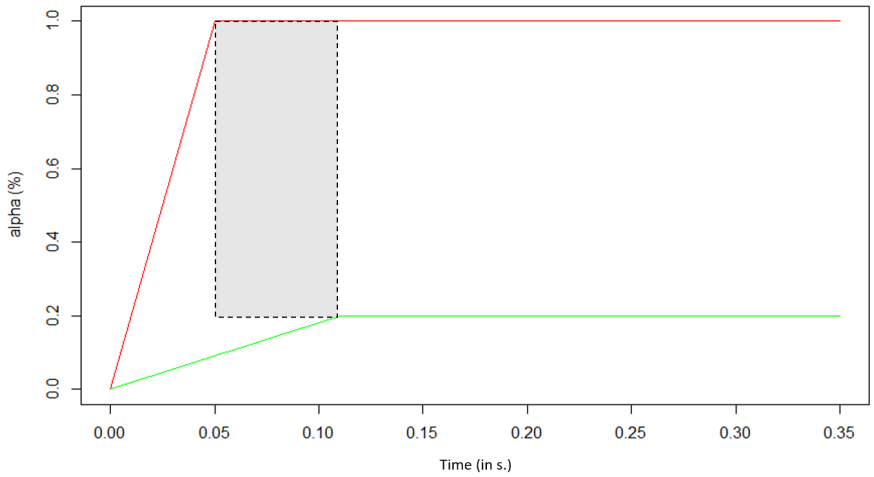


Figure 4. Range of variation of possible excitations in our scenarios. In red, the pattern corresponding to the maximum stress with a minimum activation time, In green the pattern corresponding to the minimum stress with a maximum activation time. he rectangle in grey corresponds to all possible range of values for parameters for the simulations.

2.2.2. Output

Since we are interested in the functional behaviour of the tongue after surgery that interests us, it was decided to observe the kinematics of the tongue surface at the exit; 3D movement of that whole tongue surface is therefore used to evaluate the ROM.

2.3. Learning scenarios

20 simulations were conducted to define the learning scenarios for the genioglossus and styloglossus muscles. These simulations were performed with excitation data whose parameters α and $t_{activation}$ have steps of 0.1 and $0.01s$. respectively forming a grid represented Fig. 5.

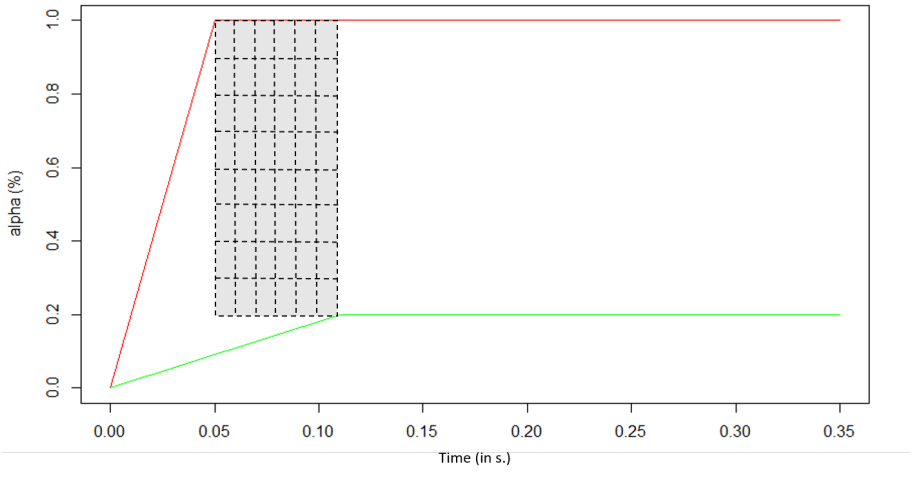


Figure 5. Set of learning scenarios for each activation cases (SG, GG and SG+GG)

2.4. Evaluation scenarios

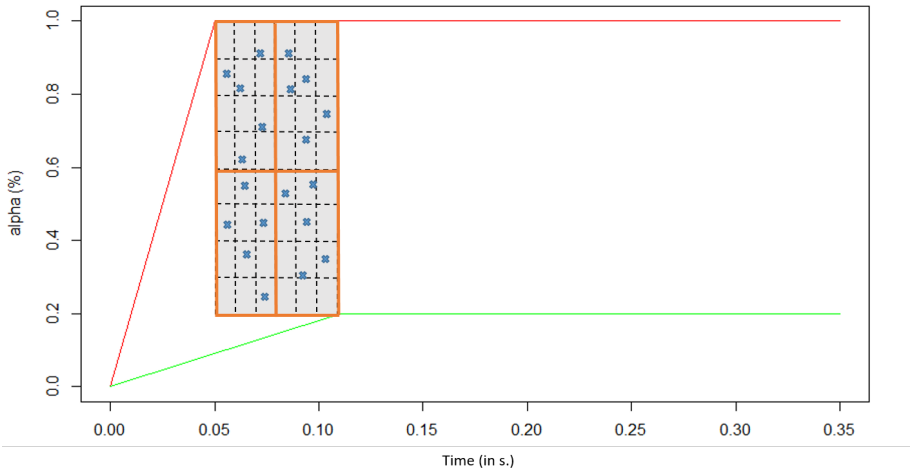


Figure 6. Example of a set of possible excitation for evaluation scenarios

The evaluation scenarios are a set of simulations of the size of the learning data, i.e. 20 evaluations simulations. These scenarios are used to evaluate the ROM in the evaluation phase represented Fig. 1.

Then excitation data are randomly distributed between four areas cutting the grid of Fig 5 to cover a wide spectrum of possibilities while not taking the results of a simulation used for learning. This distribution is represented in Fig.. 6.

2.5. Metric

To quantitatively study the accuracy of the ROM, the metric used is the root mean square error (RMS error) between the coordinates of any output point (X^x, X^y and X^z), and the coordinates of this point estimated by the ROM (X^x_{ROM}, X^y_{ROM} and X^z_{ROM}). Equation 5 allows to evaluate this RMS error of the displacement field on all scenarios and will be used to obtain the optimal learning and accuracy of the ROM (N is the number of scenarios). For our reduced tongue model to be considered of satisfactory quality it must have an RMS error of less than a few tenths of a millimeter.

$$RMSError = \frac{\sqrt{(\sum^N (X - X_{ROM})^2)}}{N} \tag{5}$$

3. Result & Discussion

3.1. Results

Figure 7 shows typical examples of the horizontal displacement of a node located on the blade of the tongue resulting from the separate activations of the genioglossus and the styloglossus as generated with the original biomechanical model, and approximated with the Reduced Order Model. Clearly the Reduced Order Model is able to provide a very good approximation of the displacement computed with the biomechanical model.

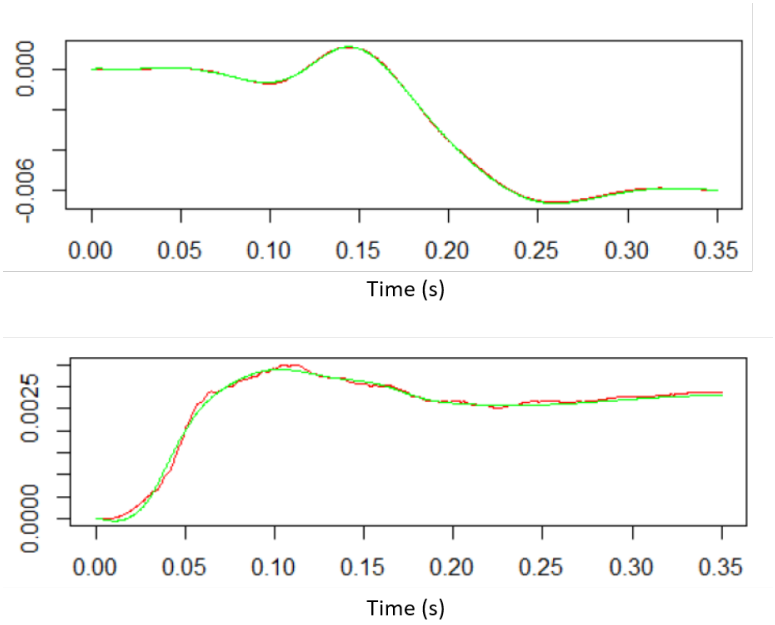


Figure 7. Horizontal displacement of a node on the blade of the tongue in response to the separate activations of the genioglossus (Top panel) and the styloglossus (Bottom panel). Red curves: displacement resulting from the simulations with the biomechanical model. Green curves: displacement computed with the Reduced Order Model

The mean RMS error of the displacement field computed on all the nodes of interest and for all the scenarios are given in Fig. 8. With respective average errors of $3.7x10^{-6}$m and $1.9.x10^{-6}$m for the styloglossus and the genioglossus (standard deviation $8.8x10^{-7}$m and $4.8x10^{-7}$m) the quality of the approximation with the reduced mode is clearly excellent.

Figure 9 illustrates the spatial distribution of the RMS error over the different nodes of the tongue for the simulations associated with the Styloglossus activation (results are similar for the genioglossus). It can be observed again that the RMS error is quite evenly distributed and in general very low, except in a specific region, namely the one where the posterior fibers of the styloglossus, arising from the Stylo-hyoid process enter the body of the tongue (in the posterior velar region). This result deserves further investigation, since it can reveal a strong non-linearity in the mechanical behavior of the element connecting the external part of the stylo-glossus with the tongue body. However at this stage of the study, it is not a crucial issue since the error remains very localized, and has a negligible impact on the global behavior of the tongue.

3.2. Discussion

The results obtained with the Reduced Order Model for the separate activations of the genioglossus and the styloglossus are very encouraging. The Reduced Order Model generates in real-time time-varying tongue deformations that are very close to those generated with the original biomechanical model, with a mean RMS error that is generally sub-millimetric. Slight differences are observed in the approximation quality between the results obtained for the genioglossus and those obtained with the styloglossus. Figure 7 provides a possible explanation for that: the trajectory generated with the biomechanical model is more noisy for the activation of the styloglossus,

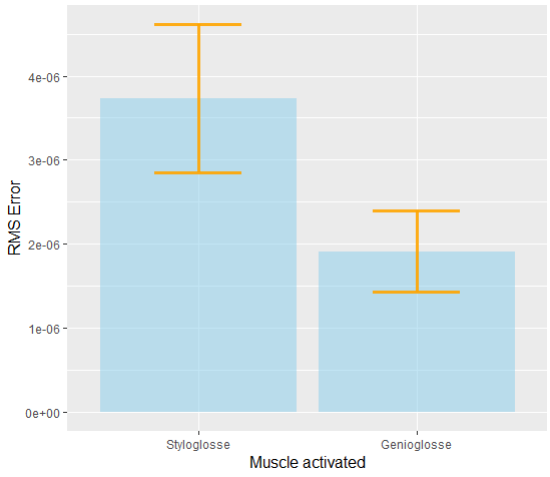


Figure 8. mean RMS error of the displacement field commputed on all the nodes of interest and for all the scenarios

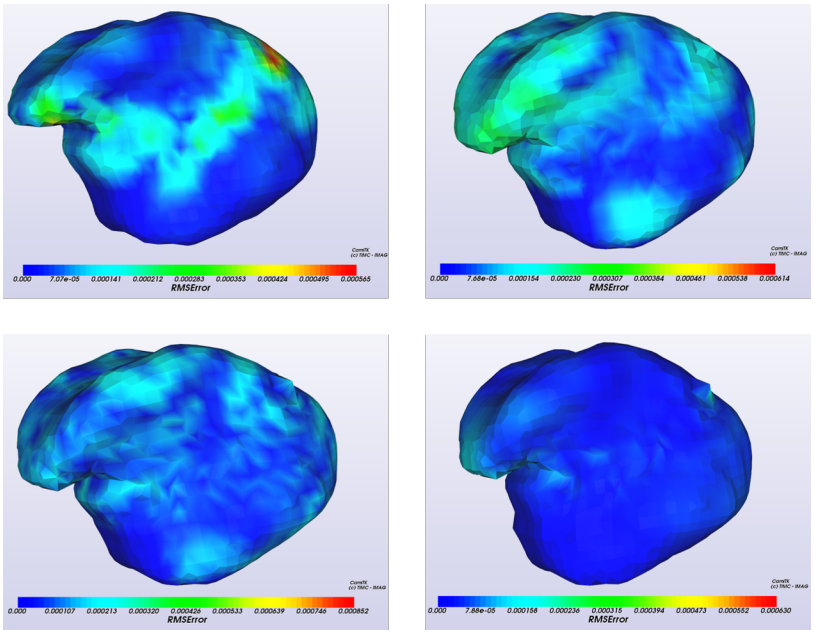


Figure 9. RMS errors for four different tongue positions resulting from 4 activations of the styloglossus. The heat map values are in meters.

due to inaccuracy in solving Lagrangian equations. Hence, this difference originates in limitations of the Finite Element computation, and not in the design of the Reduced Order Model. This point should be further investigated, at the level of the algorithms solving the Lagrangian equations in the FEM model.

Figure 7 shows that in the simulations that we have considered for this work, the node trajectories are not very complex, and are quite similar to the indicial response of a second order system with a subcritical damping factor. This could explain the high quality of the approximation obtained in our results so far. Hence further work will involve more complex scenarios in which potential non-linearities of the original biomechanical model will have stronger consequences on the time variations of the tongue shape. First of all we will consider cases in which several muscles are activated at the same time, with different timings. Such activations patterns will generate node trajectories that are more complex and differ more strongly from indicial responses of second-order models. We will also consider scenarios with strong mechanical non-linearities due to contacts between the tongue and the vocal tract boundaries (palate, teeth, pharyngeal walls). These simulations will provide a more challenging context to assess the approach used to design the Reduced Order Model.

Importantly, independently of the ability of the ROM to describe the time course of the nodes toward the final configuration, Figure 9 shows that the final configurations are predicted very accurately. This is crucial result in the context of a clinical application that would aim at assessing the extent to which the tongue can cover the whole range of speech articulation in a post-surgical state.

4. Conclusion & Future Work

In this paper, a ML-based MOR method has been used in the specific-case of the tongue that own non-linear behavior and fast movement requiring transient analysis.

The ROMs have been constructed on two kinds of activation (SG and GG) in learning a large set of off-line simulations. Then the ML algorithm found the best non-linear functions to describe this simulations.

The results show that this method is able to estimated in real-time with a sub-millimetric precision the displacement of the tongue when one muscle is activated.

Some aspects remain, however, open for future work. For instance, the possibility to estimate the tongue movement for more than one muscle or in the longer term to estimate the movement of the tongue after an exeresis and a reconstruction. However, this second step need more knowledge on the behavior of resected muscles.

5. References

References

Bijar A, Rohan PY, Perrier P, Payan Y. 2016. Atlas-based automatic generation of subject-specific finite element tongue meshes. Annals of biomedical engineering. 44(1):16–34.

Borzacchiello D, Aguado JV, Chinesta F. 2019. Non-intrusive Sparse Subspace Learning for Parametrized Problems. Archives of Computational Methods in Engineering. 26(2):303–326. Available from: https://doi.org/10.1007/s11831-017-9241-4.

Chatterjee A. 2000. An introduction to the proper orthogonal decomposition. Current science:808–817.

Chinesta F, Keunings R, Leygue A. 2013. The proper generalized decomposition for advanced numerical simulations: a primer. Springer Science & Business Media.

Cueto E, Chinesta F. 2014. Real time simulation for computational surgery: a review. Advanced Modeling and Simulation in Engineering Sciences. 1(1):11. Available from: https://doi.org/10.1186/2213-7467-1-11.

Hermant N, Perrier P, Yohan P. 2017. Human tongue biomechanical modeling. in biomechanics of living organs: Hyperelastic constitutive laws for finite element modeling. Elsevier.

Jéhannin-Ligier K, Dantony E, Bossard N, Molinié F, Defossez G, Daubisse-Marliac L, Delafosse P, Remontet L, Uhry Z. 2017. Projection de lincidence et de la mortalité par cancer en france métropolitaine en 2017. Rapport technique Saint-Maurice: Santé publique France:1–80.

Lauzeral N, Borzacchiello D, Kugler M, George D, Rmond Y, Hostettler A, Chinesta F. 2019. A model order reduction approach to create patient-specific mechanical models of human liver in computational medicine applications. Computer Methods and Programs in Biomedicine. 170:95–106.

Niroomandi S, Alfaro I, Cueto E, Chinesta F. 2012. Accounting for large deformations in real-time simulations of soft tissues based on reduced-order models. Computer Methods and Programs in Biomedicine. 105(1):1–12.

Sturgis EM, Cinciripini PM. 2007. Trends in head and neck cancer incidence in relation to smoking prevalence: an emerging epidemic of human papillomavirus-associated cancers? Cancer: Interdisciplinary International Journal of the American Cancer Society. 110(7):1429–1435.

References

[1] S. Cresswell, "Locm: A tool for acquiring planning domain models from action traces", *ICKEPS 2009*, 2009.

[2] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving", *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[3] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and practice*. Elsevier, 2004.

[4] Y. Gil, "Learning by experimentation: Incremental refinement of incomplete planning domains", in *Machine Learning Proceedings 1994*, Elsevier, 1994, pp. 87–95.

[5] E. M. Gold, "Language identification in the limit", *Information and control*, vol. 10, no. 5, pp. 447–474, 1967.

[6] J. Kučera and R. Barták, "Louga: Learning planning operators using genetic algorithms", in *Pacific Rim Knowledge Acquisition Workshop*, Springer, 2018, pp. 124–138.

[7] D. Long *et al.*, *Theory of Finite Automata: With an Introduction to Formal Languages*. Prentice Hall, 1989.

[8] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, *Pddl-the planning domain definition language*, 1998.

[9] K. Mourão, L. Zettlemoyer, R. P. A. Petrick, and M. Steedman, "Learning strips operators from noisy and incomplete observations", in *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, AUAI Press, 2012, pp. 614–623.

[10] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time", in *Pattern recognition and image analysis: Selected papers from the IVth Spanish Symposium*, World Scientific, 1992, pp. 49–61.

[11] D. Pellier and H. Fiorino, "Pddl4j: A planning domain description library for java", *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 30, no. 1, pp. 143–176, 2018.

[12] C. J. V. Rijsbergen, *Information Retrieval*, 2nd. Newton, MA, USA: Butterworth-Heinemann, 1979, ISBN: 0408709294.

[13] C. Rodrigues, P. Gérard, C. Rouveirol, and H. Soldano, "Active learning of relational action models", in *International Conference on Inductive Logic Programming*, Springer, 2011, pp. 302–316.

[14] J. Á. Segura-Muros, R. Pérez, and J. Fernández-Olivares, "Learning numerical action models from noisy and partially observable states by means of inductive rule learning techniques", *KEPS 2018*, p. 46, 2018.

[15] X. Wang, "Learning by observation and practice: An incremental approach for planning operator acquisition", in *Machine Learning Proceedings 1995*, Elsevier, 1995, pp. 549–557.

[16] Q. Yang, K. Wu, and Y. Jiang, "Learning action models from plan examples using weighted max-sat", *Artificial Intelligence*, vol. 171, no. 2-3, pp. 107–143, 2007.

[17] H. H. Zhuo, T. Nguyen, and S. Kambhampati, "Refining incomplete planning domain models through plan traces", in *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

Towards automated test generation and fault tolerance analysis for programmable logic controllers

Enzo Brignon

Supervised by: Laurence Pierre

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: Brignon Enzo, 23/08/19

Abstract

PLC's (Programmable Logic Controllers) are widely used to control industrial processes. However testing such device's program is tedious, and suffers from human errors. The work presented in this article aims at automatically generating test benches, and performing fault analysis for such systems.

1 Introduction

Industrial processes such as assembly lines or weighing/packing solutions for food industry are systems that have to be controlled and monitored by an electronic device, or collaborating networked electronic devices. Programmable Logic Controllers (PLC's) are such devices that are widely used in industry, even in critical systems [gbc, 2015]. They are connected to the plant, taking inputs from it and sending actions to it. This interaction involves control and operations to compute the actions (commands) to be sent to the plant in reaction to given inputs. This is implemented as a PLC program downloaded to the controller [uni, 2016].

Grafcet (that stands for "Graphe fonctionnel de commande des étapes et transitions") [IEC, 1999] is a graphical model to represent PLC programs in a sequential and parallel way. It has been standardised (IEC 60848) and derived into a PLC programming language, namely Sequential Function Chart or SFC (IEC 61131–3) [IEC, 2001]. It is widely used to specify the behaviour of PLC programs, more precisely as the control operated on the outputs in response to the inputs. From such a specification, an implementation is (manually) developed in one of the programming languages available for PLCs. Thus mistakes can easily be introduced in the implementation of the system, tests are mandatory to detect such errors.

The goal of this work is, given a Grafcet specification, to propose a solution for generating executable tests which will allow automatic black box testing of PLC programs that should implement the given specification. Here a test bench will be a program running on a PLC, that is connected to the PLC under test and which emulates the plant, sending inputs to the tested PLC, and receiving its outputs. This plant emulation enables to check that the reaction of the PLC under test is as expected.

[is a model of the plant available?]

In addition, forbidden states and fault scenarios will be considered. Forbidden states are states of the system that shall never occur during execution, for example harmful situation such as a press is activated while an alarm is triggered. Fault scenarios are scenarios in which faults are injected and that are used to check the error handling capacity of the system. Faults can be for example perturbations of inputs or abnormal behaviour of a component of the system.

Our approach for the automatic test generation for PLC software from a Grafcet specification is composed of several phases. First the Grafcet specification is translated into a Petri Net-style intermediate model which is more formal than the Grafcet one. Then safety and reachability are structural properties that are checked on this Petri Net model to ensure that the specification does not entail uncontrollable behaviours during execution. Checking those properties permits to get rid of ill-formed Grafcet models and to give feedback to designers for further corrections. If the Grafcet fulfils safety and reachability properties, tests can be generated and composed with forbidden state checking or fault scenarios. A test is generated as a Grafcet model that can afterwards be translated into PLC source code (e.g., Schneider Electric/Itris Glips language, which is a textual view of Grafcet). Once translated, the test program can be used in a PLC to test the system. This process is detailed all along this document.

[How ?]

2 Grafcet

Grafcet is a graphical model used to specify the control program as sequential and parallel compositions of behaviours. In this section we briefly describe the elements that compose a Grafcet model and their meaning. Figure 1 gives an example of Grafcet model in which every construct is represented. It is composed of the following elements:

(1) Steps (square boxes), that can be active or inactive. The set of active steps represents the state of the system.

(2) Transitions, that are enabled when all their input steps are active. A transition can be associated with a condition (called receptivity e.g., C1 in Figure 1) that permits it to

be fired if enabled. The receptivity $t1/X5$ below step 5 is true when a time $t1$ has elapsed since the last activation of step 5 (denoted $X5$).

(3) Actions can be associated with a step. There are several kinds of actions, the three main kinds are:

- Pulse rising edge actions, that are executed when the corresponding step becomes active (action A2).
- Continuous actions that are executed while the corresponding step is active (action A3).
- Pulse falling edge actions, that are executed when the corresponding step becomes inactive (action A4).

(4) Parallel sequences divergence (double horizontal line with a transition above), permits, from a step, to activate several steps simultaneously. It is used to define parallel sequences.

(5) Parallel sequences convergence (double horizontal line with a transition below), permits to have synchronisation between several steps being simultaneously active. It permits to define the end of parallel sequences.

(6) Alternative sequences selection (simple horizontal line with several transitions below), permits to select a sequence among several ones, depending on the values of the receptivities associated with the outgoing transitions.

(7) Alternative sequence convergence (simple horizontal line with several transitions above), permits to make alternative sequence to converge on a single step.

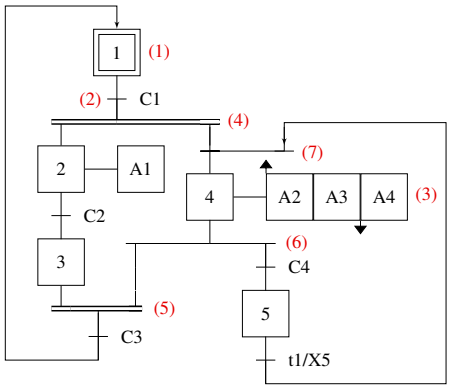


Figure 1: Grafcet example

3 Overview of the approach

In this section we summarise the successive steps of our approach.

- The first phase is the transformation of the Grafcet model into a more formal model, equipped with a formal semantics. We have chosen a variant of the Petri Net model [Peterson, 1977]. For the sake of brevity, this transformation is not presented here.
- The second phase consists in **verifying the well-formedness of this specification** i.e., the fact that it respects some properties known as "safety" and "reachability". To that goal, we have developed a specific algorithm presented in Section 5.
- If this verification succeeds, **generating test programs** is worthwhile. Section 6 presents the methodology that has been developed to that goal. The test program generates inputs for the program under test and checks whether the outputs are correctly produced in response.
- In general, and in the PLC context in particular, this is not sufficient. Indeed the specification of the PLC behaviour also often requires that the outputs (commands) behaviour is as expected, while not traversing some "forbidden states". For example, a control valve is opened at the right instant, while not managing another valve in the same interval. Thus, after producing the first raw test program, we can improve it in such a way that it also **verifies that some given "forbidden states" are not traversed**. Finally, those test programs enable to debug the PLC program in nominal situations.
- Another goal of this work is to offer the possibility to analyse the system behaviour when faults occur (for example, a sensor becomes out of order and delivers values which are out of an expected range). Thus the final phase consists in composing fault scenarios (also expressed as Grafcets) with nominal tests to produce "faulty tests". Section 7 describes how such **fault injection** is performed within test cases.

4 Related Work

Clearly, our objective is twofold: first, the verification that the specification of the PLC software is well-formed ("safety", "reachability"), then the automatic generation of test programs for the forthcoming application that should obey this specification.

There has been intensive work on the application of formal methods to reason about PLC programs. Surprisingly most of them are related to the use of model-checking techniques to verify general properties. Some approaches take as input the industrial implementation of standard programming languages to target specific applications such as [Fernández Adiego *et al.*, 2015] which applies model-checking methods on Siemens's SFC and ST implementations to verify CTL/LTL properties using nuXmv model checker. Alternatively, a more general approach consists in directly addressing the languages provided by the IEC 61131–3 standard. In [De Smet *et al.*, 2000] and [Lampérière-Couffin and Lesage, 2000] discrete models such as finite state machines to verify LTL properties, in [L'Her *et al.*, 1999] the authors use timed automata intermediate model to check Timed CTL formulas within Kronos model checker. In [Nellen *et al.*, 2015] and [Nellen and Ábrahám, 2012] the authors present an approach to translate "Hybrid SFC" into hybrid automata and perform state space exploration using SpaceEx space explorer. Finally [Mertke and Frey, 2001] uses Signal Interpreted Petri Net specifications, and uses in-house model checkers to verify EF and AG CTL properties.

To the best of our knowledge, only one significant work has addressed the well-formedness of SFC specification, and few attention has been paid to automatic test generation.

4.1 Dedicated solution for "safety" and "reachability" of SFC

Only [Huuck *et al.*, 2003] addresses the well-formedness of SFC specifications i.e., the satisfaction of the "safety" (no token proliferation) and "reachability" (no transition that is never firable) properties.

Classically their solution exploits model checking using CaSMV model checker. In particular each step is associated with one Boolean variable which is true if the step is currently active. Actions are not considered in this model because they do not play a significant role for the satisfaction of these properties. The authors give a formalisation of these two properties as temporal assertions that are meant to be verified within SMV. In order to limit the model size, they reformulate the assertions without guards expressions. Roughly speaking, safety amounts to verifying that there is always at most one token in every step, and reachability means that each converging transition (i.e., following a parallel convergence) must be firable. An example demonstrates that this method may miss some property violations in some case e.g. when the token proliferation "hides" unreachability. The state space of such approach is exponential in terms of the number of steps of the system. In addition the complexity of verifying invariant on such system is known to be PSPACE-complete.

4.2 Test generation

Here we focus on automatic test generation methods for PLC software, namely written in SFC language. Mainly Provost et al. address this issue, the goal is to generate input sequences for conformance testing.

Indeed in [Provost *et al.*, 2009], the authors give an approach to extract input test sequences from Mealy machines generated from SFC models. Targeted applications are SFC programs with only Boolean input/output variables, and without time dependent elements and pulse actions.
Roughly speaking, their method is composed of 4 steps. First, a Reachable Situation Automaton (RSA) is generated from the SFC specification. The second phase is the transformation of the RSA into a Mealy machine, performed by making the self loop explicit and letting the outputs be on the transitions. Extracting the minimum length input test sequence amounts to finding a minimum length closed walk that traverses each edge of the graph. To that goal they compute the minimum length Eulerian cycle. The third step is thus, if the graph is not Eulerian, to transform it into an Eulerian one by doubling some transitions. The final step is the generation of the input sequence from the resulting minimum length Eulerian cycle.
A test sequence is a list of pairs (input, output) that is, the series of input that is given to the PLC program and the expected resulting output. Performing such a test permits to detect two types of errors: transfer and output errors. A transfer error is an error in the step activity update (i.e., the next state is not as expected). An output error is an error in the performed action. To detect transfer errors, a distinguishing sequence may have to be generated to determine, given two states with the same output, which one is active.

An extension of this work is presented in [Provost *et al.*, 2014], in which the method is adapted to Single Input Change (SIC) testing. SIC testing can be used to model that simultaneous external events never occur.

In these solutions input sequences are classically generated from a finite state machine model. Since the model that is originally used is a Moore-style machine, this prevents from considering special actions such as pulse actions. Moreover if parallel behaviours have to be processed by building the product machine, this may lead to a huge state space size. Note also that the authors do not take into account timing constraints. In contrast, our solution supports parallel constructs. We also take into consideration all kinds of actions as well as timing constraints.

4.3 Test generation from fault models

Fault injection can be classified into three different approaches: Hardware implemented FI (HWIFI), model implemented FI (MIFI), and Software implemented FI (SWIFI). HWIFI is performed by physically providing perturbations to the system under test by inserting errors at pin level using for example electromagnetic interference or laser. MIFI corresponds to fault injection within the model under test for example by modifying the hardware description of the component to insert co-called "saboteurs". The approach that will be addressed here is based on software implemented testing, thus we will focus on this kind of FI. SWIFI is based on the insertion of fault code into different architectural levels of embedded systems. It can be implemented through mutants used on the system under test, or modification of test cases that inserts deviations that lead to fault. Different fault models exist, in [Avizienis *et al.*, 2004] a classification of fault models is given using several criteria such as: *phenomenological cause*, that is whether the fault is caused by natural phenomena or by human actions. The *dimension* criterion differentiates hardware and software faults. Moreover a fault can be either permanent or transient, which means that it occurs and remains permanent, or occurs only at certain time, this is the *persistence* criterion.

Few results have been presented regarding fault injection for PLC software, one of the groups which are particularly active is the Institute Of Automation and Information systems at the technical university of Munich. For example in the SWIFI context of [Rösch and Vogel-Heuser, 2017], they define a methodology for the automatic generation of test cases from fault models. Using the information of the fault model (associated with a Fault Injection Operator) and of the state in which fault injection should start, the method generates a test case that tests a specific fault scenario. The perturbations can be classified as follows: An *unexpected state change* or *interval violation* when the value of a variable is not as expected, or a *state change block* (resp. *force*) occurs when an event occurs too late (resp. too early). A test engine drives the test execution: it waits for the state in which fault injection should start, activates the fault from the plant and waits for the controller reaction.

In [Rösch *et al.*, 2014] the authors propose an approach to generate test cases with deviation from timing sequence diagram (TSD) using FMEA (Failure Mode and Effect Analysis) analysis. The generated test sequence first checks that a pre-condition is fulfilled before applying the deviation of the

test case. If the expected post-condition is true, then the test verdict is pass. The paths that corresponds to pre and post-conditions are extracted from the system specification (in IEC 61131–3 language).

In these solutions faults are defined as being either a change in the timing of the event occurring or an unexpected state change or absence of change. Checking the fault tolerance is performed by waiting for an expected reaction for the former one and the non-violation of a post condition for the latter. For both works, the fault is injected as soon as a precondition is fulfilled. Similarly in our solution the fault model is a permanent fault injected as soon as a precondition is fulfilled. However the verdict is a combination of different factors taking into account whether the test has passed or not, and the occurrence of a countermeasure. In addition, faulty scenarios can be composed with forbidden state detection.

5 Safety and reachability

5.1 Properties

The standardisation document [IEC, 2001] states that the evolution rules given to the SFC language cannot prevent the formulation of "unsafe" and "unreachable" SFCs. The definitions of "unsafety" and "unreachability" are as follows:

- An **unsafe** SFC is an SFC in which there is an uncontrolled proliferation of tokens. An example of such SFC is given by Figure 2a. This is due to the fact that there is no proper synchronisation between the two parallel sequences that start at S_2 and S_3. If there is a token in S_3 and if t_3 evaluates to true, the token will flow out the parallel construct and it can proliferate thanks to the loop to S_1.
- An **unreachable** SFC is an SFC in which an execution may lead to deadlock. An example of such SFC is given by Figure 2b. It is clear that if receptivities t_2 and t_3 are mutually exclusive, the execution of this SFC may inevitably result into a deadlock. When S_2 and S_3 are active, if t_2 evaluates to true, then S_4 becomes active and t_4 can be fired. However S_7 will no become active and t_6 can never be fired. Symmetrically, if t_3 is evaluated to true, then S_5 becomes active (but not S_4) and t_4 can never be fired.

The two following definitions are used in [Huuck *et al.*, 2003]:

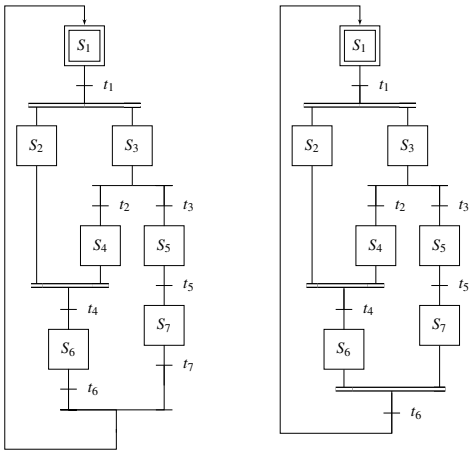
Definition 5.1 (Simple SFC)**.** *A Simple SFC is a triple* $\mathscr{S} = (S, s_0, T)$ *with:*

- S *is the set of steps,*
- s_0 *is the initial step,*
- $T \subset (2^S \setminus \{\emptyset\}) \times G \times (2^S \setminus \{\emptyset\})$ *is the set of transitions, each transition labeled with a guard g from a set G of transition conditions.*

Here $(S_s, g, S_t) \in T$ denotes a transition with S_s the set of source steps and S_t the set of target steps.

Definition 5.2 (Safe SFC)**.** *A SFC is safe if and only if*

1. *for all possible executions there is at most one token in a step,*
2. *for any converging transition, there exists an execution such that the transition can be taken.*



(a) "Unsafe" SFC (b) "Unreachable" SFC

Figure 2: Ill formed SFC examples from the standard

On this basis, the authors also define three "major violations" with respect to the structure of the SFC:

(1) The presence of a jump between different parallel sequences without synchronisation.

(2) The presence of a jump out of parallel branch.

(3) The presence of a synchronisation between multiple branches of an alternative divergence.

The purpose of [Huuck *et al.*, 2003] work is to use model checking techniques to verify whether SFCs are safe and reachable. To that goal, they formalise the problem of the evolution of the state of synchronous system, as explained in Section 4.1.

We can remark that rules (1) to (3) involve parallel branches and synchronisation. Thus we propose a **structure based** solution that detects the corresponding misuses of these constructs. The rough idea is to "colour" the branches of the graph in order to detect "incompatibilities" between "colours".

5.2 Algorithm

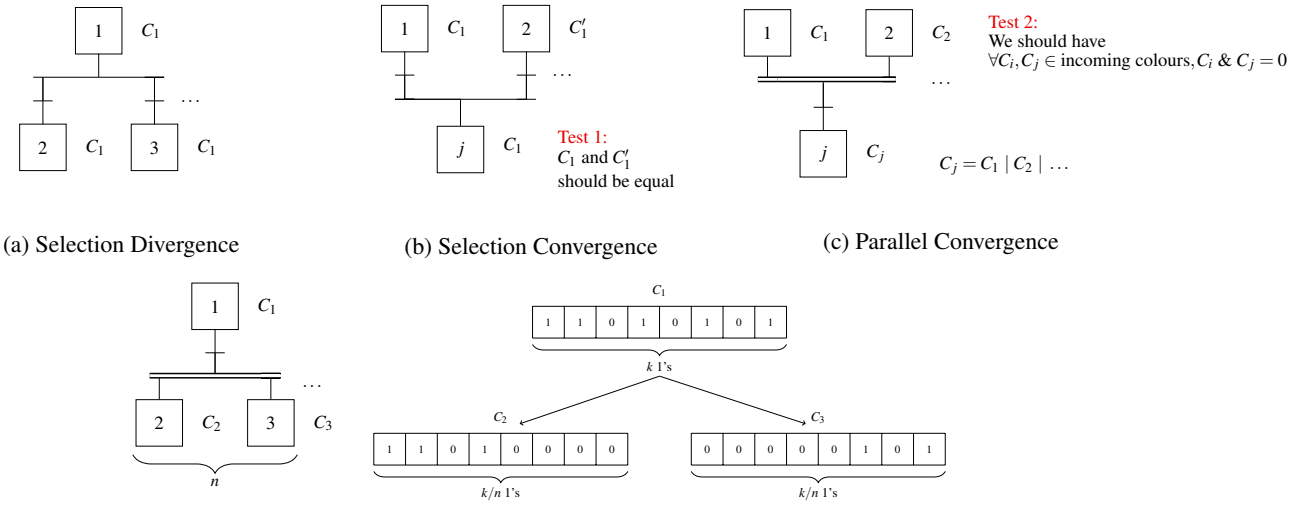
Let us first define this "colour" encoding. A "colour" is in fact a bit-vector, "black" is the vector in which every bit equals 1.

The algorithm will propagate "colours" in the graph in order to verify that

(P1) All incoming branches of a selection convergence have the same "colour"

(P2) All incoming branches of a parallel convergence have "compatible colours"

(P3) Any loop to an already visited step propagates the same colour.



(a) Selection Divergence

(b) Selection Convergence

(c) Parallel Convergence

(d) Parallel Divergence

Figure 3: Colour propagation in Grafcet

More precisely the following rules are applied to the Grafcet and its different constructions (these rules are represented by Figure 3):

1. First the initial step is coloured in black,
2. Then we repeat the following steps until the Grafcet is fully coloured:
   (a) For a transition having *one* coloured transition above it, its outgoing step takes the colour of its incoming step
   (b) On a selection divergence having its incoming step of colour C_1, every outgoing step takes colour C_1 (Figure 3a).
   (c) On a selection convergence, provided that all incoming colours are identical, the outgoing step takes this colour (Figure 3b).
   (d) On a parallel divergence with incoming colour C_1, C_1 is divided into as many colours as the number of outgoing steps and each outgoing step takes one of these new colours (Figure 3d). "Dividing" a colour into n new colours corresponds to distributing the 1's of the bit-vector to n new bit-vectors while keeping their positions.
   (e) On a parallel convergence, provided that all incoming colours are "compatible" (the bit-wise AND of the bit-vectors returns 0), the outgoing step takes the bit-wise OR of the incoming colours (Figure 3c).

This algorithm corresponds to function `Colour_Grafcet` in Algorithm 1, its parameters V and s are the set of vertices and the initial vertex respectively. It is not applied directly on the Grafcet model but on a simpler view of the graph (called NDC graph) that has three types of vertices:

N nodes that are "normal" nodes, they correspond to steps of the initial Grafcet model.

D nodes that are "parallel divergence" nodes, they correspond to transitions associated with parallel sequence divergences.

C nodes that are "parallel convergence" nodes, they correspond to transitions associated with parallel sequence convergences.

Every vertex has a colour, a parent node, and a list of child nodes. The `Colour_Grafcet` function uses functions `Process_par_conv` and `Next_colours` (not given here). The `Process_par_conv` function is used for a parallel sequences convergence, it computes the bit-wise OR after having checked whether there is a violation or not. The `Next_colours` function computes the "division" of a colour into a vector of new colours in the case of a parallel sequences divergence. `Colour_Grafcet` initialises the colour of the source vertex to black and enqueues this vertex. While the queue is not empty, it dequeues one vertex and tries to propagate colours according to the rules given above. Every new coloured vertex is en-queued.

The number of bits needed to colour the NDC graph is computed on this graph by performing a depth first search. When backtracking, the algorithm computes the number of bits needed to encode the colour of each node. This value is computed as follows depending on the type of the node:

N node The computed value is the maximum between the values carried out by the successors of the current node and 1. If the current node does not have any successor, its value is 1.

D node The computed value is the sum of the values of the successors of the current node.

C node The computed value is the value of the successor of the current node (it can have only one successor). In addition, the value that will be carried out to each of its predecessors (i.e., the value that its predecessors will use to compute their values) is computed, in such a way

```
function Colour_Grafcet (s, V):
  Input  : Vertex set: V,
           Initial vertex: s
  Returns: 0 if there is no error, 1, 2, or 3 if there is one (this number is the
           violation number).
  begin
    foreach v ∈ V \ {s} do v.colour ← "0 ... 0";
    s.colour ← "1 ... 1";
    Enqueue(s, Q);
    while Q ≠ ∅ do
      current ← Dequeue(Q);
      if D_node(current) then
        expected_colours ← Next_colours(current.colour, current.weights);
        for i ∈ [1, current.nbchildren] do
          child ← current.child[i];
          if (child.colour = 0) ∧ (child.colour ≠ expected_colours[i]) then
            exit(error);
          child.colour ← expect_colours[i];
          child.nbparents − −;
          if child.nbparents = 0 then Enqueue(child, Q);
      else if C_node(current) then
        current.colour ← process_par_conv(current);
        if child.colour ≠ 0 then exit(error);
        child ← current.child[1];
        if (child.colour ≠ 0) ∧ (child.colour ≠ expected_colours[i]) then
          exit(error);
        child.colour ← current.colour;
        child.nbparents − −;
        if child.nbparents = 0 then Enqueue(child, Q);
      else                                                    // N node
        for i ∈ [1, current.nbchildren] do
          child ← current.child[i];
          if C_node(child) then
            child.nbparents − −;
            if child.nbparents = 0 then Enqueue(child, Q);
          else
            if (child.colour ≠ 0) ∧ (child.colour ≠ expected_colours[i]) then
              exit(error);
            child.colour ← current.colour;
            child.nbparents − −;
            if child.nbparents = 0 then Enqueue(child, Q);
    exit(0);
```

Algorithm 1: Colouring algorithm

that the sum of the values carried out to all the predecessors is equal to the current number of needed bits. This means that the number of bits required by a C node is divided into as many new values as the number of its predecessors.

5.3 Examples

In this section we illustrate the algorithm on the examples given in Figure 2. The two examples will be coloured the same way however the final test will differ, for Figure 2a, the convergence of steps S_6 and S_7 is a alternative sequence convergence, thus the test that will be performed is a comparison of the two carried colours to know whether they are equal or not. For Figure 2b the convergence is a parallel sequences convergence that implies to check whether the pairwise bitwise and of all the incoming colour is equal to zero or not.

Propagating color

Computing the number of bits needed to colour all the steps of the graph returns 3. Thus the first step of the algorithm is to colour the initial step S_1 in black i.e. the vector 111. Then the colours are propagated as follows:

- Dispatch the bits of $S_1.colour$ to the two steps S_2 and S_3, $S_2.colour = 100$ and $S_3.colour = 011$.
- Propagate the colour of S_3 to its two successors S_4 and S_5.
- Propagate the colour of S_5 to step S_7.
- Check whether colours of steps S_2 and S_4 are compatible i.e. the bit-wise and is equal to 0 ($100 \ \& \ 011 = 0$) and mix them to generate colour of step S_6.

Table 1 gives the colours associated with each steps of the Grafcet.

S_1	S_2	S_3	S_4	S_5	S_6	S_7
111	100	011	011	011	111	011

Table 1: Colours given to steps of Grafcet Figure 2a

Unsafety checking

Checking the unsafety of the Grafcet Figure 2a corresponds to checking whether colours of S_6 and S_7 are equal ($111 \neq 011$) and detect that there is a violation, namely unsafety.

Unreachability checking

Checking the unreachability of the Grafcet Figure 2b corresponds to checking that colours of S_6 and S_7 are compatible ($111 \ \& \ 011 = 011$) and detect that there is a violation, namely unreachability.

6 Test generation and forbidden states

If the Grafcet specification is well-formed, test generation can be performed. In this section we first define our characterisation of test cases and the algorithm to generate them. It is illustrated on the example of Figure 4a that models the control program of a weighing-mixing machine [IEC, 1999]: two products are poured in a weighing unit (the two parallel branches), mixed, and the weighing unit is emptied.

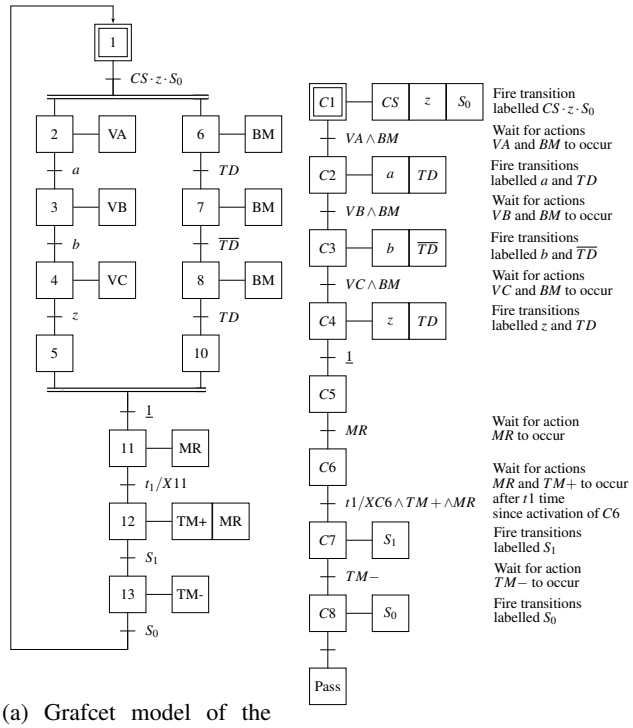
6.1 Test generation

Test cases are defined using Grafcet, that is the same model as the one which is used to specify the program under test. A test case is designed in such a way that it can be composed with the model under test in order to send it inputs and receive its outputs. The rough idea is as follows:

- For each step S of the program specification that can be activated when a condition c holds, the test program will include a step associated with an action that produces outputs (inputs to the program under test) such that c is fulfilled. For example, on Figure 4 step $C1$ is associated with an action that triggers the firing of transition labelled $CS \cdot z \cdot S_0$.
- To check that each action A associated with S is correctly performed in response to those inputs, the test program includes a transition the receptivity of which includes a condition that checks the outcome of A. For example if A is x $\leftarrow$ *value* then the condition will be x = *value*. On Figure 4 the action associated with step 11 (which is *MR*) is awaited in the test sequence by the transition labelled *MR*.

[The test coverage criterion should be given before, eg. in the introdution]

The goal of generated test sequence is to guide the execution of the system under test in a way that it fires all its transitions. If it is not possible within a single test case to have a complete transition coverage, several test cases are generated. The union of the covered transitions of all the generated test cases is the set of transitions of the tested model.



(a) Grafcet model of the weighing-mixing machine

(b) Corresponding test case

Figure 4: A Grafcet model and its corresponding test case

6.2 Algorithm

In this section we present the two main functions of the test generation algorithm: `Next_Firable` and `Gen_Test`.

First the algorithm does not reason on the structure of the Grafcet, instead it uses the interpretation of its behaviour. More precisely it considers the evolution semantics of the Petri Net model produced from the Grafcet. This semantics enables to characterise the PN state evolution, the state is defined as the set of the active places.
From this Petri Net model we generate a "next" function that expresses, for each place, the condition under which it will become active. Since we also consider actions, an "output" function enables to compute the value of each output in the next cycle.

The main function of the test generation algorithm is `Gen_Test` presented in Algorithm 3. It mainly uses function `Next_Firable` given in Algorithm 2.

This function `Next_Firable` is used to get the list of the possible next states. It takes as parameter a valuation of "next" function *st*, the "next" function and the list of steps. It returns a list of tuples (valuation of "next" function, condition, timer). Each valuation of the "next" function represents the next state, condition is the condition under which this next state is reached, and timer is the (shortest) timer associated with the transition(s). First `Next_Firable` gets the set of reachable steps E, and the associated condition. If there is only one future state in E, this future is returned. Otherwise (we are inside parallel branches), the algorithm checks whether the parallel branches can evolve simultaneously. To that goal it generates the conjunction of all the conditions associated with these next states $Conj$ and checks the satisfiability of this conjunction using the z3 tool [Microsoft, 2019]. As explained earlier, depending on this satisfiability, one or several tests are generated. To that goal, the set of reachable states is split according to the clauses of the "unsat core". This uses function `build_condition_with`. The parameters of this function are a Boolean formula x and a list of Boolean formulae u. It builds the conjunction of the all negated elements of u except the formula x which is kept in positive form.

In order to exclude timers from conditions in the generated test program, function `build_cond_transition` takes as parameters a timer c and a list of pairs (step, condition) E and it returns a condition that is the conjunction of all the conditions contained in E after removing the clause c in each element of E.

The algorithm is defined by the function `Gen_Test` (Algorithm 3). It takes as parameters the function "next" and the function "output" as well as the initial valuation of the "next" function, an accumulator (that is a Grafcet step), the set of steps and the set of output variables. The function `Call_Rec` is used to call recursively `Gen_Test` on each of the possible reachable states given by the former call to `Gen_Test`. From the initial valuation of the "next" function, the algorithm computes recursively the possible futures using `Next_Firable`. Using the information given by `Next_Firable`, a new step and an action are generated, the action is meant to satisfy the condition, and it is associated with the generated step. The "output" function is called to get the output that the program under test should return. It is transformed into a condition *Cond* to label the transition of the test program. If a timer t is returned by `Next_Firable`, no action is generated, a timing condition is added on the transition instead. If several future situations are computed, function `Call_Rec` is used to call recursively `Gen_Test` on each of the generated scenarios.

6.3 Decision about failure

As such, the test program can only check whether the output reactions are as expected. If one of those conditions never occur, the program will remain stuck at one of these steps. It is of utmost importance to get a relevant information about the failure reason. Therefore it is necessary to complete this specification with exit branches upon failure such that a specific error code is returned.

This is achieved using timeouts that will trigger if the expected reaction is not seen within a given period. An alternative sequence selection is inserted before each transition of the generated test sequence to add a transition that will

```
function Next_Firable (st, Next, S) :
  Input  : Next valuation: st          // The current valuation of Next function
  Input  : Function: Next                          // The next function
  Input  : Step set: S
  Output: List of tuple (valuation, condition, timer)
  begin
    E ← ∅;
    foreach s_i ∈ S do
      // Simplification of the formula's AST knowing state st
      A ← reduce(AST(Next(s_i)), st);
      /* A contains the condition that has to be satisfied for s_i to be active in the next cycle */
      if A ≠ ⊥ then E ← E ∪ (s_i, A) ;
    if Card(E) = 1 then
      st' ← ⋃_{s_i∈S} {(s_i, c) | c = s_i ∈ E|_c};
      condition ← cond(head(E));
      t ← first_timer(E);
      result ← List((st', condition, t));
      return result;
    else // Multiple steps are active or we are on a selection divergence
      t ← first_timer(E);
      if t ≠ Null then Conj ← build_cond_transition(t, E);
      else Conj ← ⋀_{e∈E} cond(E);
      if SAT(Conj) then          // If the conditions are compatible
        st' ← ⋃_{s_i∈S} {(s_i, c) | c = s_i ∈ E|_c};
        condition ← Conj;
        result ← List(st', condition, t);
        return result;
      else
        u ← unsat_core(Conj);
        Conj' ← remove(u, Conj);
        /* Generate a transition for each clause of u */
        result ← [ ];
        foreach x ∈ u do
          condition ← Cond' ∧ build_condition_with(x, u);
          st' ← List(s_i, reduce(a_i, SAT_instance(condition)));
          result ← append((st', C, t), L);
        return result;
```

Algorithm 2: `Next_Firable` function

```
function Gen_Test(next, output, v_0, acc, S, O):
  Input  : Function: next
  Input  : Function: output
  Input  : Next valuation: v_0
  Input  : Grafcet node: acc
  Input  : Step set: S
  Input  : Output variable set: O
  Output: A list of test cases (Grafcet)
  begin
    /* If a state (i.e. a set of step) has already been visited, then we end the test generation */
    if v_0 ∈ Already_Visited ∨ terminal_step(v_0) then
      return concat(acc, box(∅, ∅));
    else              // Otherwise we generate a new test step
      Already_Visited ← Already_Visited ∪ s_i ; // We mark the state as being visited
      L ← Next_Firable(v_0, next, s) ;       // We get the list of possible next states
      Lacc ← ();
      foreach l ∈ L do            // For each possible next states
        /* We generate the input and output corresponding to the transition to this next state */
        I_generated ← SAT_instance(cond(l));
        output ← ⋃_{o∈O} output(o, I_generated, v_0);
        Cond ← transform_into_condition(output);
        // If there is a timer, we generate a suitable condition
        if timer_cond(l) ≠ Null then
          Cond ← Cond ∧ find(step(timer_cond(l)), acc);
        Lacc ←
          concat(acc, (box(string(v_0), I_generatrd), transition(conf)), Lacc);
      /* For each generated tests steps, iterate on the next states */
      Call_Rec(next, output, L, Lacc, S, O);

function Call_Rec(next, output, L, A, S, O):
  Input  : Function: next
  Input  : Function: output
  Input  : Next valuation list: L
  Input  : Grafcet node list: A
  Input  : Step set: S
  Input  : Output variable set: O
  Output: A list of test cases (Grafcet)
  begin
    if L = () then
      ()
    else
      concat(Gen_Test(next, output, head(L), head(L), S, O),
        Call_Rec(next, output, tail(L), tail(A), S, O));
```

Algorithm 3: `Gen_Test` and `Call_Rec` functions

be fired if the expected reaction is not seen before a time-out (e.g., for transition labelled $VA \wedge BM$ after step $C1$ in Figure 4b, an alternative transition labelled with condition $timeout/XC1 \wedge \overline{VA} \wedge \overline{BM}$ will be added).

6.4 Forbidden states

In addition to checking that the model passes the test, it is often mandatory to check whether forbidden states have been encountered or not. In this section we define and explain how the occurrence of forbidden states is checked.

The principle is to continuously check if the forbidden state has been encountered until the end of the test sequence is reached. To that goal, as shown on Figure 5 the test sequence will be composed concurrently with forbidden state checking. A Boolean variable Frb is initialised to false and is set to true if the forbidden state has been encountered. A variable named $TestEnded$ becomes true whenever the test sequence reaches its final state. In that case forbidden state checking should stop.

Figure 5 shows the Grafcet model of a test sequence (hidden by the cloud) composed with a component that checks the occurrence of a forbidden state, pictured here by the Boolean formula $TM + \wedge BM$. The Grafcet component that is used to check if a forbidden state represented as a Boolean formula C has been encountered is composed of the three steps "E0", "E1" and "E2". The variable Frb is set to false on the step "E0" and set to true on step "E1". Once step "E0" has been activated, either the forbidden state is encountered (condition $C = TM + \wedge BM$ here) or the test sequence ends and Frb remains false. This latter case corresponds to an execution where the forbidden state is not observed before the end of the test sequence. If the condition C evaluates to true before $TestEnded$ is set to true, the transition from "E0" to "E1" is fired and then the step "E1" becomes active. Thus the value true is assigned to variable Frb letting the test case know that the forbidden state has been encountered. Then the transition from "E1" to "E2" is fired. At the end, when the step "E2" is active, either the test case is finished before seeing the forbidden state and the variable Frb is false, or the forbidden state has been encountered and the value of Frb. In both cases the value of Frb says whether the forbidden state has been seen or not. On the other hand Ok is true when the test sequence terminates successfully, false otherwise. At the end of the parallel composition of the test case and the forbidden state detection component, an alternative sequence selection

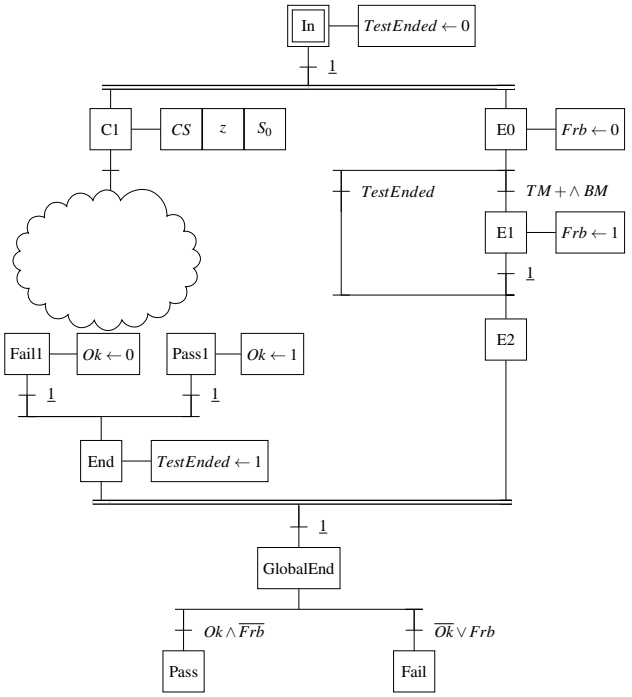


Figure 5: Test case with forbidden states

is added to give the final verdict: the test is pass if *Ok* is true and *Frb* is false, otherwise it is fail.

The construction of this parallel composition has been automated on the principles presented above, both when the forbidden state is a Boolean condition and when it is a more complex Grafcet.

7 Fault injection

In addition to test case generation and forbidden state detection, fault injection can be performed by composing a generated test case Grafcet model and fault injection Grafcet component in a similar way than for forbidden state detection.

7.1 Principle

The fault model covers external and permanent faults, i.e., faults that come from outside the system boundary and that propagate errors in the system. Permanent faults means that they remain "active" as soon as they are injected.

Thus fault injection is defined as the perturbation of an input or an output of the system that occurs when a given event is observed. Such an event can be characterised either using a Boolean formula over the input/output variables of the system, or a sequence of inputs/outputs of the system. It corresponds to a precondition C that has to be observed before injecting the fault. The fault itself corresponds to an action A executed after seeing C being true.

Two kinds of analyses can be performed, either *correction* analysis or *detection*. *Correction* is tested by checking whether the system's behaviour is still as expected after the fault has been injected. *Detection* is checked by testing whether a detection mechanism has been executed or not. These two types of analyses are specified by different Grafcet components, with few differences.

Figure 6 gives the general form of detection analysis fault injection component composed with a test case sequence. The Boolean formula D corresponds to the observation of the detection mechanism. It is true whenever this mechanism is observed. Similarly to forbidden state detection analysis, variable *TestEnded* is used to determine that the end of the test sequence has been reached. It is used to bypass the fault injection component if the precondition C for injecting fault is not fulfilled before the end of the test. In addition variables Inj and $Detected$ are introduced. Inj indicates that the fault has been injected, i.e., that the action A has been executed after meeting precondition C. $Detected$ indicates that the fault has been detected by the system under test, which means that the Boolean formula D has been evaluated to true after fault injection. If the fault has not been detected by the system under test within a time interval t after action A has been executed (i.e., after that step E1 has been activated) then transition labelled $t/XE1 \wedge \overline{D}$ is fired and the fault injection process is aborted without observing fault detection. $t/XE1$ is a Boolean variable that becomes true after a duration t from the last activation of step E1.

We can notice that it is possible to compose a test case both with some forbidden state detection components and a fault injection component. That permits to have an even more elaborated verdict.

7.2 Verdict

In this section we explain how a verdict is given in terms of the variables set by the test case.

For correction checking we have two variables of interest: Inj and Ok. For detection those two variable as well as $Detected$ are used to determine the verdict.

To give the verdict at the end of the test sequence, a Grafcet snippet has to be inserted after the last step of the sequence. This snippet is an alternative sequences selection that leads to steps corresponding to verdicts through transition labelled by the condition checking the verdict. On Figure 6 we have this alternative sequence selection after step "GlobalEnd". The possible verdicts are as follows:

PASStest SAFE The test has passed and the fault has been injected and detected.

PASStest Unchecked The test has passed but the fault has not been injected.

PASStest UNSAFE The test has passed but the fault has been injected and not detected.

OnlySAFE The test has failed but the fault has been injected and detected.

FAILtest Unckecked The test has failed but and the fault has not been injected.

FAILtest UNSAFE The test has failed and the fault has been injected but not detected.

The construction of this parallel composition has been automated on the principles presented above, and few experiments have been performed, in particular on the example of Figure 4a.

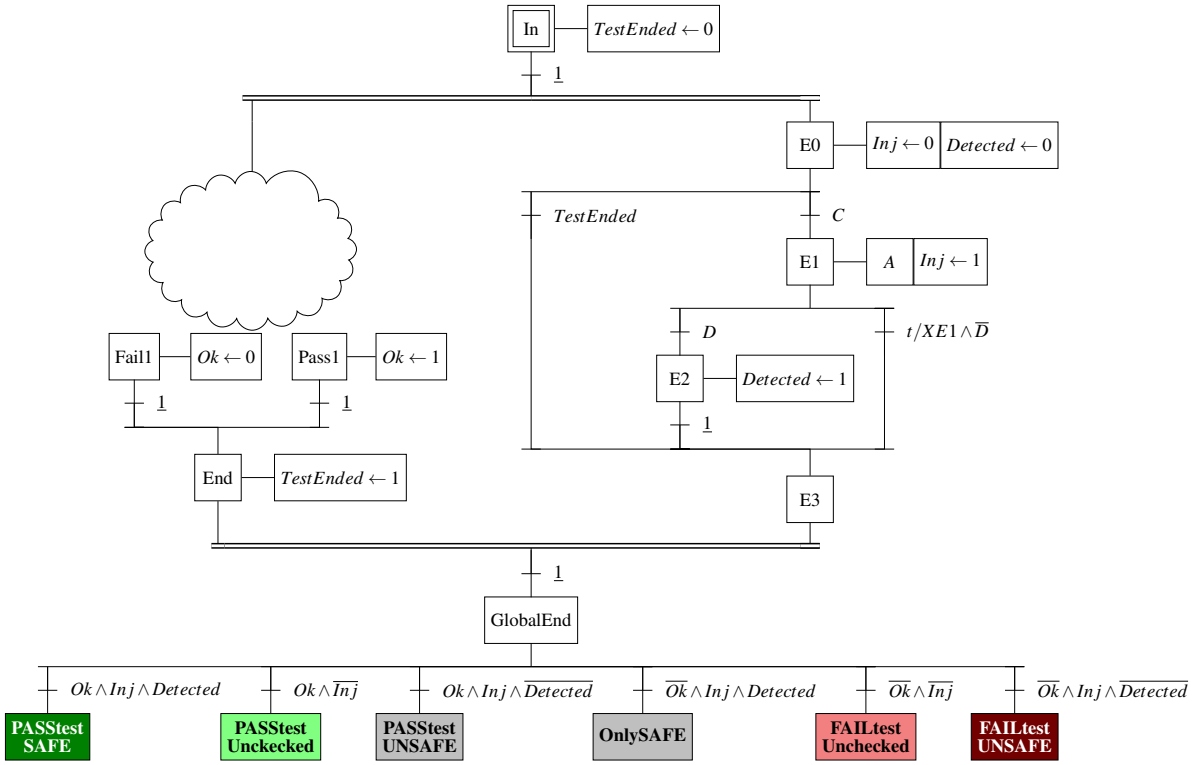


Figure 6: Test case with fault injection

8 Conclusion

Our purpose in this work is the automatic test generation for PLC programs from their Grafcet specification. Prior to producing the test program, we verify the well formedness of the specification. To that purpose, we have developed an original structure based method which has a better complexity than model-checking-based methods. Indeed this algorithm has a linear complexity: it processes (i.e., colours) each node only once, and each transition is traversed only once. Note that it can be assimilated to a breadth first search algorithm since every node that still has to propagate its color is enqueued. The complexity of breadth first search as well as depth first search algorithms is $O(V+E)$ where V is the number of vertices and E is the number of edges.

Test generation itself is made of several phases: the generation of raw test cases from the program specification, improvement of these test cases with explicit failure detection and error code emission, and the detection of forbidden states traversal. The generated test program is specified as a Grafcet that sends inputs to the system under test and receives its outputs in order to check that they behave as expected. Test cases are generated with coverage in mind, each transition of the system under test is fired at least once. This test generation solution has been specified and implemented, some experiments have demonstrated the applicability of this approach.

Moreover test case generation has been enhanced by adding support for fault tolerance analysis by defining a method for composition of test cases and fault injection components. These components are generated from a Boolean formula C which is the precondition to fulfil before injecting the fault which is represented by an action A (also given as parameter). Fault tolerance analysis is performed either by checking whether the system under test's behaviour corresponds to the expected one or by verifying that a detection mechanism (represented by a Boolean formula D) has been triggered. The automatic generation of such a component has been implemented. We have also investigated (but not yet implemented) the specification of this fault injection process when C and D are Grafcet components instead of Boolean formulas. A definition of the verdict of the fault tolerance analysis is given, several levels of gravity are translated by a rich set of verdicts. The automatic generation of the Grafcet component that gives the verdict at the end of the test case execution has also been specified and implemented.

Ongoing work includes enhancing the implementation and performing more experiments. Future work is related to the improvement of the specification of C and D when they should be more complex Grafcet components: the goal is to give the possibility to specify those behaviours as regular expressions, automatically translated into Grafcet specifications given as inputs to the fault tolerance Grafcet construction algorithm.

References

[Avizienis *et al.*, 2004] Algirdas Avizienis, J.-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and

taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1:11 – 33, 02 2004.

[De Smet *et al.*, 2000] Olivier De Smet, S Couffin, O Rossi, Géraud Canet, Jean-Jacques Lesage, H Papini, Chaire De Fabrications, and Ecole Normale Suprieure. Safe programming of plc using formal verification methods. 11 2000.

[Fernández Adiego *et al.*, 2015] B. Fernández Adiego, D. Darvas, E. B. Viñuela, J. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez. Applying model checking to industrial-sized plc programs. *IEEE Transactions on Industrial Informatics*, 11(6):1400–1410, Dec 2015.

[gbc, 2015] The World of PLCs is Closer than You Think: PLC Applications in our Everyday Lives. `https://www.gbctechtraining.com/blog/PLC-Applications-in-our-Everyday-lives`, 2015.

[Huuck *et al.*, 2003] Ralf Huuck, Ben Lukoschus, and Nanette Bauer. A model-checking approach to safe sfcs. In *IMACS Multiconference on Computational Engineering in Systems Applications*, 08 2003.

[IEC, 1999] IEC. *IEC 60848 Ed.2 Specification language GRAFCET for sequential function charts*, 1999.

[IEC, 2001] Programmable Controllers - Programming Language, IEC 61131-3, Final Draft. Technical report, International Electrotechnical Commission, 2001.

[Lampérière-Couffin and Lesage, 2000] S. Lampérière-Couffin and J.-J. Lesage. *Formal Verification of the Sequential Part of PLC Programs*, pages 247–254. Springer US, Boston, MA, 2000.

[L'Her *et al.*, 1999] Dominique L'Her, Philippe Le Parc, and Lionel Marcé. Proving sequential function chart programs using automata. In Jean-Marc Champarnaud, Djelloul Ziadi, and Denis Maurel, editors, *Automata Implementation*, pages 149–163, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Mertke and Frey, 2001] T. Mertke and G. Frey. Formal verification of plc programs generated from signal interpreted petri nets. In *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236)*, volume 4, pages 2700–2705 vol.4, Oct 2001.

[Microsoft, 2019] Microsoft. Z3 Theorem Prover. `https://rise4fun.com/Z3/tutorial/guide`, 2019.

[Nellen and Ábrahám, 2012] Johanna Nellen and Erika Ábrahám. Hybrid sequential function charts. In *MBMV*, 2012.

[Nellen *et al.*, 2015] Johanna Nellen, Erika Ábrahám, and Benedikt Wolters. A cegar tool for the reachability analysis of plc-controlled plants using hybrid automata. 346:55–78, 01 2015.

[Peterson, 1977] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[Provost *et al.*, 2009] J. Provost, J.-M. Roussel, and J.-M. Faure. Test sequence construction from sfc specification*. *IFAC Proceedings Volumes*, 42(5):299 – 304, 2009. 2nd IFAC Workshop on Dependable Control of Discrete Systems.

[Provost *et al.*, 2014] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. Generation of Single Input Change Test Sequences for Conformance Test of Programmable Logic Controllers. *IEEE Transactions on Industrial Informatics*, page 1, April 2014.

[Rösch and Vogel-Heuser, 2017] Susanne Rösch and Birgit Vogel-Heuser. A light-weight fault injection approach to test automated production system plc software in industrial practice. *Control Engineering Practice*, 58:12 – 23, 2017.

[Rösch *et al.*, 2014] Susanne Rösch, Dmitry Tikhonov, Daniel Schütz, and Birgit Vogel-Heuser. Model-based testing of plc software: test of plants' reliability by using fault injection on component level. *IFAC Proceedings Volumes*, 47(3):3509 – 3515, 2014. 19th IFAC World Congress.

[uni, 2016] `https://unitronicsplc.com/what-is-plc-programmable-logic-controller`, 2016.

Blockchain versus PKI : setting up and analysis of two architectures for securing an industrial system

Florian Barrois
Université Grenoble Alpes
Grenoble, France

Supervised by: Christine Hennebert
Laboratory LSOSP, CEA Grenoble, France

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

The security of information systems is essential in our society. The constant improvement of computers capacities brings the need to permanently elaborate new security solutions and the use of innovative technologies is relevant for this purpose. Even though these technologies are usually developed for a precise purpose, their underlying functioning may sometimes by exploited for different usages. In this article, we consider a requirement of end-to-end security of an activity involving embedded systems and conducted by an industrial company. We propose two solutions of security architecture. The first one involves the use of a blockchain multi-users wallet whereas the second one is based on digital certificates. We detail the functioning of each of these designs as well as their realization before comparing them according to many security features and performance parameters. Although both of these solutions are interesting candidates likely to be chosen to secure a system, it appears that one of them is particularly relevant for the usage we focus on.

[avoid generalities]

Key words : Blockchain, wallet, asymmetric cryptography, digital certificates, embedded systems.

1 Introduction

As the technology evolves, computer facilities become ubiquitous and the need of assurance on the systems security gets stronger. Indeed, while modern computers become faster and more powerful, the number of cyber-attacks keeps growing and these new capacities bring the need to permanently reevaluate and push forward detection and protection systems. The lucrative aspect of successful attacks make some sectors of activity like industry particularly susceptible to be targeted by such a threat. Even though the risk is known, many professionals consider this risk as theoretical and their infrastructures unlikely to be aimed, and thus choose to ignore it, which opens the door to malicious attacks, as the recent history has shown with intrusions like Stuxnet and Black Energy.

Securing a system implies the consideration of all its aspects, notably securing the data related to the company's activity and controlling the access to all devices and networks, but also studying the consequences of setting up a security architecture to protect the potential additional components and communication channels it involves. To this end, many security fixtures and technologies may be employed and combined together to provide a security level in line with the goods to protect. One may cite firewalls, biometric systems, or even blockchains or PKIs.

A blockchain is a distributed ledger recording a list of events. The legitimacy of these events, called transactions, is verified by some users contributing to the proper functioning of the blockchain, called miners, who place validated transactions in blocks, which, tied together, form the blockchain. These transactions, in addition to an event to record, notably include an identifier of their issuer, which allows to trace the activity of users on the chain. Because blocks are produced and linked cryptographically, it is not possible for a malicious user to alter the content of the ledger. Moreover, since this content can be seen by all users, anomalies can easily be detected and reported, so as long as a majority of users are reliable, the safe functioning of the blockchain is ensured. Such characteristics make of this technology a good candidate to guarantee security features like authentication, non-repudiation and traceability.

Public key infrastructures, or PKIs, handle keys employed in asymmetric cryptography and usually involve the use of digital certificates. These certificates are emitted by a certification authority and allow to associate a key to a user's identity, thus providing authentication of the operations performed with this key. Such infrastructures involve many actors and processes related to the management of keys and to the administration of certificates, and are commonly used as security architecture of modern information systems and in secure communications.

The choice of a particular security architecture remains more or less relevant depending on the usage. In this article, we focus on a complex industrial system notably involving artificial intelligence and Internet of Things, and requiring the compliance with numerous security features. We describe the elaboration and the development of two solutions entailing, for the first one the use of a blockchain wallet, and for the other one, the adoption of digital certificates. The benefits and drawbacks of these designs are then discussed by considering many security and performance parameters and synthesized to determine the best structure to adopt for this particular usage.

2 Case study

Our survey deals with a concrete need of an industrial company to secure its industrial park. This park is populated with robotic machines that involve many technologies. Each system is equipped with sensors that retrieve the indicative values required to ensure that the system is working properly, such as position or speed. Then this data is processed by an artificial intelligence (AI) unit that estimates whether the functioning is normal or not and records the adjustments to perform. The robotic system is also provided with a human-machine interface that allows operators to supervise the automated work and to take decisions in case of uncertainty from the AI. In order to guarantee both security and safety properties, the operators have to be physically present at the human-machine interface workstation of the robot they are in charge of.

In addition to the protection of the system against external attackers, the infrastructure must allow to record and trace the proofs of the behaviour of the artificial intelligence inference engine, which functioning is sum up by the Figure 1. During the system execution, both the operations ordered by the AI and the operators interventions must be recorded. Moreover, in order to start the system execution, the users must authenticate. Thus, in case of malfunction or control of the actions history, the archives can be analyzed and have the value of proofs in jurisprudence to impute responsibilities to the supervisor of the actions performed.

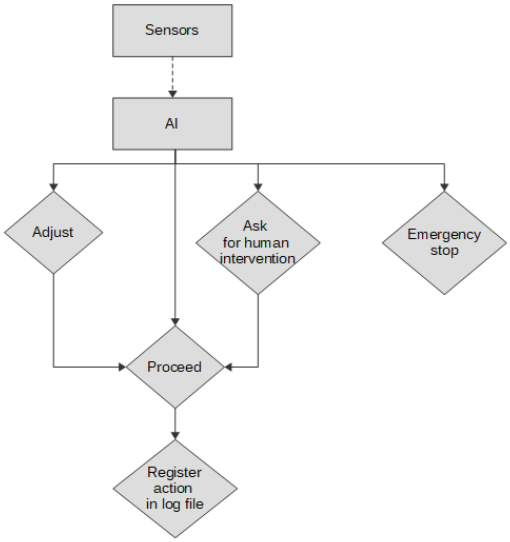


Figure 1: AI functioning

However, depending on the storage location of the traces recorded, both data and proofs may be negligently or intentionally tampered or erased. Consequently an adequate security architecture for this system must consider the securing of the data processed and the means put in place for this purpose.

In order to protect the company's activity, the following security needs have been listed out :

- The generation of proofs of the integrity of records
- The authentication of these proofs by the user present at the execution of the action recorded
- The time stamping of these proofs
- The ordering of recording of the proofs
- The ensurance of non-tampering of these proofs
- The non-repudiation of these proofs
- The traceability of these proofs over time
- The compliance with the users' privacy rights
- The resilience of the architecture.

Furthermore, in the aim of making the systems supervision more convenient and efficient, the said industrial company wished to have the possibility to monitor the machines at a distance, by using a mobile connected object.

In response to these needs, two security architectures have been studied : one involves the use of a blockchain wallet, while the other one is based on digital certificates. To be as close as possible to the real working environment, both of these solutions have been developed and tested in an ecosystem of devices simulating the functioning in an embedded system. However, because the final platform was not precisely defined in the project requirements conducted by the company, the setting up of the two proposed designs were done using a Raspberry Pi 3B+ as an intermediary test equipment.

3 Background and state of the art

We present three examples of usages of asymmetric cryptography as well as their global functioning. Each one involves many actors and different means to deal with the processes related to key management.

3.1 Digital wallets

An electronic wallet, or digital wallet, is a secure vault that contains one or many currency accounts. It may take many forms, like a web platform, a smartphone application or a physical component such as a USB key. Such wallets are generally used for the management of currencies. They notably allow to perform monetary transactions without needing a credit card while reducing the use of cash.

The usage of a digital wallet requires to create an account in an monetary organism (bank, service provider...). The user provides his personal details, his banking information and chooses a password. Depending on the account type chosen, the payments may either process a withdrawal directly from a bank account or withdraw the money placed on an electronic wallet account which must be alimented. When a wallet owner wishes to perform a payment (on a platform accepting this payment mode), he must pass an authentication step to access its wallet. Then, the banking details are no longer necessary since they have been filled at the account creation, so the password is enough to execute the transaction. Cryptocurrencies, like the famous bitcoin [1], can also be managed and exchanged via this tool.

Obviously, the monetary organism providing the electronic wallet solution must be trusted. The data should be encrypted and stored securely, and the service must respect regulations related to the protection of the users privacy. Because of these security requirements, the wallet must include all the elements, in particular cryptographic primitives, necessary to a secure and user-transparent execution.

Wallet accounts are associated to a couple of asymmetric keys that are used in the context of digital signatures operated via the Elliptic Curve Digital Signature Algorithm (ECDSA), proved cryptographically secure [5], employed by the blockchain technology to ensure the signature of all transactions and thus provide traceability of the actions initiated from each account.

There exists many types of internal wallet structure. For our usage, we focus on the one proposed in the Bitcoin Improvement Proposal (BIP) 32 [6] which describes Hierarchical Deterministic wallets. In such kind of wallet, a master seed works as a password giving access to the blockchain accounts hosted by the wallet. This master seed, which can also be more easily memorized as a sequence of words called mnemonic, must be generated randomly and known only from the owner of the wallet, and is used to generate a master account from which all other accounts are derived according to a certain path, as shown on Figure 2. This latter takes the form of a sequence of integer indices used in the calculations of the child accounts address and keys. The successive account derivations executed from the master account thus form a n-ary tree of accounts. All accounts are always generated in one and unique way, namely by following the same derivation path from a same precise seed.

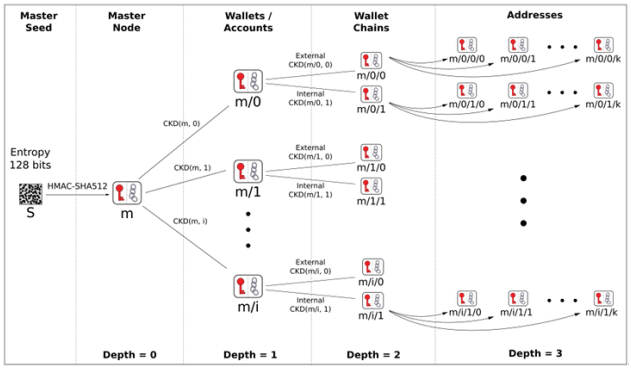


Figure 2: BIP32 Hierachical Deterministic Wallets

Other improvements from the Bitcoin Improvement Proposals themselves can be considered. BIP43, proposed by M. Palatinus and P. Rusnak [4], and based on the BIP32 standard introduces the "purpose" field in the derivation path. This intends to specify precisely the standard used for a specific HD wallet implementation among the numberless implementations claimed to be BIP32 compliant. The proposal consists in exploiting the first level of BIP32 based derivation (i.e. the second field of the key derivation path, the master node being omitted) to indicate the reference to the standard used. For instance, implementations based on BIP44 should attribute the keys a derivation path of the form m/44'/../..., which would match the index value 0x8000002C in the current hexadecimal representation.

BIP44 [3] is a particular application of BIP43, as it defines a role for the first derivation path level, but also for the next four levels. In addition to the purpose field from BIP43, this proposal includes in the path the coin type of the account used, an account field that differentiates the usages of a same coin type, a boolean change field to define whether the account address should be exploited for external exchanges or used internally as a transaction change address, and finally the index, which is the number of the child derived from this parent key. This scheme permits to order all the accounts generated by a wallet.

The usage of electronic wallets keeps growing. We distinguish many companies releasing their wallet version among which two main ones, Trezor and Ledger. These companies compete for the market of hardware wallets, which are the most secure type of wallet since they do not expose the private key to the internet when signing transactions.

The Trezor One is a hardware wallet that takes the form of a USB component. It was the first electronic wallet of this type, namely providing a secure physical storage space and limiting the exposition of sensitive data, compared to other types of wallet that store it either online or on a local but not necessarily protected computer. It offers the possibility to manage more than 500 different cryptocurrencies and relies on a PIN code chosen by the user and a mnemonic, namely a series of 24 words that allow the recovery of the master seed. Moreover, the dual authentication and an additional passphrase are optional features and act as supplementary security parameters. At a lower level, the security of the component is ensured by the bootloader that verifies the integrity of the loaded firmware and the firmware itself, which is subject to regular updates and is implemented to trigger a memory erasure of the data on the component in case of threat detection. As for the random seed generation, it is made by coupling the entropy of its internal physical random number generator (RNG) and the entropy of the computer it is wired to, what minimizes the risk related to the presence of a backdoor in RNGs.

The Ledger Nano S is another example of hardware wallet and presents the particularity to be the only certified (CC EAL5+) hardware wallet on the market. This wallet supports more than 1100 of the existing cryptocurrencies and offers a security via physical touch. Each transaction must be confirmed by pressing two buttons on the device, which makes useless the compromising of keys at a distance without owning the physical component. Like in the Trezor One wallet, the Ledger wallet bases its security on the use of a PIN code and a 24-word mnemonic and offers the dual authentication feature for transactions. But it is also composed a secured chip that contains a True Random Number Generator (TRNG) used to generate the seed.

The Ethereum blockchain

Ethereum[2] is a blockchain that introduces a new kind of usage. Its main characteristic is that it offers the possibility to manage other goods than money. Strictly speaking, the Ether (the Ethereum coin), is a cryptoasset that owns a monetary value as well. But although ether transfers are possible, the finality of Ethereum is the use of its specificity : the execution of code coming from decentralized applications called "dApps" that allow the deployment of smart-contracts. Smart-contracts are programs available on the blockchain. The Ethereum users have the possibility to execute the functionalities offered by this piece of code in exchange for Ether. Moreover, token contracts are a particular type of smart-contract that introduce a token representing a goods that can be exchanged like a cryptocurrency but which use is dedicated to the context of a particular application.

Smart-contracts present the advantage of being contracts which content cannot be modified, thanks to the underlying blockchain technology. They can thus turn out to be particularly interesting when applied to legal usages due to their high availability, auditability, transparency and neutrality.

Ethereum transactions are processed by the Ethereum Virtual Machine (EVM), a stack-based virtual machine that executes bytecode. The programs executed by this machine, the smart contracts, are implemented in a high-level language, for instance in Solidity, and are compiled to bytecode to be executed on the EVM. Since smart-contracts are computer programs, they are subject to the problem of termination unpredictability stated by A. Turing. However, the execution of smart-contracts on the Ethereum Virtual Machine is costly : in addition to the computing resources common to all computer system, it requires Ether. This limitation introduces the case of loss of Ether due to the execution of programs that never end. To overcome this complication, Ethereum establishes the mechanism of gas. The gas represents the resource spent during the execution of a transaction. By determining an amount of gas to each one of the instructions executed by the EVM, when users come to execute a functionality offered by a smart-contract, they fix a limit of the amount of gas they are ready to pay, which solve the undesirable effects of never-ending functions.

Every computation realized as a result of the execution of a smart-contract in a transaction incurs a fee. The transaction fee is paid to the miner who validates the transaction and includes the gas required to execute the functionality requested as well as a supplementary fee that can be seen as a tip for the miner and thus incites the miner to choose this transaction to validate in order to receive more Ether. The gas is thus one of the mandatory fields of Ethereum transactions and contributes to the calculus of this fee, just like the gas price. The gas price is the amount of Ether the transaction issuer accepts to spend on every unit of gas, and is measured in Wei, which is a fraction of Ether. The gas limit previously mentioned corresponds to the product of the gas price and the gas. In case the transaction sender does not provide the gas necessary to the execution of the transaction, this latter is considered as invalid and the amount of gas specified by the sender gets lost.

3.2 PKIX

PKIX is a public key infrastructure based on the use of X509 digital certificates. The data gathered in such certificates include varied indications among which :

- The subject name refers to the identity of the person associated to the public key in this certificate
- The public key, which associated key stays secret and known only by its owner
- The address of the person
- The city where resides the person
- The company or organization if needed
- The validity period of this certificate
- The signature of this certificate by its delivering organization, which attests the correspondence of the person with this public key, and thus needs to be reliable.

The PKIX architecture introduces two roles. The Registration Authority (RA) is in charge of the verification of the user's identity when one requests for the creation of a certificate. The approval is then sent to another entity called the Certification Authority.

The Certification Authority (CA) is responsible for the creation of the certificates as well as the mechanisms put in place for their update or their cancellation, notably the Certificate Revocation Lists (CRLs) which indicate the certificates that should not be used anymore for reasons of private key loss or theft, renewal or expiration of the couple of keys, or even expiration of the certificate itself.

As the security of public key infrastructures comes from the trust in a third party from the entity wishing to make sure of the belonging of a key to a certain other entity, and because actual network of relationships are complex, many Certification Authorities may be involved during an identity check process. In particular, the authenticity of a CA may be ensured by another CA. PKIX works with a hierarchical internal organization, as shown by the Figure 3, where the root CA is recognized valid thanks to an auto-signed certificate, and thus needs to be trusted anyway.

Once a user has retrieved, in a secure way, the public key of a CA, he can ask for the creation of a certificate that binds his identity to a public key. The CA then proceeds to the type of certificate creation needed. Either the user generated his couple of asymmetric keys prior to the certification request, in which eventuality the CA fills the certificate with the public key it received from him, or the user needs keys in order to get identified and the CA then generates them, communicates them to the user and fills the certificate with the public key of this couple.

The CA is in charge of the renewal of certificates, which is necessary when a certificate or the key it refers to comes to expiration. It must also deal with its own CA migration, that is to say the change of its own keys. For this purpose, it must create its new pair of keys and three certificates, which dependencies are represented on Figure 4 :

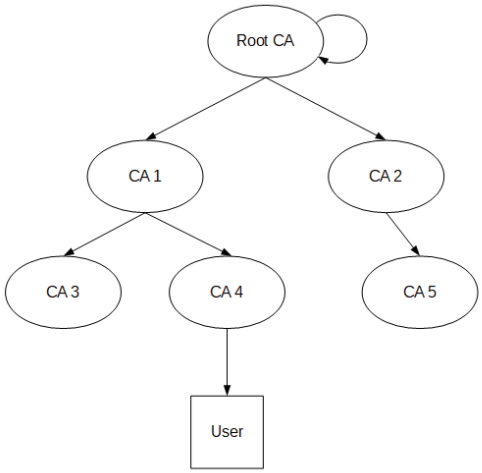


Figure 3: Certification hierarchy

1. A self-signed certificate for the new public key and signed with the new private key
2. A certificate for the new public key signed with the old private key
3. A certificate for the old public key signed with the new private key.

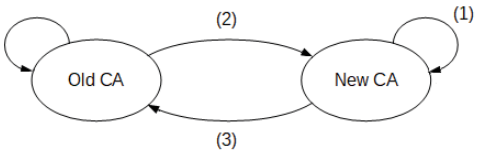


Figure 4: Certification Authority migration

This process actually consists in a particular case of peer-to-peer cross-certification, that is to say the mutual certification of two CAs, which remains another role of the CA.

Finally, the Certification Authority must support the Online Certificate Status Protocol (OCSP) if the hierarchy of CAs it belongs to employs it. OCSP requests remain an alternative to the Certificate Revocation Lists for the cancellation of certificates. They allow to directly ask a CA whether a precise certificate is valid or not, which avoids the request sender to proceed to the cascading verifications of the certificates and CRLs of the CAs of the architecture and to download and analyze the possibly long CRLs. Consequently, there is less much overhead on the client and the network. However with this mechanism the CA must deal with a significant number of requests.

3.3 PGP

Pretty Good Privacy (PGP) is another example of architecture involving asymmetric cryptography. Unlike PKIX, in PGP, the data relating to the users identity and to the functioning of the infrastructure is divided in packets, that are concatenated to form PGP files. There exists about 20 packet types that all contain a different information, among which the public key packet, the user ID packet and the signature packet, constituting the main data of digital certificates.

The content of PGP certificates is similar to the one of X509 certificates. Still, there exists a few subtleties that discern them. Notably, while in PKIX the unicity of certificates is ensured by the couple (certificate serial number, CA identifier), PGP employs the fingerprint of public keys, produced via a hash function. But the main difference between X509 and PGP certificates remains in their usage. Indeed, PGP certificates can include many signatures. The reason for this is that in this infrastructure, there is no Certification Authority. The trust comes from the number of users, spread out in a peer-to-peer network, that may have used another user's key and can thus attest that it belongs to the right person by signing the corresponding certificate. This functioning is commonly called Web of Trust.

PGP introduces several levels of trust :

- No trust, when the concerned key has never been used by the checking user and that none of the his trusted keys trusts this key.
- Partial trust, when at least m partially trusted keys applied their signature to this key. m is chosen at the model creation and must be equal to or higher than 2.
- Complete trust, when at least n ($1 \leq n \leq m$) completely trusted keys trust this key, with n chosen at the model creation.
- Ultimate trust, when the private key is known. This should only concern owned keys.

Likewise, one lists out many verification levels for signatures :

- Level 0, when no origin is provided.
- Level 1, when no verification of the identity was made.
- Level 2, when some basic verifications were made.
- Level 3, when a complete verification process was performed.

Like in PKIX, the revocation of certificates is possible with PGP. This is done via the sending of revocation packets, that must be signed by the certificate subject, to all the users who signed it and thus trusted it.

In practice, the PGP application is considered as a hybrid cryptographic system, since it uses both symmetric and asymmetric cryptographies. When a user encrypts a message with PGP, the data first goes through a step of compression. This one allows to reduce the transmission delay, to save disk storage space and to reenforce the cryptographic security. Then this encryption step is performed via many operations. First, PGP randomly creates an IDEA (International Data Encryption Algorithm) symmetric secret key. The data is then encrypted with this key. Afterwards, this secret key is itself

encrypted and transmitted to the receiver by the means of asymmetric keys. The receiver realizes the reverse process to retrieve the data. The association of symmetric and asymmetric cryptographies allows to keep the performance of symmetric encryption without decreasing the security level provided by the use of asymmetric keys.

In addition to encryption, PGP offers most of the expected functionalities of a secure messaging application, such as digital signatures and message integrity check, based on the use of MD5 and RSA, key transport, or even key exchange using the Diffie-Hellman algorithm.

4 Development of a security solution involving a blockchain wallet

The first security solution introduces the use of a digital wallet. The HD wallet implemented, compatible with the Ethereum blockchain, is inspired from an open-source implementation based on the BIP44 standard. Its development and test phases involved the use of two tools : Ganache and Pyeth.

Ganache is a simulator of the Ethereum Virtual Machine. As transactions sent to a blockchain are costly, this tool is interesting by virtue of the virtual blockchain it runs. It notably offers a graphical interface and permits to manage many predefined accounts owning ether that allow to proceed to transactions and observe the internal behaviour of the blockchain.

Pyeth is a Python library introducing functionalities to connect to the Ethereum blockchain and interact with it, both for reading blockchain information like an account balance and for updating it by sending new transactions. It was thus exploited in the scope of this work to connect to the virtual blockchain offered by Ganache and exploit it with the accounts of our wallet.

The security of this architecture is guaranteed by many factors. For the use case we focus on, the wallet application is embedded on a Raspberry Pi, as can be seen on Figure 5, which represents the personal mobile physical device required to access the system and allowing the operators to remotely interact with it. The exchanges between the device and the host system are detailed in Appendix A and can be sum up as follows.

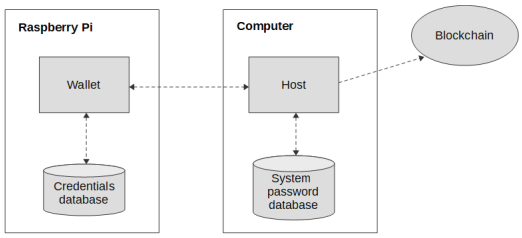


Figure 5: Ecosystem of the blockchain security solution

When a wallet connects to the host system, the user must enter on the industrial system interface its login, the password of the host application providing the access to the system and a PIN code, that is checked before unlocking its wallet account. The PIN code is sent to the device which hosts a database of credentials and unlocks the corresponding wallet account in case the PIN code provided effectively matches the user login. The wallet is then ready to use and sends the unlocked account address back to the host. The account of the wallet is generated dynamically when the PIN is recognized valid. This confers a security protection to the potential reverse engineering attacks on the wallet. Only the seed must be stored, in a secure place. Once the account available, the password provided to access the system is verified by consulting a local database, and if correct, permits the start of the system activity.

During the system activity, all actions it does are traced in a local log file. Periodically, the data recorded is used as input of a hash function to produce a digest that plays the role of proof of integrity of this data. A blockchain transaction is then formed and the proof is placed inside it. This transaction is sent to the wallet so that the private key of the current user signs it. The wallet then sends the signed transaction back to the host, which possesses a connection to the blockchain and can submit the transaction. The proof of integrity of the data recently produced is thus stored on the chain by the means of a smart-contract, implemented in Solidity, previously deployed for this purpose. Finally, the block identifier is retrieved from the chain and placed in the log file to ease auditors to exploit the proof.

The compatibility of the wallet with the Ethereum blockchain was studied and implemented on the base of the Python library eth-account[1]. This one was notably used to implement the generation of the Ethereum account address and Ethereum signatures.

Ethereum account addresses are generated from the public key of this account. The generation process is made of many steps. The public key is first serialized to bytes. The value obtained is then passed to a keccak256 hash function. The resulting account address is composed with the 20 less significant bytes of this hash, usually written as 40 hexadecimal characters. Account addresses also have a checksum form. It is used as a verification of the validity of an Ethereum address. This checksum address is similar to the normal one but some of its hexadecimal values are in uppercase. The position of the uppercase letters is defined by the raw binary keccak256 hash value of the address bytes.

Signatures in Ethereum are ECDSA signatures using the parameters given by the curve secp256k1 and are composed of three parameters. In Solidity, the function ECrecover returns an account address given these three parameters. The returned address can then be compared to the sender's address to determine whether the signature is valid or not. This verification process is notably used throughout the sending of transactions.

[1] https://github.com/ethereum/eth-account

Ethereum transactions are encoded with the RLP (Recursive Length Prefix) serialization library, and are composed of several fields :

- The **nonce** relates to the number of transactions already sent from this account
- The **gasPrice** is the amount of ether the user chooses to pay per gas used to execute the transaction. The gas can be considered as the computing resource consumed to proceed transactions. And like Bitcoin users may set higher transaction fees for their transactions to be mined faster, on Ethereum the gas can be used to incent the miners
- The **gasLimit** designates the maximum amount of gas the transaction sender accepts to pay for this transaction to be mined
- The field **to** contains the receiver's checksum address. This address may be either a user address or a smart-contract address
- The field **from** is optional, as it can be derived from the signature after signing the transaction with the private key
- The **value** is the amount of ether to send
- The **data** is specific to the transactions dedicated to smart-contracts and may contain a request for a function call or a contract creation.

The wallet developed is adapted to the usage : contrary to the classical use of an electronic wallet that must be specific to one particular user, our wallet is dedicated to be multi-users. Indeed, in our usage, the physical devices hosting a wallet application are the property of a legal entity, the client company. However, the numerous operators involved in the industrial process each have a dedicated wallet account, derived automatically from their identity at the connection to the industrial system. Thus all actions performed during their working time are electronically signed using this personal wallet account, which provides the authentication property. In case of legal procedures concerning the activity of this company, the events are traced and can be provided to the legal organization in the name of the company. But internally, the company managers are also capable of assigning responsibilities to the employees.

5 Development of a security solution involving digital certificates

The other version of the application developed involves digital certificates. These certificates are of the form of the X509 standard and constitute the element employed to guarantee the authentication of the events recorded by the system.

The global functioning is the same as in the blockchain version. The user must authenticate on the system via a PIN code associated to his login and the password of the application running the system activity. However, the personal mobile device does not contain a wallet, but a set of digital certificates belonging to the users authorized to operate on the robotic machine. When a user logs in, the global access to its physical device is provided and a certificate which subject name corresponds to the user's login is searched in the file tree of the device. The personal object then sends the digital certificate back to the host machine. The password of the application located on the system is verified and the activity starts. The system data is stored in a log file like in the blockchain wallet version and the proofs generated are sent to the device which signs them by means of the private key located on it that corresponds to the certificate used. Once the signed proof received from the physical device, the host forms a message containing the proof, the signature and the user certificate and transmits it to a remote database. All the verifications related to the proof and the user certificate are then performed on this remote database, which finally store both the proof and the signature and associates a timestamp to their registration. This time, in addition to the data and the proofs, the log file located on the robotic system contains an identifier of the user certificate.

This solution distinguishes from the first one presented by the exchange of messages instead of blockchain transactions. Furthermore, the association between a user and a key is clearly established. But also, the use of digital certificates implies the presence of other actors in the process environment and their functioning requires to be secure in the same way as the main components. Indeed, the remote server storing the proofs and their signatures must deal with the problems of access right management, data tampering and confidentiality, and denials of service, since its availability is imperative for the functioning of the industrial park. In addition to that, the users certificates must be previously generated by a certification authority. This implies that the company must endorse this role or that a list of trustable authorities has to be established. The usual mechanisms related to public key infrastructures is also to take into account. The keys and the certificates handled beg the questions of the key renewal, and of the validity period and the correctness of the signatures of all certificates and certificate revocation lists that needs to be considered during the process. These involve numerous verifications to perform, which considerably complexifies the architecture, as illustrated on the Appendix B.

In order to obtain a first relevant comparison of the two solutions implemented, the version involving digital certificates has been simplified by only considering auto-signed certificates, what reduces the number of operations to carry out during the procedure, as can be seen on Appendix C.

6 Performance study

In order to compare optimized implementations of each version, notably concerning the time and resource performance, the digital certificates employed during the tests are filled with the minimum information required, namely a subject name, a validity period, a public key and the signature of the certificate performed by the certification authority, which in our case remains the private key associated to the public key of the said certificate.

We face the two versions together along some performance indicators and the security features expected :

Integrity The integrity of the data generated is ensured in both versions. However, this is not the case concerning the integrity of the proofs. Indeed, in the variant involving digital certificates, the proofs are store in a database, which, contrary to a blockchain, may be subject to data tampering, reordering or erasure.

Time-stamping In the blockchain version, the time-stamping and the ordering of the proofs are guaranteed while, in the certificate version, the time-stamp is stored in the database and is thus subject to the risks aforementioned. Furthermore, a modification of the internal clock of the server would also skew the time-stamping of the proofs.

Authentication The need for authentication is fulfilled, whether with the signature of blockchain transactions from a wallet account or with the signature using the private key associated to the public key of a user certificate.

Privacy Because wallet account addresses provide authentication without user identification (only the account address is visible on the chain and in the log file), the use of accounts allows to respect restrictions about users privacy. On the other hand, digital certificates may contain personal data. In our implementation, these fields are left unfilled, but the user login is used as the certificate subject name. Thus, additional measures must be taken to comply with the users rights about privacy.

Identity trust The backside of the good management of the users privacy provided by the blockchain is a lack of trust in the users identity. Indeed, in the blockchain technology, the users own account numbers that are visible in all the transactions submitted, providing traceability, and that are the only public information about the transactions issuer. But consequently, this can be considered in the best case as pseudonymization and finally hides the users real identity, what makes this technology particularly popular for illegal activities. In contrast, the digital certificates handled in public key infrastructures are specifically dedicated to the association of users identity to the public keys they employ. Their identity can thus be surely established, although the trust in the issuing certification authority is still necessary.

Key management Since the wallet account addresses result from a hash digest, there is no key management to handle when exploiting the blockchain implementation. The one involving certificates, however, has a duty to keep the confidentiality of identities included in the certificates transiting through the communication channels. Consequently, a key exchange, like Diffie-Hellman, is in order. This might also imply key renewals.

Availability The functioning of both designs is limited by the availability of the remote entities the host system must communicate with. Whether a blockchain or a remote server hosting a database of proofs, this entity is necessary to permit the execution of the system tasks. Thus the company's activity remains directly dependent of the availability of this actor, which may be affected by the quality of the network communication channel for instance.

Bandwidth The amount of data transiting between the actors of the application during its execution has been measured for both versions via a network packet sniffer. By omitting the packets transmissions due to the network protocol and focusing on the ones related to the usage of the application, it has been established that, in average, 3.426 KB of data were sent using the blockchain version versus 3.180 KB with minimal certificates. Nevertheless, the second variant does not provide any cryptographic confirmation of the operations done like blockchain does thanks to transaction receipts. In addition, the receipts recorded during the tests of the blockchain version are close to 1.460 KB, due to the supplementary information specific to the blockchain included in the packets. So the blockchain version causes a greater volume of data to travel over the network, but provides an additional guarantee compared to the other implementation.

Storage memory The certificate variant requires to store the users certificates employed for the authentication, what is rather inefficient in terms of occupied storage space. As for the ledger, it offers much more storage space than a database by virtue of its distributed nature, but a significant part of this space is employed for information related to the chain.

Resilience In public key infrastructures, the trust is delegated between entities that share their knowledge network. Consequently, the leakage of a private key or the presence of a rogue certification authority may propagate through the whole hierarchy of trust. The time required for the identification of vulnerability in the architecture and its remediation via the key renewal process or the update of reliable entities may be significant and the activity of the company hosting this infrastructure might remain seriously impacted. On the other hand, a blockchain wallet embedded in a personal object manages the eventually of a key leakage in the best possible way since the possession of the physical device would be necessary for a malicious user to access to the wallet, take advantage of the leaked account and affect the company's activity. Moreover, because a blockchain is distributed, the resilience is ensured by design by the replication of the ledger between independent actors, which is not the case of a local database.

The efficiency of the solutions set up is sum up by the table below :

Criteria	Blockchain wallet	PKI
Integrity	◆◆◆	◆◆◆
Time stamping	◆◆◆	◆◆
Authentication	◆◆◆	◆◆◆
Privacy	◆◆◆	◆
Availability	◆◆	◆◆
Bandwidth	◆◆◆	◆
Key management	◆◆◆	◆
Identity trust	◆◆	◆◆◆
Resilience	◆◆◆	◆◆
Storage memory	◆◆	◆◆

Figure 6: Synthesis of the comparison between the blockchain and the PKI versions set up

Globally, the use of blockchain in the security architecture is a relevant choice for this usage, as the other version implemented requires additional technical means that bring a higher level of complexity in the ecosystem.

7 Conclusion

It appears that both of the architectures put in place may be employed in the real industrial environment targeted. Still, to confer the same security level as the one provided by the use of a wallet, the version based on a PKI introduces a greater degree of complexity due to the presence of more actors and the necessity of an increased number operations. Indeed, the certification authorities providing the users certificates must be trusted and, contrary to a blockchain, the database in charge of the storage of the proofs is not distributed and thus may be tampered, which also gives the possibility to the intruder to fake the order of the operations recorded. A solution could be to introduce a mechanism of data redundancy, but this also involves more equipment and increases the attack surface. Furthermore, the number of security verifications about the certificates and the electronic signatures is significant and may slow down the system execution speed. In the case of the use of non auto-signed certificates, this number is even greater and a connection to public repositories hosting namely certificate revocation lists is required and involves again network exchanges that need to be secured. The PKI also works thanks to a key management scheme that would oblige the company to a costly continuous maintenance.

The use of a hardware wallet, on the other hand, provides a physical security that adds to the numerous security features provided by design by the blockchain technology. The notions such as non-tampering, non-repudiation and traceability are thus guaranteed with no need of supplementary measures to be taken by the company. Moreover, the principle of a blockchain itself reenforces the security compared to a PKI that requires the trust in a third-party. This solution is also ideal as for the respect of the users privacy and, although this advantage is counterbalanced by the lack of absolute identification of the operators, it remains a relevant choice for this usage.

This survey led to the development of a Proof of Concept of a security architecture involving the use of a multi-users blockchain wallet applied to the secure management of proprietary industrial data. This implementation is a first draft example of how this innovative concept can be exploited and it may then be subject to further investigations to provide a better security level of a better performance. In particular, hardware security measures may be studied in order to provide a secure software execution and storage of the sensitive data like the wallet master seed and the accounts private keys. The performance tests conducted may also be run on different platforms to obtain a better range of results and a preview of the effect of the hardware architecture on the software. Although this study was dedicated to a precise usage, it may turn out to be a relevant start point for more global surveys.

References

[1] A.M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, Inc., 2nd edition, 2017.

[2] A.M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly Media, Incorporated, 2018.

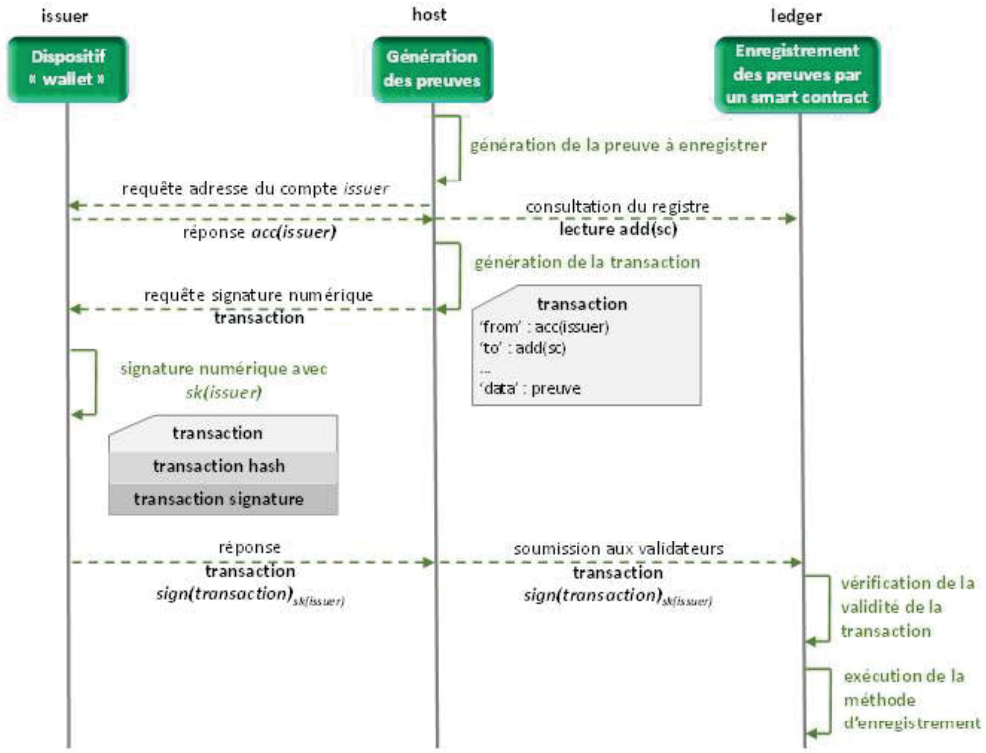
[3] M. Palatinus and P. Rusnak. Multi-account hierarchy for deterministic wallets. April 2014.

[4] M. Palatinus and P. Rusnak. Purpose field for deterministic wallets. April 2014.

[5] D. R.L.Brown. The exact security of ecdsa. 12 2001.

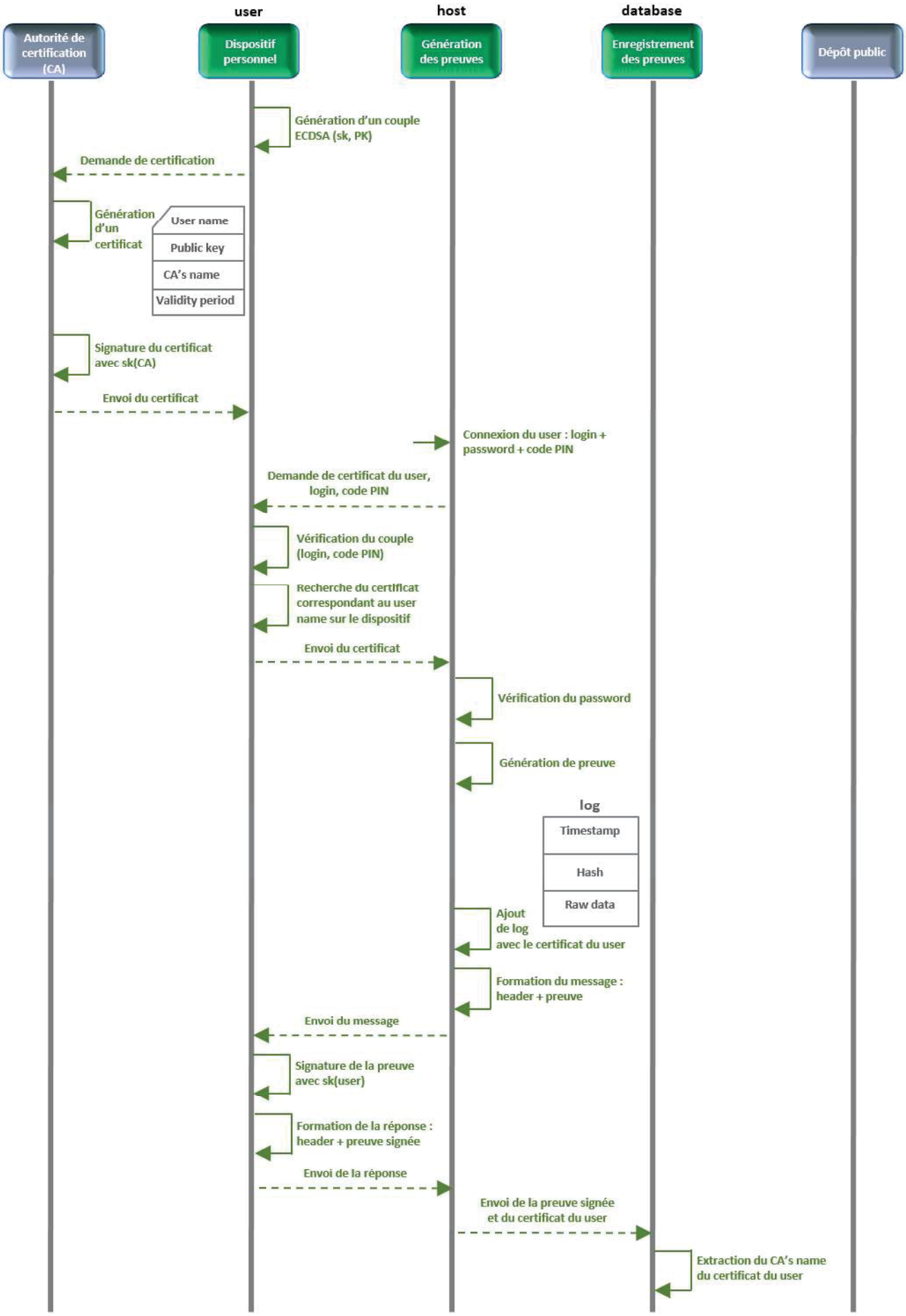
[6] P. Wuille. Hierarchical deterministic wallets. February 2012.

Appendix A

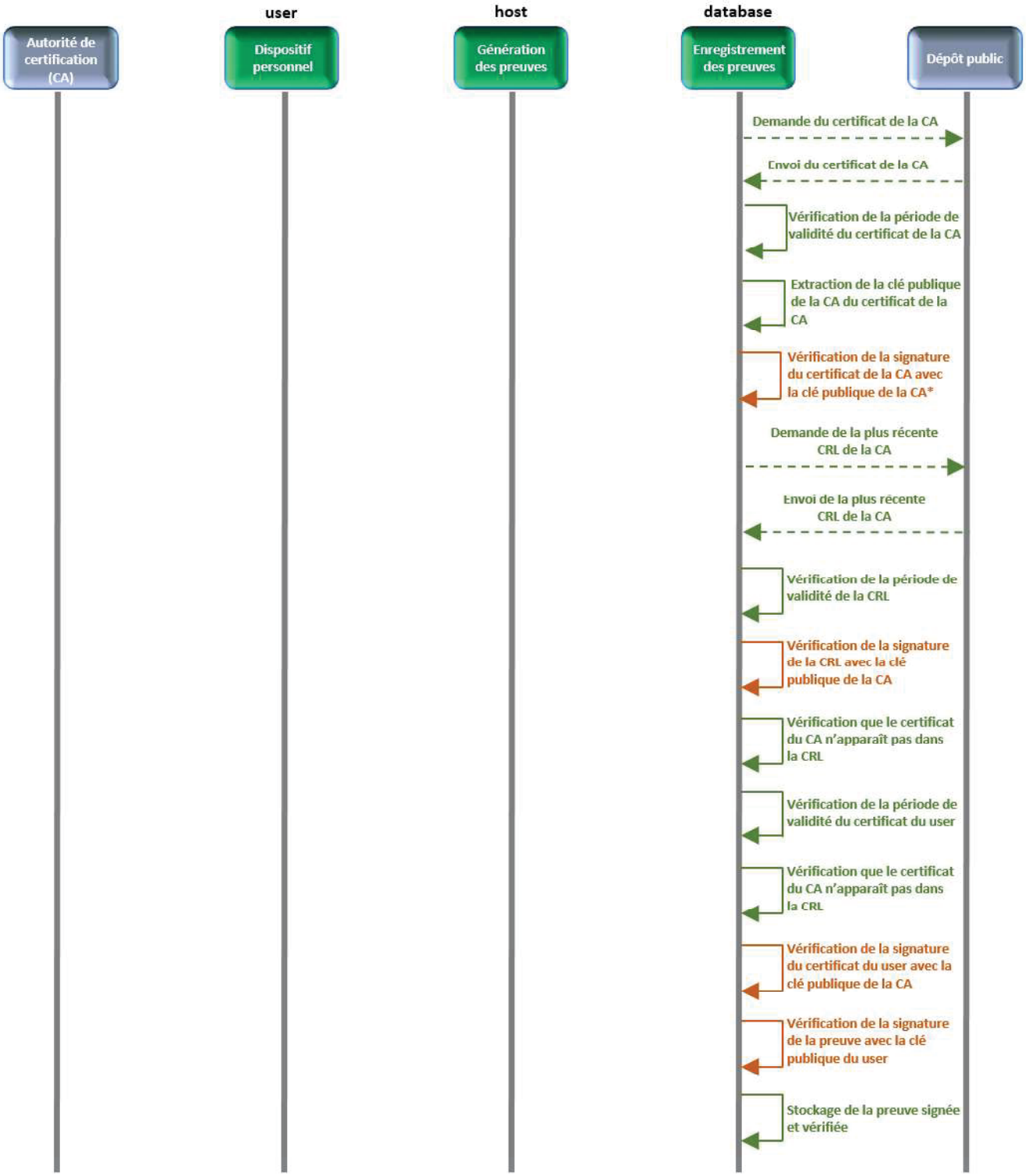


Sequence diagram of the blockchain solution

Appendix B



Sequence diagram of the solution involving digital certificates (1)



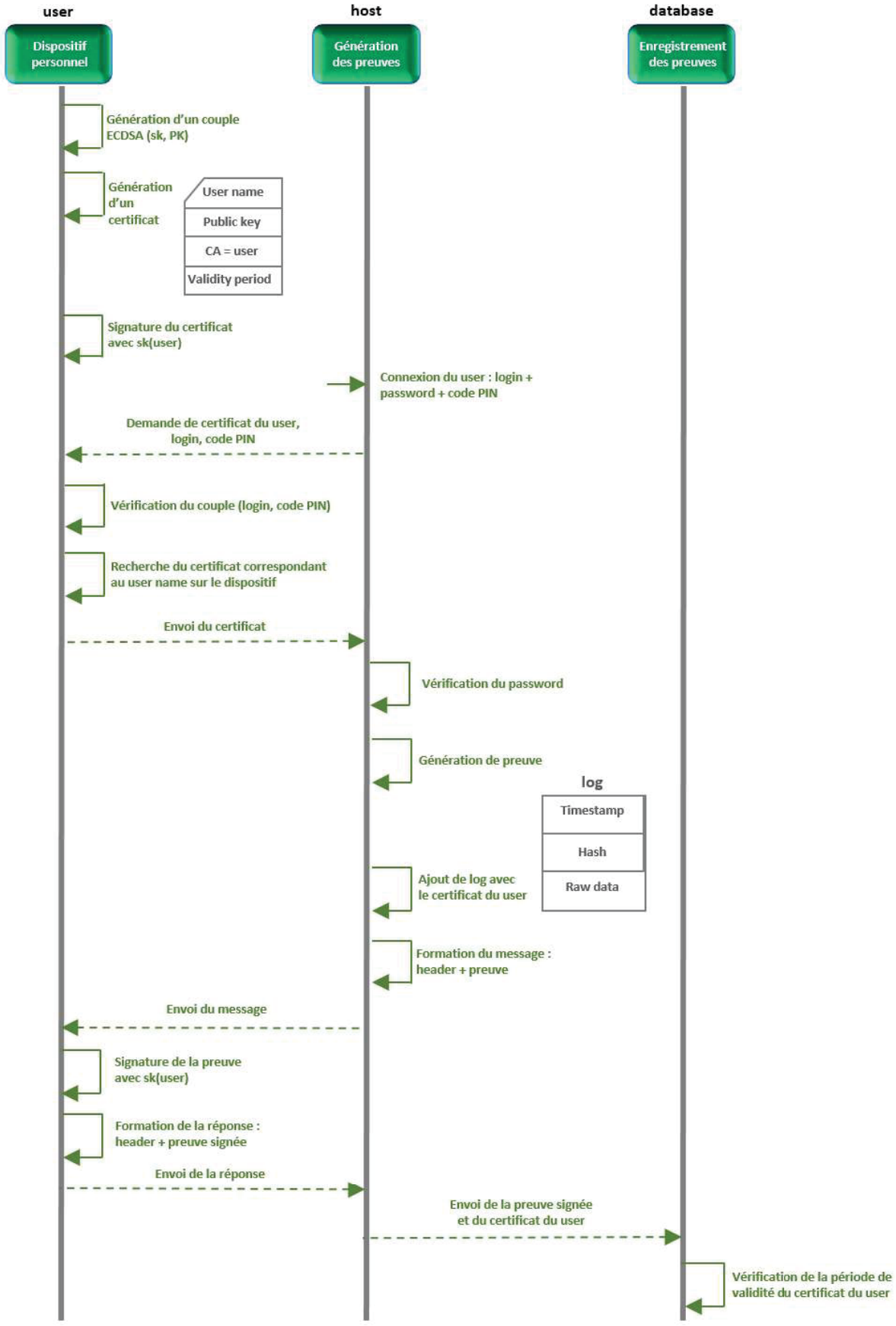
*En considérant que la CA signataire du certificat du user est autosignataire de son propre certificat (CA root).

Les **vérifications de signature** d'un document (certificat ou CRL) consistent en 3 opérations :

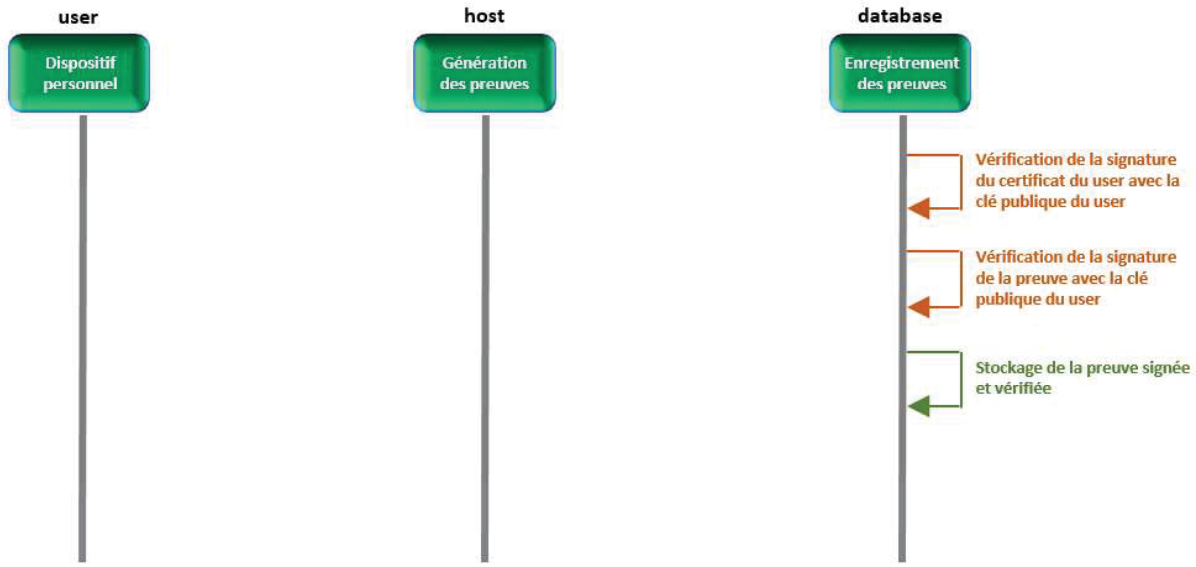
1. Calcul du haché de la partie données du document
2. Déchiffrement du haché signé présent dans le document par le biais de la clé publique de l'entité signataire.
3. Vérification de l'égalité des deux valeurs de haché obtenues.

Sequence diagram of the solution involving digital certificates (2)

Appendix C



Sequence diagram of the solution with auto-signed certificates (1)



Les **vérifications de signature** d'un document (certificat ou CRL) consistent en 3 opérations :

1. Calcul du haché de la partie données du document
2. Déchiffrement du haché signé présent dans le document par le biais de la clé publique de l'entité signataire.
3. Vérification de l'égalité des deux valeurs de haché obtenues.

Sequence diagram of the solution with auto-signed certificates (2)

Internship Report: Joint analysis in frequency and in pattern of large parallel application traces

Nils Defauw (Student), Jean-Marc Vincent (Supervisor)

Abstract

A study of a new information measure is presented along with its possible applications to large parallel application traces using regularity in pattern. The proposed measure is based on Lempel-Ziv's eigenvalue which is defined as the rate of growth of the number of different substrings found in a given string. A hypothesis over the behavior of the eigenvalue applied to strings generated uniformly at random is formulated along with ongoing work to prove that hypothesis.

1 Introduction

In High-Performance Computing, the study of application traces is crucial in order to understand the overall behavior of the system being designed. However, traces are often composed of billions of microscopic events that happened in the system during the recording of the traces, making any analysis a real struggle. As an example, we worked during this study on a trace of a parallelized mergesort algorithm implemented using OpenMP. In this trace, each OpenMP event such as the creation or destruction of a thread of execution was recorded and we could easily by zooming at the trace spot large sections of it consisting of a perfectly regular alternation of two events, such as the scheduling of a task and the termination of a task (tasks are an OpenMP concept, see the documentation for further explanations). However, due to the large number of such events in the trace, it would have been really hard to spot a break of regularity in these sections such as two consecutives terminations of OpenMP tasks without the scheduling of an other one in-between. Methods for partitioning the traces into regular sections separated by breaks of regularity would thus allow a much simpler analysis of the traces as it would quickly lead the scientist looking at the trace to the interesting parts: the regularity breaks.

Previous work [Lamarche-Perrin, 2013; Dosimont, 2015] develops this idea by providing a method for reducing huge traces composed of microscopic events to macroscopic representations by applying a two-step process to the traces. The first step of this method partitions the trace into sections recognized as regular. The second step is to represent each section of the newly-formed partition by a small macroscopic representation intended to summarize the section, a process named "aggregation". Whereas this method provided important results, and seems to be what we're looking for as the first step partitions the trace into regular sections, it is failing at capturing the regularity of our previous example with OpenMP where a perfect alternation of two microscopic events was occuring on a large section of the trace. Indeed, all the ambiguity resides in the word "regular" which doesn't have a precise meaning.

Regularity in information theory is the opposite of complexity as measured by complexity measures (also known as information measures, in the sense that a high-complexity object is difficult to compress and thus contains more information than a regular, low-complexity, easy to compress object). As the reader has probably spotted, we're talking about complexity measures at plural form, because there exists a lot of such measures. The method provided by Lamarche-Perrin and later Dosimont [Lamarche-Perrin, 2013; Dosimont, 2015] is using the homogeneity of microscopic events in a section to decide if it should be considered regular and should be a part of the partition. This method of considering homogeneous objects as regular can be seen as a form of Shannon's entropy [Shannon, 1948] which is indeed a complexity measure. We'll explain in further sections of this article the concept of entropy but note that it can't tell the difference between a sequence of uniform random coin tosses and a perfectly regular alternation of heads and tails and is thus useless at partitioning the parallel OpenMP traces we already talked about. . Note …

Hopefully, there exists one information measure that can tell apart random sequences of events and sequences made of the repetition of a pattern as this is the case with our OpenMP example, this measure is known as the Lempel-Ziv's complexity [Lempel and Ziv, 1976] and has been extensively studied and applied in compression algorithms [Ziv and Lempel, 1977; Ziv and Lempel, 1978; Welch, 1984]. As this measure is defined on strings over finite alphabets we first need to transform in a meaningful way our traces into strings, this is what is done and justified in the following section of this article. We then

focus on the properties of an alternate measure also defined by Lempel and Ziv as the "eigenvalue" that is much easier to work with and forms the foundation of the real Lempel-Ziv's complexity.

In the following of this paper, we study the properties of the normalized eigenvalue. A normalization of the eigenvalue allows to have a measure in $[0, 1]$ and thus to compare the regularity of strings of different sizes, as what was done with the normalized entropy) regarding an adaptation of Lamarche-Perrin's ideas to the partitionment of parallel application traces like the one used in our previous examples. We then prove that on strings made of the repetition of a pattern, the normalized eigenvalue drops towards 0, thus capturing the regularity of such strings and justifying our study of this measure. We highlight a behavior of the normalized eigenvalue when used to detect breaks of regularity in strings that can induce "false positives", and thus should be taken care of.

The largest final part of this article is about the study of the normalized eigenvalue on uniform random strings. We've run simulations that show that on uniform random strings, the normalized eigenvalue quickly reaches 1 but we have been unable to prove it formally so far. However, we study the behavior of this measure on de Bruijn strings [de Bruijn, 1975] which can be seen as perfect random-looking strings regarding Lempel-Ziv's eigenvalue. We adapt the results on these strings to formulate an hypothesis over the behavior of the normalized eigenvalue on real uniform random strings. We then reformulate the expectation of the normalized eigenvalue on uniform random strings as the sum (over all possible words) of the probability that those words appear in a uniform random string. nd present our unsuccessful results at computing these probabilities using Markov chains. Finally, we prove that the probability that a given word appears in a uniform random string solely depends on its autocorrelation set as defined by Guibas and Odlyzko [Guibas and Odlyzko, 1981] and as such that the number of possible different probabilities that a word of length n appears in a uniform random string is the same as the number of possible autocorrelation sets for a word of length n which is defined in The On-Line Encyclopedia of Integer Sequences as sequence A005434.

2 Representing the traces as strings over a finite alphabet

There are a lot of different and complex formats available for the representation of application traces and as such, we need a way to represent them with a unified formalization in order to be able to work with them. Many of these traces at least record the different events that happened in the system along with a timestamp for each of these events. We decided to represent them as strings over finite alphabets as strings are simple and fundamental objects in computer science for which a lot of information measures exist.

Obviously, not all traces can be represented easily as strings. For example, when working with traces on a distributed system, each event can be recorded along with the node where it occured. This kind of multi-dimensional indexing of traces has not been studied, though, as it would have deeply complicated our study and is not our primary goal of studying Lempel-Ziv's measures application to trace partitioning. As such, we focus in the remaining of this article on traces that can easily be converted into a one-dimensional string over a finite alphabet, such as time-indexed traces.

We're now facing two different cases for reducing these traces to strings over a finite alphabet. In the first case, let's suppose that the clock of the system used to provide the timestamp related to each event has a low resolution. In such a case, a lot of events happen on the same timestamp, we can then use the power set $\mathcal{P}(E)$ of the set of all possible events E as our finite alphabet and represent the traces as strings of symbols, each symbol corresponding to one timestamp being the subset of events that occured at that timestamp, with the empty set as the symbol for timestamps where nothing happened. In the second case, assuming that the clock of the system is a high-resolution clock, the probability that two different events happen with the same exact timestamp being really low, we can represent the traces as strings by taking the set of all possible events E as symbols and by considering the string of all the events that happened in the system in the order in which they happened. Remark that these two different ways of transforming the traces into strings are not equivalent as the second one doesn't take into account the amount of time elapsed between two consecutive events. The best method to use is application dependent and has not been investigated in this study.

Now that we have a convenient way of representing the traces as strings, our objective is to analyze them regarding their regularity and take apart random-looking sections of the traces and regular sections. This is the purpose of information measures as studied by information theory, which are functions associating strings with a positive number, aimed at representing the amount of information contained in the string.

3 State of the art of information measures on strings

Going back to the example given in introduction, we have a trace transformed to the string of all the events that occured during a multi-threaded merge-sort, and this string is composed of large sections where two different events A and B are alternating $\cdots ABABABABABA \cdots$. We want an information measure able to capture this kind of regularity, known as pattern-regularity because a given pattern AB is repeated along the trace. More formally, saying that an information measure f captures one kind of regularity means that on strings S regular regarding this criterion of regularity, $f(S) = 0$ whereas in order to be a real information measure, on uniform random strings R,

$f(R) = 1$.

Previous work on the partitionment of traces [Lamarche-Perrin, 2013; Dosimont, 2015] considered homogeneous sections as regular. More formally, Lamarche-Perrin and Dosimont studied traces where each microscopic event is a distribution of states and not a given event in a finite set (for example, when considering a trace of a distributed system, for each node, the distribution between the states when the node is actually computing something useful, and when it is waiting for a task to be given to it). In order to be able to measure the information contained in a section of the trace, they used the Kullback-Leibler divergence [Kullback and Leibler, 1951] between the microscopic distributions and the average distribution over the whole section. This measure can be considered as a form of Shannon's entropy [Shannon, 1948] and is unsuitable for recognizing pattern-repeating strings as regular as it doesn't take into account the ordering of the microscopic events.

Indeed, the entropy of a string $S = s_1 \cdots s_l$ of length l over a finite alphabet $\{x_i\}$ of n symbols is defined as $-\sum_{i=1}^{n} \mathbb{P}(x_i) \log_2 \mathbb{P}(x_i)$ where $\mathbb{P}(x_i)$ is estimated as the number of times the symbol x_i appears in S over l. This definition only take into account the distribution of each symbol over the string and as such can't tell the difference between a uniform random string of heads and tails and a perfect alternation of heads and tails as in the two cases, the distribution of heads and tails will be 0.5/0.5. We thus need a different information measure in order to recognize our pattern-repeating sections of traces as regular.

Another example of such measures is Kolmogorov's complexity [Kolmogorov, 1963] which measures the complexity of a string as the size of the smallest algorithm outputting the given string. As we previously said, Shannon's entropy can't capture the regularity in pattern of a string, but Kolmogorov would. Unfortunately, Kolmogorov's complexity has also been proved to be uncomputable and as such can't help us building a practical measure of information.

There still exists one measure of information that can identify patterns in strings and still is computable, this is the Lempel-Ziv's complexity [Lempel and Ziv, 1976; Ziv and Lempel, 1977; Ziv and Lempel, 1978]. This measure created by A. Lempel and J. Ziv has been studied as the foundation of several compression algorithms in use today (DEFLATE, LZW [Welch, 1984], LZMA) and as such offers a useful and deeply studied theoretical framework for studying the partitionment of traces. In this study, we focused particularly on the eigenvalue of a string $S = s_1 \cdots s_l$ at index i, $k_S(i)$, which is the number of new substrings in $s_1 \cdots s_i$ compared to $s_1 \cdots s_{i-1}$, as defined by Lempel and Ziv [Lempel and Ziv, 1976]. The reason for this choice is that on a pattern-repeating string, most of the substrings (called words) of the string should be present at multiple places so the overall number of substrings should stay low for pattern-repeating strings. As we're interested in finding the breaks of regularity in the traces, we're interested in the dynamic of this number of substrings, this is why we're looking at the number of new substrings at each symbol added to the string S. Moreover, in order to be able to compare the regularity of strings of different sizes, we choose to define a normalized version of the eigenvalue $\bar{k}_S(i) = \frac{k_S(i)}{i}$ as $k_S(i) \leq i$. This normalized eigenvalue thus belongs to $[0, 1]$.

In this article, we make the hypothesis that the eigenvalue can be used to measure the regularity of strings according to the repetition or not of patterns, we further make the hypothesis that $\bar{k}_S(i)$ has an asymptotic in $1 - \frac{\log_2 i}{i}$ and give a few insights allowing us to think this asymptotic behavior holds.

4 Eigenvalue of a string: definitions and first properties

Let $S = s_1 \cdots s_l$ denote a string of length l over a finite alphabet Σ. Let $V_S(i)$ denote the set of all substrings contained in $s_1 \cdots s_i$, $V_S(i) = \{s_j \cdots s_k \mid 1 \leq j \leq k \leq i\}$, it is the vocabulary of $s_1 \cdots s_i$, as defined previously [Lempel and Ziv, 1976]. Note that we don't include the empty word in the vocabulary in this definition. As we will look at differences between vocabularies, the choice of including it or not in this definition doesn't matter.

We're interested in the rate of growth of the vocabulary of the string at each symbol added and as such, we define the eigenvalue of S after i symbols as $k_S(i) = |V_S(i) \setminus V_S(i-1)|$. As previously proved [Lempel and Ziv, 1976], $1 \leq k_S(i) \leq i$ so we also define a normalized form of the eigenvalue $\bar{k}_S(i) = \frac{k_S(i)}{i}$, which allows our measure to be in $[0, 1]$.

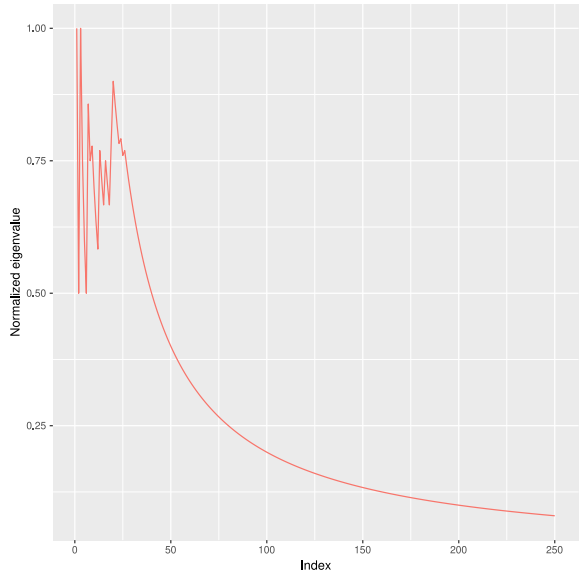
As Lempel and Ziv proved [Lempel and Ziv, 1976], the new words added to the string by the addition of the symbol i are the $k_S(i)$ longuest suffixes of $s_1 \cdots s_i$. This will allow us to prove that if a word $s_j \cdots s_i \notin V_S(i) \setminus V_S(i-1)$ then $k_S(i) < j$ and conversely that if a word $s_j \cdots s_i \in V_S(i) \setminus V_S(i-1)$ then $k_S(i) \geq j$.

Theorem 1. $V_S(i) \setminus V_S(i-1) = \{s_i \cdots s_l \mid 1 \leq i \leq k_S(i)\}$.

In all the following, we will suppose $\Sigma = \{0, 1\}$ as our study is based on binary alphabets for simplicity reasons.

4.1 First observations

A preliminary plot of the normalized eigenvalue on a pattern-repeating string of size 250, $S = p_1 \cdots p_p p_1 \cdots p_p \cdots$ with $p = 20$ seems to show a fast decrease of $\bar{k}_S(i)$ towards 0 after the first 20 symbols have been encountered.

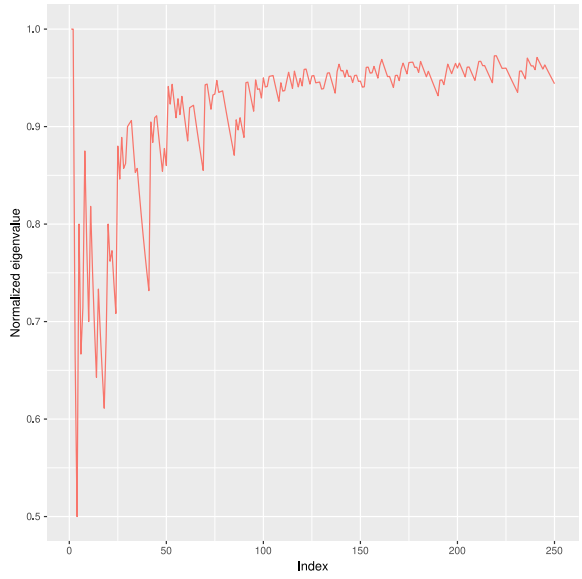


Theorem 2. *Let $S = s_1 \cdots s_l$ a string of size l and $P = p_1 \cdots p_p$ a pattern such that $\forall i, 1 \leq i \leq l$, $s_i = p_{(i \mod p)+1}$. $\forall i > p$, $\bar{k}_S(i) \leq \frac{p}{i}$.*

Proof. Let $i > p$, $s_{p+1} \cdots s_i = s_1 \cdots s_{i-p} \in V_S(i-1)$. This implies $s_{p+1} \cdots s_i \notin V_S(i) \setminus V_S(i-1)$. Using 1, $k_S(i) \leq p$, so $\bar{k}_S(i) \leq \frac{p}{i}$. □

We thus have been able to prove that on a pattern-repeating string, $\lim_{i \to \infty} \bar{k}_S(i) = 0$.

The same plot against a string of size 250 generated uniformly at random over a binary alphabet lets appear a steady increase towards 1.



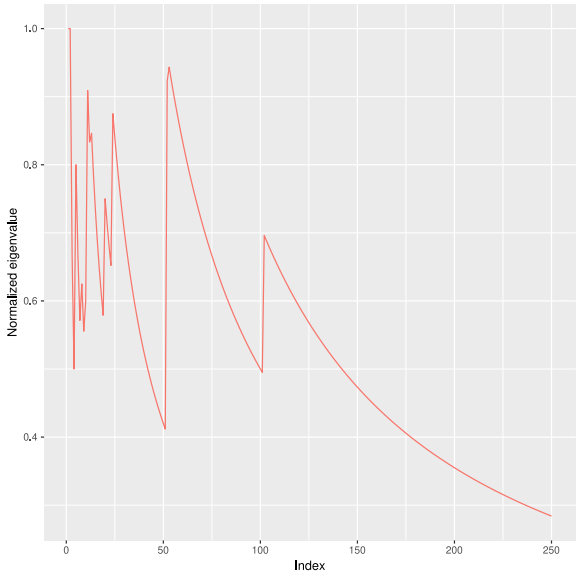
The study of this behavior being much more complicated, we let it to the second part of this article.

These two behaviors over pattern-repeating and uniform random strings are required in order for the normalized eigenvalue to be an interesting measure regarding the partitionment of the traces we're looking to achieve.

4.2 Random pattern inserted into a constant string: Rebound effect

As a preliminary study of the behavior of the normalized eigenvalue on a regular string with a little perturbation inserted in it, we studied a string of the form $p_1 \cdots p_p 0 \cdots 0 p_1 \cdots p_p 0 \cdots$.

In the following plot, $p_1 \cdots p_p$ is a uniform random pattern of size 20 inserted at the beginning and at position 50 in a constant string of 0.



We can observe two peaks appearing after index 25 (as with all pattern repeating strings, the normalized eigenvalue is noisy until the pattern has been entirely seen), the first one at 50, where the pattern in encountered for the second time and another around 100, at a position where the string is perfectly constant, we call it the "rebound effect" and have been able to explain it.

Theorem 3. *Let $P = p_1 \cdots p_p$ be a pattern of length p and considering the string $S = p_1 \cdots p_p \underbrace{0 \cdots 0}_{n \text{ times}} p_1 \cdots p_p 0 \cdots$. We thus have $k_S(2p+2n) \leq p+n$ whereas $k_S(2p+2n+1) \leq 2p+n+1$.*

Proof. $s_{p+n+1} \cdots s_{2p+2n} = p_1 \cdots p_p \underbrace{0 \cdots 0}_{n \text{ times}} \in V_S(2p+2n-1)$ so $k_S(2p+2n) \leq p+n$. $s_{2p+n+2} \cdots s_{2p+2n+1} = \underbrace{0 \cdots 0}_{n \text{ times}} \in V_S(2p+2n)$ so $k_S(2p+2n+1) \leq 2p+n+1$. □

These inequalities allow us to see the increase of p in the eigenvalue when passing from $2p+2n$ to $2p+2n+1$.

This behavior must be taken care of when trying to use the normalized eigenvalue to detect breaks in regularity in the string, indeed, the normalized eigenvalue shows peaks even at places where the string is locally perfectly regular. However, this result gives us the precise location

where these "false positives" can happen thus allowing us to suppress this effect.

4.3 Summary

Using this preliminary work, we make the hypothesis that the eigenvalue can be used as a way of measuring the information contained in a trace and would thus allow us to partition the traces based on this measure of information.

5 Study of the normalized eigenvalue on strings generated uniformly at random

As we're trying to use our measure in order to identify the information contained in traces, we need to know precisely the behavior of the eigenvalue on strings generated uniformly at random on Σ in order to compare this reference value to the measured normalized eigenvalue of a string and thus tell apart random-looking strings and regular strings.

5.1 de Bruijn strings: the most complex strings regarding the eigenvalue

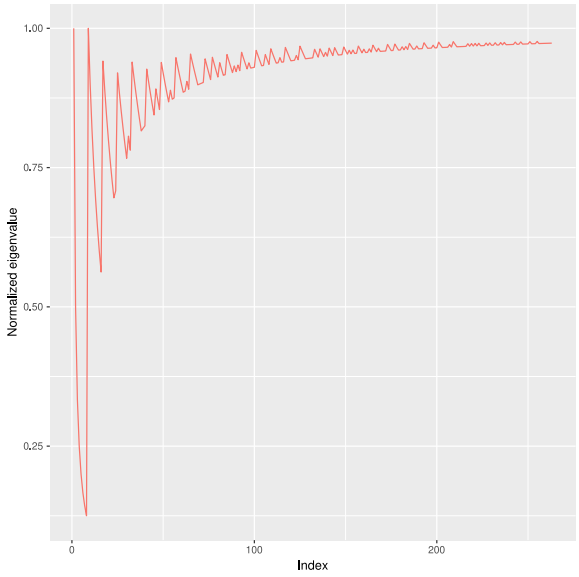
A de Bruijn string of order k over a binary alphabet Σ is a string of length $2^k + k - 1$ containing each word of Σ^k once and only once [de Bruijn, 1975]. These strings are useful to study in that they represent the most complex strings we're able to create regarding the eigenvalue, and in that we know a lower bound on their eigenvalue.

Theorem 4. *Let $B_k = b_1 \cdots b_{2^k+k-1}$ be a de Bruijn string of order k. $\forall i \geq 1, \bar{k}_{B_k}(i) \geq 1 - \frac{k-1}{i}$.*

Proof. If $i < k$: Then $i-(k-1) < 1$. As for a string $S = s_1 \cdots s_i$, $s_1 \cdots s_i \in V_S(i) \setminus V_S(i-1)$ because $s_1 \cdots s_i \notin V_S(i-1)$, then $\forall i \geq 1, k_S(i) \geq 1$, then $k_{B_k}(i) \geq 1 > i - (k-1)$, then $\bar{k}_{B_k}(i) \geq 1 - \frac{k-1}{i}$.

If $i \geq k$: $b_{i-(k-1)} \cdots b_i \notin V_{B_k}(i-1)$ by definition of a de Bruijn string of order k since $b_{i-(k-1)} \cdots b_i$ is of length k, then by 1, $k_{B_k}(i) \geq i - (k-1)$, this implies that $\bar{k}_{B_k}(i) \geq 1 - \frac{k-1}{i}$. □

We thus have a lower bound on the normalized eigenvalue of a de Bruijn string that become obvious by plotting the normalized eigenvalue over a de Bruijn string of order 8.



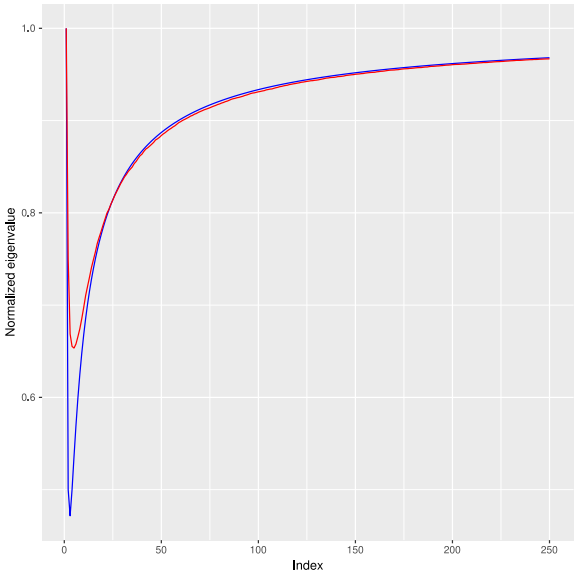
5.2 Hypothesis on the behavior of the normalized eigenvalue over uniform random strings

Using the lower bound of the normalized eigenvalue over de Bruijn strings, we can attempt to fit the expectation of the normalized eigenvalue on strings generated uniformly at random using a similar model.

Let $\Omega = \{0, 1\}$ be a sample space equipped with the discrete uniform distribution. Let $\Omega^i = \underbrace{\Omega \times \Omega \times \cdots \times \Omega}_{i \text{ times}} = \Sigma^i$ equipped with the product measure. Let $S = S_1 \cdot S_2 \cdots S_i$ be a random variable in Ω^i. We thus have $\forall j, 1 \leq j \leq i, \mathbb{P}(S_j = 0) = \mathbb{P}(S_j = 1) = \frac{1}{2}$ and $\mathbb{P}(S_1 \cdot S_2 \cdots S_i = s_1 \cdot s_2 \cdots s_i) = \frac{1}{2^i}$ for each $s_1 \cdot s_2 \cdots s_i \in \Omega^i$. Let K_i be a random variable such that $K_i(s_1 \cdots s_i) = k_{s_1 \cdots s_i}(i)$ on the sample space Ω^i.

We want to compute $\mathbb{E}[K_i]$ for each $i \geq 1$.

Below we plotted our hypothesis, which is that $\frac{\mathbb{E}[K_i]}{i} \sim_{i \to \infty} 1 - \frac{\log_2 i}{i}$. In red is represented the simulated expectation of $\frac{\mathbb{E}[K_i]}{i}$ obtained by averaging the eigenvalues of 10000 uniform random strings and in blue the function $1 - \frac{\log_2 i}{i}$.



In the following we introduce our latest work in our way to prove this hypothesis.

We will use the notation $s_1 \cdots s_l \in t_1 \cdots t_{l'}$ to mean that $\exists i, 1 \leq i \leq l' - l + 1$ such that $\forall j, 1 \leq j \leq l, s_j = t_{i+j-1}$.

Lemma 1. *$k_S(i) = |V_S(i)| - |V_S(i-1)|$.*

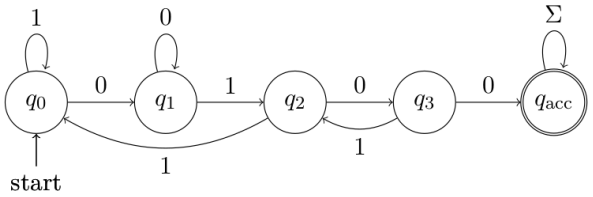
Proof. $k_S(i) = |V_S(i) \setminus V_S(i-1)|$. $V_S(i-1) \subseteq V_S(i)$. This implies that $k_S(i) = |V_S(i)| - |V_S(i-1)|$. □

This lemma tells us that instead of studying K_i we can study V_i which is a random variable over Ω^i such that $V_i(S) = |V_S(i)|$. We can further remark that $V_i(S) = \sum_{w \in \Sigma^*} \mathbf{1}_{V_S(i)}(w)$, which implies that $\mathbb{E}[V_i] = \sum_{w \in \Sigma^*} \mathbb{E}[\mathbf{1}_{V_i}(w)]$, where $\mathbf{1}_{V_i}(w)$ is itself a random variable such that $\mathbf{1}_{V_i}(w)(S) = \mathbf{1}_{V_S(i)}(w)$. Remarking that $\mathbb{E}[\mathbf{1}_{V_i}(w)] = \mathbb{P}(S \ni w)$, we can then try to compute this probability in order to compute the desired expected value.

5.3 Computing $\mathbb{P}(S \ni w)$ using Markov chains

Given a word $w = w_1 \cdots w_n$, how to compute $\mathbb{P}(S \ni w)$ with $S \in \Omega^l$ and $\mathbb{P}(S) = \frac{1}{2^l}$ (i.e. S is a binary string of length l taken uniformly at random) ? Our current approach is to model this problem by considering $S = S_1 \cdot S_2 \cdots S_l$, with S_i in $\Omega = \{0, 1\}$, and $\mathbb{P}(S_i = 0) = \mathbb{P}(S_i = 1) = \frac{1}{2}$.

Using the finite state automaton that recognizes w in a string, and relabelling the transitions 0 and 1 to $\frac{1}{2}$ each, we obtain a Markov chain. This Markov chain is composed of $n+1$ states, labelled from 0 to n, and at each step k in the process, we're in state $X_k = i$ if the i last symbols seen correspond to the prefix of w: $w_1 \cdots w_i$. The last state n is the success state, that is, $\mathbb{P}(X_{k+1} = n \mid X_k = n) = 1$. Now, $\mathbb{P}(S \ni w) = \mathbb{P}(X_l = n)$. The drawing below represents the recognizing automaton for the word 0100.



We can remark that given a word w and its dual $\overline{w}$ obtained by swapping the 0's and the 1's in w, they have both the same recognizing automaton apart from the labels of the transitions and thus have the same derived Markov chain and thus $\mathbb{P}(S \ni w) = \mathbb{P}(S \ni \overline{w})$. We can thus ask how many different ways exist of labelling a valid recognizing automaton for which the transition labels have been erased as an attempt to create classes of words for which the probability of finding them in a uniform random string is the same. Unfortunately, we prove in the following that there exists only two different ways of labelling a valid automaton. The first will recognize w and the second will recognize its dual $\overline{w}$.

In the following, we denote by $s_{i \to j}$ the label of the transition going from i to j.

Let's give the overall shape of a recognizing automaton, we will need it for the following proof. If we're in the state i, the last i symbols read correspond to $w_1 \cdots w_i$ and the state has two leaving transitions, one which is $i \to i+1$ labelled by w_{i+1} and another one in case of failure $i \to j$ with $j \leq i$ labelled $\overline{w_{i+1}}$ where $\overline{w}$ corresponds to 0 if $w = 1$ and to 1 if $w = 0$. The last state n being the success state, it has only one transition $n \to n$ labelled by Σ.

We'll need two properties of recognizing automata in order to prove the result.

Lemma 2. *$\forall i, j, j'$ with $j \neq j'$ such that there exists two transitions $i \to j$ and $i \to j'$ in the automaton, $s_{i \to j} = \overline{s_{i \to j'}}$.*

Proof. If there exists two different transitions $i \to j$ and $i \to j'$ labelled with the same symbol, then the automaton isn't deterministic. This contradicts the classical definition of recognizing automata. □

Remark that that the determinism of the automaton just gives us the fact that $s_{i \to j} \neq s_{i \to j'}$ and we're using the fact that in this study, the alphabet is binary. This is a weakness in our model that should be handled in future work.

Lemma 3. *$\forall i, i', j$ such that there exists two transitions $i \to j$ and $i' \to j$ in the automaton, $s_{i \to j} = s_{i' \to j}$.*

Proof. If $j \neq 0$: By the definition of the recognizing automaton, we are in state j if the last j symbols read are $w_1 \cdots w_j$. In particular, the last symbol read must be w_j. That means that all the transitions leading to the state j must be labelled by w_j. If $j = 0$: If $s_{i \to 0} \neq s_{i' \to 0}$, then $s_{i \to 0} = \overline{s_{i' \to 0}}$. Thus $s_{0 \to 1} = s_{i \to 0}$ or $s_{0 \to 1} = s_{i' \to 0}$. Now if $s_{0 \to 1} = s_{i \to 0}$ then after taking the transition $i \to 0$ the last symbol seen is $s_{0 \to 1}$ which is w_1 and as such, the transition should be from i to 1 and not from i to 0 and

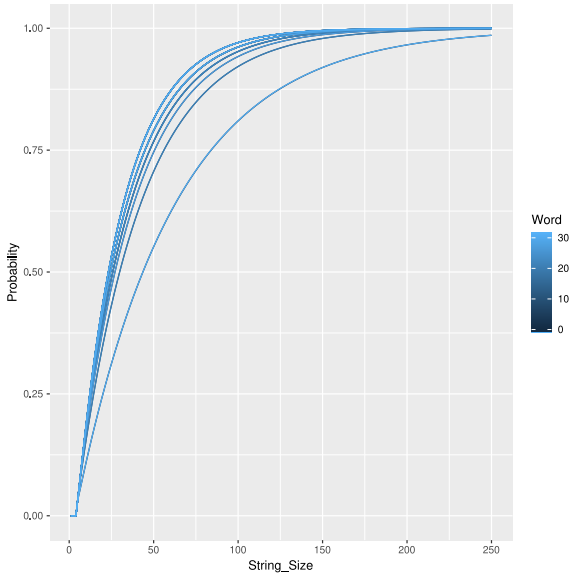
the automaton isn't valid. If $s_{0\to 1} = s_{i'\to 0}$, the same reasoning leads to the same conclusion. In all cases, the automaton isn't valid which is absurd. □

Theorem 5. *Given an unlabelled automaton for which there exists a way of labelling the transitions in order to obtain a finite state automaton recognizing a word w, there are only two ways to label it properly that would give two recognizing automata, the first one recognizing w, the second one recognizing $\overline{w}$.*

Proof. Let's choose $s_{0\to 1} \in \{0,1\}$. For $1 \leq m \leq n-1$, let P_m: $\forall i, 1 \leq i \leq m$, for all transitions $i \to j$, we know $s_{i\to j}$. Using the definition, there exists an other transition $0 \to j$ such that $j \leq 0$. Thus, $j = 0$. Using 2, $s_{0\to 0} = \overline{s_{0\to 1}}$. We thus have P_0. Suppose P_m for $m < n-1$. Let's consider the transition $m+1 \to j$ with $j \leq m+1$. If $j = 0$, then using 3, $s_{m+1\to 0} = s_{0\to 0}$. If $j \neq 0$, then there exists a transition $j-1 \to j$ and by supposition we know $s_{j-1\to j}$, using 3, we get $s_{m+1\to j} = s_{j-1\to j}$. We thus know $s_{m+1\to j}$, and using 2 we get $s_{m+1\to m+2} = \overline{s_{m+1\to j}}$. We then have P_{m+1}. Using the definition again, there exists only one transition $n \to n$ labelled by Σ and thus the whole automaton is properly labelled. We have thus proved that the labelling of the automaton depends only on $s_{0\to 1}$. As we have only two choices for $s_{0\to 1}$, we can deduce that there are only two ways to label the automaton, the first one recognizing a word w, the other one recognizing $\overline{w}$. □

As we're trying to identify the different classes of words of a given length having the same probability of appearing in a uniform random sequence, this theorem tells us that this isn't as simple as a given automaton shape for each class, as each automaton shape only recognizes two words.

We thus tried to identify the number of such classes for each word's length by simulation. In the plot below, we plotted for each of the 32 words in Σ^5 their probability of appearing in a uniform random string S against the length of S. As expected, all of these probabilities have an asymptotic in 1. Furthermore, we can observe here 6 classes of probabilities.



Using these simulation results, we tried to identify the sequence of the number of classes of probability for words in Σ^i.

Word's length	Number of classes
1	1
2	2
3	3
4	4
5	6
6	8
7	10
8	13
9	17

A quick search on the OEIS gave us the desired sequence which is known as the number of correlations of length n (Sequence A005434).

5.4 Autocorrelations of strings

The sequence A005434 of the OEIS (The On-Line Encyclopedia of Integer Sequences) of correlations of length n has been studied by Leo J. Guibas and Andrew M. Odlyzko [Guibas and Odlyzko, 1981].

In order to properly define this sequence, a few definitions are necessary. Let's consider a word $w_1 \cdots w_n$ of length n over an alphabet Σ. For $p \in \{0, \cdots, n-1\}$, we say that p is a period of $w_1 \cdots w_n$ if $\forall i \in \{1, \cdots, n-p\}, w_{i+p} = w_i$. More intuitively, a period p of $w_1 \cdots w_n$ is a non-negative integer such that, when placing a copy of $w_1 \cdots w_n$ shifted by p positions to the right above the original word, the symbols match in the overlapping part. As can be seen below as an example, 3 is a period of the word 010010.

0	1	0	0	1	0						
0	1	0	0	1	0						
			0	1	0	0	1	0			
					0	1	0	0	1	0	

We can now define the autocorrelation of a word $w_1 \cdots w_n$ as the set of its periods. Considering our

previous example 010010, its autocorrelation is the set $\{0, 3, 5\}$. It is rather obvious that 0 is always in the autocorrelation of any given word. We can also remark that if 1 is a period of $w_1 \cdots w_n$, then $\forall i \in \{1, \cdots, n-1\}, w_i = w_{i+1}$ and $w_1 \cdots w_n$ is constant and as such its autocorrelation is $\{0, \cdots, n-1\}$.

Actually, the set of possible autocorrelations for a word $w_1 \cdots w_n$ is highly constrained and Guibas and Odlyzko have been able to fully specify a minimal set of constraints a subset of $\{0, \cdots, n-1\}$ must follow in order to be a valid autocorrelation set [Guibas and Odlyzko, 1981]. Moreover, they proved that the number of possible autocorrelation sets for words of length n doesn't depend on the size of the alphabet. This property allows us to define the sequence A005434 $a(n)$ as the number of possible autocorrelation sets for words of length n.

Let's now prove why the number of classes of words $w \in \Sigma^n$ having the same $\mathbb{P}(S \ni w)$ with S taken uniformly at random in Σ^l is the same as the number of possible autocorrelation sets. We first define the autocorrelation sequence of the word w, c_i for $i \geq 0$ by $c_i = 1$ if $i \geq n$, $c_i = 1$ if $i < n$ and i is a period of w and $c_i = 0$ otherwise.

Theorem 6. *Given the autocorrelation sequence c_i of a word w, by defining the sequence $b_{l,i} = 2^{l-n} - \sum_{j=1}^{i-1} c_{i-j} \frac{b_{l,j}}{2^{\min(i-j,n)}}$, the number of strings of length l containing the word w is $\sum_{i=1}^{l-n+1} b_{l,i}$.*

Proof. For $i \leq l - n + 1$, let $S_{l,i}$ denote the set of all strings of length l that contain the word $w_1 \cdots w_n$ at position i, i.e. $\forall k, 1 \leq k \leq n$, $w_k = s_{i+k-1}$. Let $S^+_{l,i}$ denote the set of all strings of length l in which the word $w_1 \cdots w_n$ appears for the first time at position i, i.e. $S^+_{l,i} = S_{l,i} \setminus (\cup_{j=1}^{i-1} S_{l,j})$. Let's now remark that S^*_l which is the set of strings of length l containing the word $w_1 \cdots w_n$ is $\sqcup_{i=1}^{l-n+1} S^+_{l,i}$. This implies $|S^*_l| = \sum_{i=1}^{l-n+1} |S^+_{l,i}| = \sum_{i=1}^{l-n+1} b_{l,i}$ by defining $b_{l,i}$ by $|S^+_{l,i}|$.

Now let's compute $b_{l,i} = |S^+_{l,i}|$. We've seen $S^+_{l,i} = S_{l,i} \setminus (\cup_{j=1}^{i-1} S_{l,j})$, this implies $S^+_{l,i} = S_{l,i} \setminus ((\cup_{j=1}^{i-1} S_{l,j}) \cap S_{l,i})$ giving us $b_{l,i} = |S_{l,i}| - |(\cup_{j=1}^{i-1} S_{l,j}) \cap S_{l,i}|$. Now by remarking that $\cup_{j=1}^{i-1} S_{l,j} = \sqcup_{j=1}^{i-1} S^+_{l,j}$, we get $(\cup_{j=1}^{i-1} S_{l,j}) \cap S_{l,i} = (\sqcup_{j=1}^{i-1} S^+_{l,j}) \cap S_{l,i} = \sqcup_{j=1}^{i-1} (S^+_{l,j} \cap S_{l,i})$. This gives us $|(\cup_{j=1}^{i-1} S_{l,j}) \cap S_{l,i}| = \sum_{j=1}^{i-1} |S^+_{l,j} \cap S_{l,i}|$.

Let's now compute $|S^+_{l,j} \cap S_{l,i}|$ for $1 \leq j \leq i-1$. First, if $i < j + n$ and $i - j$ isn't a period of w, then this set is empty. Second, if $i \geq j + n$, then n symbols in S are constrained and $|S^+_{l,j} \cap S_{l,i}| = \frac{|S^+_{l,j}|}{2^n}$. Third, if $i < j+n$ and $i-j$ is a period of w, then only $i-j$ symbols are constrained and $|S^+_{l,j} \cap S_{l,i}| = \frac{|S^+_{l,j}|}{2^{i-j}}$. These three cases can be put together with the formula $|S^+_{l,j} \cap S_{l,i}| = c_{i-j} \frac{|S^+_{l,j}|}{2^{\min(i-j,n)}} = c_{i-j} \frac{b_{l,j}}{2^{\min(i-j,n)}}$. Lastly, $|S_{l,i}| = 2^{l-n}$, and as such, $b_{l,i} = 2^{l-n} - \sum_{j=1}^{i-1} c_{i-j} \frac{b_{l,j}}{2^{\min(i-j,n)}}$ and $|S^*_l| = \sum_{i=1}^{l-n+1} b_{l,i}$. □

This theorem gives us an explicit formula to compute the number of strings in Σ^l containing a word w and as such, if we take a string S in Σ^l uniformly at random, the probability that the word w appears in S is exactly $\frac{|S^*_l|}{2^l}$. Moreover, this formula only uses the autocorrelation sequence of the word w, thus, all words sharing the same autocorrelation sequence have the same probability to appear in a given string and as such, there is at most as many classes of probability for words as there are possible autocorrelation sets for the given words, thus establishing the link between these probability classes and the sequence A005434 of the OEIS.

6 Conclusion

Through this work, we define an information measure and study its properties on regular and uniform random strings. As required, on pattern-repeating strings, this measures drops to 0 and, as we conjectured, the measure seems to reach 1 on uniform random strings. We explain the "rebound effect" with regular strings perturbated by the insertion of a uniform random pattern at specific locations. We formulate a hypothesis on the asymptotic behavior of $\frac{\mathbb{E}[K_i]}{i}$ and give a strategy to prove it, providing a reformulation of the problem of computing $\mathbb{P}(S \ni w)$ by using finite state automata for recognition of w and Markov chains obtained by erasing the labels of these automata. We also establish a link between the probability that a word appears in a uniform random string and the autocorrelation sets defined by Guibas and Odlyzko, which appear to be a complex subject where a lot of unkowns remain.

The next steps in our study of this information measure are to end the proof of the asymptotic behavior of $\frac{\mathbb{E}[K_i]}{i}$ to be able to clearly define our measure of information by comparing $\bar{k}_S(i)$ on a given string S with $\frac{\mathbb{E}[K_i]}{i}$. Using this knowledge, we could thus experiment on ideal traces providing that we can generalize this work over bigger alphabets with $|\Sigma| \geq 3$. A better understanding of the links between the eigenvalue and the Lempel-Ziv's complexity is also desirable because that would allow us to use previous work in this well-studied field. To be able to apply this technique to real-world traces, we plan to study the behavior of this measure on noisy strings by studying the behavior of normalized eigenvalue over different models of noise and adapting it if necessary. Finally, an exploration of the different ways of transforming traces to strings, or maybe to a different object in the case of multi-dimensional traces and the ways of extending our measure on strings to these new objects would give us a real, practical technique to partition huge traces into meaningful simplified representations.

References

[de Bruijn, 1975] Nicolaas Govert de Bruijn. Acknowledgement of priority to c. flye sainte-marie on the counting of circular arrangements of 2^n zeros and ones that show each n-letter word exactly once. *Technische Hogeschool Eindhoven*, 1975.

[Dosimont, 2015] Damien Dosimont. *Agrégation spatiotemporelle pour la visualisation de traces d'exécution.* PhD thesis, École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique, jun 2015.

[Guibas and Odlyzko, 1981] Leo J. Guibas and Andrew M. Odlyzko. Periods in strings. *Journal of Combinatorial Theory*, 1981.

[Kolmogorov, 1963] A. N. Kolmogorov. On tables of random numbers. *Sankhya: The Indian Journal of Statistics*, 25, 1963.

[Kullback and Leibler, 1951] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[Lamarche-Perrin, 2013] Robin Lamarche-Perrin. *Analyse macroscopique des grands systèmes.* PhD thesis, Laboratoire d'Informatique de Grenoble, oct 2013.

[Lempel and Ziv, 1976] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, jan 1976.

[Shannon, 1948] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[Welch, 1984] Terry Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, jun 1984.

[Ziv and Lempel, 1977] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, may 1977.

[Ziv and Lempel, 1978] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, sep 1978.

Développement d'outils de traçage et d'analyse de traces pour OpenMP

Manal BENAISSA
Master 1 Informatique
Université Grenoble Alpes

Encadré par :
Vincent DANJEAN
Equipe POLARIS

1. INTRODUCTION

Les simulations en recherche scientifique, technologique et industrielle nécessitent très régulièrement de passer par des algorithmes complexes et chronophages. Le Calcul Haute Performance (HPC) a su répondre aux besoins de ces applications nécessitant de résoudre de lourds traitements. Ce large domaine fait entre autres intervenir la parallélisation des calculs, afin d'en accélérer la résolution. Lorsque la puissance séquentielle devient insuffisante, notamment en physique et en modélisation, le parallélisme offre une alternative ~~de choix~~. Il se présente sous plusieurs formes, à tous les niveaux et pour toutes granularités, aussi bien du coté matériel (avec les processeurs super-scalaires, les processeurs multi-coeurs ou les clusters), que logiciel (avec les threads, les co-routines, les tâches ou les processus communiquants). Cependant, l'implémentation de programmes usant ces méthodes peut être pénible et délicat. C'est pourquoi il existe ~~encore beaucoup d'études proposant~~ des frameworks ou des environnements facilitant la parallélisation de ces logiciels. Parmi eux, une stratégie a su gagner en popularité : la parallélisation par tâches.

2. PARALLÉLISME

L'approche la plus intuitive lors de la résolution d'un problème est la méthode dite "séquentielle" : Elle consiste à traiter les instructions une par une et repose principalement sur les principes d'une architecture SISD (Single Instruction, Single Data). ~~Malheureusement,~~ malgré les avancées technologiques et la rapidité croissante des processeurs, cette stratégie atteint ses limites en termes de temps d'exécution. Le parallélisme répond à cette limite en permettant de traiter des informations ou d'effectuer des calculs de manière simultanée. Ce procédé implique entre autre d'exploiter au mieux toutes les unités de calcul (CPU) présentes dans le système, ce qui reste un compromis à faire entre le gain de temps et la consommation d'énergie. ~~Malgré tout,~~ cette approche ~~a su~~ répondre à beaucoup de limitations temporelles, et ~~a su~~ convaincre de plus en plus de programmeurs au point d'être devenu le paradigme dominant des ordinateurs actuels. Elle pose toutefois de nombreux problèmes d'optimisation ou de synchronisation que nous verrons par la suite. Nous distinguerons notamment deux types de parallélisme : Le parallélisme à mémoire distribuée, très largement utilisée et permettant de diviser le travail à travers plusieurs machines, mais qui ne sera pas traité dans ce document, et le parallélisme à mémoire partagée, permettant de diviser le travail au sein d'une même machine.

2.1 Parallélisme à mémoire partagée

La gestion de la mémoire prend une place très particulière en parallélisation, si bien qu'elle a donnée naissance à plusieurs stratégies bien distinctes, dont une en particulier : le parallélisme à mémoire partagée. Elle fonctionne comme suit : Toutes les unités de calcul ont accès à une même zone mémoire. Elles possèdent également leur propre mémoire afin d'y stocker les informations qui ne nécessitent pas d'être partagées. Cette approche a l'avantage de réduire drastiquement le surcoût de communication au détriment d'une synchronisation plus délicate et d'un risque de conflit plus grand. Il existe plusieurs APIs gérant cette approche, dont une qui sera le sujet principal de ce document : OpenMP.

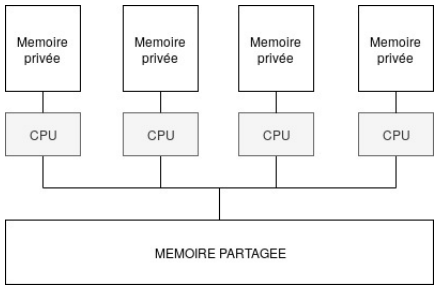


Figure 1: Parallélisme à mémoire partagée

Dans le cas de la mémoire partagée, le travail est réparti à travers plusieurs threads, qui gèrent alors une partie du code. Le thread "maître" se charge de créer les autres threads "esclaves" et de gérer l'ensemble de la section parallèle du code. Chaque thread esclave se voit attribuer un bloc de code à exécuter. Ainsi, l'ensemble des threads esclaves (ainsi que le thread maître) gère une section du code simultanément.

~~ACM ISBN 978-1-4503-2138-9.~~
~~DOI:~~ 10.1145/1235

L'attribution des blocs aux différents threads est déterminée à la compilation.

2.2 Parallélisation par tâches

Le système de tâches, largement repris en parallélisme à mémoire partagée, apporte bon nombre d'avantages dans la gestion de la charge de travail de chaque thread. Le programme est découpé en plusieurs blocs de code appelés "tâches". Le thread maître se charge de créer lesdites tâches et les place dans une file d'attente (taskpool). Chaque thread (esclave ou maître) prend une tâche dans la file et l'exécute. Dès que cette tâche est terminée, il peut reprendre une tâche dans la file, ou voler une tâche à un autre thread.

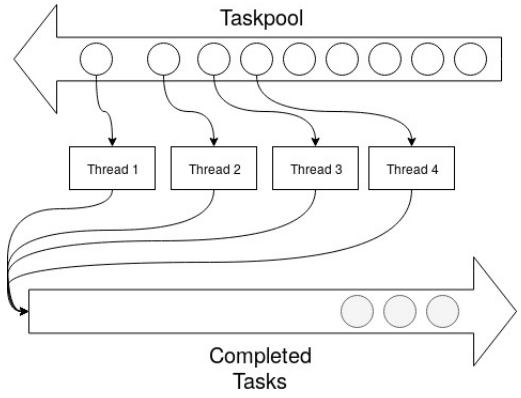


Figure 2: Gestion des tâches

Les tâches en cours d'exécution peuvent être suspendues. Le thread associé peut alors prendre une autre tâche, la terminer, puis reprendre une tâche qui a été mise en pause. Chaque thread a donc sa propre pile de tâches où il place toutes les tâches mises en attente. De même, il peut exister des dépendances entre tâches : Lorsque qu'une tâche dépend d'une autre pour être terminée, le thread peut mettre celle-ci en suspens, attendre que la tâche dont elle dépend soit terminée, puis reprendre la tâche suspendue. Pour finir, une tâche n'est pas nécessairement gérée par le même thread du début à la fin. En effet, cette tâche peut être "liée" (tied) au thread, auquel cas celui-ci gère la tâche intégralement. Sinon, elle peut être "non liée" (untied), et lorsque qu'elle a été suspendue, elle peut être reprise par un autre thread.

2.3 Implémentation et traçage de programmes parallèles

L'implémentation d'un tel système peut passer par différents modèles de programmation. Parmi eux, il en existe un qui gagne régulièrement en popularité : OpenMP.

2.3.1 OpenMP

OpenMP est une API spécialisée dans le parallélisme à mémoire partagée, destinée aux langages C/C++ et Fortran. Apparue pour la première fois en 1997, cette interface s'est développée jusqu'à sa version 5.0, sortie en novembre 2018. Elle tourne principalement autour d'un système de directives, et permet d'intégrer des sections parallèles dans un code initialement séquentiel de façon simple et intuitive. OpenMP a connu un grand tournant après sa version 3.0, grâce à l'intégration du système de tâches, en 2008. Ces nouvelles fonctionnalités se sont améliorées par la suite dans la version 4.0, permettant ainsi une plus grande stabilité dans la gestion des tâches. Les directives reprennent le modèle *Fork and Join*, et permettent de délimiter les sections parallèles tout en indiquant certains paramètres au moment de la compilation, tels que le nombre de threads à créer, la répartition du travail ou les dépendances. Dans le contexte des tâches, un thread maître se charge de créer autant de threads que nécessaire. Puis l'un d'eux se charge de créer les tâches dites *explicites*[1], et de les placer dans le taskpool. A la création des threads, ceux-ci se voient assignés d'une tâche implicite[2]. Lorsqu'un thread prend une tâche, il passe par une étape d'ordonnancement (scheduling) où la tâche implicite est suspendue puis mise dans une pile d'attente. Le thread peut alors passer à la tâche explicite précédemment choisie, puis revenir sur la tâche suspendue lorsque la tâche explicite est terminée.

```
#pragma omp parallel
{
        #pragma omp single
        {
                //TACHE 1
                #pragma omp task
                {
                        x1 = a*b;
                }
                //TACHE 2
                #pragma omp task
                {
                        x2 = c*d;
                }

                #pragma omp taskwait
                S = x1 + x2;
        }
}
return S;
```

Figure 3: Exemple de code C parallélisant l'opération S = a*b + c*d

2.3.2 OMPT

Le besoin d'effectuer une trace d'exécution se fait très souvent sentir lors de la parallélisation de code. Des outils existent déjà, notamment pour MPI, une API permettant la parallélisation sur des systèmes distribués, mais peu d'outils proposent une trace pour une parallélisation à mémoire partagée, et encore moins pour le traçage des tâches. OpenMP a également su répondre à ce besoin avec sa version 4.0, qui intègre la possibilité de tracer et de débugger les programmes usant cette API, notamment grâce à *OMPT* et *OMPD*[3]. OMPT est une interface permettant de créer soi-même son outil de traçage. Elle fonctionne sur un système de callbacks, où chaque directive envoie un évènement qui est traitée par

1. Les tâches explicites sont signalées dans le code source par la directive "pragma omp task"
2. Contrairement aux tâches explicites, les tâches implicites sont créées par le système, durant le runtime
3. OMPD permet le débug, et ne sera pas abordé dans ce rapport.

l'outil ensuite. L'outil et le programme à tracer s'exécutent dans le même processus. Celui-ci met à disposition 32 callbacks, et 19 points d'entrée pour récupérer des informations divers durant le runtime. L'implémentation de ces callbacks, qui est une part importante du traçage dans ce système, est nécessaire afin que l'outil fonctionne, et permet entre autre de personnaliser et d'adapter la trace au programme étudié. Pour finir, il suffit d'activer le mode "trace" d'OMPT en indiquant le chemin de l'outil nouvellement crée. [4]

3. PROBLÉMATIQUE

Un outil de traçage a déjà été crée dans un stage précédent [5], lorsque OpenMP n'en était qu'à sa version 4.0. OMPT était alors en cours de développement, et seule une version de test était proposée. La version 5.0 propose une version aboutie d'OMPT, avec quelques modifications. Il est donc nécessaire, dans un premier temps, de mettre à jour le précédent outil et d'y intégrer les nouvelles fonctionnalités, puis d'explorer toutes les informations qu'il est possible d'avoir par ce nouveau système. Ce rapport fera suite au précédent stage, en y apportant des analyses et des informations supplémentaires, puis en proposant des modèles de visualisation possibles de ces analyses. Un tel outil permettra par la suite de déceler un quelconque problème d'ordonnancement pouvant justifier une baisse de performance.

4. CRÉATION DE L'OUTIL DE TRAÇAGE

La conception de l'outil de traçage repose principalement sur l'implémentation des callbacks fournis par OMPT. En effet, l'interface propose un lot de fonctions qui s'exécuteront après réception de l'evènement associé. Ainsi, l'évènement *EVENT_TASK_CREATE* (lancé lorsque la directive *pragma omp task* est rencontré) déclenchera le callback *ompt_callback _task_create()*. L'ensemble des callbacks (cf Table 1) permettra de construire une trace d'exécution du programme suivi, en mettant à jour à chaque évènement un fichier log contenant les données brutes. Chaque ligne de ce fichier donnera les informations propres à l'évènement rencontré, qu'il sera possible de retraiter ensuite pour obtenir un diagramme. Il est tout à fait possible de créer plusieurs traces simultanément, en générant plusieurs fichiers logs qui permettront de créer ensuite plusieurs diagrammes différents (cf Figure 4). La conversion de ces fichiers log (au format CSV) en fichier permettant la création d'un diagramme est à la charge de l'utilisateur également.

4.1 Callbacks

L'élaboration de l'outil de trace débute avec l'implémentation de la fonction *ompt_start_tool()*. Elle permet de récupérer les callbacks et les points d'entrée voulus (notamment avec la fonction *lookup()*) et d'amorcer le traçage, avec *ompt _initialize()*. Le traçage se terminera ensuite en même temps que le programme analysé. L'outil recevra alors un évènement qui déclanchera le callback *ompt_finalize()*. L'étape suivante consiste à implémenter les autres callbacks choisis. Chacuns d'entre eux possèdent un certain nombre de

4. La variable d'environnement *OMP_TOOL_LIBRAIRIES* doit indiquer le chemin de l'outil, qui doit se trouver dans */usr/local/lib/* pour Linux.

5. Se référer aux travaux de Maxime MILLET, durant son stage en 2017

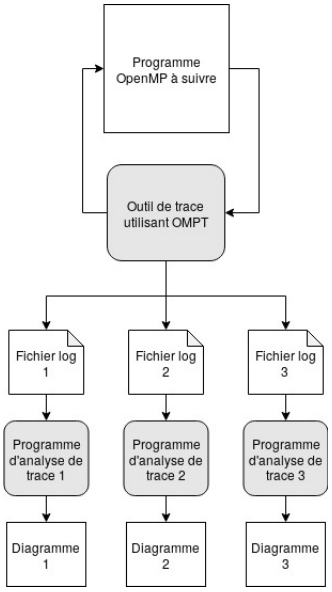


Figure 4: Étapes de construction de traces avec OMPT

paramètres : Des variables et des structures contenant un certain nombre d'informations propre à l'évènement reçu. Ces variables sont mises à jour dès la réception de l'évènement, et l'utilisateur n'a qu'à réécrire les informations de son choix dans le fichier log. Il est important de noter que certains callbacks gèrent plusieurs évènements en même temps. C'est notamment le cas de *ompt_callback_task_schedule()* par exemple. Un paramètre permettra alors de distinguer ces évènements au sein d'un même callback.

Table 1: Liste (non-exaustive) des callbacks utiles au traçage d'un programme utilisant le système de tâches

Callbacks
ompt_start_tool()
ompt_initialize()
ompt_finalize()
ompt_callback_thread_begin()
ompt_callback_thread_end()
ompt_callback_parallel_begin()
ompt_callback_parallel_end()
ompt_callback_task_create()
ompt_callback_task_schedule()
ompt_callback_implicit_task()
ompt_callback_dependences()
ompt_callback_task_dependence()
ompt_callback_work()
ompt_callback_master()

4.2 Points d'entrée

En plus des callbacks, OMPT permet de récupérer certaines informations grâce aux différents points d'entrée fournis. Là encore, ces points d'entrée doivent être récupérés dans *ompt_start_tool()* mais à la différence des callbacks, aucune implémentation n'est nécessaire. Ces fonctions permet-

Table 2: Liste (non-exaustive) des points d'entrée utiles au traçage d'un programme utilisant le système de tâches

Points d'entrée
ompt_set_callback()
ompt_get_thread_data()
ompt_get_num_places()
ompt_get_place_proc_ids()
ompt_get_proc_id()
ompt_get_state()
ompt_get_parallel_info()
ompt_get_task_info()
ompt_get_unique_id()

tront notamment de compléter les informations déjà disponibles depuis les callbacks (cf Table 2). Pour permettre la construction d'un diagramme par la suite, il est donc important que le fichier log réunisse toutes les informations nécessaires, et ce pour chaque évènement. Les informations importantes et communes à tous les évènements sont les suivantes :

- `timesamps` : Heure (en seconde) durant lequel l'évènement est survenu.
- `type` : Type de l'évènement
- `proc_id` : ID du processeur où l'évènement a été enregistré.
- `current_thread_id` : ID du thread où l'évènement a été enregistré.
- `current_task_id` : ID de la tâche actuellement gérée par le thread spécifié.
- `parent_task_id` : ID de la tâche précédemment gérée par le thread spécifié.

A cela, il est nécessaire d'ajouter d'autres informations supplémentaires spécifiques à l'évènement traité. (cf Annexe). Le manuel d'OpenMP permet de consulter les informations disponibles et les callbacks/points d'entrée permettant d'y accéder.

4.3 Enregistrement efficace des traces

Les informations capturées lors des callbacks et des points d'entrée doivent être écrites dans un fichier de trace. Pour cela, il est possible d'écrire le dit fichier grâce aux bibliothèques standards en C, notamment avec la fonction *fprintf* de *stdio.h*. Cette méthode, bien que commode, peut néanmoins être intrusive, et ralentir le traçage du programme. En effet, derrière de telles fonctions se cachent des appels systèmes coûteux qui sont appelés à chaque évènement. Le coût provient principalement de la synchronisation entre les divers threads lors de l'écriture du fichier, notamment dû à l'utilisation des verrous. Afin d'éviter ce problème, il est possible d'utiliser une bibliothèque spécialisée dans la création fichiers de trace, tel que *fxt.h*. Développé par l'équipe RUNTIME à Bordeaux, le format FXT est un format de fichier binaire, pris en charge par *fxt.h*, et particulièrement adapté à ce type de traçage. Le fichier n'est pas édité à chaque évènement : Un tampon est peu à peu rempli avec les informations capturées. Lorsque le tampon est plein, le contenu de celui-ci est écrit dans le fichier, limitant ainsi l'usage des appels systèmes. La synchronisation, directement gérée en assembleur, permet une gestion plus fluide des accès en écriture du fichier. Une fois le fichier généré, il suffit de le convertir en un fichier lisible. Le programme *fxt2csv* (déjà fournit) permet une telle conversion : Son code source peut être modifié pour personnaliser le format du fichier CSV.

5. ÉVALUATION DES PERFORMANCES DE L'OUTIL DE TRACE

Le passage au format FXT a pour but de réduire au mieux l'impact du traçage sur l'exécution du programme. Deux métriques ont été utilisés pour mesurer ce phénomène : Le temps d'exécution avec trace, et la taille des fichiers de trace obtenus. Afin de valider une telle transition, une comparaison a été effectuée entre le format classique (utilisant donc des printf) et ce nouveau format, pour un même programme de test. Dans cette expérience, trois types de programmes ont été testés : Des programmes légers, dont le temps d'exécution ne dépasse pas la dizaine de secondes, des programmes intermédiaires, et des programmes lourds dont l'exécution est volontairement longue, et dont le fichier de trace est de l'ordre de plusieurs gigaoctets.

La figure 4 montre une réelle amélioration en terme de temps d'exécution et de taille de fichier, suite au passage du format classique au format FXT. Sur des programmes de tests de taille moyenne (de l'ordre d'une à deux minutes), les résultats obtenus montrent une performance plus ou moins équivalente. La différence de taille peut être expliquée par l'ajout de méta-données dans le fichier au format FXT comme l'en-tête, qui ne compense pas la taille des informations capturées. En revanche, il est étonnant de constater qu'il existe peu de différences entre le format classique et le format FXT, en terme de temps d'exécution. L'écart est d'autant plus flagrant, et ce en faveur du format classique, lorsque la comparaison est effectuée sur de petits programmes de test. FXT semble donc offrir une réelle optimisation que lorsque les programmes à tracer sont particulièrement lourds.

6. ANALYSE DU COMPORTEMENT DES APPLICATIONS OPENMP

Le fichier de trace construit par l'outil a la forme d'une liste brute et illisible de données. La dernière étape consiste donc à traiter ces données afin d'obtenir un diagramme lisible qui mettra en valeur les informations souhaitées. Seuls les modèles de visualisation principaux et régulièrement utilisés dans un programme utilisant le système de tâches d'OpenMP ont été étudiés. Quelques uns peuvent être construits avec les données accessibles via OMPT, notamment les diagrammes de Gantt et les graphes de dépendances.

6.1 Diagramme de Gantt

Un diagramme de Gantt permet de visualiser dans le temps les diverses tâches composant le programme, et permet de représenter graphiquement l'ordonnancement de celles-ci. Un tel format est commode pour simplement visualiser dans quel ordre et par quel thread les tâches ont été exécutées. Il est donc nécessaire d'avoir au minimum les informations suivantes dans le fichier log :

- A quel moment une tâche a été crée, et par quel thread.
- A quel moment celle-ci a été prise (ou reprise), et par quel thread.
- A quel moment une tâche a été terminée ou suspendue.
- L'ID de la tâche en question.

Table 3: Exemple d'informations obtenues suite à l'évènement EVENT_TASK_CREATE

timestamp	type	proc_id	current_thread_id	parent_task_id	current_task_id	param1	param2	param3
0.000937	105	1	1	1001	1002	1003	EXPLICIT	DEPENDANT

Table 4: Performance des systèmes de traçage

Programme	Outil de trace	Temps d'exécution	Nombre de points de traces	Taille de fichier
GSL [6] (lourd)	Sans trace	254.86 sec	/	/
	Printf	4996.32 sec	3875229	46.3 Go
	FXT	1229.01 sec	3867351	42.1 Go
GSL (moyen)	Sans trace	35.25 sec	/	/
	Printf	59.87 sec	1940	6.9 ko
	FXT	59.01 sec	1479	118.1 ko
Mergesort (léger)	Sans trace	0.06 sec	/	/
	Printf	0.04 sec	128	5.17 ko
	FXT	0.07 sec	133	22.6 ko

D'autres informations peuvent être ajoutés pour enrichir le diagramme, mais elles ne sont pas forcément nécessaire pour visualiser l'ordonnancement. Ces informations, obtenables via l'outil, permettent d'obtenir les diagrammes de Gantt des Figures 5 et 6.

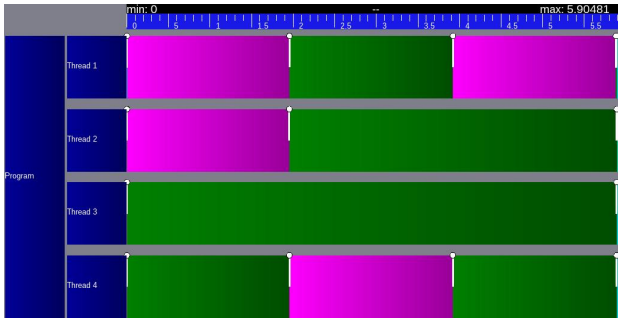


Figure 5: Diagramme de Gantt obtenu pour le programme Time.c

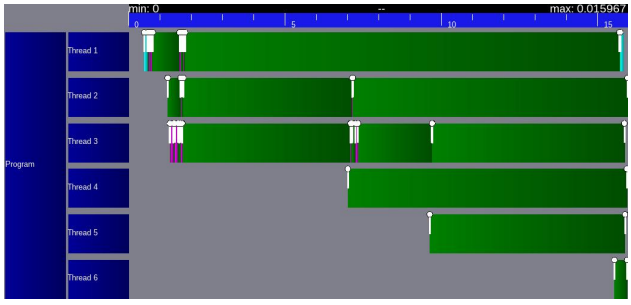


Figure 6: Diagramme de Gantt obtenu pour le programme Mergesort.c

Ces diagrammes ont été obtenus avec deux programmes : `Time.c` et `Mergesort.c`. Le premier crée de façon très basique plusieurs tâches effectuant des *sleep()*. Certaines tâches comportent également des dépendances. Le deuxième programme trie un tableau d'entier en utilisant le Mergsort. Les tâches en rose représentent des tâches explicites, celles en vert sont implicites. Il est utile de noter que certaines tâches s'exécutent trop rapidement pour être visibles. Pour finir, il est tout à fait possible de visualiser l'ordonnancement, non plus par rapport aux threads, mais par rapport aux processeurs, puisque cette information est également disponible.

Nota Bene : Le logiciel Vite, disponible sur Linux, permet de visualiser de tels diagrammes. Cela nécessite de convertir au préalable le fichier log (au format CSV) au format Pajé.

6.2 Graphe de dépendances

6.2.1 *Dépendances explicites*

Les données obtenues grâce à l'outil permettent également de construire un graphe de dépendances entre les différentes tâches. Pour cela, il est nécessaire d'avoir au minimum les informations suivantes :

- L'ID de la tâche dépendante (cf `param2`, de l'évènement de type 115 de la Table 5)
- L'ID de la tâche dont elle dépend (cf `param1`, l'évènement de type 115 de la Table 5)
- Eventuellement les valeurs/variables en jeu (cf `param3`, l'évènements de type 114 de la Table 5)

Ces données sont visibles grâce aux callbacks `ompt_callback _dependences()` et `ompt_callback_task_dependence()`. En revanche, si OMPT permet de connaitre les dépendances entre tâches, les informations liées aux variables intervenant dans les dépendances (indiqués par les évènements de type 114 de la Table 5) restent floues. En effet, pour le programme de la figure 7, il est possible d'obtenir les informations suivantes (cf Table 5) :

- Les valeurs [IN], soit les valeur entrantes (et donc attendues)
- Les valeurs [OUT], soit les valeur sortantes (et donc fournies)
- Les valeurs [INOUT], la combinaison de [IN] et [OUT]

Il est impossible, si ce n'est par déduction, et en ayant le code sous les yeux, de savoir quel tâche (dépendante) attend quelle valeur venant de quel tâche (source). Par exemple, il est difficile de savoir si la tâche 1005 attend la variable $x = 42$ de la tâche 1004 ou de la tâche 1003. En effet, la tâche 1004, qui fournit la valeur de y, ne fournit pas en revanche la valeur de x. De plus, il est impossible d'obtenir les nouvelles valeurs de x et y à chaque fin de tâche.

Table 5: Informations obtenues à propos des dépendances entre tâches

timestamp	type	proc_id	current_thread_id	parent_task_id	current_task_id	param1	param2	param3
0.000954	114	1	1	1001	1002	1003	1	[42,INOUT]
0.000984	114	1	1	1001	1002	1004	2	[42,IN][66,INOUT]
0.000990	115	1	1	1001	1002	1003	1004	
0.000998	114	1	1	1001	1002	1005	1	[42,INOUT]
0.001003	115	1	1	1001	1002	1004	1005	
0.001015	114	1	1	1001	1002	1006	2	[42,IN][66,IN]
0.001020	115	1	1	1001	1002	1005	1006	
0.001024	115	1	1	1001	1002	1004	1006	

```
int x=42,y=66;

#pragma omp parallel
{
        #pragma omp single
        {
                //TACHE 1
                #pragma omp task depend (
                    out:x)
                {
                        x=1;

                }
                //TACHE 2
                #pragma omp task depend(in:
                    x) depend(out:y)
                {
                        y=10;
                }
                //TACHE 3
                #pragma omp task depend (
                    inout:x)
                {
                        x++;
                }

                //TACHE 4
                #pragma omp task depend (in
                    :x,y)
                {

                        z=x+y;
                }
        }
}
```

Figure 7: Exemple de dépendances entre tâches

6.2.2 *Dépendances implicites*

La création d'un thread ou d'une section parallèle engendre nécessairement la création d'une tâche implicite qui lui est associé. Cette tâche, qui joue le rôle de tâche initiale (associé au thread/section parallèle en question), permet d'effectuer toutes les autres actions : créer des tâches explicites, des sous-sections parallèles etc... Cette tâche ne se termine que lorsque les actions qui lui sont associés sont terminées. Ainsi, si une tâche implicite crée une tâche explicite, celle-ci ne s'achèvera que lorsque la tâche explicite sera terminée. Ceci implique que la tâche implicite dépend de la tâche explicite.

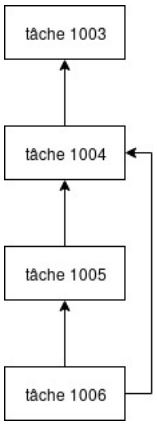


Figure 8: Graphe de dépendances obtenu avec les données de la Table 5

Il est possible d'observer les dépendances implicites grâce à un modèle de visualisation sous la forme d'un arbre, reprenant ainsi le modèle du graphe *fork and join*. La racine de l'arbre représente alors la tâche initiale. Lorsque deux tâches sont exécutées en même temps par deux threads différents, l'arbre se divise en deux branches (cf Figure 9). A la fin de ces deux tâches, l'exécution de la tâche initiale est reprise, terminant ainsi le programme. Il est important de noter que ce modèle de visualisation permet de représenter les dépendances implicites entre tâches. En effet, le nœud père dépend des nœuds fils. Dans l'exemple présenté à la figure 9, la tâche initiale ne peut pas se terminer si les tâches 2 et 3 ne sont pas terminées avant.

6.2.3 *Rayon-log*

Il existe des outils permettant de représenter en même temps le graphe de dépendance d'exécution et l'ordonnancement des tâches : c'est notamment le cas de *Rayon-logs*.[7] Cet outil, destiné à l'origine au traçage de programmes parallèles écrits en *RUST*, permet de présenter l'ordonnancement des tâches sous forme d'un arbre binaire dont les noeuds sont les tâches exécutées. Cet outil comporte néanmoins deux subtilités :

— Les tâches ne peuvent être préemptés. Elles doivent donc être coupées en sous-tâches non préemptés. Dans l'exemple de la figure 9, la tâche initiale sera considérée comme deux sous-tâches : la tâche 1 qui sera la racine

7. Voir les travaux de Frederic Wagner

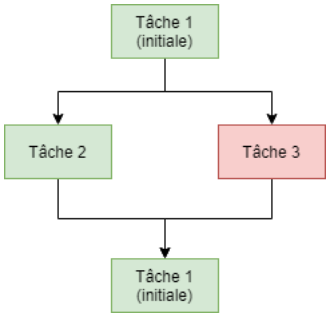


Figure 9: Graphe Fork&Join d'un programme comportant deux tâches explicites exécutées de façon parallèle par le thread vert et le thread rouge.

de l'arbre, et la tâche 4 qui terminera le programme.

— L'arbre étant toujours binaire dans le cas du Rayon-logs, il sera parfois nécessaire des créer des tâches implicites pour gérer la création de threads. Ainsi, le graphe résultant de l'exécution d'un programme à quatre threads exécutant quatre tâches explicites donnera la figure 10. [8]

Rayon-logs permet également d'ajouter divers effets pour plus de lisibilités, comme des infos-bulles à chaque tâche survolée, afin d'avoir des détails de celle-ci (temps d'exécution, ID, etc...), ou des animations permettant de visualiser le temps et l'ordre d'exécution des tâches. C'est particulièrement ce dernier point qui permettra de connaitre l'ordonnancement des tâches.

Les données requises pour construire des graphes Fork&Join sont les mêmes que celles demandées pour les diagrammes de Gantt. Néanmoins, une première étape de conversion sera nécessaire pour adapter les données à l'outil de construction de graphe, en prenant notamment en compte les deux subtilités décrites ci-dessus. En effet, l'outil attend un fichier au format JSON, récapitulant les tâches existantes (il faudra s'assurer de la non-préemption de celles-ci à ce moment là), les sous-graphes de dépendances, et éventuellement d'autres informations complémentaires sous forme de *tag*. Le développement de l'outil de conversion CSV vers JSON a été commencé au cours de ce stage mais n'a pas pu être achevé à temps.

6.3 Suivi des choix d'ordonnancement

Pour finir, si OMPT permet de suivre l'ordonnancement des tâches, il ne permet pas en revanche de prédire celui-ci. En effet, même si les expériences menés tendent à montrer que les tâches sont organisées dans le taskpool sous forme de file d'attente (Les threads prennent alors les premières tâches de la file jusqu'à que celle-ci soit vide), ni les callbacks, ni les points d'entrées ne permettent de le confirmer. Il est alors impossible de connaître exactement l'état du taskpool. En revanche, lorsqu'une tâche est suspendue, celle-ci semble être placée dans une pile réservée aux tâches en attentes. Il est possible, mais délicat de connaître l'état de la pile, grâce notamment au point d'entrée `ompt_get_task_info()`. C'est notamment grâce à celui-ci qu'il est possible de déterminer l'ID de la tâche parente et celle de la tâche courante (cf

8. La remarque à propos de préemption des tâches a été prise en compte dans ce graphe, mais sans changement de la numérotation des tâches, afin de gagner en clarté.

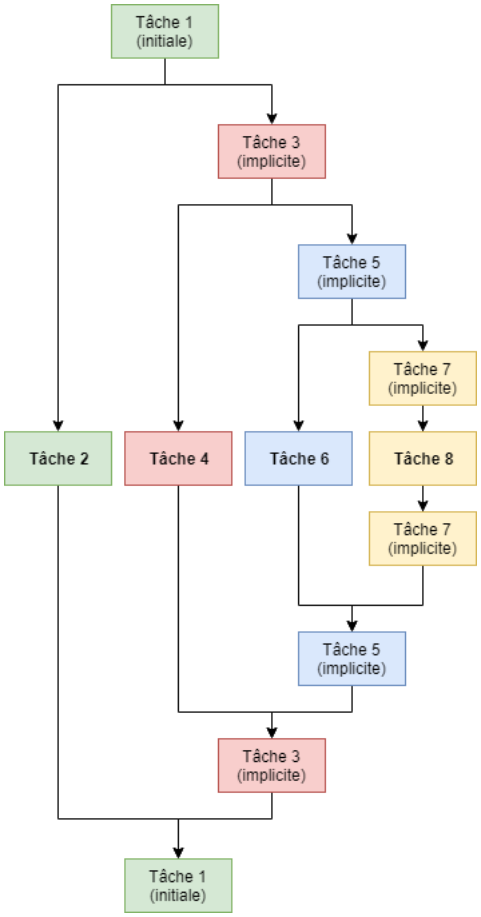


Figure 10: Graphe Fork&Join d'un programme comportant quatre tâches explicites exécutées de façon parallèle par le thread vert, rouge, bleu et jaune. Dans cette figure, seules les tâches 2, 4, 6 et 8 sont explicites.

tables 3 et 5). Il est nécessaire de connaître le *niveau* de la tâche voulue dans la pile pour accéder ses informations. La tâche courante est de niveau 0, sa tâche parente (directe) est de niveau 1. Il est donc possible de remonter jusqu'à la tâche d'origine, pour peu que nous connaissons son niveau.

Nota Bene : Pour prédire l'ordonnancement des processus/threads, d'autres outils existent. En effet, OMPT ne permet pas de tels traces pour l'instant.

7. CONCLUSION ET TRAVAUX FUTURS

OMPT est une API très riche, destinée aux programmes utilisant OpenMP, et permettant d'effectuer des traces précises. Elle est particulièrement intéressante pour les programmes utilisant le système de tâches, et présente le principal avantage de pouvoir amplement personnaliser l'outil de traçage. L'interface propose un large panel de callbacks et de points d'entrée accordant l'accès à bon nombre de données, qui pourront servir par la suite à la construction de graphes et de diagrammes. Elle offre notamment assez d'informations pour générer des diagrammes de Gantt ou des graphes de dépendances. En revanche, elle ne permet pas

de prédire les choix d'ordonnancement, ni des tâches, ni des threads ou des processus. Par ailleurs, seul les modèles de visualisation les plus populaires ont été étudiés, mais il existe d'autres modèles, dont l'étude pourrait constituer une extension de ce stage. Au cours de ces travaux, des alternatives au diagramme de Gantt ont été recherchées. Parmi elles, une a su se démarquer : les traces utilisant *Rayon-logs*. Il peut être intéressant de voir si d'autres représentations sont envisageables avec les informations qu'offre OMPT. L'équipe d'OpenMP travaillant toujours sur ce projet, il n'est pas impossible de voir une nouvelle mise à jour permettant de prédire le choix d'ordonnancement des tâches, ou d'autres fonctionnalités qui pourraient ouvrir la porte à d'autres modèles de visualisation. Le besoin de connaître la trace d'exécution des programmes parallèles est toujours très présent, et nécessite toujours plus d'études pour faciliter les utilisateurs dans cette démarche. D'autres recherches peuvent être menés pour permettre la génération de traces pour d'autres systèmes, notamment avec d'autres API, ou pour d'autres architectures.

8. REMERCIEMENTS

Tout d'abord, je souhaite adresser mes remerciements à mon maître de stage, Mr Vincent DANJEAN, sans qui je n'aurais probablement pas pu découvrir le domaine du HPC. Il a su me transmettre de façon très pédagogue les connaissances nécessaires pour comprendre mon sujet, et a su m'orienter dans mes recherches avec beaucoup de gentillesse et de prévenance.

Je tiens également à remercier Mr Frederic WAGNER, qui a pu me présenter de nouvelles méthodes de visualisation, tout en écoutant mes besoins.

Pour finir, Je souhaite remercier toutes les personnes qui ont contribué au succès de mon stage et qui m'ont aidé lors de la rédaction de ce rapport, ainsi que POLARIS, pour m'avoir admis dans leur équipe et pour m'avoir permis de faire ce stage.

Références

[1] OpenMP Architecture Review Board. OpenMP Technical Report : Version 5.0. Complete Specifications, November 2018.

[2] Maxime MILLET. *Internship report 2017*. (French) [*TER : Développement d'outils de traçage et d'analyse de traces pour OpenMP*].

[3] Edouard MARGUERITE. *Internship report 2017*. (French) [*Développement d'algorithmes parallèles pour laphysique numérique du QGP*].

[4] Frederic WAGNER, *Rayon-logs*.
`http://www-id.imag.fr/Laboratoire/Membres/Wagner_Frederic/rayon-logs.html`

[5] OpenMP : API for parallel programming,
`https://www.openmp.org`

[6] FXT, *Fast User/Kernel Tracing*.
`https://savannah.nongnu.org/projects/fkt`
Article associé :
`http://moais.imag.fr/membres/vincent.danjean/publis/DanWac05TSI.pdf`

APPENDIX

A. TABLEAU DES ÉVÈNEMENTS

Evenement	Code	Callback associé	Informations données
EVENT_OMPT_START	101	ompt_initialize()	- Temps - Code de l'évènement
EVENT_OMPT_END	102	ompt_finalize()	- Temps - Code de l'évènement
EVENT_THREAD_BEGIN	103	on_ompt_callback_thread_begin()	- Temps - Code de l'évènement - ID du processus courant - ID du thread crée - ID de la tâche parente - ID de la tâche courante - Type du thread crée (Initial, worker...)
EVENT_THREAD_END	104	on_ompt_callback_thread_end()	- Temps - Code de l'évènement - ID du processus courant - ID du thread supprimé - ID de la tâche parente - ID de la tâche courante
EVENT_TASK_CREATE	105	on_ompt_callback_task_create()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la nouvelle tâche - Type de la nouvelle tâche (explicite ou initial) - Présence (ou non) de dépendances
EVENT_TASK_END	106	on_ompt_callback_task_schedule()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche terminée
EVENT_IMPLICIT_TASK_CREATE	107	on_ompt_callback_implicit_task()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la nouvelle tâche implicite - ID de la section parallèle associée - Type de la tâche (implicite) - Nombre de threads alloués à la section parallèle associée - ID du thread appelant (index)
EVENT_IMPLICIT_TASK_END	108	on_ompt_callback_implicit_task()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche implicite terminée - ID de la section parallèle associée - Type de la tâche (implicite) - Nombre de threads alloués à la section parallèle associée - ID du thread appelant (index)
⋮	⋮	⋮	⋮

Evenement	Code	Callback associé	Informations données
⋮	⋮	⋮	⋮
EVENT_TASK_SCHEDULE	109	on_ompt_callback_task_schedule()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche source - ID de la tâche destination
EVENT_PARALLEL_BEGIN	110	on_ompt_callback_parallel_begin()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche associée à la section parallèle - ID de la nouvelle section parallèle - Type de la section parallèle - Nombre de threads alloués à la section parallèle
EVENT_PARALLEL_END	111	on_ompt_callback_parallel_end()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la section parallèle terminée
EVENT_SINGLE_OTHER_BEGIN	112	on_ompt_callback_work()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche associée au thread - ID de la section parallèle associée
EVENT_SINGLE_OTHER_END	113	on_ompt_callback_work()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche associée au thread - ID de la section parallèle associée
EVENT_TASK_DEPENDENCES	114	on_ompt_callback_dependences()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - Nombre de dépendances - Liste des dépendances
EVENT_TASK_DEPENDENCES_PAIR	115	on_ompt_callback_task _dependence()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche source - ID de la tâche dépendante de la source
⋮	⋮	⋮	⋮

Evenement	Code	Callback associé	Informations données
⋮	⋮	⋮	⋮
EVENT_SINGLE_EXECUTOR_BEGIN	116	on_ompt_callback_work()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche associée au thread - ID de la section parallèle associée
EVENT_SINGLE_EXECUTOR_END	117	on_ompt_callback_work()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche associée au thread - ID de la section parallèle associée
EVENT_MASTER_BEGIN	118	on_ompt_callback_master()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche associée au thread - ID de la section parallèle associée
EVENT_MASTER_END	119	on_ompt_callback_master()	- Temps - Code de l'évènement - ID du processus courant - ID du thread courant - ID de la tâche parente - ID de la tâche courante - ID de la tâche associée au thread - ID de la section parallèle associée

EXPLAINATION OF REVIEW NOTATIONS

[comments]

typos or incorrect english or improper formulation

not clearly explained or undefined or not enough precision

Good point that discards a previous remark