Magistère Informatique 2016

JEUDI 1er SEPTEMBRE

9h Steve ROQUES
Optimisation des accès mémoire spécifiques à une application par prechargement de page dans les manycores.

9h40 Thomas LAVOCAT
Gotaktuk, a versatile parallel launcher

10h25 Myriam CLOUET
Raffinement de propriétés de logique temporelle

11h05 Lenaïc TERRIER
A language for the home - Domain-Specific Language design for End-User Programming in Smart Homes

14h Mathieu BAILLE
Méthodologie d'analyse de corpus de flux RSS multimédia

14h40 Jules LEFRERE
Composition in Coq of silents distributed self-stabilizing algorithms.

15h20 Rodolphe BERTOLINI
Distributed Approach of Cross-Layer Allocator in Wireless Sensor Networks

16h15 Claude Gobet
3D kidney motion characterization from 2D+T MR Images

VENDREDI 2 SEPTEMBRE

8h30 Thomas BAUMELA
Integrating Split Drivers in Linux

9h10 Rémy BOUTONNET
Relational Procedure Summaries for Interprocedural Analysis

10h Quentin RICARD
Détection d'intrusion en milieu industriel

10h40 Christopher FERREIRA
Scat : Function prototypes retrieval inside stripped binaries

11h30 Lina MARSSO
Formal proof for an activation model of real-time system

14h Antoine DELISE
Is factorization helpful for current sat-solvers ?

Le Magistère est une option pour les élèves de niveau L3 à M2 leur permettant d'acquérir de l'expérience dans le domaine de la recherche.

Les soutenances, organisées sous forme de conférence, sont ouvertes au public.

Rendez-vous au grand amphithéatre de l'IMAG

UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Contents

Prefetching memory pages in manycores

Roques Steve Supervised by: Frédéric Pétrot
steve.roques@imag.fr, frederic.petrot@imag.fr

Abstract

Computer systems contain an increasing number of processors. Unfortunately, increasing the computation capability is not sufficient to obtain the expected speed-up, because then memory accesses become a bottleneck. To limit this issue, most multi/manycore architectures are clustered and embed local memory, but then what, when and how the content of these local memories is managed becomes a crucial problem called *prefetching*.

In this article, we perform a study of the state of the art around memory pages prefetching and propose a simple model to evaluate the performances of such systems. This ultimately leads us to conclude with a description of the future works to achieve.

1 Introduction

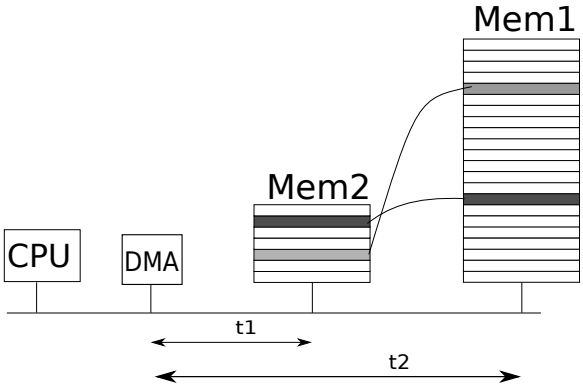
The memory hierarchy in modern embedded massively multi-processor CPUs is usually structured in the form of L1 caches (coherent or not) close to the processor, local memory (LM) dedicated to a small set of processors (8 or 16, called a cluster or a tile) which can be regarded as a L2, but whose content is the responsibility of the software, and an external global memory (GM) can be seen as a L3.

The local memories are usually small, for technological integration reasons, and have good performances (latency and throughput). But this implies that any access to the LM that fails requires access to the GM, which causes an important slowdown.

Some operating systems attempt to prefecth pages on the fly, based on previous page-fault occurrences for example. This approach is general but makes assumptions, such as that the desired pages are on disk, and thus suffer of extremely high latencies.

Let us take the example of figure 1 in which one can see one CPU, one direct memory access engine (DMA) and 2 memories. The time t1, corresponding to time to load between Mem2 and DMA is shorter than time t2, corresponding to time to load between Mem1 and DMA. It is better to use the Mem2 more often and therefore repatriate Mem1 data to Mem2. So the execution time is reduced. To do this we will see several methods.

Figure 1: Prefetch between 2 memories



The objective of this paper is to examine the state of the art in prefetching *memory pages* (typically 4 KB) to determine if the knowledge of the application can be exploited to define an offline approach able to limit the number of page faults. We start by doing a quick tour of different works already realised about prefetching memory pages. Then the evaluating system that we used for experimentation aiming at comparing these approaches will be explained. Finally, we present future works and conclusion.

2 State of the art

2.1 Comparison of several algorithms

Dini et al. [1] compare many caching and prefetching algorithms in order to explain why prefetching improves efficiency. They begin by explaining the two basic algorithms they use for comparison: Least recently used (LRU) and Most recently used (MRU). LRU is defined with this rule: every page replacement evicts the buffered page whose previous reference is farthest in the past. MRU is defined with this rule: every page replacement evicts the buffered page whose

previous reference is nearest in the past. These two algorithms probably represent the program response times upper and lower bounds in the absence of page prefetch.

Krueger et al. [2] trace the memory accesses produced by the *successive over relaxation* (SOR) target application. They simulate SOR with LRU page replacement policy and they show that LRU is inadequate under the given circumstances because it generates one page fault by iteration of the loop. However the objective of prefetch is to reduce access time not only because the data is closer, but also by reducing the number of page faults too. Krueger et al. [2] show that the replacement algorithm for page depends directly on the target application.

Unlike those algorithms, Dini et al. [1] presents two prefetch algorithms EP (early prefetch) and LP (late prefetch), that characterise different degrees of secondary memory system (SMS) activity.

EP include several phases, one for early prefetch when SMS becomes idle and until the fair replacement rule was satisfied. Fair replacement rule is defined as every page replacement evicts the buffered page whose next reference is farthest in the future, provided this page has been referenced since its most recent load from secondary memory into the buffer. Finally, the next reference is every page fetch loads the non-buffered page whose next reference will be in the future.

LP is defined by the next reference and fair replacement rules and a third rule. This is late prefetch, when the SMS becomes idle and a page is missing in the buffer, the fetch of this page is done at the earliest time making it possible to complete fetch before occurrence of the next reference to this page.

Dini et al. [1] concluded that the time necessary for the processor to control the SMS has a significant impact on programs response times. And the results of their experimental measurements made in local SMS configuration indicate that these holes, those created by allocation of buffer, are responsible for significant increases in program response time.

2.2 Compiler and Prefetching

There are also proposals which aim to include the prefetch phase directly in compilation. Thus one can gather the list of information required to perform the prefetch. There is even the version that automate this task and thus relieves the programmer of this task. Mowry et al. [3] propose and evaluate a fully automatic technique. They use a hierarchically memory cluster, named Hurricane, which is a micro-kernel based operating system that is mostly POSIX compliant. They compare many applications on this system, which represent a variety of different scientific workloads. The compiler analyses future access patterns to predict when page faults are likely to occur and when data is no longer needed. Then, operating system manage I/O to accelerate performance and minimising prefetch. Caragea et al. [4] aim at evaluating many algorithms with compiler prefetching for manycores. Manycores, or massively multi-core, describe multi-core architectures with an high number of cores. They evaluate prefetching algorithms on the explicit multi-threading (XMT) framework. The goal of the XMT is improving single-task performance through parallelism. Furthermore, they present Resource-Aware Prefetching (RAP) algorithm, an improvement over Mowry's algorithm.

Mowry's looping prefetch is a three-phase algorithm. One for determining dynamic accesses which are likely to suffer cache misses and therefore should be prefetched. A second for isolating the predicted dynamic miss instances using loop-splitting techniques such as peeling, unrolling, and strip-mining. Finally, a scheduler that prefetches the appropriate page the proper amount of time in advance using software pipelining[4].

RAP is an improved Mowry's algorithm lowering 2 iteration. Thereby, it limits resource requirements and uses them to hide as much latency as possible. Experiments with this algorithm show that it has up to 40% better performances than Mowry's algorithm.

2.3 Shared Memory Systems

Paudel et al. [5] and Speight and Burtscher [6] present how prefetching improves performance in shared memory. They showed that employing optimised coherence protocols for targeted patterns of shared-variable accesses improves application performance. Paudel et al. [5] have used different access patterns to test and optimise these access :

- Read-mostly : variables are initialised once and subsequently only read by many threads.
- Producer-consumer : variables are updated by one producer and read by many consumers.
- Accumulator : variables are updated from values generated at each node.
- Migratory : variables are read and modified one at a time by different processors.
- Stencil pattern : each variable in a multidimensional grid is updated with weighted contributions from a subset of its neighbouring variables.
- General read-write : variables are read from and written to by multiple threads.

However, when the prediction window grows, so does the numbers of page fault. This is due to the fact that the algorithms tested are not very resilient to page fault. However, the results show that the prefetch on shared memory increase performances, it is sufficient just to check that the target program uses the good memory address at the right time and reduce errors.

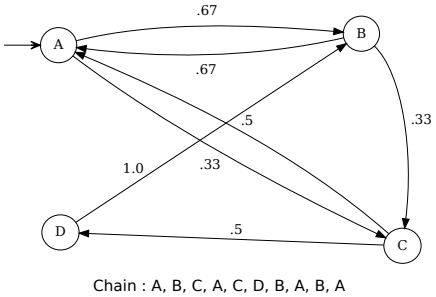
2.4 Dependence Graph Precomputation

Annavaram et al. [7] use a different approach. They use a dependence graph precomputation for prefetching memory page. This graph represents a scheduling of precomputations in the target program. Thus, the target program is precomputed as if it was compiled. The advantage of this method is that it can associate the prefetched memory page at the node graph and so define what page will be fetched after. Annavaram et al. [7] use for their simulation the CPU2000 integer benchmark suite. It is composed of the following program set: gcc, crafty, parser, gap... This set forms a coverage of target program for testing. The precomputation is necessary when parts of the target application is repeated over time. Indeed the fact of precomputing allows memoization, and returns it directly to the next request. However precomputation is not useful if the operation does not perform repeatedly. It would be advantageous to associate precomputation and prefetching in order to increase the performance.

2.5 Prefetching by Markov predictors

In this section we explain why Markov chains are interesting for prefetching. Joseph and Grunwald [8] use the miss address stream as a prediction source. They use this stream to build the Markov chain and so achieve a prediction automaton. Pathlak et al. [9] shown that Markov history table size of 32 is sufficient for cache prefetching. They modify memory hierarchy along with Markov prefetch engine and prefetch buffer. Firstly quick explanation on how the Markov chain works. In figure 2 one can see an automaton that represents the Markov chain for the string "A,B,C,A,C,D,B,A,B,A". The chain is built by the following way: If A is followed by B, we create an edge of 1 weight between A and B. Then once all the stream is traversed, we normalize the weight of edges.

Figure 2: Markov chain example



The Joseph and Grunwald [8]'s predictor works this way. Each page miss is referenced in the predictor and pushed in the queue of miss address stream. Then the Markov chain is updated with this new reference. They realize the Markov prefetcher in hardware with several registers to predict future address miss and they use these registers to request a prefetch in cache. This method works using only two thirds of the memory of a demand-fetch cache organisation. This is expensive, but this method is still advantageous and can be reworked by software.

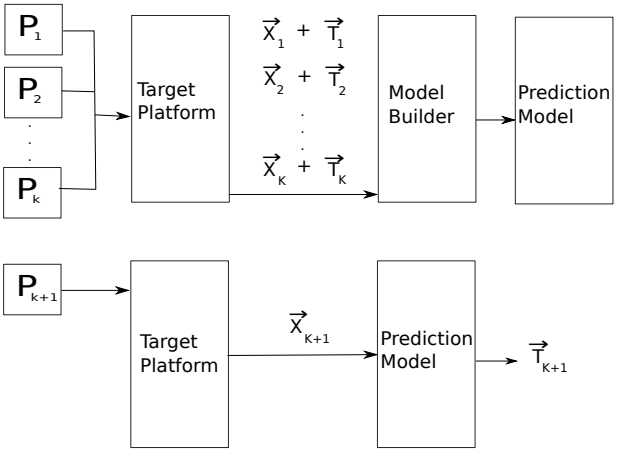
2.6 Speculative data prefetching for branching structures

Carpov et al. [10] examine two different strategies for prefetching in dataflow programs, a fractional strategy and an all-or-nothing strategy. Their analyse is specific to n-output branching structure. The fractional strategy consist on if is allowed to prefetch only fractions of branch data, or all-or-nothing otherwise. Carpov et al. [10] try to find the optimal data prefetching strategies to minimize objective functions as the mathematical expectation and the worst-case of the execution time. In both cases, problems are resume by a linear programming problem and Dantzig algorithm[11] (or simplex algorithm) can solve this problem in an polynomial time. Carpov et al. [10] prove the branch prefetching order is an optimal one.

2.7 Machine Learning based prefetch

Liao et al. [12] presents a prefetch optimization based on machine learning with several algorithm of machine learning and a benchmark data center application. They apply machine learning to predict the best configuration using data from hardware performance counters.

Figure 3: Workflow of framework construction and evaluation



In figure 3 we can see a workflow of framework construction then a workflow of framework evaluation. Firstly we construct the prediction model from programs P_1 to P_k with their entries $\overrightarrow{X}$ and their execution times $\overrightarrow{T}$. Then we use the model built to assess the program P_{k+1} with input $\overrightarrow{X}_{k+1}$. Thus we obtain an execution time $\overrightarrow{T}_{k+1}$ based on the knowledge that was acquired from previous programs. Liao et al. [12] created their prefetch algorithms from this machine learning approach.

Liao et al. [12] created four hardware prefetchers based on four machine learning algorithms.

- Data Prefetch Logic : fetches streams of instructions and data from memory to the unified L2 cache.
- Adjacent Cache Line : fetches the matching cache line in a cache line pair to the L2 cache.

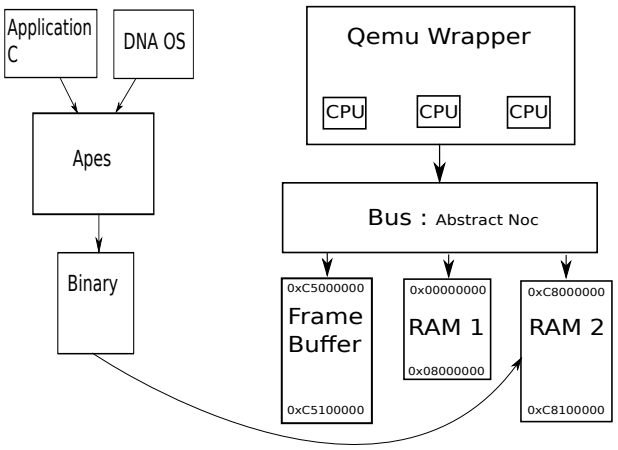
- Data Cache Unit : fetches the next cache line into the L1 data cache when it detects multiple loads from the same cache line.
- Instruction Pointer-based : keeps sequential load history information per instruction pointer and uses this data to decide whether to fetch data into the L1 cache.

By combining these algorithms and multiplying the test, they get an improvement in execution time between 1% to 75% according to target program and platform used. This can serve us to create the preftech models we wish to achieve with Markov chains. So by combining the stochastic approach of Markov chain and systematic approach to machine learning should lead to interesting results for our technical prefetch.

3 Experimentation environment

In this section we describe the environment used for simulation. It is based on the RABBITS platform, that uses QEMU to simulate processors and SystemC (a standard for modeling hardware components) to simulate the other hardware parts of the system. QEMU is a generic and open source machine emulator and virtualizer. This enables us to virtualize our system and evaluate different prefetching strategies. We test several applications: matMult (matrix multiplication), ParallelMjpeg (muti-threaded Mjpeg decoder) and Pyramid (image processing algorithm). These applications make multiple memory accesses and we will recover traces of execution that allow us to determine when and by what algorithm should we prefetch these pages. The applications run on top of DNA/OS, a simple operating system that allows to schedule threads and provides accesses to the libc functionalities, among which memory allocation and terminal output. The binary generated by the cross-development toolchain will be executed on the CPU of QEMU. The Communication Bus is realized by an abstract network on chip (NoC) as we can see in figure 4. This NoC allows to abstract communication between the CPU in QEMU and the memories modeled in SystemC.

Figure 4: Rabbits architecture and compilation chain with apes compiler

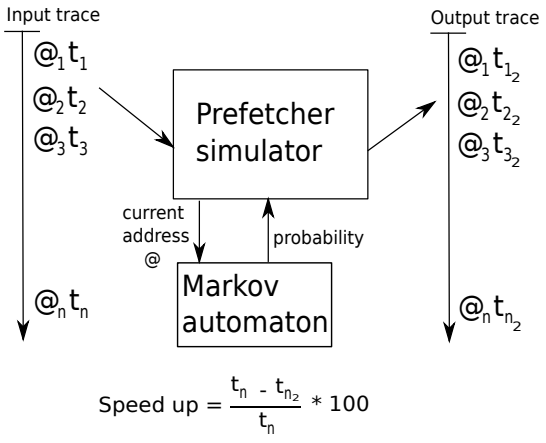


The trace of executions we generate with this system allows us to know when the page is entered and at what time. This allows us to chain the memory access and fill define at what time the pages should be prefetched so that execution are conducted more quickly. Indeed if the page has directly from the cache the CPU will not raise a page fault and therefore will directly access a resource.

4 The prefetcher simulator

The prefetcher we have implemented has two alternative algorithms based on Markov chains like Joseph and Grunwald [8] and Pathlak et al. [9] did. Markov chains were used for learning the sequence of page accesses and selecting the future page which has the highest probability. The advantage of using such an approach is that developers are not solicited to perform the prefetches themselves. The system "learns" by itself the most probable accesses to pages on a given execution path and predicts which pages should be repatriated according to the latency characteristics of the architecture and the execution of the target application. The execution traces recovered contain a list of page addresses accesses and the time at which the accesses have been made. By browsing the list, a Markov automaton is built. From the current page, this automaton gives the probabilities to access any of its successor pages.

Figure 5: Simulator



We developed a simple prefetch simulator which reads in a trace of execution with the addresses of accessed pages and their corresponding access times. Architectural information such as the latencies of memories accesses, their sizes, and the time it takes to copy a page from and to the large and small memories are also required. Then the simulator built the Markov chain by browsing the entry trace and it provided another trace that represents simulation with prefetch. The construction of the Markov chain is the same as that of described

Figure 6:

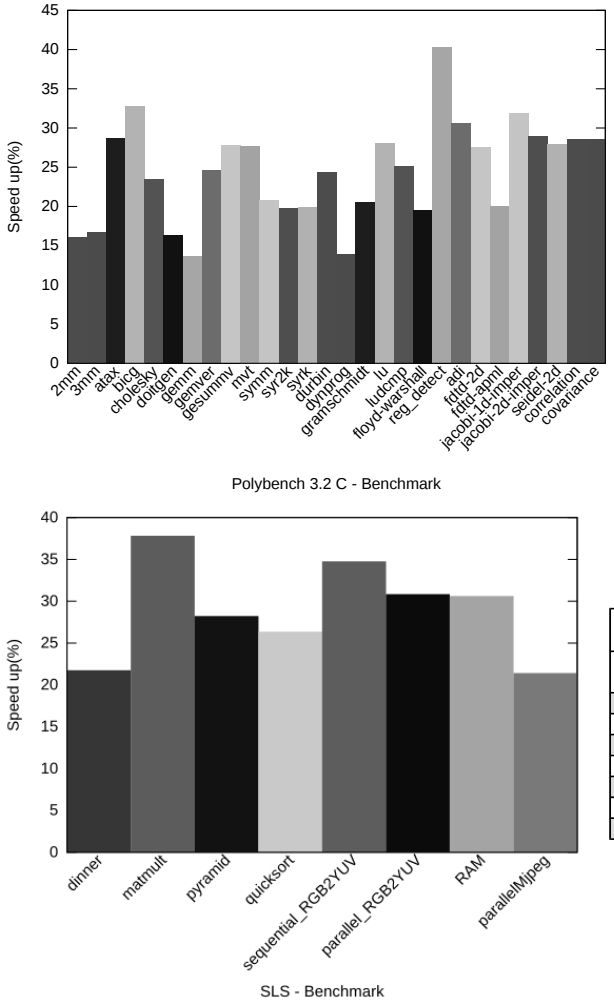


Figure 7:

Polybench (speed up in %)

Application	DATA	Linear Algebra		Medley		Stencils		Data mining	
Latencies L1 - L2(ns)		Split	Full	Split	Full	Split	Full	Split	Full
100 - 100	MINI	-24.32	-24.32	-25.27	-25.27	-27.37	-27.37	-37.85	-37.85
10 - 100	MINI	14.69	14.71	13.75	13.75	16.20	16.20	22.58	22.58
10 - 200	MINI	17.83	17.85	17.11	17.11	22.42	22.43	28.62	28.62
10 - 500	MINI	20.02	20.04	19.88	19.88	26.02	25.86	32.34	32.34
10 - 1000	MINI	20.88	20.91	20.97	20.97	27.29	27.15	33.77	33.77
10 - 2000	MINI	21.34	21.36	21.50	21.50	28.03	27.88	34.38	34.38
10 - 5000	MINI	21.60	21.63	21.77	21.77	28.42	28.26	34.73	34.73
100 - 100	SMALL	-29.47	-29.47	-38.93	-38.93	-32.68	-32.68	-55.42	-55.42
10 - 100	SMALL	16.88	15.77	25.57	25.87	18.98	18.76	31.93	21.99
10 - 200	SMALL	21.51	20.57	29.43	29.72	28.00	27.76	38.63	28.59
10 - 500	SMALL	24.98	23.92	32.12	32.41	32.10	31.26	42.70	32.67
10 - 1000	SMALL	26.92	25.72	33.18	33.29	33.93	32.28	44.11	34.32
10 - 2000	SMALL	27.96	27.33	33.58	33.95	32.42	32.84	42.40	35.58
10 - 5000	SMALL	29.52	28.65	34.99	35.43	37.05	33.12	47.28	37.92
10 - 100	STANDARD	19.04	17.75	27.05	24.89	21.13	20.83	25.56	27.47
10 - 200	STANDARD	19.69	21.07	26.57	27.90	27.18	27.26	27.29	29.01
10 - 500	STANDARD	21.44	22.96	27.57	28.90	28.17	28.19	31.34	31.94

SLS (speed up in %)

Application	Dinner		Pyramid		Quicksort		Parallel RGB_2_YUV		Parallel Mjpeg	
Latencies L1 - L2(ns)	Split	Full	Split	Full	Split	Full	Split	Full	Split	Full
100 - 100	-44.44	-44.44	-35.70	-35.70	-32.95	-32.95	-39.05	-39.05	-29.87	-29.87
10 - 100	14.36	13.99	22.27	22.34	22.87	23.05	26.77	27.09	18.03	17.55
10 - 200	22.14	21.74	27.90	28.03	26.11	26.31	30.55	30.86	22.18	21.42
10 - 500	27.56	26.49	31.38	31.56	28.16	28.35	33.70	33.95	24.90	24.34
10 - 1000	32.10	30.53	32.96	33.09	28.90	29.07	34.51	34.73	25.93	25.33
10 - 2000	30.81	29.47	33.76	33.84	29.37	29.47	34.93	35.13	26.47	26.24
10 - 5000	39.63	39.84	34.26	34.41	29.65	29.75	35.33	36.27	26.81	27.55

in section 2.5. Each page is chained to former then normalises edges to obtain the probabilities for each of these pages from previous. At last, the simulator verifies if the probability returned by the automaton is greater than the threshold of acceptability defined by the developer and charging or not the page in the buffer representing the local memory.

The simulator also uses an algorithm to simulate the ousting of the local memory that contains the pages that have been loaded. This algorithm is a simple LRU, it may be replaced by a more efficient algorithm based on the architectures and application testing. Generally the simulator is scalable and several heuristic to test anymore can be added. The algorithms described above, as EP and LP [1] or Mowry's looping prefetch algorithm [3], will be used as heuristic in the simulator.

5 Results

We test the prefetch for two benchmark. The first, named SLS, is composed of linear programming and parallel program developed by SLS team at TIMA laboratory. The second is the Polybench benchmark developed by Pouchet and Yuki [13]. Polybench 3.2 is composed of different categories:

- Linear Algebra
- Medley
- Stencils
- Datamining

We also tested different data size provided by this benchmark (mini, small and standard). This verifies that the prefetch is effective even with longer executions. Finally latencies of the two memories accesses are used as limit. If both latencies are identical then the prefetch has no interest because it just used the distant memory directly. But when the gap between memory accesses is large enough then the prefetch helps to enhance performance. The experiment shows that the prefetch is useful and helps reducing target programs execution time.

It is observed in figures 6 and 7 that when the difference between memory accesses increases the speed up obtained grows also. The executions of longer duration, thus with larger data, obtain better results than the smaller ones. Indeed the gain is proportional to the length of the execution of the program. So if the execution time is long enough, the gain between the prefetcher version and the normal version will be great. These results remain observable with both benchmarks for both sequential and parallel programs. So the fact of sharing data among several given thread does not seem to affect the prefetch data.

What these results do not show, as the simulator does not model it, is the effect of the memory required to hold the prefetcher information, i.e. the Markov chain. Indeed, traversing many pages may lead to a large automaton is no provision is taken. If this information, which must be quickly accessible, fills this memory, then the system will spend much time to flush it and allocate for application data. This can reduce the prefetcher gain or even slow down the execution of the program. This will require before realising the prefetch to conduct an analysis of the architecture of the system taking into account the size of memories, their latency and size that the prefetcher information will need. To avoid this risk, a simple solution is to bound the amount of memory and accept not fully accurate information.

5.1 Split versions

The simulator implements two strategies. A first one, called "full" that contains all the transitions since the beginning of the execution, and a second one, called "split", which uses a threshold representing the number of transitions, and thus considers only the n last transitions. This latter method allows to define how much information is retained by the prefetcher. Thus the pages that were used at the beginning or from a certain point on will be forgotten because it may not be useful in the current context of the application. In this experimentation we defined a limit of 10% of total transition because we know the number of pages used in the input trace. This represents between 1000 and 10000 transition memorised by the prefetcher. The split version destroys the automaton each time it reaches the threshold. We could also implement a version that uses a circular buffer of transitions that will keep a local context throughout the execution and therefore avoid the loss of local information.

However this limit is to be defined in terms of architecture because it generates an huge accuracy lost with respect to the full version. Indeed, whenever the prefetcher destroys the automaton and restart from the current pages, it loses time which impacts the total execution time of the application. And so if the prefetcher has emptied his memory too often the error will accumulate. This error can be positive or negative that is to say what can be increase the gain by the prefetcher or reduce it, so it will assess whether the loss is not too large compared to the target application used.

6 Future work

We achieved a prefetch simulator that takes an execution trace and estimates the execution time as if a page prefetcher was used. We observe that this is profitable and that program implementation is faster and easier, as this method has no impact on the programmer. Indeed, it can be implemented directly into operating system. To do so will require pages fault and caches miss information form the CPU to build the Markov chains. Then the rest of prefetcher will be the same as that of the simulator. With Markov automaton it predicts pages to preload. This is technically challenging, as it requires the use of a Direct Memory Access for the copy operation, and the correct management of the Translation lookaside buffer and the page tables for each process of each CPU concerned by the accessed data. Note that if the cost of doing the copy using the DMA has been taken into account in our model, the cost of the changes in the page tables and the cache misses they will induce is not. Once this implementation realised, a comparative study between the solutions proposed by this article and reality can be done, thus checking whether the simulator is correct and if not, explaining the potential gap between the expected solution by the simulator and the reality on the system studied.

We can also add several heuristic based on what has been presented in the state of the art. As the precomputation[7], algorithms EP and LP[1]. Or machine learning that could accelerate the creation of the Markov chain and refine the model for the target application. Finally, we can look learning algorithms to achieve a more accurate prefetcher. We could be using the Markov automaton as the first learning function and refined the model over execution.

7 Conclusion

To conclude, we have seen several method to prefetch pages with their advantage and disadvantage. We provide a tool to simulate the prefetch from a trace of execution. This tool allows to set up a prefetch strategy quickly, avoiding the pain to developed this strategy directly in the operating system. It remains tracks to explored, such as prefetch by learning. Finally, the prefetch allows to increase performance of a system by reducing the access time of data. There also remain several studies to conduct concerning the result of variations in the simulated version and a version implemented on a system or to studying the minimum size required for prefetcher with Markov automaton to obtain satisfactory results without taking too much memory.

Acknowledgement

I wish to thank Frédéric Pétrot for providing this internship and helping me along it, Afif Temani for his unvaluable help in setting up the environment used for the experiments and the System Level Synthesis (SLS) team members for their help in my research and experiences. Finally I would like to thank the TIMA lab for welcoming me during this work.

References

[1] Gianluca Dini, Giuseppe Lettieri, and Lanfranco Lopriore. Caching and prefetching algorithms for programs with looping reference patterns. *The Computer Journal*, 2005.

[2] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. *Proceeding OOPSLA '93 Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications Pages 48-64*, 1993.

[3] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. *Operating Systems Design and Implementation (OSDI '96)*, 1996.

[4] George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-aware compiler prefetching for many-cores. *Parallel and Distributed Computing (ISPDC)*, 2010.

[5] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. Optimizing shared data accesses in distributed-memory x10 systems. *High Performance Computing (HiPC), 21st International Conference on Computer architecture*, 2014.

[6] Evan Speight and Martin Burtscher. Delphi: Prediction-based page prefetching to improve the performance of shared virtual memory systems. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2002.

[7] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. *ISCA '01 Proceedings of the 28th annual international symposium on Computer architecture - Pages 52-61*, 2001.

[8] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *ACM SIGARCH Computer Architecture News - Special Issue: Proceedings of the 24th annual international symposium on Computer architecture (ISCA '97) Volume 25 Issue 2, May 1997 Pages 252-263*, 1997.

[9] Pranav Pathlak, Mehedi Sarwar, and Sohum Sohoni. Markov prediction scheme for cache prefetching. *Proceedings of 2nd Annual Conference on Theoretical and Applied Computer Science, Stillwater, OK*, 2010.

[10] Sergiu Carpov, Renaud Sirdey, Jacques Carlier, and Dritan Nace. Speculative data prefetching for branching structures in dataflow programms. *Electronic Notes in Discrete Mathematics*, 2010.

[11] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Knapsack problems. *Springer, Berlin, Germany*, 2004.

[12] Shih-Wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine learning-based prefetch optimization for data center applications. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[13] Louis-Noël Pouchet and T Yuki. Polybench/c 3.2, 2012.

Gotaktuk a versatile parallel launcher

Thomas Lavocat tlavocat@april.org
Supervised by: Olivier Richard

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

Abstract

In the High Performance computers (HPC) field, computers are attached to each other to create clusters that are made of thousand of computational nodes and continue to grow bigger. As computers needs maintenance, administration task, and eventually to execute some jobs, there is a need to follow the growing cluster's complexity with more efficient tools than yesterday. This report focus on the building of *Gotaktuk* a versatile tool that deploys commands and schedule tasks with adaptation to the network configuration.

1 Introduction

In the field of High Performance Computing the next generations of platform will become bigger and Grids will involve more and more nodes [6]. Actual tools are still usable for thousand of cores, but what about millions of them ? There is a need for a quicker way to continue the daily administration and exploitation tasks. Administration tasks often consist of broadcasting commands over a range of nodes. Exploitation tasks often consist to process a set of jobs with the higher efficiency.

As definitions, executing a command on a computational node is called a remote executions deployment and is made of some steps : The first and most time consuming part is to connect the nodes, the second is to execute the final command and fetch the output [1] and the final step is to shutdown each node. If the program need to be copied on the remote, this task is called the propagation and it is an additional cost to the deployment.

Gotaktuk is also a refactor of *Taktuk* a versatile tool that is able to handle thousands of computers. Those two will be compared in this report. Finally, this tool has been built using *Grid5000* [10] a large-scale and versatile testbed for experiment-driven research in all areas of computer science.

This report focuses on the ideas behind the development of *Gotaktuk* and is organized as follows, section 3 will go over the developed solution of *Gotaktuk*, section 4 presents the experiments and to finish, section 5 presents the related work.

2 Problematic

As said above, there is a need to have a faster versatile launcher, with some needed qualities to be usable on heterogeneous grids. First of all, the launcher need to propagate itself over the nodes to avoid an additional administration task for installation and maintenance, to obtain this quality, the program should not have too much dependencies to external libraries and must run out of the box on standard exploitation systems and architectures. Second, as platforms of computational nodes are heterogeneous and involve a lot of computers, the program need to be fail resilient and at least be able to terminate in every situation and not let any phantom program running on nodes that will impact performance. Third, the program need to be efficient in term of CPU usage and minimize its impact in order to gain efficiency while scheduling tasks. Fourth, to keep maintainability, the program need to be write with high level libraries to minimize the code volume and minimize bug sources.

3 Solution

3.1 Chosen technologies

To follow the need of maintainability and speed, two main choices have been made. *Golang* as language and *Zmq* as networking library.

Golang a fast and compiled language *Taktuk* was written in *Perl*. A Script language like *Perl* is a good solution for several reason. For self deployment purpose, because of the program's weight and for portability, because of native presence of the *Perl* interpreter on computers. From one side, interpreted languages are slow compared to a compiled one, but as *Taktuk* is most IO intensive than computational intensive this cost can be neglected [1]. This is not the case of *Gotaktuk*, because this tool is also able to schedule a lot of task and this functionality is computational intensive.

From the other side, *Perl* suffer from usabilities issues, like dynamic typing, or its trickyness making more difficult to maintain a well done program for a non *Perl* expert. A rebuild of *Taktuk* should consider a more easy to use language

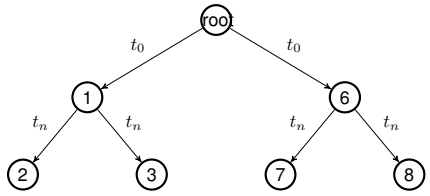


Figure 1: Deployment sequence shown on a binary tree. Times t_0 and t_n correspond to the time needed to launch a node in the deployment tree. Those two times are equivalent, the imposent only mean the floor of the tree. This time is the one that can not be reduce or overlapped, and to be optimized, a node should use this "dead" time to launch as many children has possible.

to be maintained. *Golang* has been chosen for several reason, it has a native high performance concurrency system [8] usefull for scheduling and allow writing simpler and more reliable code [7] and can be linked with external libraries.

Deploying *Gotaktuk* on a node means sending the executable on the remote and run it. As said, *Golang* is a compiled language, and the *Golang* compiler produces an *ELF* format on GNU/Linux systems. For the moment, we do not take care of the grid heterogeneity, this a engineering concern. Luckily on *Grid5000* we can ensure that our test benches will run on *Debian* with the deployment facilities provided by *Kadeploy* [10].

A quick networking library A reliable and high performance communication layer is needed to link nodes together. *ZeroMq* is a high-performance reliable library for asynchronous communications [9] that is not in an experimental state and used on production's programs. It provide out of the box easy to use communication patterns.

Unfortunately, *Zmq* is a *C* library, it needs to be present on the node, to keep the propagation facilities a workaround consists to statically link the library inside the executable. This has not been made for now.

The *Gotaktuk*'s engine is described in this section, explaining how it work and how it answer the problematic.

3.2 Deployment and auto propagation

Gotaktuk reduces the complexity of the all execution by splitting its work through a communication tree called "deploying tree". The width can be fixed or variable depending on the "work stealing" activation, we will define this mechanism later. Each node has for visibility its direct neighbours, x children and 1 parent, except for the root node that has no parent and leaf nodes that have no children.

The first action made by a node is to initialize and launch its children. This operation is called a "launching sequence" and is considered done only when the launched node has send its first message to its parent.

A node cannot start working until it is properly launched. On the diagram presented on figure 1 this time is represented by t_0. This constant time cannot be overlapped and is equals to the needed time to open a `ssh` connexion.

When the remote computer a node try to launch has no *Gotaktuk* installed, the parent node will install *Gotaktuk* ont it, this is an auto propagation sequence and its made over `scp`. In the auto propagation case, a copy time must be added to t_0 and is equals to the required time to upload *Gotaktuk* on the remote computer.

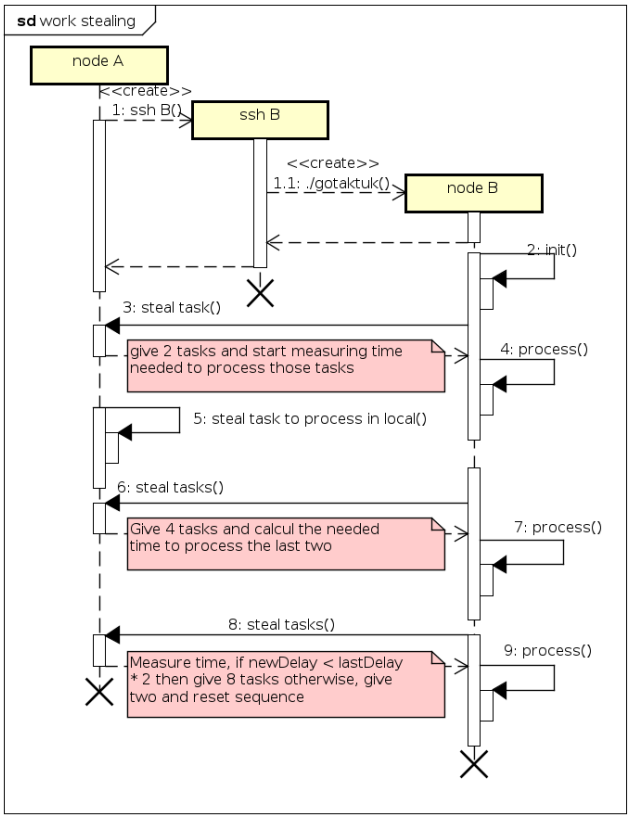


Figure 2: Workstealing between a node and its parent, the request to new jobs always come from the child, between two request the parent measure the time spent executing the requests. If the time is lower than twice the last one, then the parent give twice more commands to execute, otherwise the parent will restart the number of given tasks to two and start over the time computations.

To optimize the deployment sequence, a node should continue launching children until the first one is ready, this would mean that the t_0 time would have been spent entirely doing something useful and not just waiting.

It may be a good idea to limit the nary tree, because a node may launch a lot of children before the first one start to respond and it may affect the IO performance of the parent.

3.3 Work stealing

Gotaktuk is inspired of *Taktuk*'s workstealing behavior. And use this well known algorithm to bring adaptivity over the heterogeneous grid[4] [2].

The work stealing is used two times in *Gotaktuk*. One to create the deploying tree, one to steal jobs when *Gotaktuk* is in schedule mode. The key idea behind work stealing is to let the slow nodes less active than the quicker ones.

The algorithm behind is quite simple and shown on figure 2. When receiving a task request a node will give 2 task

and save the demand's timestamp. On the second time a demand is received, it will give 4 tasks, mesure the timestamp again and save the difference as 'time needed to accomplish two tasks'. After that, on the next ask if the new delay was lower to twice last one, then the parent will give $2^{(lastPower+1)}$ otherwise, as TCP's most basic slow start, it will start from the beginning.

A node has for the two types of tasks that can be stolen, a pool of task that can be stoled and memorize for a son which tasks has been stolen, in order to get them back in case of failure.

This heuristic has some drawbacks, it only detect when a node is slowing down but because there is no comparison between nodes it's not possible to discriminate a slow one. A way to do this would be to keep a reference time to process a task, but this may only work when the task are homogeneous in term of execution time.

For now this basic heuristic has given some good results that we will see later.

3.4 Asking tasks

A node is responsible of its work, this mean that, a node need to ask tasks to do to its parents. In the first version of Gotaktuk a parent had no way to contact its children, only to answer them. In the last version, the functionality has been added but only in broadcast mode for synchronisation purposes. This is useful for recursive task asking. We said above that a node has only a local visibility of its neighbours and can only ask tasks to do to its parent. If the parent has no more task to do, it will ask its parent and then to the root node if necessary to find job to do. Once a parent has fetch tasks to process, it will send a broadcast message to its children asking them to send their potential question again. It leaves the responsibility to tell that there is no more work to do to the root node. To minimize the network impact, each node follow the same rule, do not send again a question until it got an answer about it. Per example, on the graph 1 if 1 receive a task request from 2 and need to request the root about it and in the mean time 3 ask the same question, 1 will wait *root* answer before sending again a potential question. And what if *root* never answer ? We will see that failure detection later.

3.5 Broadcast commands and shutdown sequence

Gotaktuk is able to broadcast commands. This functionality goes along with the shutdown sequence. Because for both of them the deployment need to be over to complete the task.

A broadcast command is simply a command that will be executed on every nodes. And those will have to send back their results. On a deployed tree like the one above, 1 would have to wait for 2 and 3 answer before sending the result to the root. And 2 and 3 would have to be sure that there is no more child to launch before taking the decision to send their answer.

At the end of the deployment sequence, 2, 3, 7 and 8 will ask to their parent if there is another node to launch, 1 and 2 having no child to launch, will ask to the root with a recursive request. After receiving a bootstrap request from all its direct children, the root node will declare that the bootstrap phase is over, freeing 2, 3, 7 and 8, after propagation by 1 and 6, to send their broadcast results.

The shutdown sequence need the same precondition. The root node will tell its direct children to shutdown only if every node is launched and ready. And will wait for its direct children to shutdown (this is done recursively along the tree). This ensure a clean stop of the deployment tree leaving no orphan process on each node.

3.6 Work scheduling

Gotaktuk is able to schedule tasks over a deployed tree. The scheduling engine use a process per core of the computer to execute tasks in parallel.

Tasks are given by the parent node following the work stealing algorithm. And the results are send immediately after execution, even if in fact, to avoid sending too much small messages, the node groups its answer to send messages less often.

On heavy load of schedule tasks to execute we notice the first nodes of the tree will have a greater network charge than the leafs. This is caused by the nature of a tree for communications.

For now, we juste observed that the work stealing give more work to children that are parents and we did not reach the network limit. Further experimentations need to be done here.

We maybe would have to rethink the connexion between nodes and create a graph with different routes to obtain work to do.

3.7 Failure detection

On a distributed program, failure are often to append. A node can be shutdown because of many reason or its network link may die too.

Each *Gotaktuk*'s node embed a failure detection mechanism that allow it to detect a failure parent or a failure child. This functionality is build over a heart beat to detect defective children, once a child has been declared failed, all its broadcast task are filled with a failure message and all its non done schedule task are taken by other nodes.

The parent failure detection is triggered when the parent stopped answering to questions (heartbeat is a question too). Once a node has detect its parents as failed, it began its shutdown procedure so in cascade, thanks to heart beating, its children will shutdown too and so on to the end of the deploying tree.

3.8 Reliability

For now, the only insured reliability is that *Gotaktuk* will eventually terminate in a finite time.

A further work would be to have backup link to great parents allowing a node to continue working and not cut the tree for a single failure.

4 Experiments

The bellow presented graphs are from two kind. Fixed nodes number and progressive growing nodes number. For the fixed nodes number, three time measures are shown, those are given by the *Debian* time command.

- **real** is the elapsed wall clock time used by the process
- **user** is the elapsed time spent in user mode
- **system** is the elapsed time spent in kernel mode

For the progressive growing nodes number, the number of nodes is increasing by 10 from 10 to 250 and by 50 from 250 to 2000.

> Note that the experiments didn't use 2000 different machines to run the tests. This configuration is obtained by deploying 200 nodes per computer. This will impact the second test bench set where we need 'real' computers to measure propagation cost.

Experiments have been made to be reproducible and ran on *Grid5000*. The operating system has been built with the Inira's *Kameleon* tool[3].

4.1 Deployment Experiments results

Those result shows a comparison between *Taktuk* and *Gotaktuk* at a development stage where work stealing wasn't developed in *Gotaktuk*.

Executions without propagation

Flat launch comparison : Figure **3** shows a comparison between two flat launch. As we can see on this graph, for *Gotaktuk* the **user** time is really greater than the **real** time, it shows a multi-core usage, thing we do not have on *Taktuk*. It lead in fact to a problem corrected by the work stealing algorithm, *Taktuk* has a sliding window feature limiting the number of parallelized launch. We do not have this on *Gotaktuk*'s flat launch and with a greater number of nodes, performance begin to crush.

The purpose of this test is to measure the deployment cost on a fair execution, this is not relevant in real world application because the execution is limited by the master node characteristic and is generally not scalable [1].

Tree comparison : Figure **4** show the mean and standard deviation *Gotaktuk*'s and *Taktuk*'s executions. Taktuk is in work stealing mode (its faster mode) and *Gotaktuk* in binary tree launch. Even in this non linear tree configuration, *Gotaktuk* is faster compared to *Taktuk*. In this configuration *Taktuk*'s speed is increased by two, and its capacities are closer than a real world execution.

This time, *Gotaktuk*'s **user** time is almost zero ($2ms$), it can be explained because, a node only use CPU resources to launch two nodes, and then, spend it time waiting for its children to ask questions or giving answers.

Increasing tree comparison : Figure **5** shows the **real** time execution over an increasing work load. *Taktuk* is launched with work stealing and *no-numbering* and as it was an opportunity to test different deploying tree for *Gotaktuk*, we launched it with different fanout values, 2 stands for a binary tree, 4 for a four nodes by level tree and 8 for an height nodes by level tree. We can notice the logarithmic shape for *Gotaktuk*.

That graph shows that there is not much differences between the 4, 2 and 8 level tree, it can be explained because in

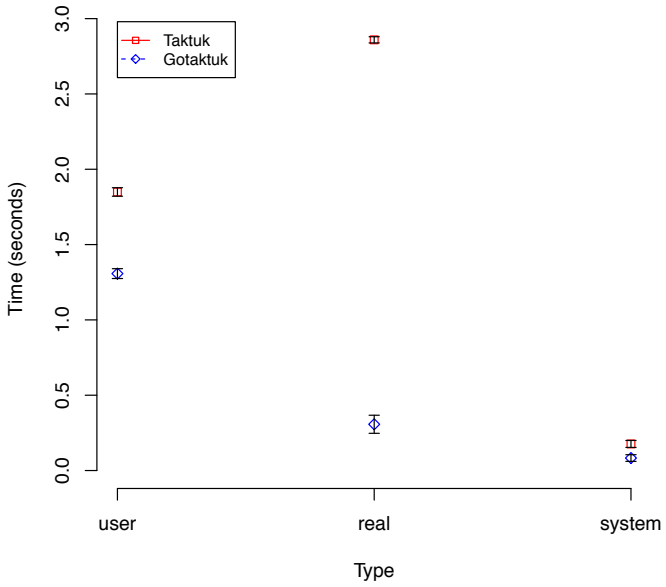


Figure 3: Mean an standard deviation of flat launch for *Taktuk* and *Gotaktuk* over hundred nodes. There is 30 measure for both of the program.

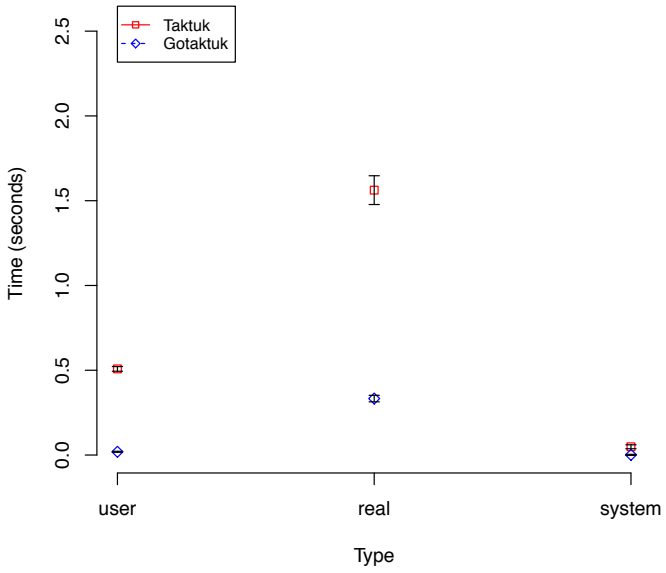


Figure 4: Mean and standard deviation of work stealing launch for *Taktuk* and btree for *Gotaktuk* over a hundred nodes. There is 30 measure for each program.

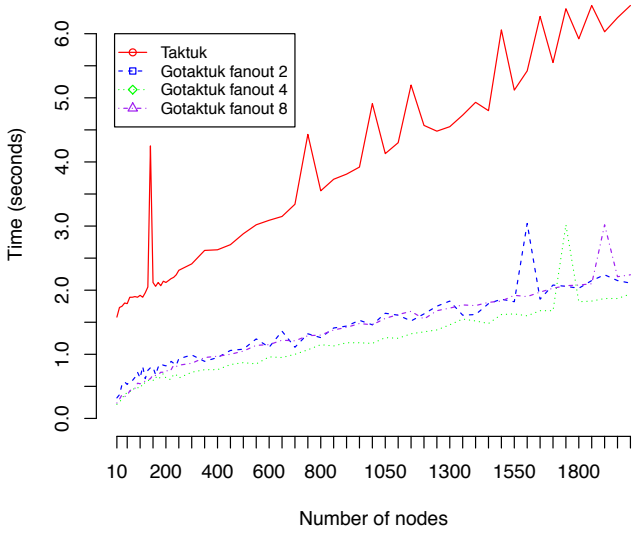


Figure 5: Increasing workload from 10 to 2000 nodes. *Gotaktuk* is launched in different n-ary tree (2,4 and 8) *Taktuk* is in work stealing mode, the efficiency of *Gotaktuk* follow a more sensible logarithmic shape than Taktuk who seems more linear. There is only on measure per point.

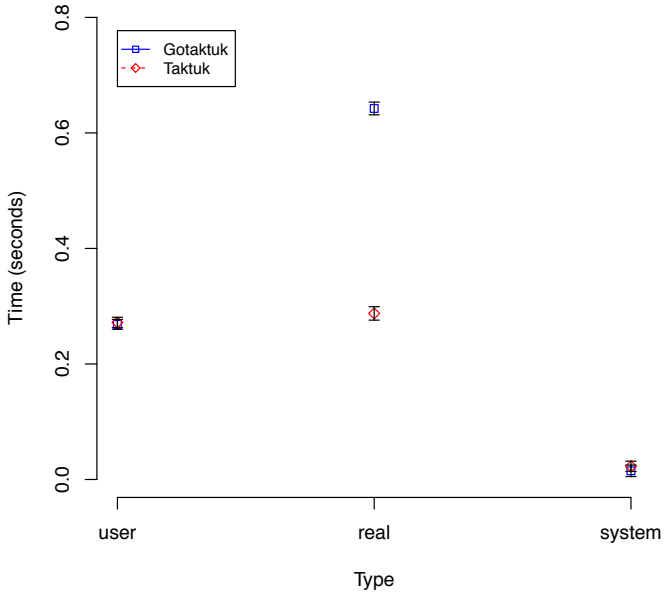


Figure 6: Self propagation binary tree for *Gotaktuk* and work stealing for *Taktuk* over 10 nodes. There is 30 measure for each program.

term of complexity, logarithms function are equivalent. Even if the 4 one seems a little faster, there is only one measure per point, and we need, more measure per point, a mean and a standard deviation for each of them to really see if the 4 is faster than the others. It can also be explained that in term of complexity, logarithms functions are equivalent.

Execution with propagation

Figure **6** and **7** show two executions with propagation. In this configuration, *Gotaktuk* is slower than *Taktuk*. This can be easily explained because of the executable format. Even stripped, *Gotaktuk*'s dynamically linked executable weights $4.5MB$ and *Taktuk*'s $220KB$.

There is still an advantage for the go version, as it installs itself on the remote computer and it does not have to pay the propagation cost each time a node is contacted.

4.2 Tasks scheduling

Gotaktuk's was not good or efficient with realy quick jobs like the echo command and we wondered why. This inefficiency lead to the building of a basic scheduling program only able to do the basics of *Gotaktuk*, like parsing input schedule commands, and use all the cores of the computer to process them. Figure 8 shows on the red curve the growing efficiency of the scheduler with longer tasks to process. This curve represent the best than can be achieved for now in term of efficiency.

Gotaktuk's full engine can be compared to this best possible curve. As the full engine work in this case with 10 nodes it

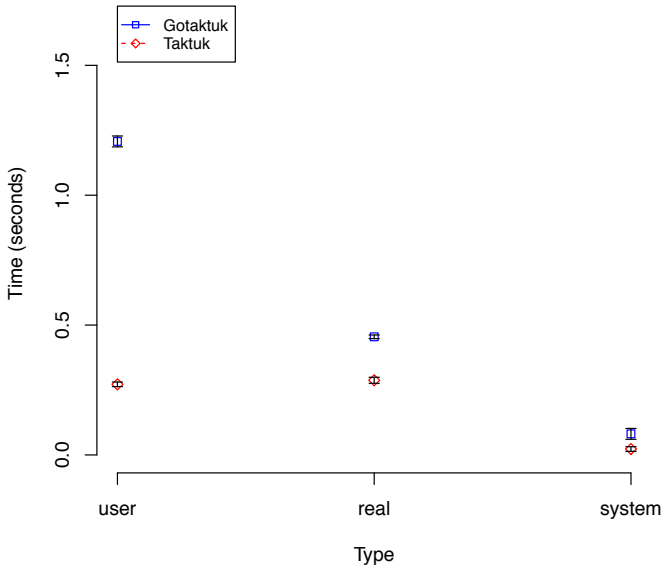


Figure 7: Self propagation with a flat launch tree for *Gotaktuk* and *Taktuk* over 10 nodes. There is 30 measure for each program.

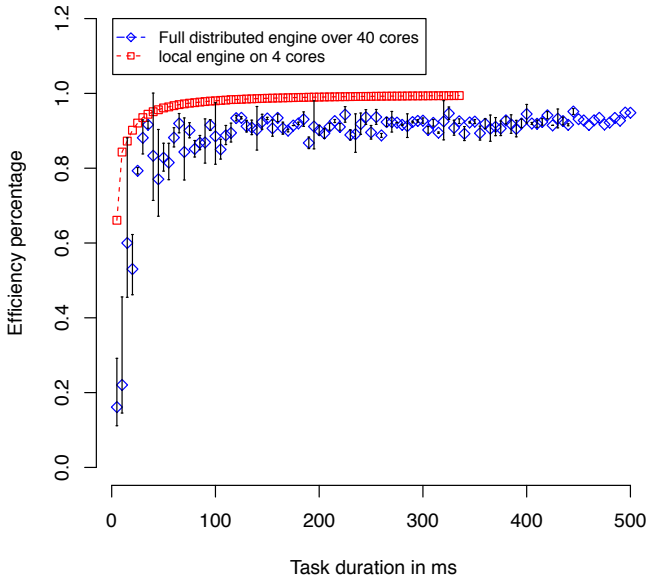


Figure 8: Comparison of the efficiency of Gotaktuk with full functionality (network, children handling) and the core engine of Gotaktuk, which is just a loop launching parrallel program without any network functionality, while executing 5000 commands burning CPU for a determined time ($5ms$ to $500ms$) The simple engine which is software brick responsible of Gotaktuk's task execution has the "best efficiency" we may achieve on Golang for now. The reason of comparison is to measure the efficiency loss with network handling.

cannot be as stable as the local engine. So standard deviation and mean are shown.

We can see on this diagram that *Gotaktuk* begin to be usable with $30ms$ tasks instead of $10ms$ for the local engine.

Further test campaign must be done, in order to see the efficiency evolution with a greater number of nodes and a greater number of tasks to proceed. We also need to compare the efficiency of *Gotaktuk* with other tools, *Taktuk* has a mode that enable it to execute single tasks and this functionality can be measured in term of efficiency.

The percentage is processed looking to the perfect time needed to accomplish the task and is equals to $(5000_{tasks} * x_{ms})/y_{cores})$.

5 Related work

Regarding broadcast execution, there are some software doing the job too, Taktuk, Clustershell [5], pdsh [1], gexec [5].

The table 1 from *Taktuk*'s [1] paper has been reused to include *Gotaktuk* and *Clustershell* in it. It shows the functionalities diffrencies between Gotaktuk and its peers.

From its paper, *Clustershell* [5] seems really fast but it has not the ability to adapt its deployment topology dynamically. We could not reproduce the results because the tree topology

	No remote installation required	New connector plugin	Can mix several connectors	Insensitive to nodes failures	Distributed deployment	Compiled engine
TakTuk [1]	Yes	Immediate	Yes	Yes	Yes	No
Gotaktuk	Yes	Simple	Simple	No	Yes	Yes
Clustershell [5]	No for distributed deployment	unknown	unkown	Yes	Fixed	No
pdsh [1]	Yes	Simple	Yes	Yes	No	Yes
gexec [5]	No	No	No	No	Yes	Yes

Table 1: Functionality comparison between *Gotaktuk* and its peers

need to be statically declared and intermediate nodes in the deployments needs to host the program, there is no self propagation facilities.

Pdsh only launch itself through a flat deployment tree but even if it's a fast program [1][5] this is not scalable [6] and suitable for heavy deployment configuration. In comparison *Gexec* has a distributed tree but cannot spread itself over the network and is intrusive as it need to run as a server on each nodes. [1].

This first version of *Gotaktuk* use the same system with a fixed n-ary tree and follow the same paradigm that *Taktuk* as it embeds some of the needed functionality to run on a Grid environment : - self propagation - high performance - and failure detection.

Regarding scheduling capabilities of *Gotaktuk* and except for *Taktuk*, none of the previous tools provide this kind of functionalities. *Ipyparallel* [13] is one that provide a scripting way to launch parallelized tasks this one is the closest to Gotaktuk, even if in our case, we only have the execution engine. For a further work we need to compare *ipyparallel*'s engine to *Gotaktuk*'s to situate where our software is compared to concurrence.

Another kind of parallelizer is *Swift* that is also scripted parallelizer, this one is a really fast and compiled one. *Gotaktuk* can be compared to *Swift* [14] [15] in term of efficiency but those two kind of systems are not on the same level. *Gotaktuk* is more a versatile tool than an high performance scheduler, at least for now.

6 Conclusions and further work

Gotaktuk has extra functionalities regarding *Taktuk*, like task scheduling. It is not simple rebuild in an other language and it opens perspectives. *Gotaktuk* can connect efficiently a wide range of computational nodes and make them work. The next step may be to allow people to write script that uses Gotaktuk to run. And for sure there is a need to explore other communication graph to connect nodes, bring some reliability, improve performance and scalability.

With a limited amount of code, thanks to the different used libraries, Gotaktuk seems to be a nice way to explore scalability on large scale platforms.

Gotaktuk is fast and do the job its build for. Even the propagation time is not that bad. A good point, is that this program only weights 2.5 thousand lines of code. That means that all the high level libraries we've used have allowed us to create a functional program very quickly. See on table **2** the lines number counting via the cloc tool for *Gotaktuk*. And on the

Language	files	blank	comment	code
Go	16	252	573	2701
Bourne Shell	2	1	2	25
SUM:	18	253	575	2726

Table 2: *Gotaktuk* code line volume

Language	files	blank	comment	code
Perl	32	672	753	3218
C	11	200	201	1590
SUM:	42	872	954	4808

Table 3: *Taktuk* code line volume

table **3** the one for *Taktuk*.

7 Acknowledgment

Thanks to Olivier Richard for accompanying me during this internship, Michael Mercier for his help with *Kamelon*, Baptiste Pichot for his help on my first steps on *Grid5000* and Vincent Danjean for helping me find this internship.

References

[1] Benot Claudel, Guillaume Huard, Olivier Richard. TakTuk, Adaptive Deployment of Remote Executions. Proceedings of the International Symposium on High Performance Distributed Computing (HPDC), 2009, Munich, Germany. ACM, pp.91-100, 2009,

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In Proceedings of the 35th Symposium on Foundations of Computer Science , pages 356 368, 1994.

[3] Joseph Emeras, Olivier Richard, Bruno Bzeznik. Reconstructing the Software Environment of an Experiment with Kameleon. [Research Report] RR-7755, INRIA. 2011.

[4] Martin, C., Richard, O., SAINT MARTIN-France, Montbonnot (2003). Algorithme de vol de travail appliqu au dploiement dapplications parallles. Renpar15, 64-71.

[5] THIELL, Stphane, DEGRMONT, Aurlien, DOREAU, Henri, et al. Clustershell, a scalable execution framework for parallel tasks. In : Linux Symposium. 2012. p. 77.

[6] Buchert, T., Jeanvoine, E., & Nussbaum, L. (2014, May). Emulation at Very Large Scale with Distem. In Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on (pp. 933-936). IEEE.

[7] Meyerson, J. (2014). The Go Programming Language. Software, IEEE, 31(5), 104-104.

[8] Togashi, N., & Klyuev, V. (2014, April). Concurrency in Go and Java: Performance analysis. In Information Science and Technology (ICIST), 2014 4th IEEE International Conference on (pp. 213-216). IEEE.

[9] Georgiev, D., & Atanassov, E. (2014, May). Extensible framework for execution of distributed genetic algorithms on grid clusters. In Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on (pp. 301-306). IEEE.

[10] Nussbaum, L. (2015, November). Grid'5000: a Large Instrument for Parallel and Distributed Computing Experiments. In Journes SUCCES-Rencontres Scientifiques des Utilisateurs de Calcul intensif, de Cloud Et de Stockage.

[11] Cappello, F., Caron, E., Dayde, M., Desprez, F., Jgou, Y., Primet, P., ... & Mornet, G. (2005, November). Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (pp. 99-106). IEEE Computer Society.

[12] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 558-565.

[13] https://github.com/ipython/ipyparallel

[14] OUSTERHOUT, John. PARALLEL SCRIPTING FOR APPLICATIONS AT THE PETASCALE AND BEYOND.

[15] WILDE, Michael, HATEGAN, Mihael, WOZNIAK, Justin M., et al. Swift: A language for distributed parallel scripting. Parallel Computing, 2011, vol. 37, no 9, p. 633-652.

Refinement of requirements for critical systems

Myriam CLOUET

Supervised by : Laurence PIERRE
Laboratory : TIMA
Team : AMfoRS

Abstract

The context of this paper is the verification of Systems on Chip (SoC's) during their design flow. These systems are described as virtual platforms in SystemC at Transaction Level (TL) then they are refined step by step to be ultimately refined at Register Transfer Level (RTL). At each level, the behaviour can be verified with Property Specification Language (PSL) assertions according to the system specifications. The goal of this work is to reuse the assertions throughout the design flow. Due to different timing granularities, they cannot be directly reused. A set of assertion refinement rules was previously defined. These rules allow to refine assertions according to given timing conditions. Here we extend and generalize this set. This paper discusses this issue and proposes a new PSL assertion refinement set. The illustrative examples use properties of an industrial case study.

1 Introduction

Systems on Chip (SoC's) are more and more commonly used. A SoC is composed of several components. For example, these components are processors, memories, busses... There are many different kinds of SoC's in embedded systems. Those systems can be more or less expensive, more or less complex. They can be found in washing machines, smartphones, planes, nuclear plants ... Most of these SoC's are very complex to design and to verify.

A usual design methodology is based on the description of a virtual platform of the system under development. This virtual platform is defined at Transaction Level Modeling (TLM) abstraction level in SystemC. A virtual platform at TLM is a coarse-grained model. At TLM we abstractly represent the components of the system and their communications. We do not specify precisely the internal behavior of these components or how the communications are precisely made. Each communication action is seen as an atomic operation.

Then this platform is refined step by step. It is ultimately given at the Register Transfer Level (RTL). At RTL we see the clock of each component and the details at signal level (bit). We specify more precisely the components of the system and their communications. For example we can select a bus protocol, and the communications are defined with this protocol in mind. In this case, to respect the chosen protocol, we use the signals and the timing of the protocol. And communications take this timing into consideration. For example, at TLM we can have "*write X at address A*". This is an atomic action. At RTL, according to the chosen protocol, it can become a series of events which are spread over time. This is no more atomic.

At every abstraction level, the system behavior must be verified. To that goal, we can simulate the virtual platform. Then we analyze the trace generated by the simulation. We verify that communications are made correctly.

For example consider a system with a processor and a Direct Memory Access (DMA). A DMA is a component which is used to transfer data between memories. The processor configures the DMA, and then the DMA transfers the data. When the DMA finishes the transfer, it sends an interrupt to the processor. We may want to verify "*The DMA must not be configured again before the end of transfer interrupt*". So we simulate the virtual platform, and we check on the generated trace that the behavior respects this specification.

Traces generated by the simulation can be very long. So, the manual verification of the respect of specifications can be tedious and error-prone. Therefore we formalize properties in Property Specification Language (PSL)[IEEE, 2005b]. These properties correspond to specifications of the system behavior. Properties in PSL can be used by differents tools to automatically verify on the generated trace that the behavior respects the specifications.

In that design and validation context, the project of the team aims at giving the possibility to use these properties throughout the refinement of the virtual platform. But communications in virtual platforms at TLM and virtual platforms at RTL can be different. Therefore the properties on their communications, at TLM and at RTL can be different. We need to refine the properties concurrently with the refinement of the virtual platform, in particular the properties that use components which are refined.

The team has previously defined a set of transformation rules [Pierre and Bel Hadj Amor, 2013] that allow to semi-automatically refine properties. The resulting properties can be used in tools to verify the behavior of the virtual platform.

This set of transformation rules was very restrictive therefore we define new rules to extend and to generalize this set.

The remainder of the paper is organized as follows. In Sec-

tion 2 we present the PSL monitoring for SystemC. In Section 3 we provide the existing assertion refinement rule set and its extention. In Section 4 we illustrate the assertion refinement with an industrial case study. In Section 5 we briefly present the related work and in Section 6 we conclude.

2 PSL monitoring for SystemC

2.1 SystemC TLM modeling

SystemC [IEEE, 2005a] is a C++ library which allows to define virtual platforms. The components of the platform are represented by class instances. SystemC has a specific class to define components of platforms : sc_module. For example, to define a processor of the platform we will define a new class Processor which extends the class sc_module, and implements a specific behavior. Then the virtual platform includes an instance of this class Processor, this is the representation of the processor of the system.

The communication channels between the components are represented by class instances too. SystemC has a specific class to define the communication channels : sc_channel. For example, to define a bus of the platform we will define a new class Bus which extends the class sc_channel, and implements a specific behavior.

The interrupts in the system are represented by instances of a SystemC class : sc_signal.

Throughout this paper we use a running example that was designed by Astrium in the SoCKET [Soc, 2010] project. The behaviour of this platform, described in [Pierre and Bel Hadj Amor, 2013], is : This image processing platform performs spectral compression on incoming images, see Figure 1. Raw data are first sub-sampled to reduce the data set to its most significant part, free of optical aberrations (left part, with the "Leon_a" processor and the "DMA_a" component). A 2D-FFT is applied to obtain the corresponding spectrum. The latter is then compressed and encoded to reduce the output throughput (right part of the figure, with the "Leon_b" processor and the "DMA_b" component). Each processor has its own 32-bits wide bus, with a memory and a DMA component.

The IO module generates periodic IRQs, received by Leon_a, wich configures DMA_a to copy data from the IO module to Mem_a. At the end of the transfer, Leon_a subsamples the data and writes the result to Mem_a. It then configures DMA_a to copy the results form Mem_a to Mem_b. Leon_b configures the FFT module to perform 2D-FFT on these data in Mem_b, and then compresses the obtained spectrum. Compressed data are then transferred by DMA_b to the IO port. The processing platform can start processing a new image before finishing the processing of previous ones, thus increasing the overall throughput.

This platform has been described in SystemC. The Leon_a, Leon_b, Mem_a, Mem_b, DMA_a, DMA_b, FFT, and IO_module are instances of classes which extend sc_module. The three busses are instances of a class which extends sc_channel. The interrupts between DMA_a and Leon_a, between DMA_b and Leon_b, between FFT and Leon_b are instances of sc_signal.

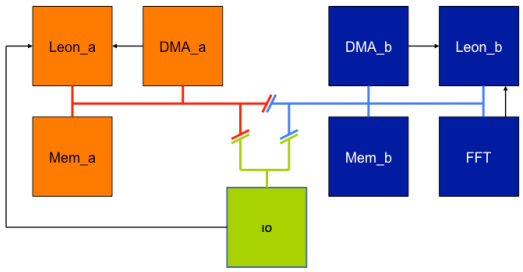


Figure 1: Astrium Platform

The communications on the bus are implemented by functions read_block and write_block that represent read or write requests, and that take the necessary values as parameters. For example the first parameter of write_block is the address and the second parameter represents the data to be written.

2.2 Overview of PSL

Property Specification Language (PSL) [IEEE, 2005b] is a standard language that enables to formalize temporal logic formulae. Its Foundation Language (FL) class extends Linear Temporal Logic (LTL). PSL uses temporal operators like : *next, until, always, ...* PSL uses Sequential Extended Regular Expressions (SERE) like : *, +, ...

PSL allows to interpret properties on a finite or infinite word, or trace. A trace is a set of successive observation points, denoted : $v = v^0 v^1 v^2 v^3 \dots$

We briefly explain some notations in PSL.

Let v be a trace, b a boolean, φ a FL formula. The length of v is noted $|v|$. v^i denotes the $(i+1)^{th}$ observation point of v. For $j \geq i$, $v^{i..j} = v^i v^{i+1} \dots v^j$ and for $j < i$, $v^{i..j} = \epsilon$. $v^{i..}$ denotes the suffix of v starting at v^i. $v^i \models b$ means v satisfies b. $v \models \varphi$ means v satisfies φ.

We briefly present some operators in PSL. Let v be a word, r, r_1, r_2 SERE formulae, φ, ψ FL formulae.

- $v \models r_1; r_2 \Leftrightarrow \exists v_1, v_2 \; s.t. \; v = v_1 v_2, v_1 \models r_1, and \; v_2 \models r_2$.
- $v \models \varphi \; until! \; \psi \stackrel{def}{=} \exists k < |v| \; s.t. \; v^{k..} \models \psi, and \; \forall j < k, v^{j..} \models \varphi$.
- $v \models \varphi \; until \; \psi \stackrel{def}{=} (\varphi \; until \; \psi) \vee always(\varphi)$.
- $v \models always(\varphi) \stackrel{def}{=} \forall j < |v| \; s.t. \; v^{j..} \models \varphi$
- $v \models next!(\varphi) \stackrel{def}{=} |v| > 1 \; and \; v^{1..} \models \varphi$
- $v \models next![i](\varphi) \stackrel{def}{=} \underbrace{next! \; next! \; \dots next!}_{i \; times}(\varphi)$
- $next\_a![i..j]\varphi \stackrel{def}{=} next![i](\varphi) \wedge \dots \wedge next![j](\varphi)$

until! is called the strong until and *until* is called the weak until. The weak until does not require the occurence of ψ.

PSL supports parameterized SEREs and FLs. Let f be a PSL formula, S a set of constants, integers or boolean values and p an identifier.

- $for\ p\ in\ S : ||\ f \stackrel{def}{=} \bigvee_{s \in S} f[p \leftarrow s]$
- $forall\ p\ in\ S : f \stackrel{def}{=} \bigwedge_{s \in S} f[p \leftarrow s]$

Here, PSL will be used to formalize and to verify properties at TLM and RTL levels of abstraction. The evaluations of RTL and TLM properties require the construction of traces with different sampling schemes.

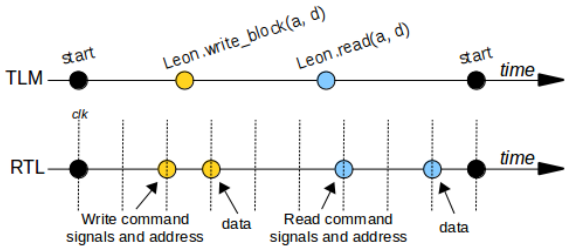


Figure 2: Example of temporal refinement

Figure 2 shows an example of a TLM trace and a RTL trace. At TLM level, the trace is sampled on communications. At this level, an observation point contains the communication information (call, parameters). At RTL level, the trace is sampled on clock ticks. At this level, an observation point contains the signals set at this clock tick.

At TLM the transactions are atomic, but at RTL these transactions can correspond to several actions. On this example, at TLM the FFT write was atomic, at RTL, the data is sent 1 cycle after the address. At TLM the leon read was atomic, at RTL the data read can occur several cycles after the read request.

2.3 Instrumentation with assertion checkers

The team has developed a tool called ISIS [Pierre and Ferro, 2008] which allows to check PSL assertions for a virtual platform, during simulation. This tool can deal with properties at TLM level.

With the description of a virtual platform and PSL properties for this platform, the tool generates assertion checkers. It combines generated assertion checkers with the virtual platform. This new design can be used in simulation to verify if the system behavior respects the specifications.

Let us continue with our example of Astrium platform. One property on this platform is : *An input data packet must be read before the IO module generates a new interrupt*[Pierre and Bel Hadj Amor, 2013].

This correspond to : "each time the IO module generates an IRQ, the DMA_a must read the data before the next IRQ ".

i.e. It is $always$ true that, if "IO module generates an IRQ" $then$ from the $next$ evaluation point, "the DMA_a must read the data" $before$ that "IO module generates an IRQ" (again). Therefore the property will have the form : $always(A => next(B\ before\ A))$, with A which respresents "IO module generates an IRQ", and B which represents "the DMA_a must read the data".

The IRQ generation by the IO module is represented by $io\_module.generate\_irq\_CALL()$.

The read by the DMA_a at IO module address is represented by $dma\_a.read\_block\_END()$ && $dma\_a.read\_block.p1 == io\_module\_add$

So the corresponding PSL property is :

```
ALWAYS(io_module.generate_irq_CALL()
=> NEXT((dma_a.read_block_END() &&
         dma_a.read_block.p1 == io_module_add)
        BEFORE! (io_module.generate_irq_CALL()))
       )
   )
```

The team has also developed HORUS [Morin-Allory *et al.*, 2008], a tool which allows to check PSL assertions during simulations at RTL level. This tool has a similar behaviour. With a description at RTL level and RTL PSL properties, the tool generates assertion checkers. It combines generated assertion checkers with the RTL design. This new design can be used in simulation to verify the system behavior.

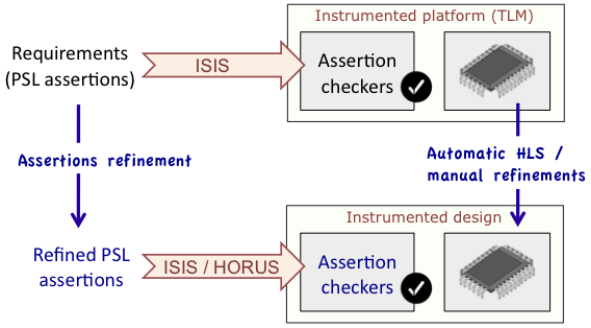


Figure 3: TLM-to-RTL verification flow [Pierre and Bel Hadj Amor, 2013]

Figure 3 represents the targeted verification flow. When a Design Under Verification (DUV) at TLM level is defined, TLM PSL assertions can be used with ISIS to verify if the system behavior respects the specifications.

Then the DUV is refined manually or automatically. PSL assertions at RTL level can be used with HORUS to verify if system behavior respects the specifications.

The goal is to reuse the TLM assertions at the RTL level. But TLM PSL assertions cannot be used at RTL level because RTL assertions represent a more fine-grained specification, as illustrated in section 2.2. Therefore we have to refine TLM assertions to RTL assertions according to the PSL semantics and specific constraints due to the platform refinement.

3 Assertion refinement

3.1 Existing rules

A set of refinement rules has been defined in [Pierre and Bel Hadj Amor, 2013]. It defines specific refinements for some

expressions commonly used in TLM properties, and some specific timing schemes.

Refinement of a communication into a sequence of events
Transformation 1 :

- expression : $A\ before!\ (B\ and\ C)$
- time constraint : C occurs X cycles after B
- refined expression : $((B\ \rightarrow next![X](\neg C))\ until!\ A)$ $and\ next\_event(A)(B \rightarrow next![X](\neg C))$

Transformation 2 :

- expression : $(A\ and\ B)\ before!\ C$
- time constraint : B occurs X cycles after A
- refined expression : $(prev(A, X)\ and\ B)\ before!\ C$

Transformation 3 :

- expression : $A\ and\ B \rightarrow C$
- time constraint : B occurs X cycles after A
- refined expression : $(A\ and\ next\_a![1..(X - 1)(\neg B)\ and\ next![X](B)) \rightarrow next![X](C)$

The transformation that was originaly proposed did not actually correspond to this timing constraint. The refinement has been reworked here.

Simplification, considering the specific delay of a communication
Transformation 4 :

- expression : $A\ until!\ B$
- time constraint : B expected 1 clock cycle after A
- simplified expression : $(A\ and\ \neg B)\ and\ next!\ B$

Transformation 5 :

- expression : $A\ until!\ B$
- time constraint : B expected X clock cycles after A
- simplified expression : $next\_a![0..X - 1](A\ and\ \neg B)$ $and\ next![X](B)$

Transformation 6 :

- expression : $A\ before!\ B$
- time constraint : B expected X clock cycles after A
- simplified expression : $next\_e![0..X - 1](A)\ and$ $next\_a![0..X - 1](\neg B)\ and\ next![X](B)$

The correctness of each refinement rule has been proven in [Pierre and Bel Hadj Amor, 2013]. Here we recall one of these proofs :

Proof of transformation 1 Since we are under the constraint C occurs X cycles after B, A must hold before the clock tick in which B and next![X](C) holds, which means that we have to consider: $v \models A\ before!(B\ and\ next![X](C))$,

i.e., $v \models \neg(B\ and\ next![X](C))\ until!$
$(A\ and \neg(B\ and\ next![X](C)))$

i.e., $\exists k < |v|\ s.t.$
$v^{k..} \models (A\ and \neg(B\ and\ next![X](C)))\ and$
$\forall j < k, v^{j..} \models \neg(B\ and\ next![X](C))$

i.e., $\exists k < |v|\ s.t.$
$v^{k..} \models (A\ and \neg(B\ and\ next![X](C)))\ and$
$v^{k..} \models A\ and$
$\forall j < k, v^{j..} \models (\neg A) \vee (B\ and\ next![X](C))\ and$
$\forall j < k, v^{j..} \models \neg(B\ and\ next![X](C))$

i.e., $\exists k < |v|\ s.t.$
$v^{k..} \models (A\ and \neg(B\ and\ next![X](C)))\ and\ v^{k..} \models A\ and$
$\forall j < k, v^{j..} \models (\neg A)\ and$
$\forall j < k, v^{j..} \models \neg(B\ and\ next![X](C))$

i.e., $v \models (\neg(B\ and\ next![X](C))\ until!\ A)\ and$
$next\_event(A)(\neg(B\ and\ next![X](C)))$

i.e., $v \models ((B \rightarrow next![X](\neg C))\ until!\ A)\ and$
$next\_event(A)((B \rightarrow next![X](\neg C)))$

that avoids having a temporal operator under a negation.

3.2 Creation of new rules

Each existing rule was based on a time constraint with a fixed number of clock cycles (X or 1). Often with bus protocols the delay of a communication request is unknown.

Therefore we extend the set of rules with more general transformations.

Refinement of a communication into a sequence of events
Generalization of transformation 1 :

- expression : $A\ before!\ (B\ and\ C)$
- time constraint : C occurs any number of cycles after B
- refined expression : $(\neg C\ and\ \neg A)\ until!$ $(\neg C\ and\ A\ and\ next!((\neg C\ and\ \neg B)\ until\ B))$

Generalization 1 of transformation 2 :

- expression : $(A\ and\ B)\ before!\ C$
- time constraint : B occurs any number of cycles after A
- refined expression : $(\neg C\ and\ \neg A)\ until!$ $(\neg C\ and\ A\ and\ next!((\neg C\ and\ \neg B)\ until!\ (\neg C\ and\ B)))$

Generalization 2 of transformation 2 :

- expression : $(A\ and\ B)\ before!\ C$
- time constraint : B occurs any number of cycles after A, before a given event E (usually the final event of a communication)
- refined expression : $(\neg C\ and\ \neg A)\ until!$ $(\neg C\ and\ \neg B\ and\ \neg E\ and\ A\ \ and$ $next!((\neg C\ and\ \neg E\ and\ \neg B)\ until!$ $(\neg C\ and\ B\ and\ \neg E\ and\ next!((\neg C\ and\ \neg E)\ until!$ $(\neg C\ and\ E)))))$

Extension of transformation 3 :

- expression : $A\ and\ B \rightarrow C$
- time constraint : B occurs at most X cycles after A
- refined expression : $forall\ i\ in\ \{1 : X\}$: $(A\ and\ next\_a![1..i-1](\neg B)\ and\ next![i](B)) \rightarrow next![i](C)$

Generalization of transformation 3 :

- expression : $A\ and\ B \rightarrow C$
- time constraint : B occurs any number of cycles after A
- refined expression : $A \rightarrow next\_event(B)(C)$

Simplification, considering the specific delay of a communication
Extension of transformation 5 :

- expression : $A\ until!\ B$
- time constraint :B expected at most X cycles after A
- simplified expression : $for\ i\ in\{0 : X-1\}$: $||\ (next\_a![0..i](A\ and\ \neg B)\ and\ next![i+1](B))$

We have proven the correctness of these rules. We give below one of these proofs.

Proof of generalization of the transformation 1 B and C was simultaneous at TLM level, but at RTL level C occurs any number of cycles after B. Therefore $v \models A\ before!(B\ and\ C)$ becomes $v \models \{(\neg C\ and\ \neg A)[*]; \neg C\ and\ A; (\neg C\ and\ \neg B)[*] \vee \{(\neg C\ and \neg B)[*]; B\}\}$.

From the definition of $r_1; r_2$
$\exists i < |v|\ s.t.\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and\ v^{i+2..} \models (\neg C\ and\ \neg B)[*] \vee \{(\neg C\ and \neg B)[*]; B\}$

i.e $\exists i < |v|\ s.t.\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$(v^{i+2..} \models (\neg C\ and \neg B)[*] \vee$
$v^{i+2..} \models \{(\neg C\ and \neg B)[*]; B\})$

From the definition of $r_1; r_2$
$\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$(v^{i+2..} \models (\neg C\ and \neg B)[*] \vee$
$(\exists j, i+1 \leq j < |v|\ s.t.$
$v^{i+2..j} \models (\neg C\ and \neg B)[*]\ and\ v^{j+1} \models B))$

But $v \models b[*] \Leftrightarrow \forall k < |v|, v^k \models b.$

Thus $\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$(v^{i+2..} \models (\neg C\ and \neg B)[*] \vee$
$(\exists j, i+1 \leq j < |v|\ s.t.\ \forall m, i+2 \leq m \leq j,$
$v^m \models \neg C\ and \neg B\ and\ v^{j+1} \models B))$

From the definition of $until!$
$\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$(v^{i+2..} \models (\neg C\ and \neg B)[*] \vee$
$v^{i+2..} \models (\neg C\ and \neg B)\ until!\ B)$

But $v^{i+2..} \models b[*] \Leftrightarrow v^{i+2..} \models always(b).$

Thus $\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$(v^{i+2..} \models always(\neg C\ and \neg B) \vee$
$v^{i+2..} \models (\neg C\ and \neg B)\ until!\ B)$

i.e. $\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$(v^{i+2..} \models always(\neg C\ and \neg B) \vee (\neg C\ and \neg B)\ until!\ B)$

From the definition of $until$
$\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$(v^{i+2..} \models (\neg C\ and \neg B)\ until\ B)$

From the definition of $next!$
$\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1} \models \neg C\ and\ A\ and$
$v^{i+1..} \models next!((\neg C\ and \neg B)\ until\ B)$

i.e. $\exists i < |v|\ s.t\ v^{0..i} \models (\neg C\ and \neg A)[*]\ and$
$v^{i+1..} \models \neg C\ and\ A\ and\ next!((\neg C\ and \neg B)\ until\ B)$

But $v \models b[*] \Leftrightarrow \forall k < |v|, v^k \models b.$

Thus $\exists i < |v|\ s.t\ \forall j < i+1, v^j \models (\neg C\ and \neg A)\ and$
$v^{i+1..} \models \neg C\ and\ A\ and\ next!((\neg C\ and \neg B)\ until\ B)$

Form the definition of $until!$
$v^{0..} \models (\neg C\ and \neg A)\ until!$
$(\neg C\ and\ A\ and\ next!((\neg C\ and \neg B)\ until\ B))$

$v \models (\neg C\ and \neg A)\ until!$
$(\neg C\ and\ A\ and\ next!((\neg C\ and \neg B)\ until\ B))$

We obtain the refined rule$(\neg C\ and \neg A)\ until!$ $(\neg C\ and\ A\ and\ next!((\neg C\ and \neg B)\ until\ B))$. This rule has the form : $b\ until!\ \varphi$, where b is a boolean, φ is a temporal expression. This seems to be an issue because this property is not in the PSL simple subset.

In [IEEE, 2005b] the simple subset is defined as : a subset that conforms to the notion of monotonic advancement of time, left to right through the property, which in turn ensures that properties within the subset can be simulated easily.

This subset must be used in the context of dynamic verification. It limits the use of some operators, in particular for the operator $until!$: its right operand must be boolean.

Therefore our property is not in the simple subset as defined in [IEEE, 2005b] and seems to violate the notion of monotonic advancement of time.

However, in [Ben-David *et al.*, 2005] they define a subset of safety Regular-LTL (RLTL) formulae, called RLTL^{LV}. RTLT extends LTL with regular expressions.

They define : *If b is a boolean expression, r is an RE*

and φ, φ_1 *and* φ_2 *are* RLTL^{LV} *then the following are in* RLTL^{LV}:

1. b
2. $\varphi_1 \wedge \varphi_2$
3. $(b \wedge \varphi_1)$ W $(\neg b \wedge \varphi_2)$
4. $(b \wedge \varphi_1) \vee (\neg b \wedge \varphi_2)$
5. Xφ
6. $r \mapsto \varphi$

The formula 3. is interesting in our case. The W operator corresponds to the weak until in PSL. Here we see that both operands of the until are not constrained to be boolean expressions.

But they explain that their subset is less restrictive than the simple subset in [IEEE, 2005b] : for the operators $\vee$ and W the simple subset allows only one operand to be non-boolean, whereas RLTL^{LT} allows both non-boolean, conditioned they can be conjuncted with some boolean and its negation.

Our property has pattern : $b\ until!\varphi$. But we notice that b is $\neg A \wedge b_1$ and φ is $A \wedge \varphi_1$.

So, our property is $(\neg A \wedge b_1)\ until!\ (A \wedge \varphi_1)$, where b_1 is a boolean and φ_1 is a temporal expression. We have the right operand conjuncted with a boolean (A), and the left operand conjuncted with the negation of this boolean ($\neg A$).

Therefore our property is in the simple subset as defined in [Ben-David *et al.*, 2005] and conforms to the notion of monotonic advancement of time.

4 Experiments

Let us continue with our example of Astrium platform. For the experiments we use a platform defined in SystemC at TLM level. This platform is concretized with buses that respect the WISHBONE protocol [WIS, 2010]. This protocol works according to a master-slave principle. A master reads or writes at an address on a slave.

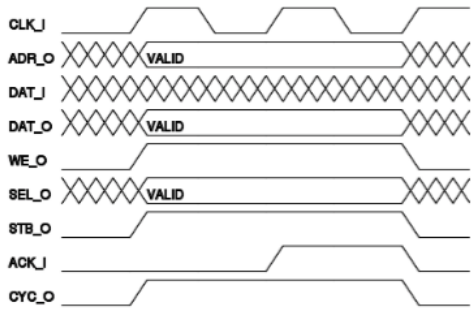


Figure 4: SINGLE WRITE cycle [WIS, 2010]

Figure 4 shows the signals of a master when it sends a write request, and when the slave responds in one cycle.

The CYC_O signal requests use of a bus from an arbiter. The signal is asserted for the duration of all bus cycles. It is asserted during the first data transfer, and remains asserted until the last data transfer. The STB_O signal indicates a valid data transfer cycle. The slave asserts the ACK_I, ERR_I or RTY_I signals in response of every assertion of STB_O signal. The acknowledge input ACK_I, when asserted, indicates the normal termination of a bus cycle. The write enable output WE_O indicates whether the current local bus cycle is a READ or a WRITE cycle. The signal is negated during READ cycles, and is asserted during WRITE cycles. In the single write of WISHBONE protocol, most control signals, address and data are set simultaneously.

One of the properties on this platform is *The FFT module must not be configured before the end of the computation*.

This is : each time leon_b processor finishes the FFT configuration, the end of the computations occurs before the next configuration.

The end of the FFT configuration corresponds to leon_b which writes in the register $read\_addr$ of FFT. This is represented by leon_b.write_block_CALL() && leon_b.write_block.p1 == read_addr.

The end of the computation corresponds to leon_b reads a value equal to image size in register $write\_length$ of FFT. This is represented by leon_b.read_block_END() && leon_b.read_block.p1 == write_length && leon_b.read_block.p2 == img_size.

This property is in PSL :

```
ALWAYS (leon_b.write_block_CALL() &&
leon_b.write_block.p1 == read_addr
=> NEXT((leon_b.read_block_END() &&
          leon_b.read_block.p1 == write_length &&
          leon_b.read_block.p2 == img_size
         )
         BEFORE!(leon_b.write_block_CALL() &&
                  leon_b.write_block.p1 == read_addr
                 )
        )
);
```

Using ISIS [Pierre and Ferro, 2008], we instrument the platform with the previous property. If we run the simulation for 10 io cycles, the monitor is triggered 6548 times and it is evaluated 10 times(number of times the premise of the implication is true).

After we refine the platform buses. We need to refine the property.

TLM to RTL, temporal refinement : With the WISHBONE protocol, there can be a delay between the read request of Master and the effective read of the data provided by the Slave.

In our case, at TLM the read request and the data read were simultaneous but at RTL, the data read can occur any number of cycles after the read request.

We are in the case of a refinement of a communication into a sequence of events. Our property is of shape $always(X \implies next(Y))$ and Y has the form $(A\ and\ B)\ before!\ C$.

Where :

A = leon_b.read_block_END() && leon_b.read_block.p1 == write_length,

B = leon_b.read_block.p2 == img_size and

C = leon_b.write_block_CALL() && leon_b.write_block.p1 == read_addr.

In our case, the time constraint is : B occurs any number of cycles after A. Therefore we can use the Generalization 1

of transformation 2 rule.

TLM to RTL, structural refinement : Moreover, boolean expressions must be structurally transformed in terms of the corresponding conditions on signals. In PSL, the function rose(signal) returns true when there is a rising edge of the signal. With WISHBONE protocol [WIS, 2010],

- leon_b.write_block_CALL() && leon_b.write_block.p1 == read_addr corresponds to :
  rose(leon_b_CYC_O) && leon_b_STB_O && leon_b_WE_O && leon_b_ADR_O == read_addr.
- leon_b.read_block_END() && leon_b.read_block.p1 == write_length corresponds to :
  rose(leon_b_CYC_O) && leon_b_STB_O && !leon_b_WE_O && leon_b_ADR_O == write_length.
- leon_b.read_block.p2 == img_size corresponds to :
  leon_b_DAT_I == img_size && rose(fft_ACK_O).

Finaly the property becomes :

```
ALWAYS (
leon_b.rose(CYC_O) && leon_b_STB_O && leon_b_WE_O &&
leon_b_ADR_O == read_addr
=> NEXT( ( !(rose(leon_b_CYC_O) && leon_b_STB_O &&
             leon_b_WE_O && leon_b_ADR_O == read_addr
            ) &&
           !(rose(leon_b_CYC_O) && leon_b_STB_O &&
             !leon_b_WE_O && leon_b_ADR_O == write_length
            )
         )
         UNTIL!
         ( !(rose(leon_b_CYC_O) && leon_b_STB_O &&
             leon_b_WE_O && leon_b_ADR_O == read_addr
            ) &&
           (rose(leon_b_CYC_O) && leon_b_STB_O &&
            !leon_b_WE_O && leon_b_ADR_O == write_length
           ) &&
           NEXT!( ( !(rose(leon_b_CYC_O) && leon_b_STB_O
                      && leon_b_WE_O &&
                      leon_b_ADR_O == read_addr
                     ) &&
                    !(leon_b_DAT_I == img_size &&
                      rose(fft_ACK_O))
                  )
                  UNTIL!( !(rose(leon_b_CYC_O) &&
                            leon_b_STB_O && leon_b_WE_O
                            && leon_b_ADR_O == read_addr
                           ) &&
                            leon_b_DAT_I == img_size
                            && rose(fft_ACK_O)
  ) ) ) ) );
```

Using ISIS, we instrument the refined platform with this property. An example of a simulation trace excerpt is in Figure 5. The text displayed by the monitor is between the arrows ==> <== and just below. The rest of the text comes from the normal simulation of the platform. As explained in section 2.2 the trace is sampled on clock ticks, and period clock is 12 ns. If we run the simulation for 10 io cycles, this property is triggered 583334 times and it is evaluated 10 times. The evaluation number is the same as at TLM though the trace has been sampled on clock ticks. On the trace at 454932 ns, leon_b writes at the address of the fft read_addr register : CYC_O is rising, WE_O is asserted, and ADR_O is egal to the address of read_addr. Before that observation point, the property was not pending (was not being evaluated), here its pending variable becomes true. Due to lack of space, intermediate steps are not displayed. At 584472 ns, leon_b sends a read request at the address of the fft write_length : CYC_O is rising, WE_O is negated, and ADR_O is egal to the address of write_length. 17 cycles after, at 584676 data are read by the leon_b : fft_ACK_O is rising . At TLM the read was atomic, here the data read can occur any number of cycles after the read request. On this particular case in the simulation it is 17 cycles. In this case, leon_b is granted the bus 16 cycles after its read request. Several cycles later at 1154916 ns leon_b writes at the address of the fft read_addr register, as previously. Here the checking variable of the property is true, because its current evaluation is finished. The pending variable should be false, because its evaluation is finished, but the next evaluation of the property starts. Therefore the pending is true again.

5 Related work

There have been few results on methods to refine TLM properties to RTL properties.

In [Bombieri *et al.*, 2007] they define a method to reuse TLM properties with the refined design at RTL level. Their method is based on transactors [tra, 2016]. A transactor is a hardware component which is manually defined to translate TLM transactions into RTL transactions and vice versa. Here a transactor is linked between the assertion checker of the TLM property and the design. The transactor automatically translates the communications from the property to the design and the communications from the design to the property.

This method is not based on semantic transformations. The transactor must be defined for each property for each design.

In [Ecker *et al.*, 2007] and in [Steininger, 2009] they define a new language of linear temporal logic, the Universal Assertion Language (UAL). This language allows to formalize TLM properties and RTL properties. They also propose a method to refine TLM properties to RTL properties. This method is based on the syntactic tree of the property. In this tree, nodes are logical and temporal operators and leaves are boolean expressions. In their method they just consider refinements which do not change the structure of the syntactic tree of the property. i.e. they just consider the refinement of the boolean expressions, the structural refinement.

This method is restrictive because they do not consider the temporal refinements. However the temporal refinement can be necessary, as seen previously.

6 Conclusion

In this paper we recalled an existing set of rules which allows to refine a property at TLM level to a property at RTL level according to a time constraint. We extended this set with new rules which are more general because the time constraints are no more with a fixed number of cycles. We illustrated our refinement method with an industrial case study.

Future work is to include these new rules in the refinement rule data base in ISIS and define new rules considering multiple data transfers such as bursts.

References

[Ben-David *et al.*, 2005] Shoham Ben-David, Dana Fisman, and Sitvanit Ruah. The safety simple subset. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November, 2005, Revised Selected Papers*, 2005.

[Bombieri *et al.*, 2007] Nicola Bombieri, Franco Fummi, Graziano Pravadelli, and Andrea Fedeli. Hybrid, Incremental Assertion-Based Verification for TLM Design Flows. *IEEE Design & Test of Computers*, 24(2), 2007.

[Ecker *et al.*, 2007] Wolfgang Ecker, Volkan Esen, Thomas Steininger, and Michael Velten. Requirements and concepts for transaction level assertion refinement. In *Embedded System Design: Topics, Techniques and Trends, IFIP TC10 Working Conference: International Embedded Systems Symposium (IESS), May 30 - June 1, 2007, Irvine, CA, USA*, 2007.

[IEEE, 2005a] IEEE. *IEEE Std 1666-2005, IEEE Standard SystemC Language Reference Manual*. IEEE, 2005.

[IEEE, 2005b] IEEE. *IEEE Std 1850-2005, IEEE Standard for Property Specification Language (PSL)*. IEEE, 2005.

[Morin-Allory *et al.*, 2008] K. Morin-Allory, Y. Oddos, and D. Borrione. Horus: A tool for Assertion-Based Verification and on-line testing. In *Proc. MEMOCODE'08*, June 2008.

[Pierre and Bel Hadj Amor, 2013] Laurence Pierre and Zeineb Bel Hadj Amor. Automatic refinement of requirements for verification throughout the soc design flow. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013, Montreal, Canada*, pages 29:1–29:10, October 2013.

[Pierre and Ferro, 2008] Laurence Pierre and Luca Ferro. A Tractable and Fast Method for Monitoring SystemC TLM Specifications. *IEEE Transactions on Computers*, 57(10), October 2008.

[Soc, 2010] SoC toolKit for critical Embedded sysTems. http://socket.imag.fr, 2010.

[Steininger, 2009] Thomas Steininger. *Automated Assertion Transformation Across Multiple Abstraction Levels*. PhD thesis, Technische Universität München, November 2009.

[tra, 2016] ZeBu Transactor and Memory Model Solutions . https://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/emulation-validation-ip.aspx , 2016.

[WIS, 2010] *Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores* . 2010.

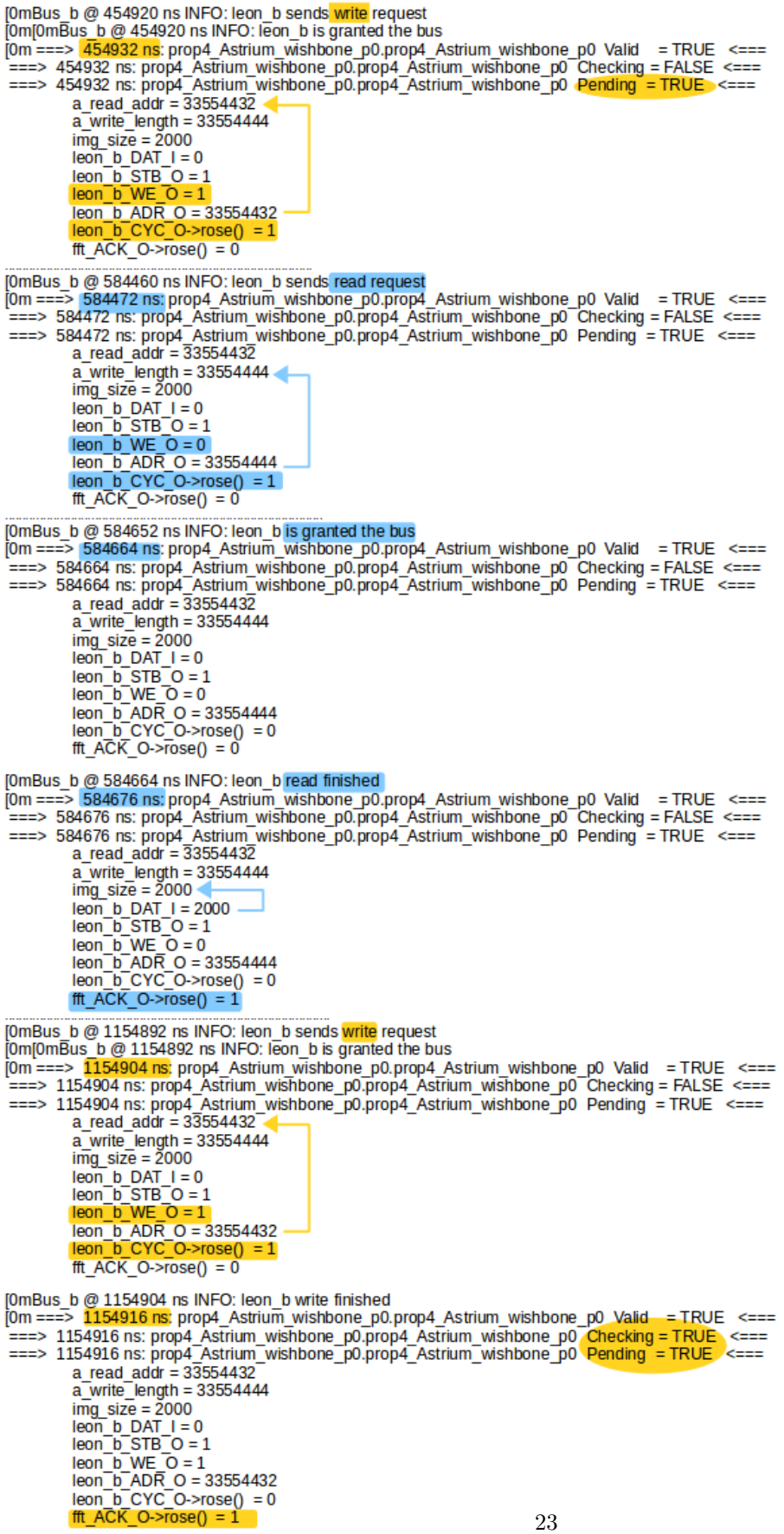


Figure 5: Excerpt of a Simulation Trace for the Property

A language for the home

Lénaïc Terrier
Supervised by: Sybille Caffiau and Alexandre Demeure

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

Abstract

Smart Homes progammed by the end-user seems to be the standard in the industry. We study several languages currently available, their features and their flaws. We discuss some important issues that are mention in the research and finally we propose a new language to programm smart homes which combine the best features of the dominant paradigm: ECA and the properties highlighed by researchers.

1 Introduction

Smart Home is defined in [Mennicken *et al.*, 2014] as "*a home that either increases the comfort of their inhabitants in things they already do or enables functionalities that were not possible through the use of computing technologies*". Technically a smart home is composed of sensors (thermometer, clock, motion tracker, switches...), actuators (lights, sound systems, digital displays...) and networked services (calendar, weather, TV program, traffic...) that together can be used to provide assistance to the inhabitants. Typical usage of the smart home is to increase comfort as said in [Mennicken *et al.*, 2014] but also as noted in [Holloway and Julien, 2010] to ease everyday life chores (automatic vacuum cleaner), to save energy (heating system optimisation), to increase security (alarm systems)...

Although the technical environment already exists and is provided mostly by **Home Automation Systems** it is not what makes a home *smart*. As discussed in [Davidoff *et al.*, 2006] the user should be in control at all time and should easily predict the behavior of his home. Several approaches aim to provide a system adapted to each domestic context.

Automatic learning consist of a system which adapt its behaviour depending on the usage and habits of the users. In the industry this way is used for certain tasks; for example, Hot Water Recirculation Systems can learn the hours of consumption of hot water of a household to deliver hot water just in time, saving energy and minimising heat loss while increasing comfort of the inhabitants [Frye *et al.*, 2013]. Although this solution works well for specific tasks which have few variables, it doesn't cover all the usages of smart homes envisioned by researchers.

Another way is via **End-User Development** which is defined in [Lieberman *et al.*, 2006] as follow: "*a set of methods, techniques and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact*". In this case, the inhabitants have to express to the system the behaviors they want. Because the user is directly programming the behaviours he wants to his home automation system, the solution provide a more precise tailoring of his needs.

Currently the most widespread solution for expressing behaviours to a home automation system via end-user development both in the industry and in the research is the **Event Condition Action** (ECA) paradigm and it's subsets like the *trigger-action* paradigm [Ur *et al.*, 2014]. Granted ECA has great properties for end-user development in the context of a smart home, it's subject to limitations.

While studying the published works on end-user development in the smart home, we noticed a lot of papers focus either on tools (like [García-Herranz *et al.*, 2010]) or on usage (like [Demeure *et al.*, 2015], [Brush *et al.*, 2011], [Lucci and Patern, 2015]), but a few broach the underlying language. And of those which do, they rarely put ECA in competition against other languages [Ur *et al.*, 2014].

This work aims to propose a end user development language which allows inhabitants of a smart home to express the behaviours they want to their home automation system. This language should bring together the best features of existing ECA-based solutions and missing properties that are asked by current users or highlighted by research works.

We first list the properties the language should have in section 2. Then we define use cases that exhibits those properties and try to implement them with existing solutions. By studying the ECA paradigm and by putting it against other solutions we hope to extract useful properties to specify our language. This is done in the section 3. Finally we expose the detail of the solution we propose in section 4.

2 Desired properties of our programming language

First we consider general properties that are required by all programming languages. First off, the language should have

a **low threshold**: what is simple to mentally model (lit a light) should be easy and simple to express (`light.lit()` for example). Moreover, since end-user development is addressed to non professional developers, the basics should be easy to learn [Burnett and Scaffidi, 2011]. On the other hand, simplicity shouldn't mean less expressiveness, the language should have a **high ceiling**: complex tasks should be possible to express, allowing more advanced users to work on bigger projects [Patern, 2013]. It's particularly relevant in home automation system where as observed in [Demeure *et al.*, 2015] some users need challenge.

Another property which is mandatory is the possibility to **encapsulate** and **abstract** pieces of code. In addition to the organisation and grouping needs observed in [Demeure *et al.*, 2015], this allow advanced users to work on big projects and for novice users to use complex tools without knowing the internal architecture. Combined with tools that check **code quality** and rightness, this enable **code sharing** as stated in [Burnett and Scaffidi, 2011]. Although [Burnett and Scaffidi, 2011] express concerns about sharing end-user programmed code, stating that bad code will end up in more household which will be exposed to attacks, it's a well known fact that sharing code leads to more people reading the code and though more chances to detect faulty code.

Next, we focus on properties highlighted in research works. For instance, [Huang and Cakmak, 2015] points out that the distinction between states and events should be made clear, and that the resulting action timing (punctual, extended or maintained) should be explicit. Lack of commentary support and variable naming has been observed in publicly available solutions in [Demeure *et al.*, 2015] forcing users to maintain spreadsheets of correspondence.

Because, smart homes are *homes* there is specific issues that arise. For instance, [Davidoff *et al.*, 2006] explain that the behaviours wanted by different members of the same family are almost never the same, this is also supported in [Demeure *et al.*, 2015]. For that reason it's mandatory to support **access level** just like in classical system where users don't all have the same rights. But event then, the system would eventually encounter a conflict between two users with the same level of access, it should be possible to define **politics of conflict resolution** depending on several factors. A solution is proposed in [Ur *et al.*, 2014]: the most recently added rule wins. Altough this solution could prove effient, we think they may be better ways to deal with this.

Finally, since non professionnals developers programs actions which have consequences in the real world, the system should support an **override** mechanism, thus preventing scenarii where the shutters are trapping inside or outside the household inhabitants. In short the system should give priority to security over comfort.This is supported by [Davidoff *et al.*, 2006] where the author highlight the concept of *routine crisis* and explain that the system should be alble to react to an exceptional scenario.

2.1 Use cases

In order to compare different solutions of ECA-based end user development, we need to design use cases that highlight precises properties noted above. Most of the use cases are purposely design with vague action or devices to allow a larger pool of solutions to be studied. All use cases mentioned in this section can be found in the appendix.

The use case UC 1 is a basic binding of an *event* to a *punctual action*. The only particularity is the repetition in "*every day*". This use case should be simple to implement, it highlights the *low threshold*.

The use case UC 2 a bit more complex. An *event* based sensor is bound to a *maintained* action. This use case highlight the issue with *non sequitur* reasoning. More detailed use cases are provided at UC 2.1 and UC 2.2

The use case UC 3 highlight a simple pure state oriented behaviour.

The use case UC 4 is almost the same as UC 3 but there is a small difference. We know at all point the status of one device from the other one. There is no unknown.

The use case UC 5 highlights a sequence of *states* bind to maintained actions.

The use case UC 6 highlights the properties of abstraction and encapsulation. The Domicube[1] is a cube that act as a remote controller for the home. To control a device, the user turns upward the corresponding face of the cube and rotate it on a vertical axis.

3 State of the art: existing ECA-based solutions

ECA seems to be accepted *by default* as the solution for end-user programming. So much, that it is used by most of home automation systems and resembling tools on the market as it is observed in [Demeure *et al.*, 2015] and other works: eeDomus[2], Crestron[3], Vera 3[4], Zibase[5], HomeSeer 3[6], Zipabox[7], IFTTT[8] and Tasker[9].

The simplest of these ECA based tools is probably **IFTT**. It is based on the action-trigger paradigm and has a very low threshold (graphical use interface, guided steps). The problem is it also has a low ceiling: it can only handle one trigger for one action. The figure 1 shows a IFTTT *recipe* that implement the use case UC 1.

Systems like **eeDomus** and **HomeSeer** offer more advanced capabilities, enabling end-user to combine multiples conditions and actions in a same rule. Because they are pure product of the industry, we do not have access to a full description of their system. However, as a result of a capitalization of years of experience in the domain of home automation, it is interesting to notice that the constructors added functionalities such as expressing duration for observed state (the temperature has been lower than 10 degrees for 5 minutes).

[1] `https://amiqual4home.inria.fr/projet-domicube/`
[2] `http:/www.eedomus.com/`
[3] `http://www.crestron.com/`
[4] `http://getvera.com/controllers/vera3/`
[5] `https://www.zodianet.com/`
[6] `http://www.homeseer.com/`
[7] `http://zipabox.domadoo.com/`
[8] `https://ifttt.com/`
[9] `http://tasker.dinglisch.net/`

eeDomus even refined ECA into ECAN (N standing for notifications), explicitly stating that notifications are semantically different from other actions (such as opening shutters).

Tasker is a tool to program automation on Android smartphones. It can use almost every input of the phone as a trigger and almost every output as an action. Although it has a high ceiling, it also has a high threshold. You have to know the app and its concepts to use it properly. Unlike its competitor Tasker distinguishes the types of actions and the types of triggers. It provides checkboxes and options to control the behavior of the automation regarding this aspect. Because Tasker is designed for power users, it allows its user to tune some very precises options peculiar to rules programming. With Tasker, we quickly reach the limits of the ECA paradigm when we want complex behaviors. When the number of rule grows, the predictability of the system falls.

Another solution, a little more far to smart homes than the rest is the graphical tools for programming such as **Scratch** or **Blockly**. These tools are basically a graphical representation of code. Therefore they have a high ceiling (almost as high as the underlying language). It is possible to lower their threshold by hiding parts of the language and by exposing methods from a custom made framework. This is done in several applications that use them like Blockly Games[10]. Since these tools aren't designed to control smart home but to teach programming, the properties and structures are general to programming and not specific to automation. It is worth noting that the automation system Zipato use Scratch for its end-user development.

4 Cascading Context Based Language

4.1 Context

The first concept to understand is the **context**. Almost all the features of the language revolve around this idea. A context is the association of a **selector** and an **instruction block**. The selector refers to events and states of devices and services, it determine if the instruction block should be taken into account by the system. The code sample 1 show the basic structure of a context.

```
SELECTOR {
  INSTRUCTIONS
}
```

Code sample 1: Structure of a context

4.2 Instructions and actions

Actions possible on the system are of three types : instantaneous, extended and sustained. These are defined in [Huang and Cakmak, 2015] :

- Instantaneous: do not change the state of the system, completed within one step and the system should be ready to make another instantaneous action at the next step (send a email)
- Extended: completed in a limited amount of time and then revert to its original state (brewing coffee).
- Sustained (maintained): involve changing the state of an actuator, the state doesn't revert back automatically (lit a lamp).

In the language, we use method calls to represent instantaneous and extended actions and variable affectation to represent maintained actions. Only one kind of instruction is allowed inside a instruction block depending on the type of selector of the context. A context with an event selector can only use method calls. An context with a state selector can only use variable affectations.

4.3 Selector

A selector is a expression very close to a boolean expression. It must always be assessable to true or false at any point in time. The selector can depict either a state as in the code sample 2 or an event as in the code sample 3.

```
Switch.isOn {
  Lamp.isOn: true;
}
```

Code sample 2: When the switch is on, the lamp is lit.

```
Button.onPressed {
  Phone.sendSMS(John, "The button has been pressed.");
}
```

Code sample 3: Each time the button is pressed, send a SMS to John.

State selector

Using a state selector implies that the values affected in the instruction block will be maintained to this value during the time that the selector holds up. In the code sample 2, we've shown a simple example with a device (the switch) that already supply a state. Obviously depending on the device, it could supply only events or both events and states. Because of that, we need to be able to make states from events.

From an event based device we can watch it as a state based device using the operator **until** as shown in the code sample 4. In this example, the TV is forced to be off between 9pm and 8am. The events 9pm and 8am have been used to define a state that is true between this two events.

```
Clock.at(21, 0) until Clock.at(8, 0) {
  TV.isOn: false;
}
```

Code sample 4: Between 9pm (21:00) and 8am (8:00) the TV stays off.

It is also possible to specify the state using a duration as shown in the code sample 5. Here, the state begin with the

[10] https://blockly-games.appspot.com/

motion sensor picking up a movement and is set to end 20 seconds after. In this time interval, the lamp in the garage will remain lit.

```
MotionSensor.onMovement for 20 seconds {
  GarageLamp.isOn: true;
}
```

Code sample 5: If they were a mouvement in the last 20 seconds, the garage lamp stays on.

The language also support classic booleans operators like **and** and **or**. They can be use with states, as shown in code sample 6.

```
// While the window or the front door is open the air conditioning stays off
// system
Window.isOpen or Frontdoor.isOpen {
  AirConditioning.isOn: false;
}

// While the window is open and it's above 20 C, the shutters stay closed
Window.isOpen and Thermometer.externalTemperature < 20 {
  Shutters.areClosed: true;
}
```

Code sample 6: *and* and *or* on states

Event selector

From the other side we can also watch a state based device as an event based device using the operator **during** as shown in the code sample 7. In this example, if the fridge door is let open for at least 20 seconds, the lamp will glow red until the fridge door is closed. The events are broadcast at the end of the time check, so when the fridge closes, the lamp will stay up for one more second. Note that using during with until, we actually build a state selector.

```
FridgeDoor.isOpen during 20 seconds until FridgeDoor.isClosed during 1 second {
  Lamp.isLit: true;
  Lamp.color: "red";
}
```

Code sample 7: If the fridge stays open for at least 20 seconds, lit the lamp red until the fridge stay closed for at least 1 second.

Because states have a starting event and a ending event, it's also possible to us a native device state as a trigger. This is shown in the code sample 8. Note the keyword `do` that signal a event selector. This keyword used on a state selector will consider the last event of the state: the ending event. `do before` tells the system to use the starting event and not the ending event.

And and **or** can also be used with events, as shown in code sample 9. Notice the keyword `within` that indicate the period of sensitivity.

```
// When the state isNight toggles to false, send a sms.
Clock.isNight do {
  Phone.sendSMS(John, "It's dawn.");
}

// When the state isNight toggles to true, send a sms.
Clock.isNight do before {
  Phone.sendSMS(John, "It's dusk.");
}
```

Code sample 8: When the state isNight ends or begins, send a sms

```
// When the front door or the back door open, send a sms
Frontdoor.opens or Backdoor.opens do {
  Phone.sendSMS(John, "Someone is in the house.");
}

// When both the front door and the back door open in a 5 seconds interval,
// sens a sms.
Frontdoor.opens and Backdoor.opens within 5 seconds do {
  Phone.sendSMS(John, "This is probably SWAT.");
}
```

Code sample 9: *and* and *or* on events

4.4 Context organization

The goal of the language is to avoid the errors of ECA, one of which is the accumulation of rules without hierarchy. In this language we provide several way to organize the contexts based on Allen's interval Algebra.

X takes place before Y ($X < Y$)

$$\underline{\quad X \quad} \qquad \underline{\quad Y \quad}$$

This relation can be achieved using the keyword `until` and the composition of selectors. This keyword used between two states X and Y will create a state between the ending event of X and the starting event of Y. The composition of selector uses the keyword `&` allowing the executions of actions in the middle of a selector. This is shown the code sample 10. The period of time between the end of X and the beginning of Y isn't limited. The code sample 11 is an example more concrete.

```
state1 {
  // Maintained actions during state1 (X)
} & until state2 {
  // Maintained actions during state2 (Y) (if X happened before)
}
```

Code sample 10: X takes place before Y

X meets Y (XmY)

$$\underline{\quad X \quad} \qquad \underline{\quad Y \quad}$$

```
X {
  // Maintained actions during X
} & until Y {
  // Maintained actions during Y (if X happened before)
}
```

Code sample 11: X takes place before Y

This relation means that the end event of one state should be the starting event of another. It's important to understand that in this case, we can't use two states from devices like in the previous case. We can't join states we don't create. We need to make a state using events and the keyword `until` as shown in the code sample 12. The code sample 13 is an example more concrete.

```
// CASE 1
event1 until event2 {
  // Maintained actions between event1 and event2 (X)
} & until event3 {
  // Maintained actions between event2 and event3 (Y)
}

// CASE 2
state1 {
  // Maintained actions during state1 (X)
} & until event1 {
  // Maintained actions between end of state1 and
       event1 (Y)
}
```

Code sample 12: X meets Y

```
// From 6:00 to 6:30 set lamp light intensity to 10
// Then from 6:30 to 7:00 set lamp light intensity to 20
// Then from 7:00 to 7:30 set lamp light intensity to 30
Clock.at(6, 0) until Clock.at(6, 30) {
  NightLamp.intensity: 10;
} & until Clock.at(7, 00) {
  NightLamp.intensity: 20;
} & until Clock.at(7, 30) {
  NightLamp.intensity: 30;
}
```

Code sample 13: Progressive light to wake up

X during Y (XdY)

$$\frac{X}{Y}$$

To achieve this relation we simply put a context into another. This setup means the subcontext can only be activated while it's parent in in activity. This is shown in the code sample 14. The code sample 15 is an example more concrete.

X starts Y (XsY)

$$\frac{X}{Y}$$

We achieve this relation by using the same method plus the keyword `begins` that represent the starting event of the

```
state1 {
  // Maintained actions during state1 (Y)

  state2 {
    // Maintained actions during state1 and state2 (X)
  }
}
```

Code sample 14: X during Y

```
TV.isOn {
  TV.maxVolume: 100;

  Phone.isInCall {
    TV.maxVolume: 10;
  }
}
```

Code sample 15: While the TV is on, when someone is on the phone, the TV maximum volume is restricted.

parent context. This is shown in the code sample 16. The code sample 17 is an example more concrete.

```
state1 {
  // Maintained during state1 (Y)

  begin until event1 {
    // Maintained between start of state1 and event1 or
         end of state1 (X)
  }
}
```

Code sample 16: X starts Y

```
Phone.isInCall {
  begin until Phone.onPickUp {
    Phone.isRinging: true;
  }
}
```

Code sample 17: When someones call, until the phone isn't picked up, the phone rings.

X finishes Y (XfY)

$$\frac{X}{Y}$$

As for above, we use the keyword `end` that represent the ending event of the parent context. This is shown in the code sample 18. The code sample 19 is an example more concrete that show a simple security system.

4.5 Priority and Predictability

On a complex system with a lot of contexts simultaneously running, it must remains simple to ascertain which context should prevail over the others when both wish to act upon the same device. Unlike ECA we wish to be able to tell without doubt at any point of time which context is acting upon which

```
state1 {
  // Maintained during state1 (Y)

  event1 until end {
    // Maintained actions between event1 and the end of
        state1 (X)
  }
}
```

Code sample 18: X finishes Y

```
House.isLockedDown {
  MotionSensor.onMovement until end {
    SecuritySystem.isInAlert: true;
  }
}
```

Code sample 19: When the house is under locked down, from the first movement detected from the end of lockdown, turn on the alert system

device. To do so, whether a context can act upon a device must me independent from the order of execution and of the successions of events. In order to achieve that we set up rules that put into order all the *competing* contexts.

Implication

If the selector of a context A implies the selector of a context B, then A has priority over B. For example if A is `X and Y and Z` and B is `X and Y`, then the context of A has priority.

This is compatible with the notion of inclusion. A subcontext is active only if its parent is active, so the subcontext has priority over its parent. Generally speaking, any context has priority over any of its ancestors (parents of parents). This successions of nested contexts is called the **cascade**.

Complexity

If a selector of a context A which is more *complex* than the selector of a context B, then A has priority over B. The notion of complexity is vague but can be rationalize using several indicators. For example, the numbers of operands : the more operands and operators a selector contains, the more priority it gets. This can be useful in the case of sequential context like the code sample 11, where this context should have priority over a context tha only uses one of the states in its selector.

Order

To avoid any case of two contexts having the same level of priority we need an arbitrary rule that will always give priority to one. Here, the earlier processed by the system a context is the more priority it gets.

4.6 The Cascade

As mention earlier it is possible to put contexts inside the instruction blocks of other contexts. This principle allow a very powerful organization. The contexts are classified on a tree with the default context($\star$) at its root. When a context is disable (the selector is false), all the descending contexts are also disabled. The code sample 20 shows a simple example of powerful organization using this principle.

In this example, we control the shutters of the house. IN the default context we set the shutters to be opened. Then, we define different behavior for night and day. At night the shutters are supposed to be closed, except it the temperature rises above 40C. In the day the shutters are opened, except if the light of the sun is too bright.

```
* {
  Shutters.areClosed: false;

  Clock.isNight {
    Shutters.areClosed: true;

    InternalThermometer.temperature > 40 {
      // Abnormal temperature, risk of fire
      Shutters.areClosed: false;
    }
  }

  Clock.isDay {
    Shutters.areClosed: false;

    LightMeter.lux > 500 during 2 minutes until
    LightMeter.lux < 500 during 2 minutes {
      // Strong light
      Shutters.areClosed: true;
    }
  }
}
```

Code sample 20: Simple cascade example

4.7 Grammar

The BNF grammar in the figure 2 should provide further details on the language.

4.8 Abstraction

The code sample 21 shows the abstraction capabilities of the language. We can create class that can be instantiated. From several low level devices, we can make a higher level, more user friendly virtual device. Advanced user are then able to build devices that act like applications and then share them with non experienced users. This, as mention in the section 2, allows users to read other people codes and learn new ways of using their home automation system.

5 Implementation

We've developed a prototype using the NodeJS environment. This prototype is able to build a coherent system with devices and contexts and to maintain the system in a proper state depending on the contexts and the input. To simulate the home environment we've simulated 10 lamps than can be either lit or turned off, 5 boolean switches and 5 buttons that send punctual events into the system. We've also implemented a simple user interface that allows the user to explore the devices and the contexts to see their status.

The source code is available on at .

5.1 Devices

Devices in the system are abstracted to a class that holds **channels**. Each channel represent a value applied to the device. For example: the device Lamp will have at least 1 channel : isOn. Depending on the lamp, the device could have

other channels like color, intensity, heat, direction, ... A channel can be a state (isOn) or an event (incomingMail). Channels can be either only readable (Calendar.isSummer) by the system or also writable (Lamp.isOn). Writable channels have to manage a stack of values that are pushed by contexts that want to change its value.

5.2 Priority stack, implementation of the cascade

To implement the priority in the prototype, we used an ordered stack. Each context which has to act upon a device, subscribes to the device's stack. The device then process which context has the priority and then change it's value accordingly. The same process is done when a context unsubscribes from the device's stack. Using the method, the device knows which devices wish to change its value and can easily adapt when a contexts is disabled.

5.3 Tests

In order to make a resilient software, we've implemented tests executions that can assert the accuracy of the software. Using the test framework Mocha and the assertion framework Chai, we've implemented tests for each operator available in the software. The test, launch a specific set of contexts and then runs a scenario in which buttons are pressed and lights turn on and off.

Because the system is strongly dependent of time, we've two clocks for the system. The first one is simply plugged to the underlying system clock, time is synchronized with reality. The second one is fixed, time doesn't automatically flow. The system or the programmer has to manually advanced in time. To do so he has two method he can call : `step()` will leap forward to the next event programmed in the clock (a selector using the keyword `for`, for example) and `progress(time)` (which take a duration in milliseconds) will leap forward the specified amount of time. This simulated time allow the tests to simulate times and test scenarios with big amount of time.

6 Conclusion

In this work we have partially designed a language which is centered on states and durations. It has a expressiveness at least a important as ECA languages since it includes the ECA paradigm. This language is designed around the concept of contexts which is a direct application of Allen's interval algebra. By construction it makes explicit *non sequitur* since the main way to express contexts is via states. The clear separation between the several types of actions (maintained, extended and punctual) and their associations with states and event, resolve the problems highlighted in [Huang and Cakmak, 2015]. It has a low threshold as the simple rules don't require handling any complex concepts. It also has a high ceiling because of its abstraction capabilities.

We've implemented a prototype that can operate devices according to a tree of contexts organized as a cascade and that have advanced testing capabilities.

Although we thoroughly tested the language capabilities with a wide variety of use cases, they is still a lot of work to do to verify its integrity and consistency. We also need to test this language and its underlying logic with real end-users. Since in this work we do not provide a preferred way of representing the language for end-users, this has to be done and also tested with real end-users.

Acknowledgements

I would like to express my thanks to my intership supervisors Dr. Alexandre Demeure and Dr. Sybille Caffiau for giving me the opportunity to work with them and for their constant support. I would like to thanks Jean-Jacques Parrain for his useful remarks and insights.

References

[Brush *et al.*, 2011] AJ Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home automation in the wild: challenges and opportunities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2115–2124. ACM, 2011.

[Burnett and Scaffidi, 2011] Margaret M Burnett and Christopher Scaffidi. End-user development. *Encyclopedia of Human-Computer Interaction*, 2011.

[Davidoff *et al.*, 2006] Scott Davidoff, Min Kyung Lee, Charles Yiu, John Zimmerman, and Anind K Dey. Principles of smart home control. In *UbiComp 2006: Ubiquitous Computing*, pages 19–34. Springer, 2006.

[Demeure *et al.*, 2015] Alexandre Demeure, Sybille Caffiau, Elena Elias, and Camille Roux. Building and using home automation systems: a field study. In *End-User Development*, pages 125–140. Springer, 2015.

[Frye *et al.*, 2013] Andrew Frye, Michel Goraczko, Jie Liu, Anindya Prodhan, and Kamin Whitehouse. Circulo: Saving energy with just-in-time hot water recirculation. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, pages 1–8. ACM, 2013.

[García-Herranz *et al.*, 2010] Manuel García-Herranz, Pablo A Haya, and Xavier Alamán. Towards a ubiquitous end-user programming system for smart spaces. *J. UCS*, 16(12):1633–1649, 2010.

[Holloway and Julien, 2010] Seth Holloway and Christine Julien. The case for end-user programming of ubiquitous computing environments. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 167–172. ACM, 2010.

[Huang and Cakmak, 2015] Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 215–225. ACM, 2015.

[Lieberman *et al.*, 2006] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-user development: An emerging paradigm*. Springer, 2006.

[Lucci and Patern, 2015] Gabriella Lucci and Fabio Patern. Analysing how users prefer to model contextual event-action behaviours in their smartphones. In *End-User Development*, pages 186–191. Springer, 2015.

[Mennicken *et al.*, 2014] Sarah Mennicken, Jo Vermeulen, and Elaine M Huang. From today's augmented houses to tomorrow's smart homes: new directions for home automation research. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 105–115. ACM, 2014.

[Patern, 2013] Fabio Patern. End user development: Survey of an emerging field for empowering people. *ISRN Software Engineering*, 2013, 2013.

[Ur *et al.*, 2014] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014.

A Benchmark use cases

UC 1 "*Notify me every day at 10:03 am.*"

UC 2 "*When I do a specific action, a specific device turns on. When I do the action again, the device turns off.*"

- **UC 2.1** "*If I push a button turn the lamp on, if I push it again, turn if off*"
- **UC 2.2** "*When I receive a sms from a specific contact, turn on WIFI, when I receive another one turn it off.*"

UC 3 "*One device should always be off when a second is on*"

- **UC 3.1** "*The TV volume should always be off when the phone is on.*"

UC 4 "*One device must only be on when a second is on*"

- **UC 4.1** "*The ambilight must only be on when the TV is on.*"

UC 5 "*When the temperature goes below 0C for at least 10 minutes, turn the heat system on. When the temperature goes above 0C for at least 10 minutes, turn the heat system off.*"

UC 6 "*Build a Domicube.*"

B Figures

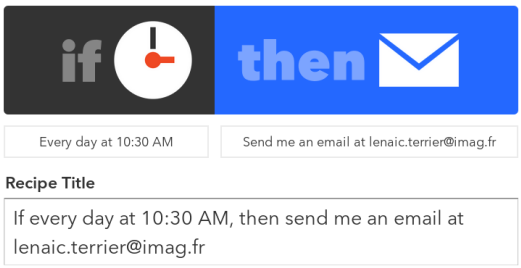


Figure 1: Usecase 1 with IFTTT

```
class Domicube(acc, gyro) {
  States {
    face:          enum{none, 1, 2, 3, 4, 5, 6};
    rotation:      float;
    isRotating:    boolean;
    isFreeFalling: boolean;
  }

  Events { }
  Variables { }

  Instructions {
    * {
      face: 0;
      rotation: 0;
      isRotating: false;
      isFreeFalling: false;

      acc.X = 0 and acc.Y = 0 and acc.Z = 0 {
        isFreeFalling: true;
      }

      acc.X = 0 and acc.Y = 1 and acc.Z = 0 {
        face: 1;
      }

      acc.X = 0 and acc.Y = -1 and acc.Z = 0 {
        face: 6;
      }

      acc.X = 1 and acc.Y = 0 and acc.Z = 0 {
        face: 2;
      }

      acc.X = -1 and acc.Y = 0 and acc.Z = 0 {
        face: 5;
      }

      acc.X = 0 and acc.Y = 0 and acc.Z = 1 {
        face: 3;
      }

      acc.X = 0 and acc.Y = 0 and acc.Z = -1 {
        face: 4;
      }

      face = 1 or face = 6 {
        gyro.Y > 10 or gyro.Y < -10 {
          isRotating: true;
          rotation: gyro.Y;
        }
      }

      face = 2 or face = 5 {
        gyro.X > 10 or gyro.X < -10 {
          isRotating: true;
          rotation: gyro.X;
        }
      }

      face = 3 or face = 4 {
        gyro.Z > 10 or gyro.Z < -10 {
          isRotating: true;
          rotation: gyro.Z;
        }
      }
    }
  }
}
```

Code sample 21: Domicube

⟨stateSel⟩ ⊨ ⟨state⟩ (1)

⟨eventSel⟩ ⊨ ⟨event⟩ `do` | ⟨state⟩ `do` | ⟨state⟩ `do before` (2)

⟨state⟩ ⊨ ⟨event⟩ `until` ⟨event⟩ | ⟨state⟩ `until` ⟨event⟩ | ⟨state⟩ `until` ⟨state⟩ (3)

⟨state⟩ ⊨ ⟨state⟩ `or` ⟨state⟩ | ⟨state⟩ `and` ⟨state⟩ | ⟨state⟩ `for` ⟨time⟩ ⟨unit⟩ (4)

⟨state⟩ ⊨ *State of a device* (5)

⟨event⟩ ⊨ ⟨state⟩ `during` ⟨time⟩ ⟨unit⟩ | *Event of the system* (6)

⟨time⟩ ⊨ $\mathbb{N}$ (7)

⟨unit⟩ ⊨ `millis` | `seconds` | `minutes` | `hours` | `days` | `month` | `years` (8)

⟨instruction⟩ ⊨ ⟨deviceState⟩ `:` ⟨value⟩ `;` (9)

⟨methodCall⟩ ⊨ ⟨device⟩ `.` ⟨method⟩ `(` ⟨params⟩ `)` `;` (10)

⟨value⟩ ⊨ $\mathbb{R}$ | ⟨string⟩ (11)

⟨device⟩ ⊨ ⟨ascii⟩ (12)

⟨string⟩ ⊨ `"` ⟨ascii⟩ `"` (13)

⟨param⟩ ⊨ ⟨ascii⟩ (14)

⟨method⟩ ⊨ ⟨ascii⟩ (15)

⟨deviceState⟩ ⊨ ⟨ascii⟩ (16)

⟨ascii⟩ ⊨ `a` ... `z` (17)

⟨stateBlock⟩ ⊨ ⟨stateSel⟩ { ⟨instructions⟩ ⟨stateBlocks⟩ ⟨eventBlocks⟩ } (18)

⟨eventBlock⟩ ⊨ ⟨eventSel⟩ { ⟨methodCalls⟩ ⟨eventBlocks⟩ } (19)

⟨stateBlocks⟩ ⊨ ⟨stateBlock⟩ ⟨stateBlocks⟩ | ϵ (20)

⟨eventBlocks⟩ ⊨ ⟨eventBlock⟩ ⟨eventBlocks⟩ | ϵ (21)

⟨methodCalls⟩ ⊨ ⟨methodCall⟩ ⟨methodCalls⟩ | ϵ (22)

⟨instructions⟩ ⊨ ⟨instruction⟩ ⟨instructions⟩ | ϵ (23)

Figure 2: BNF Grammar

Méthodologie d'analyse de corpus de flux RSS multimédia

Baille Mathieu
Supervisé par :
Vincent Jean-Marc
Studeny Angelika

ABSTRACT

Is the propagation of the media information can be described by a model ? The science of medias can be helped by different tools : internet and the RSS feed. The goal of analyzing this data is to create a predictive model to anticipate the spread of an information.

We usually talk about propagation in epidemiological domain and there are different ways to modeling the spread of a virus : deterministic way and stochastic way.

To apply our model, we need to take a sample who represents a precise event and observe its characteristics. We chose the Ebola crisis because of his precise localization and the possibility to base on several other research. And we chose the Ferguson event because of his precise depart point in term of time and localization.

Actually, we found a correlation between the propagation of the Ebola information and an epidemiological deterministic model, but the experimental protocol needs to be applied in some other sample to be proved.

In the end, the objective is to compare the deterministic approach to the stochastic model and define a virality factor of an event. The concept of rumor and deformation of the information can be added to this model and the possibility to find changes in a sample.

I. INTRODUCTION

L'apparition d'internet a permis la propagation de flux d'information, en particulier ceux générés par les médias. La science des médias s'intéresse à la description de tel flux et cherche à comprendre leurs formes et leurs structures afin d'expliquer les phénomènes sociaux, économiques, politiques... Le projet Géomédia, composé de chercheurs en sciences des médias, en géographie et en informatique a pour objectif de construire des modèles d'événements médiatiques à partir d'un corpus fourni par les journaux sous forme de flux RSS. Un flux RSS (Really Simple Syndication) est un flux au format XML permettant un système d'abonnement afin de récupérer automatiquement une partie ou la totalité d'un article nouvellement crée ou mis à jour. La récupération des données peut se faire à l'aide d'un logiciel agrégateur, celui-ci nous est fourni par le projet Géomédia sous la forme d'un serveur TomCat procédant à la captation de nouveaux articles toutes les heures. L'intérêt de cette méthode est de permettre de suivre facilement un ou plusieurs flux d'informations sans devoir se rendre manuellement sur le site à l'origine de l'article. À partir de ces données, la thématique établit est l'étude des relations internationales avec un intérêt particulier pour les structures spatiales (relation de voisinage, distance, liens culturels, conflits d'intérêt) et temporelles (aspect historique).

Un événement médiatique est un ensemble de faits remarquables détectés dans un ensemble de flux média. Il est décrit comme un point de rupture ou sursaut médiatique et détaillé chronologiquement avec un avant et un après. Son importance peut être relevée quantitativement grâce au changement de médiatisation de l'événement à partir des relevés journaliers. On peut décrire son ampleur spacio-temporelle grâce aux zones géographiques touchées (national, international, mondial) ainsi que la période sur laquelle il s'étale. Si l'événement est localisé, on parle de voisinage dans la propagation de l'information. Comme exemples étudiés dans le cadre de l'ANR, on peut citer la guerre civile en côte d'ivoire en décembre 2010, l'accident nucléaire du Fukushima en mars 2011 ou encore plus récemment l'épidémie Ebola en fin d'année 2014, chacun de ces événements a été fortement médiatisés dans les pays touchés et relayés par beaucoup de journaux.

Toutes ces informations médiatiques regroupées constituent notre corpus d'articles. Ceux-ci proviennent de 300 flux dans huit langues différentes, ils sont répartis en plusieurs catégories : général, international, une. Le nombre de pays représentés est au nombre de 59, étalés sur les six continents. la plage temporelle couverte par nos articles est du premier janvier 2014 au premier mars 2016 et comptabilise trois millions d'articles. On décrit un article grâce aux caractéristiques suivantes : un identifiant unique, un titre, une description, un contenu, une catégorie, un lien url et les dates de récupération et de publication. À cela on ajoute les caractéristiques du flux à l'origine de l'article : un identifiant unique, un type de flux et le nom du journal. Un système de taggage par sujet via des dictionnaires permet de signaler l'emploi de mots clés, par exemple, le cas *Ebola*

regroupe 90.000 articles.

Avec une grande quantité de données, on peut utiliser des méthodes statistiques afin de les analyser. Pour définir l'environnement de travail, on caractérise les propriétés de l'échantillon total pour permettre d'avoir un point de vue global du comportement des données. On choisit ensuite un sous-échantillon atypique (événement médiatique) sur lequel on applique les outils statistiques. L'intérêt de ce choix est de commencer sur une échelle plus petite et d'élargir l'analyse sur de plus grande quantités de données. Cette approche permet la création de modèles prédictifs et a pour intérêt de pouvoir être appliqué sur plusieurs sous-échantillons différents. Pour créer un modèle, on peut essayer de chercher des schémas récurrents, effectuer des courbes de tendance, appliquer une régression linéaire ou se baser sur un modèle pré-existant.

Afin de trouver un modèle expliquant les données, nous proposons de les analyser en terme épidémiologique. En effet, on décrit souvent l'information comme un virus lorsque l'on parle de sa propagation. Il semble alors intéressant de savoir si ces dires sont fondés. Il existe deux classes de modèles épidémiologiques. Les modèles déterministes cherchent à décrire l'évolution de la population par des équations mathématiques (équations différentielles, équations aux dérivées partielles). Les modèles stochastiques quant à eux, sont basés sur les processus aléatoires (chaîne de Markov). Un autre mécanisme est basé sur le concept de la rumeur, soit la déformation et la propagation d'informations déformées. Dans le cadre de ce travail de Magistère, il a été décidé de commencer l'étude des modèles déterministes car la question posée par les thématiciens des modèles (géographe et sciences des médias) est déterministe. Les modèles stochastiques seront étudiés ultérieurement.

Est-ce que la propagation de l'information médiatique peut être décrite par un modèle épidémiologique ?

Les événements constituant les échantillons à analyser sont l'épidémie *Ebola* et les manifestations de *Ferguson*, ce choix est motivé par une connaissance des bornes temporelles, de la localisation de sa zone de propagation ainsi que du travail déjà effectué par des pairs. Pour le cas de l'épidémie *Ebola*, l'intérêt second est l'étude de la médiatisation de la propagation d'un virus, ce qui augmente la facilité de raisonnement dû à la proximité avec l'épidémiologie. Pour le cas *Ferguson*, le fait d'avoir un point de départ clair qui est le 9 août 2014 dans la ville de Ferguson permet de fixer une première borne temporelle et géographique claire. Le fait de choisir deux événements distincts permet de comparer leurs analyses et d'estimer la différence de leurs propagations.

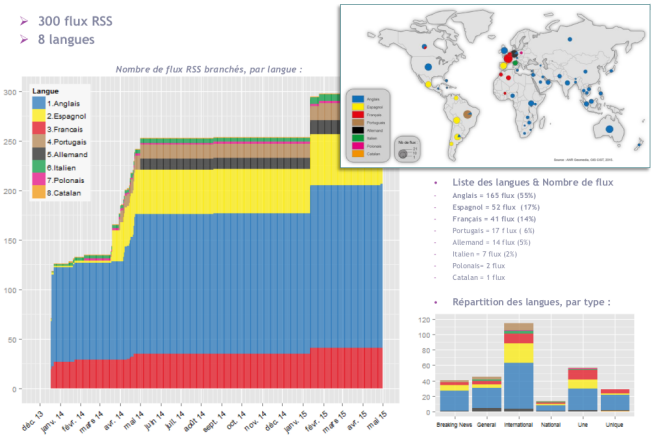
Notre méthode d'analyse comporte trois étapes. La première est de trouver une forme de modèle épidémiologique déterministe. La seconde est de déterminer l'adéquation entre nos données et le modèle, pour cela il faut définir un critère de qualité et d'acceptation de l'hypothèse. Enfin, on décide d'accepter ou de rejeter le modèle.

Actuellement, les résultats semblent montrer une adéquation entre un modèle déterministe et les jeux de données. Afin d'accepter l'hypothèse selon laquelle la propagation de l'information se comporte de la même façon qu'un virus, il faut appliquer le protocole précédent à d'autres événements de types et d'envergures différents. De plus, il faut renforcer le modèle par des contraintes tierces non prises en compte initialement. On peut par exemple citer la récurrence d'un événement (coupe du monde, élection), l'imbrication d'événements similaires (séismes), l'importance relatives des événements entre eux (séismes et magnitudes), la distance physique de la propagation ou encore la déformation de l'information par le mécanisme de la rumeur.

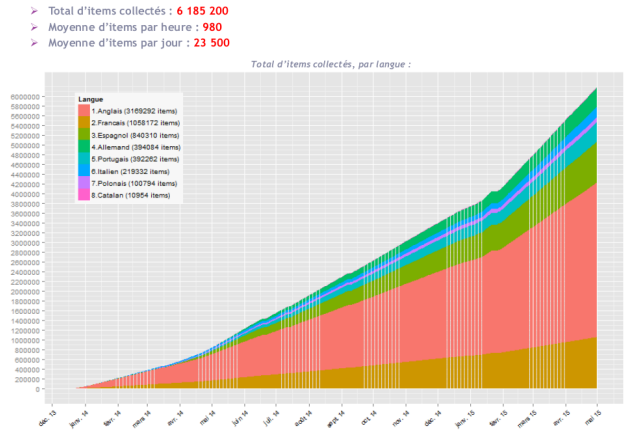
II. Définition du contexte

A. *Description globale*

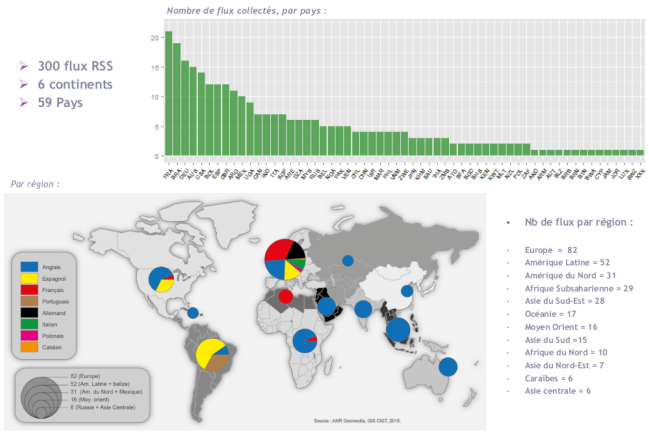
Le mécanisme de captation des flux RSS a commencé à partir de 2014, la base de donnée comprenait alors plus d'une centaine de flux. Il en a été ajouté par pallier afin d'avoir un nombre suffisant pour constituer au mieux un résumé du paysage géopolitique. Le nombre de flux actuel est de 300 dans huit langues différentes.



La base des articles croit de manière linéaire et comprend actuellement plus de onze millions articles. L'ajout de nouveaux flux de manière ponctuelle a permis une augmentation de la pente de la droite décrivant le nombre d'articles.



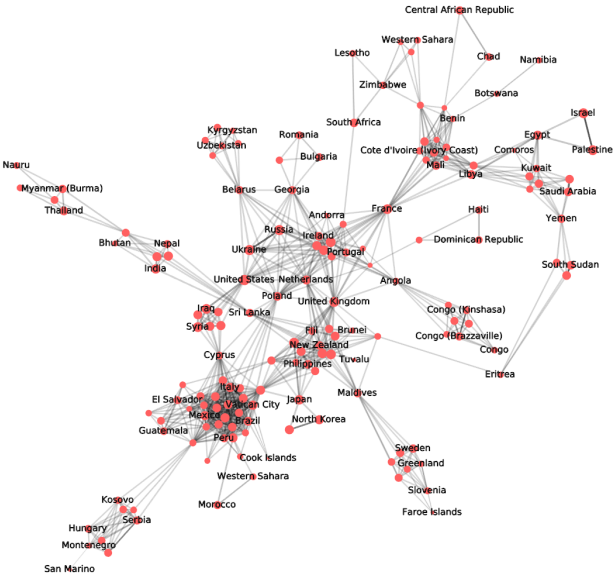
L'origine de ces articles nous vient de 59 pays répartis sur les six continents, le but de cette répartition est d'avoir plusieurs points de vue sur un événement. On peut alors observer les différents impacts sur un conflit interne à un pays et les différents opinions vues de l'extérieur.



Les articles sont compartimentés en plusieurs catégories qui correspondent à leur orientation : vers l'intérieur du pays ou vers l'extérieur, informations de dernière minute ou générales. Il peut toutefois que certains articles se retrouvent dans plusieurs catégories, ils seront alors considérés comme deux articles différents.

Unique	General	Une	Breaking news	International	National
Un seul flux RSS est proposé par le journal/média	"Home"	"Top Head Lines"	"Ultimas Noticias"	"International"	"National"
	"Home page"	"Portada"	"Dernières News"	"Internacional"	Nom du pays
	"All News"	"Top stories"	"Latest News"	"Monde"	
	"Todas las noticias"	"Une"	"Hot News"	"Mundo"	
		"Titulares"	"News"		
		"Actu"	"Fil info"		
		"Actualidad"			

On peut, grâce aux articles citant les autres pays, estimer un mapping de l'information donnant un aperçu des liens entre les pays ainsi que les différentes *communautés*. Celles-ci sont définies comme des blocs d'entités connectées entre elles et faiblement connectées avec le reste du réseau. Le but de ce mécanisme est de simplifier le réseau et de chercher à savoir si ces groupes ont du sens dans la thématique ou d'un point de vue géopolitique.



B. *Forme d'un article*

Un article est composé de plusieurs champs :
– ID Item : l'identifiant unique de chacun des items
– Titre : le titre de l'article
– Description : l'article avec les liens des images
– Catégorie : indique le contenu général de l'article, référencé par le journal, par exemple *Sport*
– Lien : l'url de l'article
– Date Recup : la date de récupération par le serveur
– Date publication : la date que l'éditeur de l'article note (peut être vide ou un autre format que GMT)
– ID flux : l'identifiant unique du flux RSS
– typeFlux : indique le contenu général du flux RSS, référencé par le serveur, par exemple *International*
– Journal : l'identifiant du journal

C. *Système de taggage*

Le système de taggage est mis à disposition sous forme de script utilisant un dictionnaire. Celui-ci regroupe plusieurs mot-clés représentatifs de l'événement en cours d'étude. Par exemple, pour l'épidémie *Ebola*, les mots peuvent être : ebola, ebolafieber, ebolavirus. Pour les manifestations de *Ferguson*, les mots peuvent être : Ferguson, Michael Brown, Darren Wilson. Le choix de ces mot-clés est important et doit être assez précis pour ne pas englober d'autres potentiels événements. Il se peut toutefois arriver qu'un nouveau mot-clé synonyme arrive en cours d'événement, c'est pourquoi il faut rester

attentif aux changements et ajouter un mécanisme de détection (voir concept de la rumeur).

III. Approche épidémiologique

Pour rappel, l'épidémiologie est l'étude des facteurs influant sur la santé et les maladies de populations. Elle se rapporte à la distribution, la fréquence ainsi que la force des états pathologiques.

Les approches utilisées en épidémiologie reposent au départ sur une analyse empirique des données, celle-ci regroupent les individus d'une population. Puis, des modèles analytiques sont proposés, en général, basés sur des équations différentielles ayant des paramètres explicatifs (taux de natalité, mortalité...). Le livre Epidemic Modelling An Introduction - D. J. Daley - 2001 regroupe plusieurs modèles d'analyse de propagation épidémiologique. On va étudier l'un des modèles dit déterministe.

On va définir trois états pour les éléments de notre population (on considère qu'un malade se soigne ou meurt instantanément) :

- susceptible : état de base de la population, permet d'entrer dans tous les autres états et de contracter la maladie
- immunisé : état potentiellement acquis après s'être soigné de la maladie, ne peut plus contracter la maladie
- mort : état atteint de manière naturelle ou à cause de la maladie

On va définir l'évolution de la population avec mort naturelle de la manière suivante :

$$\dot{\xi}(t) = -\mu(t)\xi(t) \tag{1}$$

Avec :
- $\dot{\xi}(t)$ l'évolution de la population en fonction du temps (dérivée partielle)
- $\mu(t)$ le taux de mortalité de base de la population (sans la maladie)
- $\xi(t)$ la population à un instant t

On manipule l'équation de la manière suivante :

$$\begin{aligned}
\dot{\xi}(t) &= -\mu(t)\xi(t) \\
-\mu(t) &= \frac{\dot{\xi}(t)}{\xi(t)} \\
-\int_0^t \mu(u)du &= \int_0^t \frac{\dot{\xi}(u)}{\xi(u)}du \\
&= [\ln(\xi(u))]_0^t \\
&= \ln(\xi(t)) - \ln(\xi(0)) \\
e^{(-\int_0^t \mu(u)d(u))} &= e^{(\ln(\xi(t))-\ln(\xi(0)))} \\
&= \frac{e^{(\ln(\xi(t))}}{e^{(\ln(\xi(0))}} \\
&= \frac{\xi(t)}{\xi(0)}
\end{aligned}$$

Finalement on obtient :

$$\xi(t) = \xi(0)e^{(-\int_0^t \mu(u)du)}$$

A partir de cette étape, on introduit $M(t)$ correspondant au hasard cumulatif, il est définit comme tel :

$$M(t) = \int_0^t \mu(u)du$$

On ajoute à la mort naturelle, la mort par la maladie défini par β, la constante d'infection :

$$\dot{x}(t) = (-\mu(t) + \beta)x(t)$$

Avec :
- $\dot{x}(t)$ l'évolution de la population en fonction du temps (dérivée partielle)
- $x(t)$ la population à un instant t
- β la constante d'infection de la maladie, soit la possibilité de tomber malade, si $\beta = 0$, on revient au modèle précédent.

De la même manière que précédemment :

$$x(t) = x(0)e^{(-\int_0^t \mu(u)du)}e^{(-\beta t)}$$

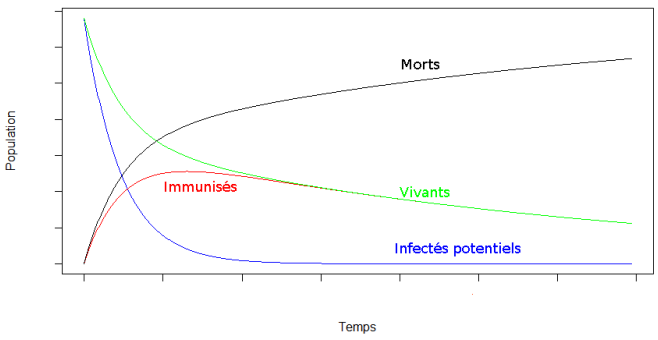
On ajoute maintenant les éléments de la population qui deviennent immunisés après avoir été infectés :

$$z(t) = \xi(0)e^{-M(t)}(1-\alpha)(1-e^{\beta t})$$

Avec :
- $z(t)$ la population des immunisés à un instant t
- $\xi(0)e^{-M(t)}$ correspondant à la mort naturelle d'un immunisé
- $(1-\alpha)(1-e^{\beta t})$ correspondant à la probabilité de devenir immunisé après avoir survécu à la maladie La population totale est donc équivalente à $x(t) + z(t)$, on la nomme $\xi_\beta(t)$ et est décrite par :

$$\xi_\beta(t) = \xi(0)e^{-M(t)}[e^{(-\beta t)} + (1-\alpha)(1-e^{\beta t})]$$

Afin de visualiser le modèle, on peut le représenter graphiquement :



IV. Application du modèle aux données

A. Comparaison du modèle avec les données

Notre objectif est maintenant de relier nos modèles déterministes à nos données. Pour ce faire, on a tout d'abord besoin de spécifier à nouveau les variables importantes d'un modèle épidémiologique :

- l'ensemble de la population
- la population morte
- la population immunisée $z(t)$
- la population susceptible d'être infectée $x(t)$
- la population des vivants $\xi_\beta(t)$
- le taux de mortalité $\mu(t)$
- la constante de viralité β
- la constante d'immunisation α

Il nous faut alors relier ces variables à la propagation de l'information médiatique et à la base de données des articles à notre disposition.

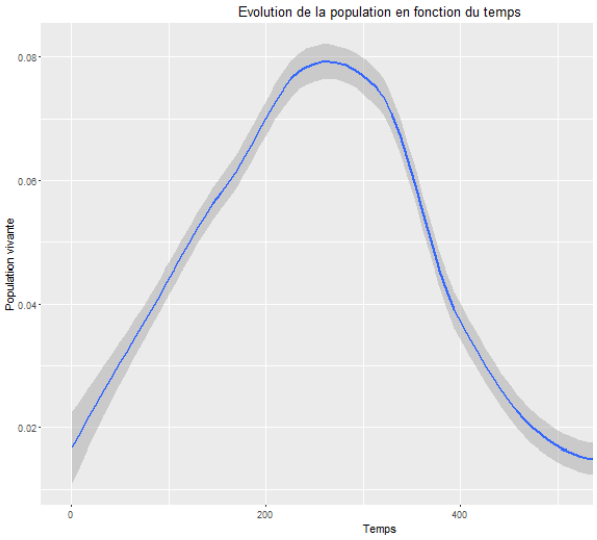
L'ensemble de la population correspond à la totalité des articles. Une population à un instant t est déterminé par à un nombre d'article sur une période de temps choisie, c'est à dire, une heure, un jour, un mois... Le choix de la granularité dépendra de la période sur laquelle s'étale l'événement.
La population morte correspond aux articles ne traitant pas de l'événement.
La population immunisée correspond aux articles traitant de l'événement.
La population susceptible d'être infectée correspond aux articles qui n'ont pas parlé de l'événement mais qui étaient susceptibles de le traiter.
La population des vivants correspond à la somme des immunisés et des susceptibles.
La constante de viralité β correspond à la probabilité qu'a un article dans l'état *susceptible* d'en parler.
La constante d'immunisation α correspond à la probabilité de traiter encore du sujet après en avoir déjà parlé.

B. Régression linéaire et corrélation

Étant donné que la population des articles parlant de l'événement sont les plus simples à détecter, nous allons les relier à la population des immunisés.

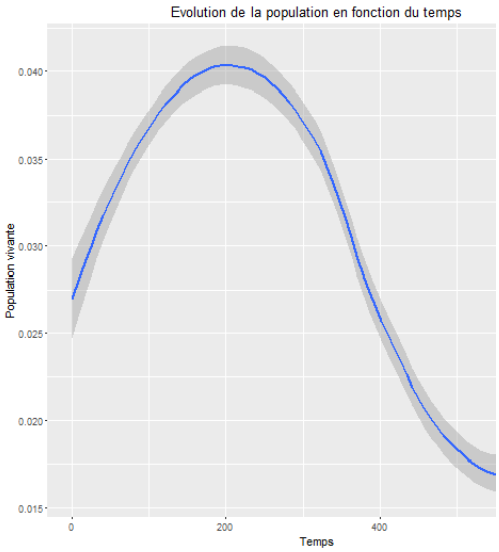
Pour estimer la corrélation entre un modèle épidémiologique et nos données, nous avons besoin de représenter graphiquement le nombre d'articles par jour traitant de l'événement. Ces courbes ont été obtenue grâce à une méthode de régression linéaire reposant sur la méthode des k plus proche voisins se nommant *LOESS* (LOcal regrESSion).

Pour l'épidémie Ebola :



Pour étudier l'intensité de la liaison entre les modèles épidémiologique et nos données, nous avons utilisé le coefficient de corrélation linéaire de Bravais-Pearson. Nous avons chercher à trouver la corrélation maximum en faisant varier les variables taux de mortalité $\mu(t)$, constante de viralité β et constante d'immunisation α. Celui-ci atteint un maximum de 0.76 avec $\mu(t) \simeq 1$, $\beta \simeq 0.515$ et $\alpha \simeq 1.92$.

Pour les manifestations de Ferguson :



La corrélation pour les manifestations de Ferguson atteint un maximum de 0.62 avec $\mu(t) \simeq 0.6x$, $\beta \simeq 2.52$ et $\alpha \simeq 0.01$.

C. Conclusion

Le coefficient de corrélation de Bravais-Pearson est compris entre -1 et 1. On dit qu'une corrélation est faible si elle est comprise entre -0.5 et 0.5. Si elle est comprise entre -1 et -0.5, on dit que cette corrélation est fortement négative, dans le cas où elle se trouve entre 0.5 et 1, on dit qu'elle est fortement positive.

Dans nos deux jeux de données, la corrélation avec

le modèle est fortement positive, ce qui va dans le sens de l'argument que la propagation de l'information médiatique peut être expliquée par un modèle épidémiologique.

Or, le fait que deux variables soient *fortement corrélées* ne démontre pas qu'il y ait une relation de causalité entre l'une et l'autre, il nous faut donc accepter ce modèle jusqu'à démontrer son inefficacité ou sa désuétude face à un autre modèle. Il nous faudra donc tester ce modèle sur d'autres événements afin d'éprouver son efficacité.

V. Perspectives

Pour rappel, l'intérêt d'expliquer les données par un modèle est de constituer en cours d'événement, un modèle prédictif de l'évolution future de la propagation de l'information. Dans le cas où l'événement serait déjà fini, on peut lui attribuer un facteur de viralité dépendant des valeurs α et β correspondant aux taux permettant la corrélation la plus forte. L'intérêt est alors de pouvoir jauger un événement médiatique et le comparer à un second, pour ce faire, il est important que les événements soient indépendants.

En effet, la possibilité de trouver des facteurs de viralités proches pour des événements identiques récurrents est une piste à suivre. De même pour des événements similaires mais ayant des caractéristiques variables, par exemple les magnitudes sur des séismes.

Par la suite, la comparaison entre les modèles déterministes et stochastiques pourrait s'avérer utile s'ils sont applicables ensemble ou en opposition, il est possible qu'un modèle soit applicable uniquement dans certains types d'événement.

Ensuite, on peut ajouter le mécanisme de la rumeur dans l'analyse de nos données, en effet de fausses informations peuvent être propagées dans le monde et un moyen de l'endiguer pourrait se trouver dans les contrôles d'épidémie. Les moyens employés sont communément : le système de quarantaine, l'immunité par le contact et l'éducation. Un système de quarantaine ne semble pas viable sur internet, mais l'éducation par la mise en lumière de mécanisme de désinformation peut être mis en place. Le concept d'immunité contre les médias reste encore à définir.

De plus, la création d'un programme calculant automatiquement la corrélation maximum entre la propagation d'un événement et un modèle épidémiologique faciliterait la précision des résultats.

Enfin, la comparaison avec les modèles définis par les pairs permettrait d'améliorer ou de rejeter le modèle présent.

Remerciements

Jean-Marc VINCENT pour son suivi tout au long de ce Magistère
Alice CHOURY pour la relecture de ce rapport de recherche
Jérôme FERRAFIAT pour son soutien au long de ce Magistère
François et Françoise BAILLE pour leur soutien et leur curiosité

Références

[1] Epidemic Modelling An Introduction - D. J. Daley - 2001
[2] PtProcess : An R Package for Modelling Marked - Point Processes Indexed by Time - David Harte -2010
[3] Construire et utiliser un corpus : le point de vue d'une sémantique textuelle interprétative - Bénédicte Pincemin - 1999
[4] Découverte et caractérisation des corpus comparables spécialisés - Lorraine Goeuriot - 2009
[5] Analyse macroscopique des grands systèmes : émergence épistémique et agrégation spatio-temporelle - Robin Lamarche-Perrin - 2013
[6] Be-CoDiS : A Mathematical Model to Predict the Risk of Human Diseases Spread Between Countries - Validation and Application to the 2014-2015 Ebola Virus Disease Epidemic - Ivorra, B., Ngom, D. & Ramos, Á.M. Bull Math Biol - 2015

Composition of Self-Stabilizing Algorithms in Coq

Jules Lefrère

Supervised by: Karine Altisen, Pierre Corbineau, and Stéphane Devismes

Abstract

We are interested in certifying the composition of self-stabilizing algorithms in Coq. To that goal, we define a composition operator, which creates a compound algorithm from two other (simpler) ones. We want to propose a composition theorem for this algorithm's specifications. To illustrate our purposes, we have created a case-study algorithm which computes the disjunction of Boolean Inputs, one per node. We propose two versions of this algorithm: a raw one, which does not use the composition operator, and another one which is the actual composition of two simple algorithms. All of this work has been made in Coq, using the PADEC's Library[1].

1 Introduction

1.1 Context

Self-stabilizing algorithms.

If one believes the forecasts, we will count around twenty billions of objects connected per days in 2020[1]. Such objects are linked together by different means of communications (*e.g.* Wi-Fi, Ethernet, *etc*...). We can see all these objects as node of a graph: an interconnected network is usually represented by a graph. On such networks, particulars algorithms are running, named distributed algorithms. An interconnected network and distributed algorithms form a distributed system like Internet, where each site of a distributed system is autonomous, asynchronous, interconnected and only have a local vision of the system (as opposed to centralized systems). The actual size of networks and the distance between each site usually do not permit human intervention in case of dysfunction, yet, the more a network is large, the more the probabilities of failure is large. Consequently, the algorithms running in such system have to be fault-tolerant. There are two main approaches:

[1] Gartner (02/05/2016). URL http://www.journaldunet.com/solutions/expert/64225/insecurite-des-objets-connectes—-comment-conjuguer-l-iot-et-la-securite.shtml

Pessimistic approach. This approach favors the correction and consists to verify that all is well consistent with the specification given at each step of the algorithm, especially the different variables, and that the global algorithm follows its specification. Here, we are talking about robust system, *i.e.* that the effect of the faults is hidden. This approach gives to the user the appearance of a system without failure.This is a costly approach because a lots of resources are used in order to control the specification at each node at each new calculation step. System's performances can be momentary slowed. Furthermore, such approach often assumes that the majority of the system stays correct.

Optimistic approach. In this approach, when a problem occurs, the algorithm is allowed to deviate of his specification, proving that it converges within a finite time to an expected behavior. To allow this convergence, it is assumed that failures can affect the entire network, but occur rarely: between two failures, the algorithm has the time to converge. One other hypothesis is that faults do not corrupt the code of the algorithm: only the data can be corrupted but they are corrupted only in their domain of definition. This kind of fault is named transient fault. This approach is the one we will more specifically use in this report. According to the hypotheses in this approach, the self-stabilization is a natural answer to this kind of problem. A self-stabilizing algorithm is able, in a finite time and from an incorrect state, to converge to a correct state, whose behavior will be conformed to its specification. Moreover, this kind of algorithm does not need any initialization.

Certification & Coq.

The first self-stabilizing algorithms [4] were short and simple, they were dedicated to simple environments(*e.g.* first algorithm of Dijkstra works in a oriented ring with leader); construct and verify its proof was an easily feasible thing for a human. Nowadays, algorithms progresses lead to take a particular interest to harder problems (*e.g.* clustering) dedicated to more complex environments. Consequently, algorithms currently developed become harder to write and to prove. Thus, it is now more complicated to ensure that the algorithm is correct and that there were no omissions of limit case. The use of a proof assistant becomes necessarily. We use Coq [2;

8], a tool which aims at helping formal proofs implemented from a derivative of Ocaml. The PADEC library[1], allows us to formally describe models of distributed algorithms and to construct the proof of these models. These models will be verified mechanically thanks to Coq: this gives us certified proofs, safer than those wrote manually. We obtain structured proofs that are both complete and correct.

With the return on experience, recurrent schemes have been identified automatized thanks to Coq. This automating allows a reuse and a certification of the currently designed proofs of algorithms.

Composition.
As the self-stabilized algorithms currently designed are now more complex, proofs are more complicated to implement. One classic method consists to split the initial algorithm into sub-algorithms, each one running one part of the algorithm. This method follows the abstraction logic of sequential functions and is one method of composition [3]. More precisely, we will focus on the one called hierarchical collateral composition which is a deviate of the collateral composition [7]. There is other techniques of composition [5]. In this internship, we are focused on one method. the used method simplifies considerably the certification of the algorithm since, with this split in sub-algorithms, we only need to prove these parts and to prove that their assembly is correct. Assuming that the parts have been intelligently created, proving these corrections will be easier because they can be considered as independents algorithms of one another. So, their proofs can be done separately. Afterwards, this method requires to exhibit that the complex algorithm corresponds to the composition of sub-algorithms previously proved and that the composition allows to prove in a generic way the composition of specifications. After that, when parts will all have been proven, and thanks to the proof of correction of the composition, this will be the same as having to prove the initial algorithm.

PADEC. [1] The purpose of the internship is to describe in Coq the hierarchical collateral composition and to construct the certification in Coq. This composition simplifies the proof of complex algorithms build according to this method. This internship is part of the project AGIR PADEC[2] whose purpose is to create a library in Coq allowing the certification of self-stabilizing distributed algorithms.

1.2 Model

Distributed Systems. Distributed systems, that we are using here, are a finite set of nodes V interconnected *via* channels. In these systems, each node executes its own code, stocks its own state which is a set of variables, and interact with the others nodes *via* a shared memory. A node can interact in function of its neighbors's variables *i.e.* all nodes that are connected directly to that node, but without the possibility to modify them. This model called "the state model" allows to abstract communications between nodes: rather than discussing *via* messages between nodes, one node can read the variable of its neighbors accordingly, it can modify its own variables. For studying the behavior of an algorithm, it is necessary to follow the value of the variables of its nodes. A configuration represents the set of states at a given point. The trace of an algorithm is a sequence of configurations of this algorithm: we can follow the behavior of an algorithm by looking at its traces.

Rules. In our model, an algorithm is defined by a finite set of rules of the following form: $(guard) \rightarrow (statement)$.

- The guard is a Boolean predicate on the values of the node's variables and those of its neighbors.
- The statement modifies node's variables, as said above, even if a rule can access to its neighbor's variables, it cannot modify them.

These rules can vary for each node and a node can have multiples rules. One rule is said enabled if its guard is evaluated to True. When a rule is activated, it is done atomically.
A step of the algorithm allows to proceed from one configuration to the next, by applying at most one activatable rule per node and at least one activatable rule across the network.

Variables. In our model, variables are divided in three classes: they can be input variables which will not be modified but which are used by the algorithm's guards, internal variables serving to intermediate problems or outputs variables serving to stock results of the algorithm. Outputs variables will be modified by the statement part of rules.

Silent algorithms. [6] In the scope of the internship, we are focusing to a specific class of algorithms: silents algorithms. In our model, a silent algorithm is deterministic and stop in a finite number of steps, which means that after this number of steps, the values of the outputs variables of the nodes will not be modified anymore. A silent algorithm takes inputs and, in a finite time, gives a result in its outputs variables in function of the inputs.

2 Case-study

In the scope of my work, and to rely on a support, I have created examples allowing me to test my work.

2.1 Full algorithm

The algorithm which is implemented in this example is an algorithm of value propagation in a network. We follow the hypothesis that the topology is a spanning tree noted $T(r)$, an acyclic connected graph which have a specific node named root (r) and as it exists a unique path from the root to the given node.

This algorithm, for a Node p, has two inputs: A Boolean constant (I_p) and a constant which have for value $\perp$ if p is root or which is worth the father of p, that is to say the closest neighbor of p to the root (par_p)

The purpose of the algorithm is to calculate an output (R_p) which will be equal to the disjunction of the set of inputs: the algorithm converges to a state from which the following predicate is checked forever:

[2]Preuves d'Algorithmes Distribués En Coq.

$$\forall p, R_p = \bigvee_{q \in V} I_q$$

For this output, the algorithm uses an internal variable (S_p). Its value must converge to the disjunction of the inputs of the sub-tree of p. When the algorithm converges, the following predicate is verified:

$$\forall p, S_p = \bigvee_{q \in T(p)} I_q$$

Where $T(p)$ is defined as:

$$T(p) = p \vee \bigvee_{q \in child(p)} T(q)$$

For the rest of the explanation: $child(i)$ is the set of children of the node i; the root has the number 0, and S_0 refers to the value S of the root. The objective of this algorithm is that each node take a value among those which are proposed by the other nodes.
In a first time, it will be necessary that the values are calculated; in a second time that a choice is made and diffused among the tree. Each node will have to make its disjunction going up to ensure the following specification:

$$\forall p, S_p = I_p \vee \bigvee_{q \in T(p) \setminus \{p\}} S_q$$

This comes to an induction on the tree with the case base being: if the node is a leaf, its sub-tree is empty, so in the specification, $T(p)$ is equal to $\{p\}$, and since $q \in T(p) \setminus \{p\}$ and $\{p\} \setminus \{p\}$ is the empty set, a leaf will have to ensure: $S_p = I_p$. The value will be calculated only one time because I_p is a constant. For the rest of the tree's nodes, we apply the formula. As a node only knows its neighbors, we will only see the S value of them, its value is itself calculated with the same formula and this recursively until a leaf (and then it is the same case as the base case we have seen before). The sub-tree of the root is equivalent to the whole tree, thus S_0 is equivalent to the disjunction of each value I of the tree. Once we have this, we only have to spread this value.

Following our model, one rule will have the purpose of computing the disjunction whereas one other will have to spread the result of the disjunction.

Rule 1 :

$$S_i \neq \left(I_i \vee \bigvee_{c \in child(i)} S_c\right) \quad \rightarrow S_i := \left(I_i \vee \bigvee_{c \in child(i)} S_c\right)$$

Rule 2 :
root:

$$S_0 = \left(I_0 \vee \bigvee_{c \in child(0)} S_c\right) \wedge R_0 \neq S_0 \rightarrow (R_0 := S_0)$$

non-root ($i \neq 0$):

$$S_i = \left(I_i \vee \bigvee_{c \in child(i)} S_c\right) \wedge R_i \neq R_{par_i} \rightarrow (R_i := R_{par_i})$$

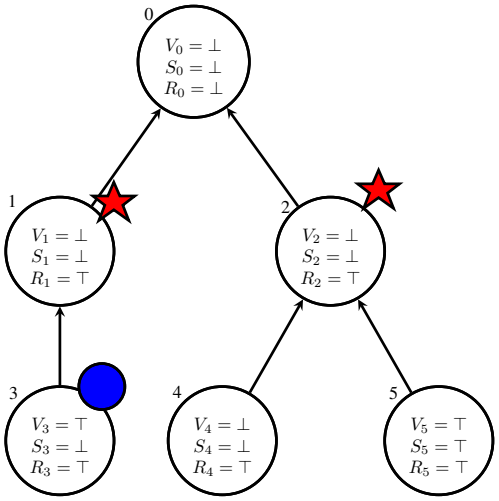


Figure 1: Scheme of the algorithm at a certain step

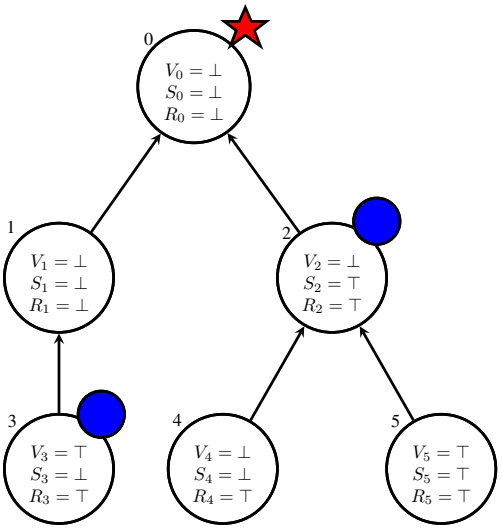


Figure 2: Scheme of the algorithm at the next step

In these scheme, $\top$ stands for true and $\bot$ for false. The node 1 is enabled by the second rule and the nodes 2 and 3 are enabled by the first rule. We can see in this example that both rules can apply simultaneously on different nodes and that on a step, all nodes are not necessarily activated: here, nodes 1 and 2 are activated (red star) and the node 3 is enable but not activated (blue circle). As we can see on the figure 2, the node 1 has change its value of R for $\bot$ and the node 2 its S value for $\top$. From one step to another, we can see that nodes which have been activated can have three behaviors:

- From enabled to non-enabled:
  The node 1 has change its value and both of its guard can't be enable.
- From non-enabled to enabled:
  Since values on the node 2 have change, and the node 0 is in its vicinity, we have the case where one guard of

this node that has become enable (here, its the first one).

- From enabled to enabled:
  We can see this case on the node 2, here, the first rule is not enable anymore but its the second one which is.

  In a more general case, there can be multiples reasons for this behavior:
  - The node had multiples rules enabled but only some were activated (that is the case here).
  - One of the neighbors was activated at the previous state, which let the node enabled.
  - The rules that were activated activate other rules in the node.

2.2 Splitting in two sub-algorithms

In the writing of the second rule, we can notice that the first part of the guard (the condition on S_i) is only here to give the priority on the first rule. On the second rule, and in fact in most of the self-stabilizing algorithms, there are priorities between rules of a node so that at most only one rule can be activated for a given state of the algorithm. The different rules are mutually exclusive. Thereby, the more rules there are, the bigger their guard are getting: hence the idea to separate the rules in several algorithms in order to simplify the guards of these rules.

Afterward, I have created two algorithms which are respectively the first and the second rule described above. Delegating the priority management to the composition operator.

First algorithm. The first algorithm has to do the disjunction of the nodes of the tree and to store the result in S_0. Under the hypothesis of having a spanning tree in input, this algorithm calculates the disjunction between each node of the tree. After stabilization, it must verify:

$$S_0 = \bigvee_{q \in V} I_q$$

Its work the same way as explain in the precedent section, by an induction on the tree where each node must verify:

$$\forall p, S_p = I_p \vee \bigvee_{q \in T(p)} I_q$$

Second algorithm. The second algorithm is only the diffusion of the result through the tree (see Rule 2). The rule is slightly different if we are in the root or not. The one of the root is the affectation using only variables of the root. The other recopies the value of its father. As previously described, the algorithm executes itself in a finite time: the number of nodes is fixed, once the root has made its choice, it will never change, then its children will choose only one time, *etc*. Eventually, the algorithm verifies the following property: $\forall n, R_n = S_0$.

Composition. It must be remembered that the two algorithms execute themselves concurrently and S_p has becommed an output variable of the first algorithm and an input one for the second algorithm. This means that the second algorithm will be able to stabilized only when the first algorithm will be stabilized.

When the second stabilizes, all nodes will have decided on the same value, – this value was in the set of the input values of at least one node of the first algorithm –. We obtain a specification which is the conjunction of the two algorithms:

$$\forall n, R_n = S_0 \wedge S_0 = \bigvee_{q \in V} I_q$$

It is equivalent to the specification of the section 2.1, because, by replacing S_0 by its definition, we obtain the initial formula.

The rules are identical with the exception however of the fact that the first rule must be validated before applying the rule 2: the rule of the first algorithm has a higher priority than the rule of the second algorithm and, in a node, the second algorithm must always wait that the first one ends before executing itself. This priority will be implemented through the composition operator.

3 Composition

As we have seen in the case-study, splitting in sub-algorithms allows to simplify the rules and to build more concise proofs which are less confuse for humans. What interests us is to go from the cutting to the composition in order to have the same behavior as the complete algorithm. According to this case-study, we have constructed a generalization for the composition: to do so, we have created an operator.

3.1 hierarchical collateral composition operator

This operator creates, from two given algorithms, the composition $A1 \oplus A2 = A3$ where $A1$ and $A2$ are the sub-algorithms and $A3$ is the algorithm resulting from the composition. Variables of $A3$ are the union of $A1$ and $A2$ variables. $A3$ have as input variables the ones of $A1$ and as output variables the one of $A2$. All others variables become internal variables of the algorithm.

This operator is not symmetrical, thus $A3$ has all rules of $A1$ ans $A2$, to the difference that guards of $A2$ are slightly modified in a way that they can only be held if guards of $A1$ cannot. This results by the conjunction of the negation of each rules of the first algorithm in guards of each rules of the second algorithm.

3.2 Composition Theorem.

We now assume two silent self-stabilized algorithms $A1$ and $A2$ and this for their respective specification. The desired property for the composition operator is that the algorithm $A3$ is self-stabilized and silent for the conjunction of the specifications of its subs-algorithms.

4 Contribution

All the work we have done, and the one we will have to do for this internship, is written in Coq and is based on the framework PADEC[1]. The first step of this internship was to familiarize myself with Coq[2] and to appropriate myself this framework. This internship is both a TER and a magistère internship.

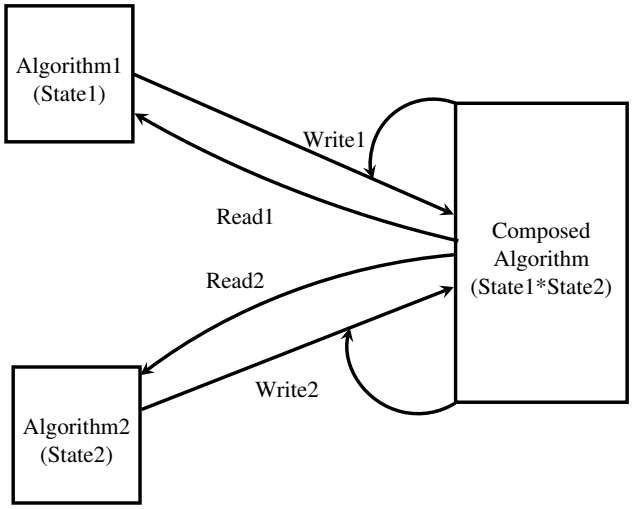


Figure 3: Scheme of the composition

4.1 Work already done

In order to properly define the composition operator, we had to formalize the communication between algorithms:

Like indicated on the figure 3, several projections must exist:

- Read allows to transform from a state of $A3$ to the corresponding state of one sub-algorithm; one projection for $A1$ and one for $A2$ are necessary.
- Write allows to transform from the state of a subalgorithm to the corresponding state of $A3$. for the write functions, we have to reconstitute a state of $A3$ from a state of one sub-algorithm which does not contain all variables of $A3$. That is why we also need of the current state of $A3$ to add to the state of the sub-algorithm, we will only take the variables which are not in the state of the sub-algorithm, and all we have to do, is the union between theses variables and these of the state we want to transform to create the new state of $A3$.

Once these projections are defined, we can execute the three algorithms while working on a state of $A3$. However this is not enough, because even if now we know how to execute them, nothing guarantees that they do not interfere with themselves to keep going. To do so, we will have to verify the input hypothesis: the read-only variables of a sub-algorithm are not modified by this sub-algorithm and the variables modified by one algorithm must not be modify by a further sub-algorithm. For example, $A2$ can accept output variables of $A1$ as input variables. But the converse is impossible. $A1$ has for input the input of the entire algorithm.

This operator was approximately defined during my TER internship, its definition has been refined during the rest of the internship. Once it was fully defined, we were able to recreate the entire algorithm from the split. There are also two definitions of the entire algorithm: the first one with just the algorithm and the second constructed from the two sub-algorithms which have been composed thanks to the operator giving an entire algorithm.

It remains to write the composition theorem and to prove it (*c.f.*3.2). Once again, to ensure that the composed algorithm is provably correct, there is some hypotheses that needs to be verified. Also, since this work is not finished, it is possible that some hypotheses are not found yet.

A good argument for additional hypotheses is to prevent the asynchronous daemon from postponing the execution of $A1$ steps indefinitely. This may be possible if that at some point of the program, there are some rules in $A2$ that enable divergent behaviors. One way of preventing this from happening, is to assume that *from any configuration of the first algorithm, the second one will stabilize in a finite time*. This will imply that, in a finite time, any daemon avoiding to execute $A1$ steps will run out of options and thus will be forced to make $A1$ progress. Such hypothesis is sufficient to ensure the stabilization of an algorithm.

That is not the only solution, one other is to change the execution model by adding a fairness assumption to the scheduler. For example, a weakly fair scheduler will have to eventually activate any rule that stays enabled for a sufficiently long time. This is what will make $A1$ progress in this solution.

Theses two solutions are not comparable, there may be cases where will be applicable and not the other.

The formal proof of the composition theorem is not finished, but before starting to work on it, it has been necessary to prove the correctness of the sub-algorithms.

4.2 Work to be done

The specification of the example is done, but it is necessary to verify the composition. The composition theorem is for the most part already written but most of the proofs are incomplete, but we think that with the condition above, the composition will work. Once this work is finished, we will have to apply it on the case-study. An other task will be to prove that the entire algorithm and the composed one answer to the same specification: this work is optional and only concern the example, although its realization will help to understand the generalization.

5 Conclusion

This internship was about certification of self-stabilized distributed algorithms in Coq. In this context, we have defined the composition of such algorithms and we have applied the composition on an example. This whole internship progressed one step at a time: I started by defining my example and writing its specification in a modular way. Later I had to define the composition operator. These were the steps done in my TER. Next, I had to make sure that the composition worked, and then I had to completely prove my algorithm. At each step, I had to understand a new part of the PADEC library, its concepts and its proof techniques. A little during the TER and mainly during the magistère, I had to define the general operator of composition and to write the specification and correctness proof of the sub-algorithms, which has caused some real difficulties.

References

[1] Altisen, K., Corbineau, P., Devismes, S.: A framework for certified self-stabilization. In: E. Albert, I. Lanese (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9688, pp. 36–51. Springer (2016)

[2] Bertot, Y.: Coq in a hurry. CoRR **abs/cs/0603118** (2006). URL http://arxiv.org/abs/cs/0603118

[3] Datta, A.K., Larmore, L.L., Devismes, S., Heurtefeux, K., Rivierre, Y.: Self-stabilizing small k-dominating sets. IJNC **3**(1), 116–136 (2013)

[4] Dijkstra, E.W.: Self-Stabilizing Systems in Spite of Distributed Control. Commun. ACM **17**, 643–644 (1974)

[5] Dolev, S.: Self-Stabilization. MIT Press (2000)

[6] Dolev, S., Gouda, M.G., Schneider, M.: Memory Requirements for Silent Stabilization. In: PODC, pp. 27–34 (1996)

[7] Tel, G.: Introduction to Distributed Algorithms, 2nd edn. Cambridge University Press (2001)

[8] The Coq Development Team: The Coq Proof Assistant, Reference Manual. URL http://coq.inria.fr/refman/

Distributed Approach of Cross-Layer Resource Allocator in Wireless Sensor Networks

Franck Rousseau and **Olivier Alphand** and **Rodolphe Bertolini**
Grenoble Informatics Laboratory (LIG), France

Karine Altisen and **Stephane Devismes**
INRIA Grenoble, France

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: Rodolphe Bertolini, 15/06/2016

Abstract

Wireless Sensor Networks (WSN) are resource-constrained by their architecture: reduced memory, powerage, calculation capabilities, connectivity. Despite those constraints, they need to organize themselves in a tree-like topology along which all the data will be relayed towards the root of the network. To increase the life expectancy of the network, nodes should spend most of their time sleeping while still ensuring an acceptable delay for data to reach the root. In this paper, we present the sketch of the distributed version of a WSN simulator from the state of the art, that was initally using centralization. We show that distribution creates contention in the shared slot, and that increasing the number of shared slots heals this contention.

1 Introduction

Wired sensor networks installation cost can be up to 10 times the market share value of sensors. WSN lower this cost, and can also provide an ubiquity of the information gathering, that we call the Internet of Things (IoT). Those reasons among others can explain the growth of WSN usage.

We worked on an open-source simulator, developped at Berkely University by the 6tisch working group [Palattella *et al.*, 2016], that focuses on the On-The-Fly (OTF) Bandwith Reservation algorithm implementation [Dujovne *et al.*, 2014]. OTF is an algorithm that determines how many time slots are needed for a sensor in order to communicate with its parent(s) to minimize congestion in its packet queue.

We would like to compare the actual probabilistic approach of allocation, caused by the random choice of a cell, to a deterministic algorithm. But to have interpretable results, we first need to make the simulator more realistic concerning the simulated network traffic.

We thus modified the simulator in two main ways:

- so that it also simulates some control traffic (requests and answers triggered by OTF, and later DIO)
- so that this algorithm operates in a distributed manner: instead of having an omniscient view of the network that allows a sensor to get the needed resource instantaneously, there are communications that occur between sensors, and sensors use the information in those communications to make decision about resource allocation.

We start in section 2 by introducing the State of the Art of WSN simulators, and we continue in 3 with the definition of the graph generated by the sensors connectivity, and an overview of TSCH protocol. Then, we present in section 4 the simulator that we made more realistic, and we explain the modifications that we did. We see in 5 that those modifications impact the bootstrap duration, and we explain in section 6 some improvements that can be done in order to reduce the bootstrap duration when using shared slots. In 7 we analyse and discuss the results using improved communication through shared slot, and we bring in 8 some ideas of what can be done as further research with the simulator.

In section 9, we finally conclude about the effects of the decentralization of the overall behavior, and bring ideas on other improvements to speed up the bootstrap and to reduce the number of collisions in data packet transmission that occur more often as the number of node increases.

2 State of the Art

Great effort were made by the IETF to standardize a complete IP-compliant IoT stack : a specific transport layer (CoAP [Shelby *et al.*, 2014]) as well as a routing protocol (RPL [Winter, 2012]) and some specific schemes (6LowPAN [Hui *et al.*, 2010]) to compress IPv6 addresses. For the physical layer, the 802.15.4 technology was the most widespread one because of its low consumption capabilities. However no MAC layer was clearly supported by the IETF. The WSN we are interested in this article specifically aim at addressing the requirements of a harsh industrial environment. Therefore, beside an enhanced WSN life expectancy achieved by sleeping as much as possible, robustness and time contraints are vital requirements. To do so, the MAC layer of the original 802.15.4 standard was not robust or deterministic enough. To overcome those problems, additional behaviors were introduced at the MAC layer in the 802.15.4e standard [Group and others, 2011]. One important feature is channel hopping which allows to mitigate multi path fading and

external interferences experienced in 802.15.4 that uses a unique channel. Several modes were defined, including synchronous and asynchronous mode.

We focus on Time Slot Channel Hopping (TSCH), a protocol defined by IEEE 802.15.4e standard [Mirzoev and others, 2014]. TSCH is a Medium Access Control (MAC) layer protocol that provides a mean to sensors so that they can communicate properly.

DeTAS [Accettura *et al.*, 2015] provides an interesting decentralized scheduler, but allows only 5 shared slots, which is a limitation when, as we see in this paper, the network has more than 10 nodes.

Orchestra [Duquennoy *et al.*, 2015], a routing-aware, autonomous, random-access TSCH allocator, gives a solution for ressource allocation at MAC layer level: it allocates time slots to a sensor in order to communicate longer with its parent when the incoming data is too large. But the randomness of this algorithm does not prevent from a sensor from a branch of the graph to allocate the same slot that a sensor from an other branch of the graph also allocated. Non preventing this would, in some cases, lead to interferences and collisions. For instance, a sensor can give to its child the same cell that has been allocated in a close neighborhood. If they communicate at the same time, and because the medium used is radio wave, the signal cannot be read by any of both receivers because collision occurs.

3 Communication between sensors

3.1 Through a DODAG ...

To collect data, sensors need to structure a Destination Oriented Direct Acyclic Graph (DODAG, see Figure 1) that is built using Routing Protocol for Low-Power and Lossy Networks (RPL) among other protocols. The aim of RPL is to build the shortest route from any sensor to the root gateway sensor that will transmit information to a server that will process this information.

RPL states that a sensor should send broadcast to its neighbors containing at least its own rank in the DODAG (number in parenthesis in Figure 1). Because the traffic is from leaf to root (plain lines in Figure 1), the root sets its DODAG rank to 0 (because there is no hop between the root and the server) and starts sending the DODAG Information Object (DIO, plain and dotted lines in Figure 1) [Winter, 2012], in which its rank information is contained, to all of its neighbors. When a sensor receives a DIO that has a lower rank than its own, it updates its rank as $received\_rank + 1$ and broadcasts a new DIO containing the updated rank information. If the rank in the received DIO is higher, it ignores it. In the first case, a sensor can also decides to ignore the DIO if the medium has a poor quality (low RSSI for instance).

To send a data packet towards the root, a sensor has to send its packet to the sensor that has a lower rank.

Figure 1 shows the different paths (set up by DIO) that data may follow to reach the root (R). The node rank is put inside brackets.

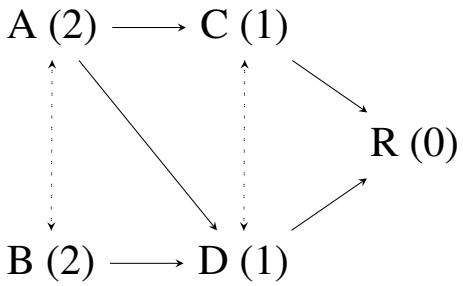


Figure 1: Example of a DODAG with 5 nodes

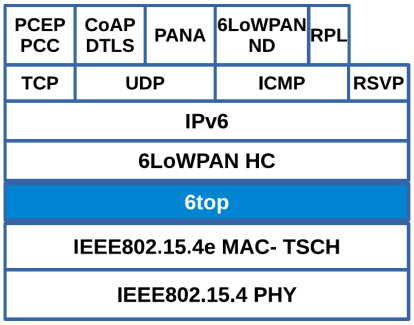


Figure 2: TSCH is a MAC layer protocol [Palattella *et al.*, 2016]

3.2 ... using TSCH

The medium through which sensors communicate is defined by TSCH, a MAC layer (Figure 2 [Palattella *et al.*, 2016]) protocol first outlined in IEEE 802.15.4e standard. A sensor X is assigned a time slot and a channel that corresponds to a specific communication with sensor Y : from X to Y (Tx: transmission), from Y to X (Rx: reception), or unicast and broadcast. Let's take sensor D from Figure 1 and see Table 1. The couple of time slot and channel offset (1,2) and (2,3) correspond to Rx cells but are Tx cells for respectively A and B, and (4,0) to a Tx cell for D but Rx cell for the root.

We initially use the couple (0,0) as the mean for unicast and broadcast informations (DIO messages, control packets). This cell corresponds to the shared cell.

Channel offset \ Time slot	0	1	2	3	4
0	DIO Control	$A \rightarrow C$			$D \rightarrow R$
1				$C \rightarrow R$	
2		$B \rightarrow D$			
3			$A \rightarrow D$		
4					

Table 1: Time slot & Channel offset matrix

4 Presentation of the simulator

We have modified an open-source WSN simulator that has been developped at Berkeley University [Palattella *et al.*, 2016]. The standard parameters of a simulation follow the Minimal 6TiSCH Configuration [Vilajosana and Pister, 2016]: a time slot lasts 10 ms, a slot frame contains 101 time slots , and 16 frequency channels are available. The nodes are located in a 1km wide square.

This simulator has a centralized view of the network: each node has the knowledge of neighbors and parents properties. For instance, to add a cell in its scheduler, a node calls a function that gets the free cells from its parent and picks randomly the needed number of slot.

Moreover, the simulated network traffic only consists of data packets while control packets (request for a new cell, reply, DIO (see section 3.1) ...) are considered non-existent. The centralization is made possible with calls from a cell of its parents' / neighbor's function. In addition, the fact that slots are randomly picked induces the same drawback as the one Orchestra induces.

4.1 Original simulator

The topology of the network is created randomly: a sensor is placed at a random position, the number of neighbor(s) is computed using RSSI calculation, and if there are too few neighbors, the sensor is replaced until the number of neighbors reaches a certain threshold. The first placed sensor is the DODAG root, the destination of all the data collected by the sensors, and is to be at the center of the network: if we consider that the topology is bordered by a rectangle, then its coordinates are $x = width/2$, $y = height/2$.

The DIO messages are not simulated in the network traffic, sensor notifies its neighbors of its rank using a centralized function. This function is called each slot frame by all of the sensors, whereas they are supposed to be sent depending on a trickle timer : while a sensor does not receive a DIO that proposes a lower rank, it doubles its DIO period. When a lower rank DIO is received, it starts again with a period of 1 slot frame.

When a sensor scheduler activates a cell in the matrix shown in Figure 1, it gathers information about what to do: if the cell is Rx then it has to listen for possible incoming packets; if the cell is Tx then it has to send the first packet in its data queue.

When a sensor detects that it needs more time slot to communicate with its parents (thanks to OTF algorithm), it calls a function of a parent sensor. This parent randomly selects the needed number of slots that are not used in both itself's scheduler and requester's (child) scheduler, it adds them into its scheduler as Rx cells, and it tells the requester to also add them as Tx cells.

Most of the functionalities are using a centralized abstraction, that permits perfect transactions between nodes. To bring a more realistic model, we modified this centralized behavior for some of the transactions.

4.2 Actual simulator

Modification in the behavior

In order to make the simulator more realistic and to make it fit TSCH standard, we first added a new type of cell: the shared cell, that corresponds to the cell (0,0) on which every sensor can listen to and send information.

This special cell allows us to make the resource allocator distributed: instead of a parent to get the information of the slot requester's scheduler through omniscient function calls, the requester sends through the shared cell all of the information that the parent needs to process the request: MAC address of the requester, a list of its already used time slots (so that to compute the available slots, the parent can subtract it from its non-used time slots list), and the number of needed slots. The parent's answer is also sent through the shared cell. We implemented the 3-way transaction [Wang and Vilajosana, 2016] :

- mote A sends a "n-cells request" packet to mote B
- mote B receives the packet, adds n cells to its scheduler (tagged as Rx) and sends the n cells list to mote A
- mote A receives the cells list, adds it to its schedule (tagged as Tx) and sends a confirmation to mote B

If the answer from mote B does not contain enough slots, mote A adds the cells and send a new request with the new needed number of slots and the updated list of slots being in use by mote A.

To make communications through the shared slot the most coherent possible, we added a transaction aborting system. If one of the node is dropped, it aborts the transaction and reverse the changes that have been done during the aborted transaction. We also added a sequence number for control packets. If a mote receives from a neighbor a control packet that has a different number as it expected, it drops the packet and aborts the currently handled transaction if any.

To be coherent with the 6top protocol, a node cannot have two pending transactions at the same time, which means that it has to wait for the transaction to be completed or aborted before starting a new transaction.

We modified the simulation engine so that a callback can take parameters. Thus, the same callback can be used to send both request and answer in the shared cell.

Modification of the launcher

We modified the simulation launcher as following:

- Addition of parameter `queuing`: if 0 then the simulation will be as original simulator (nothing through cell 0), 1 then simulation will be as new simulator behavior (slot request and answer using shared cell)
- Addition of parameter `bootstrap`: if used with `queuing 0` $\rightarrow$ no effect. If used with `queuing 1`, then the bootstrap of the network will be managed through shared cell (no Rx / Tx cell at the beginning). If not used, and `queuing 1` is used, then the bootstrap time slot request and answer will be managed by the omniscient function, then use the shared cell.
- Addition of parameter `topology`: if used, the simulator can save the topology into a file (with parameter `rw`

`w`) or read a topology from a file (with parameter `rw r`). The file name is `topologyX.txt`, X being the number of sensors.

- Addition of parameter `numSharedSlots`: it determines the number of shared slots per slot frame. We quickly understood that only one shared slot restricts the communications during bootstrap when there are more than 10 nodes in the square area. The slot number is given by the following formula : if i is the shared slot number (from 0 to *numSharedSlots* excluded), n the number of shared slots and l the slot frame length, then $i * \lfloor l/n \rfloor$

As we explain more deeply in 5, the use of this new cell brings more realism in the simulation, but also brings delay and latency in the communications.

5 Performance of communication through shared slot

For all simulations using the new version of the simulator, we compared the results with the original version using the same parameters to understand what does distribution change in the network behavior.

We first used a topology of 10 sensors. The simulation ran for 200 cycles (slot frame), that is to say $200 * 101 * 0.010 = 202$ seconds. We wrote dumping of the topology using a first run, then we read the topology dump for 600 runs: 300 without distribution (parameter $queuing$ 0) and 300 with distribution (parameters $queuing$ 1 and $bootstrap$). We separated the results into two categories: without and with distribution. Then, we calculated mean and confidence interval of: collisions in the shared cell (Figure 3)(does concern only distributed version); number of Tx cells allocated in all the network (Figure 4).

We then ran 100 times the same protocol but we increased the number of sensors from 10 to 25.

First of all, we focus on the new shared cell $(0, 0)$, through which all of the requests and answers are transmitted. We can see in Figure 3 that at the beginning of the bootstrap, there is a peak of 8 collisions in one cycle. Then the number of collisions decreases progressively to tend to 0 after 200 seconds.

Figure 4 shows the difference in the evolution of the number of cells that are allocated as Tx (transmission cells) without and with distribution. We can see that the steady state of 15 cells, that is reached after the 5th cycle without distribution, is now reached around the 175th cycle using distribution.

We can see in Figure 5 and 6 that performances decreases when we increase the number of sensors : even after 200 cycles, there are still around 10 collisions per slot frame in the shared slot, thus network has trouble allocating more than 15 Tx cells, whereas original simulator allocates 50 cells around the 10th cycle.

The high number of collisions in the shared cell at bootstrap is easily explained by the fact that none of the sensors has Tx cell to communicate with its parent. Thus, OTF algorithm triggers a "cell add" request, cells use the shared cell to

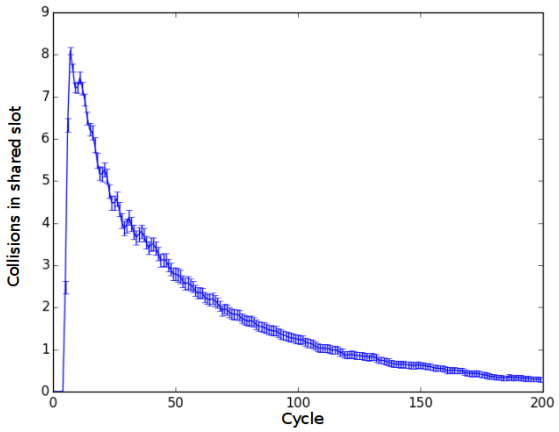


Figure 3: Collisions in the shared cell $(0, 0)$, 10 sensors

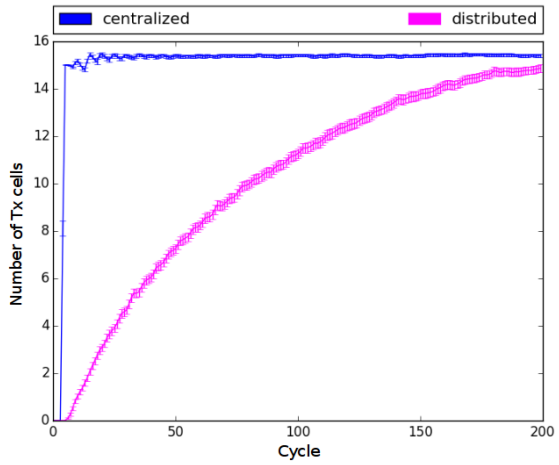


Figure 4: Evolution of cells allocated as Tx, 10 sensors

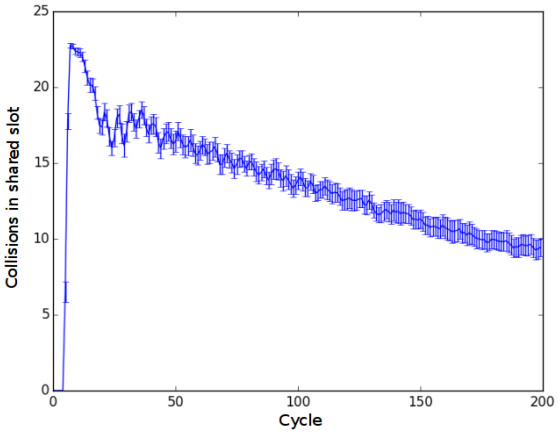


Figure 5: Collisions in the shared cell $(0, 0)$, 25 sensors

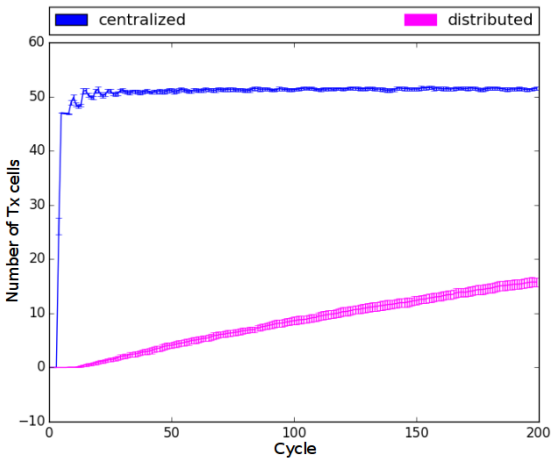


Figure 6: Evolution of cells allocated as Tx, 25 sensors

ask for a cell to their parents, resulting in a heavy contention in this cell.

The delay that it takes to the network in its globality to reach a steady state in the distributed version of the simulator can be explained by the lack of omniscient view of the network. Indeed, instead of a sensor to get all of the resources needed at the next slotframe, exchanges through the network have to be done. But the medium used for this resource allocation request is the same for all of the sensors: the shared cell. We implemented TSCH CSMA/CA algorithm provided by IEEE standard, but as we can see it is not enough to allow a fluent communication in the shared cell.

Indeed, the backoff algorithm is just a window that increases progressively while the control packets is not received. When a sensor sends a control packet, it waits for a random number (picked in the windows) of shared slots before sending again the packet. At the beginning, the windows is $2^{number\_of\_try} - 1$. Thus, it takes at least 4 tries before having a windows that allows a reasonable collision probability for 10 sensors to chose the same delay (window after 4 tries : $[0; 15]$).

Reducing contention in shared slot would reduce the time needed for the network to reach a consistent network. It would also allow us to add other important traffic in the shared slot such as enhanced beacon [Li *et al.*, 2012], DIO, and OTF deletion request that is triggered when a sensor considers that it has too many cells that are not used.

To accelerate the process of resource allocation, there is a priority for answers over queries: if a sensor has both queries to be sent to a parent and answer to be sent to a child ((i.e. both query and answer are present in the control paquet queue), then it will send the answer. Indeed, if no priority is handled, the shared cell would be even more flooded by request packet. This flood would dramatically increase the delay for a node to get the answer.

6 Improving communications through shared slot

6.1 Increasing the number of shared slots

As we showed in previous sections, the shared slot becomes quickly overloaded as the number of nodes increases. A study from the Polytechnic Institute of Porto [Koubaa *et al.*, 2006] showed that increasing *macMinBE* (the backoff exponent used in the CSMA/CA retransmission algorithm when collision occurs) enhances the throughput of the network. In our case, the throughput is limited by the number of shared slots, that is set to one slot according to Minimal 6TiSCH Configuration [Vilajosana and Pister, 2016].

We show in this section that the number of shared slots has a significant influence in the proceedings of transactions.

To have an overview of this impact, we chosed the following parameters for the simulations :

- *numMotes* : 25, 50 and 100
- *numSharedSlots* : 3, 10 and 20

We kept the slot frame to a length of 101, because even with 100 nodes the slot frame is not fullfilled thanks to the 16 different channels.

6.2 Inband communication

A 6top cell request transaction can be a 2-way or 3-way transaction. We decided to implement the 3-way because it ensures a coherent scheduler state on both two nodes.

We implemented inband communication for two of this 3-way transaction. It means that when a node has a Tx cell allocated to its parent, the next cell request transaction will take place in the allocated cell for two packet sending : from the child to its parent, when it sends the request and when it sends the confirmation.

A full inband communication cannot be handled by the receiver of the request. Indeed, in the 6TiSCH matrix, a cell cannot be tagged as Tx and Rx by the same node, thus the receiver is not able to send its answer through the allocated cell to its child. Hence, the answer is sent through a shared slot.

7 Results and discussions

Figures 7, 8 and 9 show the allocated cells in the 6TiSCH matrix. The most important thing that comes from those figures is that as the number of shared slots increases, the speed at which the cells are allocated also increases. An other thing to notice is the presence of a shared slot number lower bound depending on the number of nodes. Indeed, we can see that the 25 nodes topologies reach 60 allocated cells in 100 cycles or less with 10 and 25 shared slots, while it has a tough time when there are only 3 shared slots.

We can also point that the cells are allocated in the same proportion with 50 nodes in a 10 shared slots configuration and 100 nodes in a 20 shared slots configuration.

This cells allocation pace impacts directly the flows the data goes through. We saw that the more shared slots are present, the fastest the cell allocation is proceeded. Now if we focus on figures 10, 11 and 12, we can see that the more

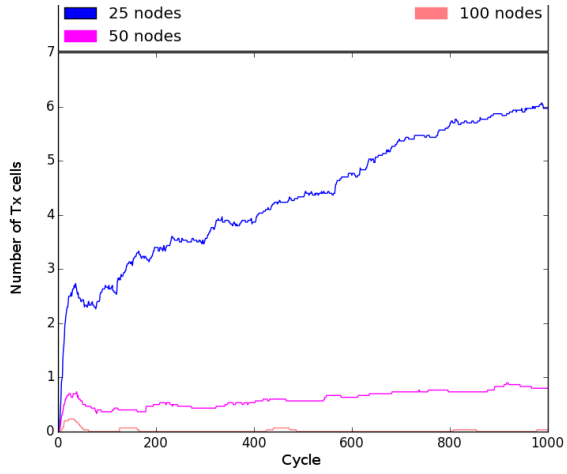


Figure 7: Evolution of cells allocated as Tx, 3 shared slots

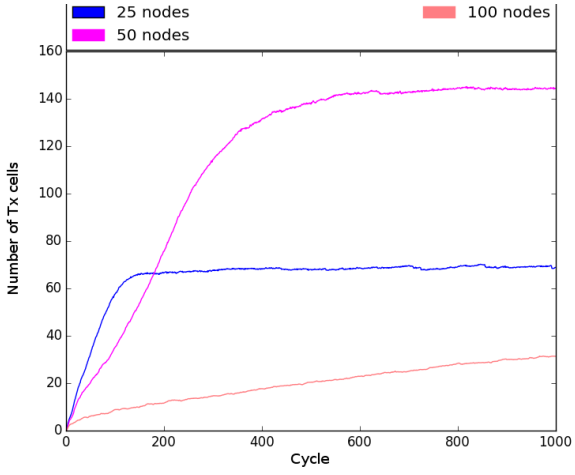


Figure 8: Evolution of cells allocated as Tx, 10 shared slots

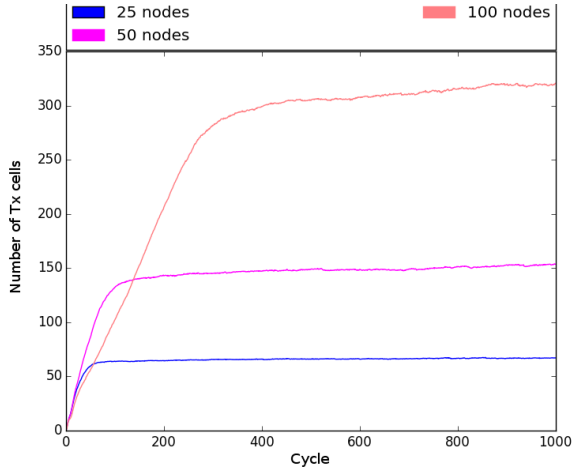


Figure 9: Evolution of cells allocated as Tx, 20 shared slots sensors

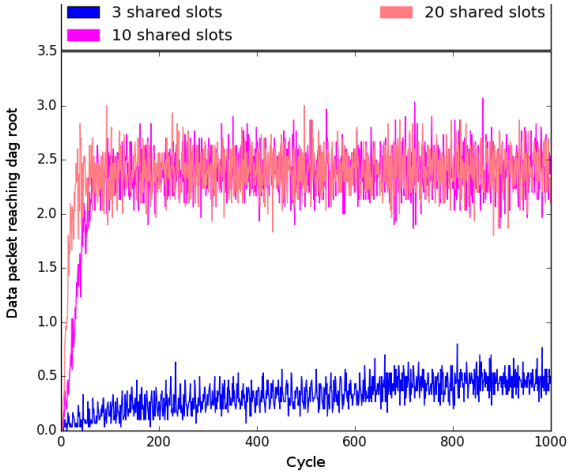


Figure 10: Data packet that reaches the dagroot, 25 nodes

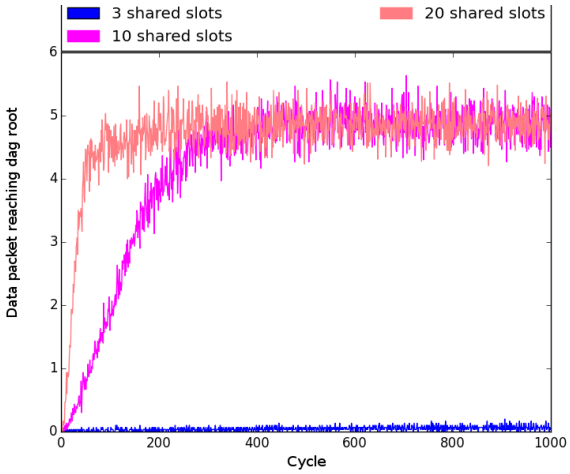


Figure 11: Data packet that reaches the dagroot, 50 nodes

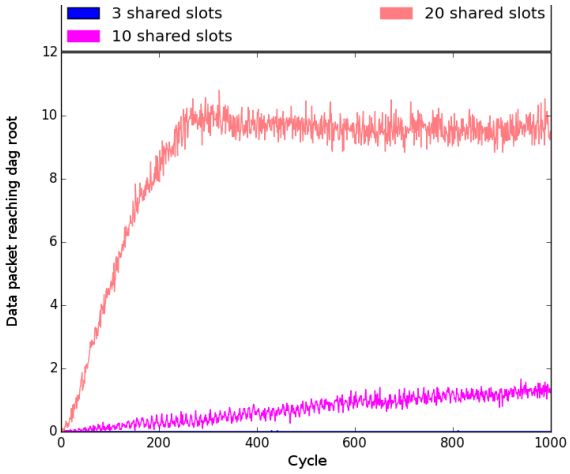


Figure 12: Data packet that reaches the dagroot, 100 nodes

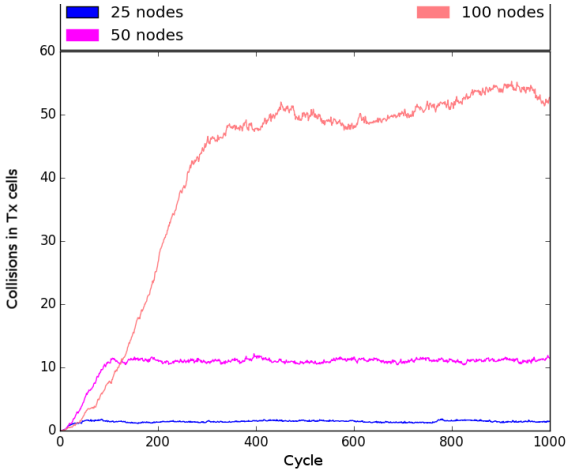


Figure 13: Collisions in the Tx cells

shared slots are present, the fastest the data packets are transmitted to the root of the DODAG.

Indeed, we saw in figure 7 that with 25 nodes and 3 shared slots, some cells are allocated but it grows at a slow pace. We can see in 10 that in this case, only a few data packets reach the root of the dag : 0.5 packets per slot frame in average; while it reaches 2.5 packets per slot frame for the 10 and 20 share slots cases.

We understand from figure 11 that the faster the Tx cells are allocated, the faster the path from a leaf to the root is built: there is an average of 5 data packets that reach root of the dag from the 25th slot frame with 20 shared slots while this average is reached after 300 slot frames with 10 shared slots. This is directly linked with the Tx cells allocation pace. Indeed, from figure 8 and 9 we see that the 25 nodes topology allocates around 65 cells. This state of 65 allocated cells is reached faster with 20 than with 10 shared slot.

Figure 13 shows the number of collisions that occur per cycle, depending on the number of nodes. We see that as the number of nodes increases, the number of collisions increases dramaticaly. The collisions are caused by the lack of knowledge that a node has about its physical neighbors scheduler. Indeed, there is no control that are broadcasted to prevent a node to allocate a cell that has already been allocated by one of its neighbor. It has thus the possibility to allocate this already allocated cell to a communication with its child, inducing collisions. There is no CSMA/CA retransmission algorithm in the allocated slots, so when a collision occurs, it will keep occuring until one or both nodes drop the packet.

We saw that to have a distributed behavior of the cell allocation, the network has to have a medium through which nodes can all communicate. Since this medium, the shared slot, is shared between all of the nodes, there is a very high contention, mostly during network bootstrap. The solution of this contention we brought is to increase the number of occurence of this shared medium.

8 Future work

Having a distributed version of the simulator and resource allocator might permit further works: implement Lamport Local Mutual Exclusion (LLME) algorithm to prevent a node to start a transaction until all its neighbors transactions ended. The ”transaction end” broadcast packet would contain the cells that have been allocated. All the nodes that receive this packet would tag those cells as ”allocated by neighbor”, and will not be able to allocate them (or in a last resort : if it has to allocate a cell and it does not have any free cell). Thanks to this knowledge, a sensor would give a slot that is not allocated or that has been allocated in a far enough place that has few probabilities to induce interferences.

We believe that this algorithm would increase the contention in the open slots - that can be solved as we saw in this paper - but would prevent from most of the collisions in the Tx slots.

9 Conclusion

Transforming the resource allocator into a distributed version has needed to add in the simulator the shared cell. This cell is shared among all nodes to broadcast informations and when communication has to occur between nodes that do not have any other communication medium. We saw that the shared cell is subject to contention, mostly during bootstrap.

This contention induces collisions, that themselves induce a delay in the request/reply transaction. This delay affects the time needed for the network to reach its steady state, but also reduces interferences between sensors during this delay. We solved the congestion problem by adding more shared slots, depending on the number of nodes in the network.

We believe that those results are reliable in a 6TiSCH context, but also for any WSN that makes use of open slots on which a considerable number of nodes can communicate, thus on which a non negligeable number of collisions may occur.

Acknowledgments

I would like to thanks Mr. Franck Rousseau, Mr Olivier Alphan, Ms Karin Alisten and Mr. Stephane Devismes. They always took time to listen to my questions and answer to them. They also asked me questions that deepened and widened a lot my view of the problem.

References

[Accettura *et al.*, 2015] Nicola Accettura, Elvis Vogli, Maria Rita Palattella, Luigi Alfredo Grieco, Gennaro Boggia, and Mischa Dohler. Decentralized traffic aware scheduling in 6tisch networks: Design and experimental evaluation. *IEEE Internet of Things Journal*, 2(6):455–470, 2015.

[Dujovne *et al.*, 2014] D Dujovne, LA Grieco, M Palattella, and N Accettura. 6tisch on-the-fly scheduling draft-dujovne-6tisch-on-the-fly-04 (work in progress). Technical report, IETF, Internet Draft, Jan. 2015.[Online]. Available: http://tools. ietf. org/html/draft-dujovne-6tisch-on-the-fly-04, 2014.

[Duquennoy *et al.*, 2015] Simon Duquennoy, Beshr Al Nahas, Olaf Landsiedel, and Thomas Watteyne. Orchestra: Robust mesh networks through autonomously scheduled tsch. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 337–350. ACM, 2015.

[Group and others, 2011] IEEE 802 Working Group et al. Ieee standard for local and metropolitan area networkspart 15.4: Low-rate wireless personal area networks (lr-wpans). *IEEE Std*, 802:4–2011, 2011.

[Hui *et al.*, 2010] Jonathan Hui, Pascal Thubert, et al. Compression format for ipv6 datagrams in 6lowpan networks. *draft-ietf-6lowpan-hc-13 (work in progress)*, 2010.

[Koubaa *et al.*, 2006] Anis Koubaa, Mário Alves, and Eduardo Tovar. A comprehensive simulation study of slotted csma/ca for ieee 802.15. 4 wireless sensor networks. In *5th IEEE International Workshop on Factory Communication Systems*, pages 183–192. IEEE, 2006.

[Li *et al.*, 2012] Xiaoyun Li, Chris J Bleakley, and Wojciech Bober. Enhanced beacon-enabled mode for improved ieee 802.15. 4 low data rate performance. *Wireless Networks*, 18(1):59–74, 2012.

[Mirzoev and others, 2014] Dr Mirzoev et al. Low rate wireless personal area networks (lr-wpan 802.15. 4 standard). *arXiv preprint arXiv:1404.2345*, 2014.

[Palattella *et al.*, 2016] Maria Rita Palattella, Thomas Watteyne, Qin Wang, Kazushi Muraoka, Nicola Accettura, Diego Dujovne, Luigi Alfredo Grieco, and Thomas Engel. On-the-fly bandwidth reservation for 6tisch wireless industrial networks. *Sensors Journal, IEEE*, 16(2):550–560, 2016.

[Shelby *et al.*, 2014] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). 2014.

[Vilajosana and Pister, 2016] Xavier Vilajosana and Kris Pister. Minimal 6TiSCH Configuration. Internet-Draft draft-ietf-6tisch-minimal-16, Internet Engineering Task Force, June 2016. Work in Progress.

[Wang and Vilajosana, 2016] Qin Wang and Xavier Vilajosana. 6top Protocol (6P). Internet-Draft draft-ietf-6tisch-6top-protocol-02, Internet Engineering Task Force, July 2016. Work in Progress.

[Winter, 2012] Tim Winter. Rpl: Ipv6 routing protocol for low-power and lossy networks. 2012.

3D kidney motion characterization from 2D+T MR Images

Claude GOUBET and Celine FOUARD
Grenoble, France
claude.goubet@e.ujf-grenoble.fr, celine.fouard@imag.fr

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:
Claude Goubet on june 15th 2016

Abstract

The motion of the kidney hardens the clinicians' work while performing kidney punctures. TIMC-IMAG is developing a robot which automatically performs kidney punctures. In order for the robot to be able to reach a moving target in the kidney, it must be able to have a live representation of the kidney motion. We present in this paper a methodology which allows to have a first characterisation of the kidney motion from 2D+T MR images as well as a first 3D model of the kidney motion. We propose to decompose the motion in two characteristics, a translation and a rotation and to define a model using the amplitude and the instantaneous phase of the kidney motion predict its 3D motion. The translation will be estimated thanks to the kidney gravity center displacement. The rotation will be estimated by comparing the kidney directions. The 3D model will be built by estimating a 2D b-spline of the motion. Results allowed us to compute a 3D model and open a discussion over its accurateness. The complete implementation of the methodology will allow us to characterize the kidney motion and make a 3D model based on these characteristics.

1 Introduction

Computer Assisted Medical Interventions (CAMI) aim at both reducing intervention time and improving the patient's operating conditions. Abdominal organs motion induced by breathing [St-Pierre, 2012] raises difficulties for the clinicians during interventions. This affects the patient's comfort as well as the quality of the interventions [Bussels *et al.*, 2003]. Among viscera, kidneys happen to be subjected to a clear motion following the thorax [Giele *et al.*, 2001].

In this context, the TIMC-IMAG laboratory is currently developing a medical robot aiming to assist clinicians to perform kidney punctures [Bricault *et al.*, 2008]. Such operations are time consuming and difficult for the patient due to the fact that the clinician often has to perform several times the punctures before hitting the target.
Today, the practitioner requires the patient to come into apnea in order to stop the breathing motion and be able to target a fixed point previously calculated thanks to preliminary MRI acquisitions. These conditions are not comfortable and may be difficult to perform for patient that are by definition not in health. The objective for this robot is to be able to accurately estimate the position of the kidney at each moment to make it possible to perform punctures during settled breathing.
Position information can't be given by live MR imaging, since the acquisition time is to slow. A 3D model of the kidney motion built prior to the intervention is then necessary for the robot to predict the kidney's position.
We decided to develop an approach characterizing the kidney motion from 2D+T MR images, acquired prior to the intervention, matched to breathing motion information in order to build a 3D model which will be used to predict the kidney's position during the intervention. Under the hypothesis of a rigid motion of the kidney motion, characterizable from local translation between planes. 2D+T images allow to display the organs position through time by representing the same slice of the body in several images acquired in different time T. From these we will characterize the motion as solid translation on x, y, z axes and two rotation angles around x and y axes.
The purpose of this paper is to propose a first solution going from the characterization of the kidney motion from 2D + T MR images to the 3D modeling of this motion. The first part of this paper will be a presentation of the state of the art. Secondly we will state the materials used, the framework on which we are developing, the images and their acquisition. Then we will present the method used to generate motion vectors, i.e. a description on the pre-processing which consists in generating the features and a description of the methodology for the characterization as well as the building of the 3D model. Will follow a description of the experiment. And finally a discussion will be done about the correctness of the 3D model.

2 State of the art

Most of the articles about abdominal organs motions aim to follow an interest point on the organ, either for observation or during image guided intervention.
Solution given for the interest point observation include image registration [Gupta *et al.*, 2003] [Giele *et al.*, 2001]. These solutions correct the translation of the kidney on the images, but doesn't characterize this motion.
Solutions for image guided interventions such as radiation therapy propose to build a motion model of the organs. Hostler et Al. propose to use a model of the pressure of the thorax on the organs [Hostettler *et al.*, 2010a; 2010b]. By measuring the the abdominal skin motion they are able to predict the position of every organs. To apply this method you first need to make a pressure model on the organs.
Fayad et Al. propose a solution for building a motion model using Splines [Fayad *et al.*, 2009]. Although there is no description for the characterization of the motion.

In this paper we will describe both the characterization of the motion and build a model of this motion using splines.

2.1 Building the 3D motion model

A great review of respiratory motion models was made by McClelland et al. [McClelland *et al.*, 2013]. We chose to focus on Fayad et al.'s work [Fayad *et al.*, 2009] which builds a direct model of the respiratory motion by estimating a 2D-Spline of the motion. They use as surrogate data the amplitude and the phase of the movement and the displacement factors to generate a continuous patient specific model function. To recover the position one just needs to know the current phase and amplitude of the breathing and refer to the 2D-Spline.

We will now describe the materials we used in order to characterize the motion and build our model.

3 Materials

3.1 Camitk

The algorithms presented in this paper (section 4) were implemented on CamiTK [Fouard *et al.*, 2012]. CamiTK (Computed Assisted Medical Intervention Tool Kit) is an open source framework which regroups all of the projects developed in this laboratory. In this way, researchers and clinicians are able to work on the same tools and easily share know-how. CamiTK can handle medical images, surgical navigation and bio-mechanical simulations. It is developed in C++ using Qt libraries for the user interface development and VTK libraries [Schroeder *et al.*, 1996] for the visualization.
This software features components, which represent data and actions, which are operations you can perform on a component, such as filtering.
As an outcome of this research, a new action will be integrated which will characterize the kidneys motion from 2D+T MRIs.

3.2 Image acquisition

In order to observe the motion of the kidney we used MR images from healthy volunteers. They were asked to follow a specific breathing pattern to allow regularity on the breathing movement during the acquisition.
The set of data we used for our experiment was acquired with a Philips 3T MRI scanner (Achieva 3.0T TX Philips, Grenoble MRI facility IRMaGE). The dynamic MRI modality allowed us to observe kidney motion by performing acquisition during time. We used a kt-BFFE (Balanced Fast Field Echo) machine's modality with a FOV of 360x360 mm and a slice thickness of 10 mm. With a dynamic MRI modality and a resolution of 256x256 pixels as in figure 1.
The experiments were proceeded on 4 subjects with different time interval between slices.

- Subject 1: dt = 80ms
- Subject 2: dt = 70ms
- Subject 3: dt = 100ms
- Subject 4: dt = 130ms

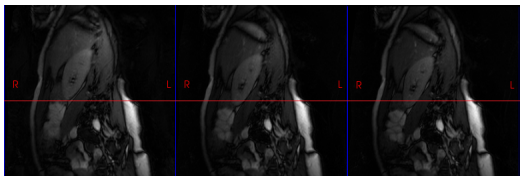


Figure 1: Motion on the coronal plane

The MR images were taken according to the kidneys planes as seen in figure 2. Thus, the acquisitions were not done accordingly to the human planes.

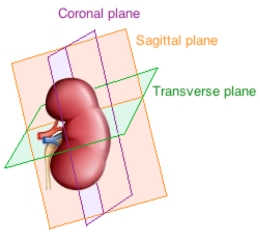


Figure 2: Kidney's planes
extracted from [Neree PAYAN, 2015]

4 Methods

In order to build a 3D model from 2D+T images we must specify characteristics of the 3D motion which can be extracted from the 2D+T data. We decided to take as hypothesis a solid motion of the kidney. Based on the hypothesis we characterized the kidney motion as a translation in the 3D space, on x, y and z axes, and a rotation based on the estimation of direction change of the kidney on coronal and sagittal views. The rotations will occur on z and y axes. It is based on these motion characteristics that we will build a 3D model

of the kidney motion.
We will now describe the different steps of our methodology. Two main stages will be described, the data extraction and the data modelling.

4.1 Data extraction

This phase aims to extract motion characteristics from MR images. We want to process the displacement and rotation of the kidney. The whole process is illustrated Figure 3.
We take as inputs raw MR images. in order to focus the image on the target, we segment the kidney on the images. We then calculate the gravity center of the kidney and its direction in order to be able to calculate the displacement and rotations.

Kidney Segmentation

The kidney segmentation on MRIs allows to focus the attention of the image on the kidney. The automatic segmentation of viscera is still a challenging subject and is not part of our objectives. The contours of the kidney in our MRI sets are not precisly defined and lead both threshold and region growing segmentation to unsatisfying results. Therefore we decided to use manual segmentation. The segmentation was performed on the MR images using ITK-Snap framework [Yushkevich *et al.*, 2006] in order to get binary segmented images of kidney used the kidney gravity center and the kidney direction computation. Figure 4 displays a segmented kidney on coronal and sagittal views.

Kidney gravity centers computation

We chose the gravity center as a marker point since it can be determined in a robust way on a segmented kidney. Also, since it is a repetitive feature of the kidney, following our hypothesis, tracking this point makes it possible to highlight a translation. Gupta Et Al. already used this characteristic point on shifted image registration [Gupta *et al.*, 2003].
The input images are segmented binary kidney slices (a kidney is represented with pixels set to 1 while background is set to 0). The gravity center of the kidney corresponds to the sum of non zero pixels' position divided by the number of pixels.
Gravity center = (M_x, M_y) with:

$$M_x = \frac{\sum_j \sum_i I(i,j) x_{ij}}{\sum_j \sum_i I(i,j)}$$

where $x_{i,j} = i * voxelSizeX$

$$M_y = \frac{\sum_j \sum_i I(i,j) y_{ij}}{\sum_j \sum_i I(i,j)}$$

where $y_{i,j} = j * voxelSizeY$

Each slice represent a time instant t of MRI acquisition. Since there is only one gravity center per slice, each couple of indexes are stored in a list, respectively to the acquisition time of acquisition of the MRI slice they were computed from.

Kidney directions computation

The kidney directions will be used for finding rotations. The directions can computed the PCA algorithm [Tipping and Bishop, 1999]. PCA algorithm finds the directions of the

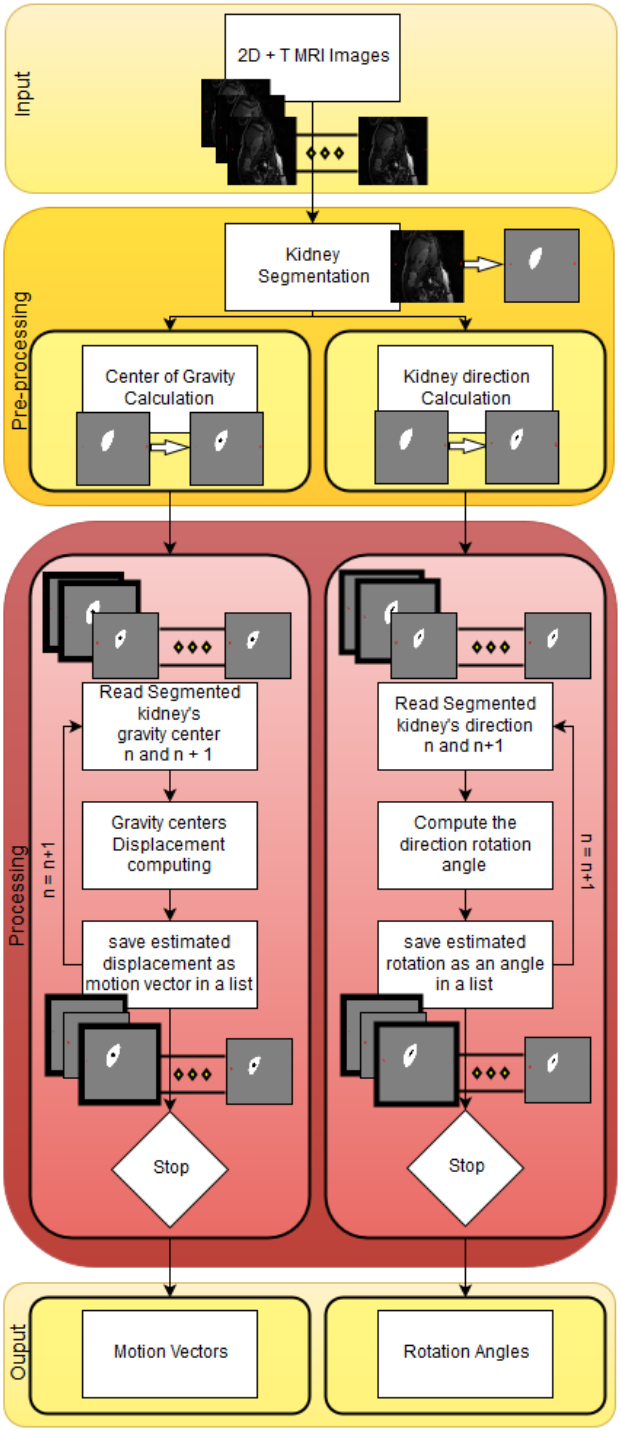
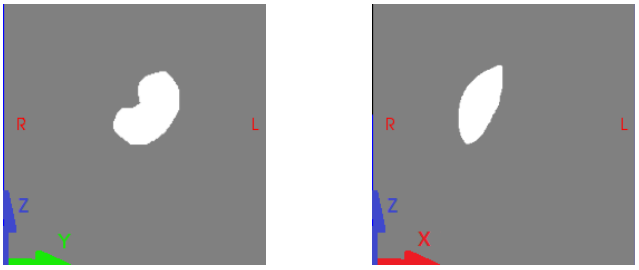


Figure 3: Characterization methodology Diagram.
Inputs are 2D+T images. Pre-processing consists in the kidney segmentation, followed by the calculation of the gravity center and direction of the kidney on each slice. Processing contains two loops done on a set of images. The image n is the search image when the image n+1 represents the reference image. The outputs are two lists of translation vectors and rotation.



(a) Coronal view (b) Sagittal view

Figure 4: Coronal and sagittal view of segmented kidney

maximal variance of data. It is actually the eigenvector of the covariance matrix. Finding an angle between two vectors is preformed using the dot product of vectors [Banchoff and Wermer, 2012].
We used the implementation of the PCA from the VTK library [VTK.org, 2016]. VTK is highly convenient for this task. Using the "vtkPCAStatistics" we have a complete abstraction of the algorithm.
For the same reason as for the gravity centers, the angles are stored in a list, respectively to the acquisition time of the images they were extracted from.

Gravity centers displacement

Gupta Et Al. propose to perform image registration by measuring the displacement between two successive masked kidney gravity center [Gupta *et al.*, 2003]. They obtain the shift values which enables them to translate a masked image of the kidney over the next image. We will base our method this algorithm to find the displacement coordinates which will be saved as motion vectors.
Under the hypothesis of local translation between pairs, our algorithm consists in the evaluation of the gravity center displacement between the reference and the search segmented images. The reference image being the first image of the acquisition and the search image being the current image. For each couple reference (r) and search (s) gravity center the relative displacement δX, δY will be calculated as:

$$\delta X = M_x^0 - M_x^s$$

$$\delta Y = M_y^0 - M_y^s$$

The result $vect = (\delta X, \delta Y)$ will represent the estimated vector of displacement between the two images.
In order to make this model more informative, we decided to add to the displacement, data related to the rotation of the kidneys. The rotation will be computed using directions variation. It is hence necessary to compute the direction of each kidney segment.

Rotation

Under the hypothesis of local rotation between pairs, we also calculate an angle of rotation between kidney directions.
The direction of each kidney was previously done done and stored in a list. We follow the same process of comparison between reference (initial) and search directions. Supposing the 2D slice axis (x,y), the angle θ_i is defined as

$$cos(\theta_i) = \frac{(\vec{v}_0 \bullet \vec{v}_i)}{||\vec{v}_0||||\vec{v}_i||}$$

with $(\vec{v}_0 \bullet \vec{v}_i)$ the dot product of $\vec{v}_0$ and $\vec{v}_i$ [Banchoff and Wermer, 2012]:

$$\vec{v}_0 \bullet \vec{v}_i = v_0^x \times \vec{v}_i^x + \vec{v}_0^y \times \vec{v}_i^y + \vec{v}_0^z \times \vec{v}_i^z$$

We now characterized the kidney motion. We will next make a model which estimates the position and direction of the kidney.

4.2 3D model

We focuses our model on Fayad et al.'s work [Fayad *et al.*, 2009] which builds a direct model of the respiratory motion by estimating a 2D-Spline of the motion. They use as surrogate data the amplitude and the phase of the movement and the displacement factors to generate a continuous patient specific model function. To recover the position one just needs to know the current phase and amplitude of the breathing and refer to the 2D-Spline.
This 3D model is based on Lee et al. algorithm for Scattered Data Interpolation with B-Splines [Lee *et al.*, 1997]. The B-spline is generated using surrogate data (amplitude and phase of the motion) matched to the induced displacement/rotation computed in the previous part. Control points of the B-spline are represented as a function $\phi_{(i,j)}$.
3D positions displacement on axes x, y and z will be recovered using respectively the functions p_x, p_y and p_z.
The 2D direction rotation on y and z will be recovered using the functions d_y and d_z.
These functions are defined as bellow:

$$\begin{aligned}
p_x(r,\varphi) &= \sum_{k=0}^{3}\sum_{k=0}^{3} \beta_k(s)\beta_l(t)\phi^x_{(i+k)(j+l)} \\
p_y(r,\varphi) &= \sum_{k=0}^{3}\sum_{k=0}^{3} \beta_k(s)\beta_l(t)\phi^y_{(i+k)(j+l)} \\
p_z(r,\varphi) &= \sum_{k=0}^{3}\sum_{k=0}^{3} \beta_k(s)\beta_l(t)\phi^z_{(i+k)(j+l)} \\
d_y(r,\varphi) &= \sum_{k=0}^{3}\sum_{k=0}^{3} \beta_k(s)\beta_l(t)\phi^{d_y}_{(i+k)(j+l)} \\
d_z(r,\varphi) &= \sum_{k=0}^{3}\sum_{k=0}^{3} \beta_k(s)\beta_l(t)\phi^{d_z}_{(i+k)(j+l)}
\end{aligned} \tag{1}$$

with i = $\lfloor r \rfloor - 1$, $j = \lfloor \varphi \rfloor - 1$, $s = r - \lfloor r \rfloor$, $t = \varphi - \lfloor \varphi \rfloor$ and $\beta_k(t)$ and $\beta_l(t)$ the uniform cubic B-spline basis function defined as:

$$\beta_0(t) = \frac{1-t^3}{6}$$

$$\beta_1(t) = \frac{3t^3 - 6t^2 + 4}{6}$$

$$\beta_2(t) = \frac{3t^3 + 3t^2 + 3t + 1}{6}$$

$$\beta_3 = \frac{t^3}{6}$$

We will first describe our surrogate data then present the B-spline generation method.

Surrogate data

By using the estimation functions (1) we will recover the kidney displacement from 1st image of the kidney on a specific axis.
In order to use these functions, we need to have the r and φ values, which in our case will be the amplitude and phase of the breathing movement. But we also need to determine a learning method taking the gravity center displacement matched with the Amplitude and the phase of the breathing movement at the time of the image acquisition in order to compute ϕ_{ij}.
Our Surrogate data will be the amplitude and phase of the breathing movement and the kidney gravity center displacement.

- z: The displacement or rotation of the kidney will be extracted as showed in section 4
- r: The amplitude of the breathing movement was recorded by a belt during the image acquisition. Although, since there is a correlation between the breathing movement and the kidney motion, during our experiments we used the amplitude of the position set of the kidney gravity center on a given axis as the amplitude of the breathing movement.
- φ: The phase which was extracted from the amplitude set using the Hilbert Transform [Freeman, 2007].

For the learning process, a subset of these surrogate values will be scaled and put together in order to make a proximity set of control point $\phi_{i,j}$ is represented as

$$P_{ij} = \{(r_c, \varphi_c, z_c) \in P | i-2 \leq r_c < i+2, j-2 \leq \varphi_c < j+2\} \quad (2)$$

Spline estimation

With P we can now estimate a B-splane defined by control points ϕ so that ϕ_{ij} satisfies:

$$z_c = \sum_{k=0}^{3} \sum_{l=0}^{3} \omega_{kl} \phi_{kl}$$

for each point $c = (r_c, \varphi_c, z_c)$ in P_{ij} we then build ϕ_c as:

$$\phi_c = \frac{\omega_c z_c}{\sum_{a=0}^{3} \sum_{b=0}^{3} \omega_{ab}^2}$$

with $\omega_c = \omega_{kl} = \beta_k(s)\beta_l(t)$ and $k = (i+1) - \lfloor r_c \rfloor$, $l = (j+1) - \lfloor \varphi_c \rfloor$, $s = r_c - \lfloor r_c \rfloor$, $t = \varphi_c - \lfloor \varphi_c \rfloor$

An example of learning set is plotted in figure 5

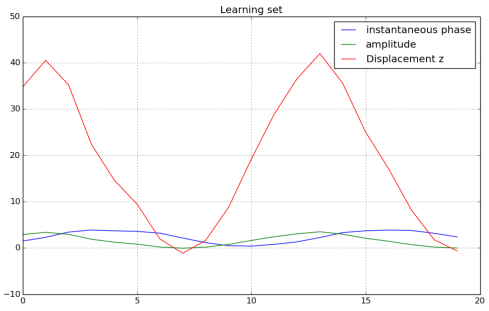


Figure 5: Learning set example

5 Experiments

5.1 Experiments setup

We experimented this model on four subjects performing deep breathing. The first step of the experiment is to compute the motion characteristics (x, y, z position and rotation angles) for each subject, as an example the subject 3 in figure 6.
Then, their breathing motion amplitude and phase and the kidney gravity center displacement are computed for the whole acquisition. Although, only image 5 to 25 data are used for the learning (as in figure 5). The rest of the data will be subject to the same scaling as the training set which was imposed by (2) and used for the estimation of the movement and for its validation.
As you can notice from figure 4, coronal and sagittal views share the z axis. We will use for the model the z axis with the best correlation with the actual values.
One model will be made per subject.

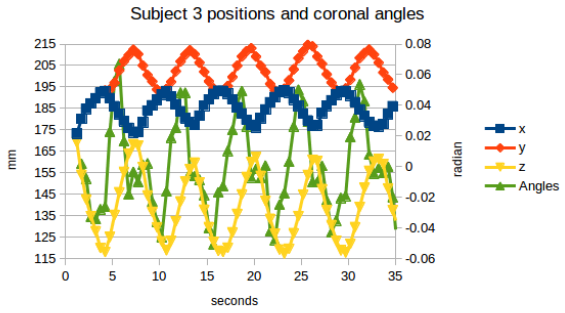


Figure 6: Subject 3 motion characteristics

5.2 Model performances evaluation

The covariance correlation between the estimation derived from the estimation functions (1) and will be used to evaluate the similarity of their behaviour. The correctness of the estimations will be evaluated in terms of their mean squared difference.

5.3 Results

Figure 7 regroups the results in terms of performance of the estimation compared to the actual results.
Figure 8 displays the averages of performances.
Figure 9 represent the results on axial plan.

Subject	axis	Correlation	Squared difference
1	x	0.9684	0.3222
	y	0.9780	1.1786
	z	0.9862	4.3213
	S. Rotation	0.5179	6.1753E-05
	C. Rotation	0.5861	0.0009
2	x	0.9977	0.3222
	y	0.9780	0.7522
	z	0.9894	4.6199
	S. Rotation	0.6264	8.8814E-05
	C. Rotation	0.5315	0.0006
3	x	0.9826	3.7626
	y	0.9805	2.3272
	z	0.9761	12.9134
	S. Rotation	0.9252	0.0004
	C. Rotation	0.9607	7.7086E-05
4	x	0.8529	8.0733
	y	0.7373	0.1348
	z	0.9629	11.0296
	S. Rotation	0.6512	0.0001
	C. Rotation	0.5390	0.0005

Figure 7: Model performances

Average correlation	0.8371
Average difference	2.4782
Displacement Average correlation	0.9503
Displacement Average difference	4.1301
Rotation Average correlation	0.6673
Rotation Average diffrence	0.0003

Figure 8: Mean performances

Subject	axis	Correlation	Squared difference
1	x	0.7308	1.5102
	y	0.8176	0.4749
	Rotation	-0.0648	6.1753E-05
2	x	0.9851	0.9272
	y	0.9593	0.2020
	Rotation	0.6660	0.0013
3	x	0.9791	4.2678
	y	0.9692	0.2130
	Rotation	0.6910	0.0058

Figure 9: Axial model performances

6 Discussion

The four subjects had different breathing characteristics. Their breathing motion was more or less settled, only subject 3's kidney seemed to have a notable kidney rotation (as you can tell with the example of the comparison between the subject 2 Figure 10 and subject 3 Figure 11).
Although, Figure 8 shows that with this model, the estimation has a high global correlation of coefficient 0.84 and a low average squared difference of coefficient 2.48.
These results show that this model allows to estimate the kidney's motion. We may also note that the average correlation is lowered down by the rotation prediction of the 3 subject who don't have a rotating Kidney. These prediction try to predict noise, not performing rotation prediction on such subject will allow to have even better results.

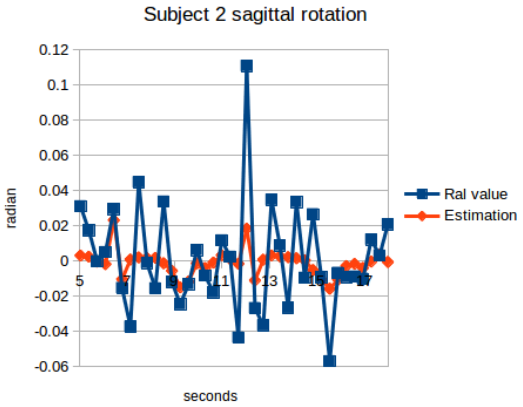


Figure 10: Estimation of rotation on subject 2

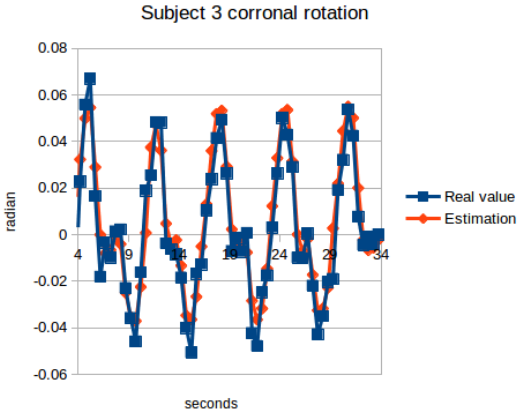


Figure 11: Estimation of rotation on subject 2

We can notice some irregularity in the efficiency for the displacement prediction, in terms of correlation for subject 3 x and y axes, and in terms of squared difference for subject 2 z axis with the highest coefficient of 12.91.
These irregularity highlight two conditions that must be avoided while using this model.

Subject 4 was having the less settled breathing during the acquisition (which does not respect our objective acquisition during regulated breathing). Figure 13 highlights these irregularities. Although the estimation seems to follow the the actual value, by having pikes at the same time, there is a non significant loss of precision. This imprecision comes from the fact that the learning set is built with higher values amplitudes than the data used for the prediction. Which is getting the average of the proximity control points higher than it should be.
This model must then be performed on settled breathing. This does fit our requirements.

Subject 3 has a really high correlation coefficient, although it seems to suffer from a lack of precision regarding its squared difference from the actual data.
This case is the opposite to the previous one, higher amplitude in the predictions input than the values used in the learning provoke a going out of bound of the model and induces lower estimation. This highlights that a specific care must be taken to make sure that the floor value of the scaled amplitude and phase of the recovering set does not exceed the maximal value for the floor of the training set.

We may also notice from Figure 9 that axial view models suffer from lower correctness efficiency. We may point out that the axial results are not taken into account in Figure 7 nor 8. Indeed, if subject 1 has estimations on x, y and z axes in Figure 7 for which the correlations with actual values don't go bellow 0.96, the x and y axis correlations with the actual values of the axial view don't go above 0.8. Also it's rotation has a negative correlation coefficient. We may also notice that most of the rotation squared differences have significantly higher values on axial view than the other view. These irregularities also come from the nature of the breathing motion and the shape of the kidney. Referring to Figure 2, the axial view is taken from above the kidney. Knowing that the kidney is more or less to the craniocaudal direction and that the major part of the kidney motion is on that same direction [Neree PAYAN, 2015], each image acquisition represent a different slice of the kidney and sometimes disappears from the MRI slice (which is noticeable Figure **??** where the cycle is truncated). Which is against our objective to characterize the motion of the kidney from a rigid translation through 2D+T images. Hence, the kidney displacement acquired is not necessarily due to the motion, but rather to the irregularity of the kidneys shape. The rotation prediction lack of precision comes from the shape of the kidney which is, from the axial view, a circular shape and may varry along the kidney. The PCA variation then does not necessarily give us information about the rotation.
Although, coronal and sagittal characterisation giving us information about x, y and z position, and two rotation angles, it is possible to generate a 3D model of the motion from both of these views.

Under the conditions of settled breathing our model shows good results. A condition which is less invasive than the current methods which requires unwell patients to go into apnea in order to stop the breathing motion.

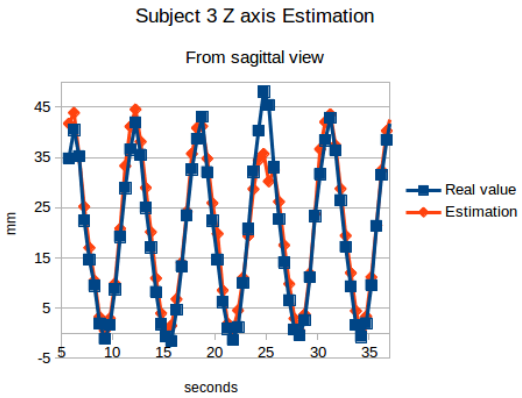


Figure 12: Estimation of z axis on subject 3

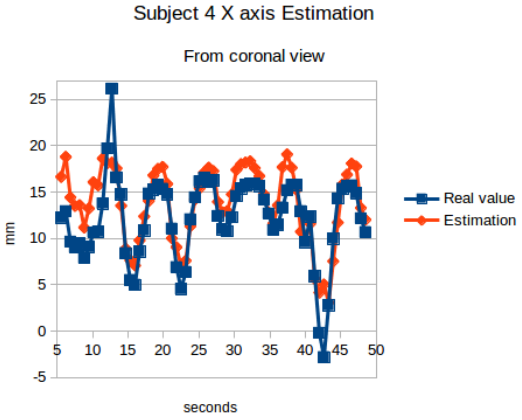


Figure 13: Estimation of x axis on subject 4

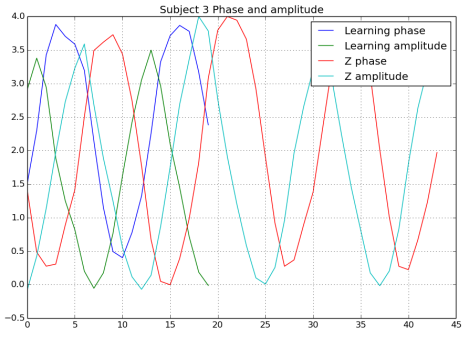


Figure 14: Subject 3 phase and amplitude

7 Conclusion

The kidney motion characterization is a crucial step for automatizing the kidney punctures. In this paper we expressed a methodology which allows to have a first characterization of the kidneys motion and we were able to build a 3d model of this motion with satisfying results.
We characterized the motion with a translation and a rotation per 3D direction. Two algorithms were presented In order to estimate them. One using the kidney gravity centers to find the kidney's translation and and the other which using the direction generated with the PCA algorithm to find the rotation of the kidney through time. We used a 2D B-spline to build a model for each axis of the 3D space and for the rotations. This model has had good results but we were able to highlight some weakness, coming from both the nature of the kidney motion and the model itself.
One first downside of the model is that not all kidneys happen to have a clear rotation, if we stated in the discussion that the estimation of the rotation on the non rotating kidneys were less noisy than the acquisition by performing a mean value filter, our data set doesn't allow us to evaluate the accurateness of this estimation.
The second issue with this model is that the breathing movement is not naturally settled. A breathing cycle different from the training set will damage the precision of this model.
We also noticed that this motion characterisation can't use all of the coronal, sagittal and axial views as input, the axial view not being appropriate to our kidney motion characterization algorithm. Although, corronal and sagittal view are suffishent to characterise the 3D motion. Careful care must be taken to make sure that the subject is breathing regularly and preliminary check weather the subject's kidney rotates during the breathing movement.

References

[Banchoff and Wermer, 2012] Thomas Banchoff and John Wermer. *Linear algebra through geometry*. Springer Science & Business Media, 2012.

[Bricault *et al.*, 2008] Ivan Bricault, Nabil Zemiti, Emilie Jouniaux, Céline Fouard, Elise Taillant, Frédéric Dorandeu, and Philippe Cinquin. Light puncture robot for ct and mri interventions. *IEEE Engineering in Medicine and Biology Magazine*, 27(3):42–50, 2008.

[Bussels *et al.*, 2003] Barbara Bussels, Laurence Goethals, Michel Feron, Didier Bielen, Steven Dymarkowski, Paul Suetens, and Karin Haustermans. Respiration-induced movement of the upper abdominal organs: a pitfall for the three-dimensional conformal radiation treatment of pancreatic cancer. *Radiotherapy and Oncology*, 68(1):69–74, 2003.

[Fayad *et al.*, 2009] Hadi Fayad, Tinsu Pan, Christian Roux, Catherine Cheze Le Rest, Olivier Pradier, and Dimitris Visvikis. A 2d-spline patient specific model for use in radiation therapy. In *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 590–593. IEEE, 2009.

[Fouard *et al.*, 2012] Céline Fouard, Aurélien Deram, Yannick Keraval, and Emmanuel Promayon. Camitk: a modular framework integrating visualization, image processing and biomechanical modeling. In *Soft tissue biomechanical modeling for computer assisted surgery*, pages 323–354. Springer, 2012.

[Freeman, 2007] W. J. Freeman. Hilbert transform for brain waves. 2(1):1338, 2007. revision 91355.

[Giele *et al.*, 2001] ELW Giele, JA De Priester, JA Blom, JA Den Boer, JMA Van Engelshoven, Arie Hasman, and M Geerlings. Movement correction of the kidney in dynamic mri scans using fft phase difference movement detection. *Journal of Magnetic Resonance Imaging*, 14(6):741–749, 2001.

[Gupta *et al.*, 2003] Sandeep N Gupta, Meiyappan Solaiyappan, Garth M Beache, Andrew E Arai, and Thomas KF Foo. Fast method for correcting image misregistration due to organ motion in time-series mri data. *Magnetic Resonance in Medicine*, 49(3):506–514, 2003.

[Hostettler *et al.*, 2010a] Alexandre Hostettler, Daniel George, Yves Rémond, Stéphane André Nicolau, Luc Soler, and Jacques Marescaux. Bulk modulus and volume variation measurement of the liver and the kidneys in vivo using abdominal kinetics during free breathing. *Computer methods and programs in biomedicine*, 100(2):149–157, 2010.

[Hostettler *et al.*, 2010b] Alexandre Hostettler, SA Nicolau, Y Rémond, Jacques Marescaux, and Luc Soler. A real-time predictive simulation of abdominal viscera positions during quiet free breathing. *Progress in biophysics and molecular biology*, 103(2):169–184, 2010.

[Lee *et al.*, 1997] Seungyong Lee, George Wolberg, and Sung Yong Shin. Scattered data interpolation with multilevel b-splines. *IEEE transactions on visualization and computer graphics*, 3(3):228–244, 1997.

[McClelland *et al.*, 2013] Jamie R McClelland, David J Hawkes, Tobias Schaeffter, and Andrew P King. Respiratory motion models: a review. *Medical image analysis*, 17(1):19–42, 2013.

[Neree PAYAN, 2015] Celine FOUARD Neree PAYAN, Julie FONTECAVE. Grenoble alpes university, timc-imag. Master's thesis, Grenoble Alpes University, 2015.

[Schroeder *et al.*, 1996] William J Schroeder, Kenneth M Martin, and William E Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th conference on Visualization'96*, pages 93–ff. IEEE Computer Society Press, 1996.

[St-Pierre, 2012] Christine St-Pierre. *Évaluation des impacts anatomique et dosimétrique des mouvements induits par la respiration*. PhD thesis, Université Laval, 2012.

[Tipping and Bishop, 1999] Michael E Tipping and Christopher M Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.

[VTK.org, 2016] VTK.org. VTK examples pcademo, 2016. [Online; accessed 7-June-2016].

[Yushkevich *et al.*, 2006] Paul A. Yushkevich, Joseph Piven, Heather Cody Hazlett, Rachel Gimpel Smith, Sean Ho, James C. Gee, and Guido Gerig. User-guided 3D active contour segmentation of anatomical structures: Significantly improved efficiency and reliability. *Neuroimage*, 31(3):1116–1128, 2006.

Integrating Split Drivers in Linux

Tima Laboratory, Grenoble, France
Magistère Informatique de Grenoble

Thomas Baumela
Supervised by:
Pr. Olivier Gruber, **Pr. Frédéric Pétrot**

Abstract

Recent studies show that 70% of Linux Kernel crashes are caused by device driver bugs. Device drivers are hard to write because they are highly dependent on both the specifics of each operating system and the minute details of hardware devices. Furthermore, since improved or new devices constantly hit the market, providing safe drivers is a never-ending struggle. Also, device driver developers are under time pressure because new devices must have their driver to be usable. This is of course something important for hardware manufacturers but it is also important for operating system providers if they want their operating system to be able to run on the latest hardware.

To address these problems, we advocate that split drivers is a promising direction. The concept of split drivers has been popularized by the USB standard [21], with a convenient plug-and-play experience for many hardware devices. With a split-driver, the usual device driver is split in two parts, a front-end driver and a back-end driver. The back-end driver is the complex hardware-related part of the driver, but it now executes on the device, not on the host. The front-end driver executes on the host, loaded by the operating system, but it is now a simpler and safer class-generic driver. The two parts, the front-end and back-end drivers, communicate using messaging, which provides the necessary failure isolation.

We have extended the device driver bus framework of Linux to integrate the concept of split drivers. The Linux bus framework is modeled after the PCI hardware bus [12], assuming a plug-and-play approach where newly plugged-in devices are matched with compatible drivers. Once a driver is matched to a device, it is assumed that the driver interacts with its device through memory-mapped input-output registers, interrupts, and direct memory transfers. Through our extension, drivers and devices communicate through message-oriented channels.

This assumes that devices are self-described and implements message-oriented interfaces. This rely on the idea of standardized classes of devices, defining the messaging protocol for each class. We argue this trend is the right direction, but the class standardization must be independent from the hardware specifics of the underlying bus technology. Our work is a first step in that direction.

1 Introduction

Writing a device driver is hard, some say it is black magic. Writing a device driver requires a very detailed understanding of the kernel hosting that driver, the Linux kernel in our case. But writing a device driver also requires a perfect understanding of the hardware device itself. Not only writing a device driver is hard, but buggy device drivers usually mean an unstable machine and most often a brutal kernel crash. Of course, device drivers become more and more stable over time, like any other software, but hardware devices constantly change and new devices appear all the time. Writing device drivers and getting it right is therefore a never-ending struggle.

Since the 1960's device drivers are based on writing and reading values into device registers. But in contrast to user-space programming, writing and reading regular memory locations, reading and writing device registers are about communicating with the device, following a communication protocol that is specified for each device. For instance, this implies that some of these operations need to be performed in a particular order and sometimes using precise timing. The necessary information is of course available in the hardware documentation, if the device manufacturer provides it, otherwise, retro-engineering is necessary. Even when available, hardware specifications are complex and notoriously incomplete from the perspective of device driver writers. The work of [9] and [11] shows well how complex drivers are and how difficult it is when it comes to update them because of the huge proliferation and interaction of them.

Device drivers are often provided by the manufacturer itself, who deeply knows the behavior of its device. However, the challenge becomes the required knowledge of the hosting operating systems, such as Windows, Mac-OS, or Linux [10]. In the embedded world, many other operating systems exist, such as WindRiver [22], eCos [4], or variants of Linux like uClinux [19]. This is a challenging situation since it is hard to find developers that are both hardware and software experts. Furthermore, drivers must be maintained as both the hardware and software evolve, either because new versions of devices hit the market or because new versions of the operating system are released. It is therefore challenging for device driver developers to stay current and deliver up-to-date and safe device drivers.

To make things worse, device drivers must be available in a timely manner. This is obviously important for device manufacturers since devices can only be used if they have available drivers in the mainstream operating systems. In the embedded world, the situation is even worse with the wide diversity of proprietary operating systems. Therefore, driver availability also becomes important for operating systems if they want to be able run on the latest hardware, being able to manage the latest hardware devices. This issue of the availability of device drivers has been plaguing the industry for decades now.

In that respect, the USB standard [21] has been a game changer, providing a successful plug-and-play experience for a wide variety of devices. The key concept is the concept of split drivers, with both parts communicating through messaging. In the USB world, devices belong to standardized classes, each class of devices defining message-based interfaces. For instance, a mass storage understands SCSI [16] commands. Similarly, they are interfaces defined for different media devices (sound, video) as well as human interface devices (HID) [5]. In this world, operating systems can host class-generic drivers, communicating with devices by exchanging standardized messages.

This means no more black magic, no more complex technical reference manuals that are thousands of pages long. Drivers are just exchanging messages, something that any software developer knows how to do. Drivers become easy to write. More importantly, drivers become safer. First, they only use messaging, no more playing around with hardware registers, interrupts, and direct memory transfers. Second, they are class-generic drivers that will last, allowing them to become safer over time as bugs are ironed out.

Unfortunately, the USB revolution has been perceived by the operating system community as only yet another bus technology. For instance, Linux has integrated support for the USB bus, like it did for other bus technologies such as I2C [7], PCI, or Thunderbolt. While the approach certainly did the job of allowing the use of USB devices on machine running Linux, it failed to foresee the potential of the approach as a game changer for the operating system community. Of course, the USB standard is about a single bus technology, but it shows a path to solve the driver challenge by evolving the software-hardware frontier. Moving away from memory-mapped registers, interrupts, and direct memory transfers towards messaging changes everything.

Unfortunately, the USB specification has been lacking the necessary separation of concerns, mixing the standardization of device classes along with the standardization of a message-oriented bus technology. The USB standard covers everything from define what a mouse is, down to the bus technology, and even the wire and plug formats. The SCSI standard made the same mistake early on but corrected it early on. Indeed, the SCSI standard started with mixing the definition of a parallel bus and the SCSI messages. Today, the SCSI messages can flow through many different bus technologies, such as PCI, USB, fiber optics, and the Internet. We therefore believe it is important to push towards split drivers, based on standardized messaging, but independently from any bus or network technologies.

This work is a first step in that direction, looking at how to extend the Linux device-driver bus framework, including the concept of split drivers communicating through messages. This document is structured as follows. First, we will discuss the related state of the art in Chapter 2. Second, we will present the current design for the Linux device driver bus framework in Chapter 3. Third, we will present our split-driver extension to that Linux framework in Chapter 4 Fourth, we will discuss our validation and experiments in Section 5. Finally, we will conclude in Section 6.

2 State of the Art

This section is about the state of the art related to devices, buses, and device drivers. Since the dawn of our industry, the software-hardware frontier has been based on memory-mapped input-output registers (MMIO), interrupts, and direct memory access transfers (DMA). As a result of this frontier, operating systems must include dynamically loaded modules that act as device drivers. As the name suggests, these modules *drive* hardware devices, reading and writing MMIO registers, responding to interrupts, and controlling DMA transfers.

As bus technologies evolved, two schools of thought appeared. One school fought to preserve this software-hardware frontier, thereby preserving existing drivers, while the other pushed for a change based on messaging. The PCI bus is the most successful champion of preserving the software-hardware frontier. In contrast, the I2C bus is advocating messaging through a serial bus. Today, even though most bus technologies adopted serial technologies for faster bandwidth, including PCI, these two schools of thoughts are still very present. The modern USB and Thunderbolt[18] standards are illustrative of these two schools of thoughts. The USB follows in the I2C footsteps, pushing for messaging, while the Thunderbolt pushes to retain a PCI interface.

Along with pushing for messaging, the USB standard has been pushing for split drivers based on the stan-

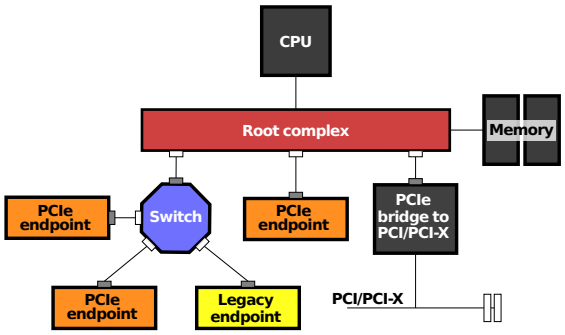


Figure 1: Example of PCIe Topology (Wikipedia Commons)

dardization of device classes, each class implementing message-oriented interfaces. This is important as this permits operating systems to host class-generic devices. This means that a driver is no longer a driver of a specific device, it is the driver of any device belonging to a given class. This changes everything. For instance, a single driver is necessary to drive all USB mass storage, that they are simple USB keys or that they are full external disks, no matter the vendor, and no matter the internal technology (flash, SSD, or traditional disk).

As a side note, this state of the art is mostly based on specifications and available documentations from the Web, from Wikipedia to various corporation announcements. The descriptive parts of the sections are freely inspired from those sources. Most of the figures are borrowed from the same sources. However, the analysis and comparison of the presented technologies represent our own personal evaluation and opinions; they do not necessarily represent the evaluation and opinions of the associated trademark owners.

2.1 PCI Bus

PCI bus is a board-local bus for attaching hardware devices. The PCIe [13] stands for PCI-express and is a serial replacement of the older parallel PCI bus. PCIe is based on point-to-point topology, as depicted in Figure 1, with separate serial links connecting every device to the root complex (host). PCIe bus link supports full-duplex communication between any two endpoints. PCIe communication between two PCIe devices is encapsulated in bus transactions, allowing both devices to send and receive ordinary PCI requests (configuration, I/O or memory read/write) and interrupts.

PCIe supports read/write requests over three address spaces: memory, I/O address, and configuration. Memory addresses are 32 bits or 64 bits and support caching. I/O addresses are for compatibility with the Intel x86 architecture's I/O port address space. The configuration space provides access to 256 bytes of special configuration registers per each device. These registers are used to configure devices when hot plugged, assigning to them memory and I/O address ranges. Overall, a host controller can configure a network with up to 256 buses, each with up to 32 devices, each supporting eight functions. For each device, as depicted in Figure **??**, the first 64 bytes of its configuration space are standardized; the remainder are available for vendor-defined purposes.

Our analysis of the PCIe bus technology is that it marked a turning point for device driver management because of its configuration feature. The configuration feature of PCIe is fundamental not only in supporting hotplug devices but also in helping with matching the correct driver with a device. It is important to note that PCIe devices are required to be self-described and configurable. The approach was clever though to avoid over specification; the PCIe specification standardizes only a minimum, mostly about devices describing what they are. This minimum permits a safer matching of drivers onto devices. From that point on, the dialog between a driver and its device is a classical one, it is based on MMIO registers, interrupts, and DMA transfers. In other words, drivers remain entirely device specific.

2.2 Universal Serial Bus

The USB technology was standardized in the mid-1990s with the intent to define the cables, connectors, and communication protocols used for connecting peripherals such as keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters.

The goal was to make it fundamentally easier to connect external devices, replacing the chaotic jungle of connectors at the back of the personal computers in the mid-1990s. This explains that the USB standard is about wires, plugs, and electrical specifications. But the USB standard is also about simplify software configuration of connected devices, always striving for greater data rates for external devices that could benefit from it.

A physical USB device may consist of several logical sub-devices that are referred to as device interfaces. For example, a web-cam with a built-in microphone would provide two interfaces, one video function and one audio function. Communications to and from a interface is based on messages flowing through pipes. A pipe is a logical communication channel from the host to an endpoint on a device. Such endpoints are attributed to interfaces. Four types of pipes are defined by the USB standard: control, interrupt, bulk, and isochronous.

- A control pipe is bi-directional and used for sending short and simple commands and receiving status responses—hence the name control pipe as they serve to control devices.
- Interrupt pipes are about devices that are sources of data such as keyboards, mice, or ethernet cards.
- Bulk pipes are about large sporadic transfers, they are typically used for mass storage devices.
- Isochronous pipes have guaranteed data rate, but with possible data loss. Isochronous pipes are typically used for realtime audio or video.

These four types are related to the complex scheduling of data frames over the wire. Since the early version of the USB technology had low bandwidth, multiplexing pipes over shared wires was challenging. It was important that devices could always be controlled, no matter if there were other pipes with high bandwidth requirements. So the scheduling of frames always reserves a percentage of the physical bandwidth for control frames.

The introspection of devices is interesting to detail. The USB technology requires each device to be fully self-described, via messages on the endpoint 0 (the only endpoint not associated to an interface). In other words, a USB device always supports the endpoint 0 and always understands the basic configuration messages defined by the USB specification. Through such messages, the host can obtain a complete description of a device, from that device. The description is mostly about the device interfaces and endpoints. That way, the host operating system can discover which devices are plugged in and safely match corresponding drivers. Furthermore, drivers know about the available endpoints and their type.

Device interfaces are meant to be standardized, that is, their message-oriented protocols must be defined independently from any specifics of any particular device. Accordingly, the USB promotes the concept of device classes, with interfaces being standardized for each class of devices. For instance, a mass storage device would implement the SCSI standard (both commands and status). This standardization permitted the implementation of class-generic front-end drivers as opposed to device-specific front-end drivers. As a result, the availability and quality of front-end drivers dramatically improved, leading to a satisfactory plug-and-play experience for end users and consequently a very attractive technology for device manufacturers—quite a win-win situation that ensured the USB success.

It is our analysis that although the USB technology is inspirational about split drivers based on class-generic message-oriented interfaces, the USB technology is also an example not to follow in that it binds together different specifications that ought to be independent. In that regard, the SCSI specification ought to be an inspiration as an SCSI driver can drive an SCSI device across a wide variety of buses and networks, such as PCIe, USB, fiber optics, and IPv6. We ought to define a software bus that can host and match front-end and back-end drivers, while fully abstracting away any specifics of the underlying bus or network.

The concept of message-oriented communication channels delivers just that. This is fact quite similar to traditional sockets or message queues in software middleware, providing hardware-independent communication channels. But just like the Internet differentiate between TCP and UDP on their properties, the USB specification has introduced different semantics for endpoint pipes (control, interrupt, bulk, and isochronous). But are these distinct semantics still necessary or are they leftovers from early versions of the USB specification?

It would be wise to preserve as much as possible of the

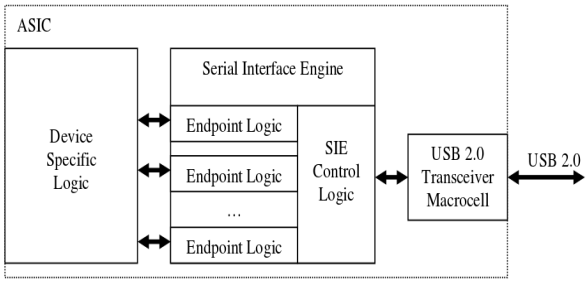


Figure 2: USB Serial Interface Engine (Intel USB specification)

USB specification of device classes and their corresponding message-oriented interfaces. Linux has done exactly that when it adopted the HID specification from USB. As it will be clear later, our approach has been designed in order to be able to wrap a USB network, protecting the overall investment that the industry made in USB software and hardware stacks. But longer term, we would expect a better layering to happen, both in software and hardware.

Device side, the situation is more delicate to handle. The USB specification has put a lot of effort in making sure that device manufacturers could easily and cheaply move their devices to the USB technology. First, this means that the extra hardware cost induced must be low. Second, the complexity of managing endpoints and messages must be also be low. In particular, it was important that USB devices could still be simple hardware automata. Requiring even a small microcontroller per device would have been catastrophic.

Figure 2[1] presents the USB solution, device side. First, to preserve slow ASIC (around 40MHz to 60MHz), the PHY layer of the USB stack was moved to a independent transceiver macrocell, running at 480MHz at USB 2.0, even faster for newer releases of USB. Second, a VHDL drop-in module was provided that encapsulates the management of endpoints and pipes. The SIE Control Logic contains the USB address recognition logic, and other sequencing and state machine logic to handle USB packets and transactions. The Endpoint Logic contains the endpoint specific logic: endpoint number recognition, FIFOs and FIFO control, etc.

In order to preserve the investments made in USB by our industry, we must consider solutions that preserve most of the USB bus technology at an hardware level, which means preserving both the host controller and Serial Interface Engine control logic. Our solution should permit exactly that, while also permitting to use split

[1] http://www.intel.fr/content/dam/www/public/us/en/documents/tech specifications/usb2-transceiver-macrocell-interface-specification.pdf

drivers over other bus and network technologies.

3 Linux Device Driver Bus Framework

This section gives an overview of the Linux driver model. We will see how Linux organizes drivers and devices and discuss the corresponding programming model given to kernel developers.

3.1 Linux Bus Model

Device drivers are organized in Linux following the concept of hierarchical buses. Some of these buses are matching hardware buses, such as PCI or USB. Other buses are about software abstraction such as the bus grouping devices that are Human Interface Devices. For each bus, three steps are important—reified devices, registered drivers, and matching.

First, for each bus, devices must be reified. This can happen as the result of processing a given description of which devices are plugged on that bus. An example of such description is the device tree given to the kernel on ARM that describes the different hardware buses and the hardware devices that they host. A bus can also support hot plugging of devices, such PCI buses, allowing to enumerate hardware devices. The enumeration is used by the bus implementation to reify devices that represent on the bus at any given time.

Second, for each bus, the necessary device drivers must be registered. In Linux, each device driver is written to register to a given bus, something that happens when the module containing the device driver is loaded. Modules can be pre-loaded or they can be loaded on demand as the result of enumerating new devices. This is the case for example with the USB bus, looping back to the udev mechanism that controls which modules are loaded for driving devices.

Third, each bus matches locally drivers to devices. To do this matching, the bus relies on devices describing what they are and on drivers describing which devices they can driver. This is typically done through different identifiers such as device IDs and vendor IDs.

Then, Linux supports stacking buses, where device drivers in one bus reifies themselves as devices in other buses. For instance, the driver of a USB mouse, on the USB bus, might reify itself as an Human Interface Device (HID) in the HID bus. Stacking buses help organize the overwhelming number of devices and their drivers into a hierarchy of concepts that separate concerns, encapsulate details, and help the necessary composition. For instance, the HID bus can compose different pointing devices as a single pointer that can be used by the window manager to drive the mouse icon on the screen.

3.2 Device, Drivers, Buses

Linux bus model articulates three core concepts: drivers, devices, and buses. Each device represents a software view of an hardware device such as a PS/2 mouse, an Ethernet controller, an audio controller, or a USB controller. Devices can be very close to their hardware device or very abstract, or even software devices with no corresponding hardware device. Drivers are pieces of software that bind to devices they can handle and comunicate with them using the bus programing model.

The Linux bus model offers a generic way to organize devices and drivers, but it does not get in the way of drivers interacting with their devices. In fact, some details of this interaction are often bus specific. For instance, drivers often need more details on their devices than a generic framework can provide. PCI drivers need to access device configurations. USB drivers need to list device interfaces and endpoints. Moreover, traditionally, drivers communicate directly with their devices through I/O ports, I/O memory, and interrupts. This is what makes drivers so hard to write and utterly device specific. The PCI bus is a typical example of this philosophy. Other buses, such as USB or I2C, have adopted a different philosophy based on messages, yielding simpler and safer drivers.

4 Split Driver Bus Extension

We propose to extend the Linux device driver framework with the concept of split drivers based on messaging and the definition of device classes and interfaces. Our work was inspired by both the USB specification and the Linux HID framework, yielding a proposal for a Split-Driver Bus (SDB). Our extension is optional. Indeed, some extisting buses might not be able to support a message based communication. This section first describe the programming model of our extension, and its internals.

4.1 Programming Model

The programming model is described in three steps. First, we presents the data structures and functions. Then, we discuss the management of the buffers. Finally, we discuss an example, a mouse driver.

Data Structures and Functions

Devices are identified using an ID data structure described by the Listing 1. This ID contains various identification numbers such as the vendor and the product numbers of a device, its class and the protocol it uses, and finally the Bus Controller Driver that created it. This ID data structure will be used by the bus to bind devices with drivers.

To send and receive messages, devices have ports (inspired from USB endpoints). Ports will be used by drivers to connect channels used to exchange messages between devices and drivers. A port has a number identifying it, protocol and class IDs allowing to attach them a semantic, and a type. Port types are the messaging scheme a driver must use when sending and receiving messages through the port. We defined two port types:

- Symmetric port: Drivers and devices can send and receive messages at any time. This is typical of master-master relationship.
- Asymmetric port: Drivers can send messages but must request to receive messages from a device. It

means a device must wait for the driver to request data, before sending a message. This is typical of a master-slave relationship.

Ports are bundled into interfaces (inspired by USB interfaces). An interface is identified by its interface number, protocol and class IDs, and the list of ports they hold. New devices are registered using the *msg_register_device()*. Listing 2 shows the full data structures a device hold.

Listing 1: Device ID Structure

```
struct msg_device_id {
    /* Flags used to match ids */
    int    match_flags;
    /* Product specific matching */
    int    vendorId;
    int    productId;
    /* Class and protocol matching */
    int    classId;
    int    protocolId;
    /* Bus Controller Driver matching */
    int    bcdId;
};
```

Listing 2: Device Data Structure

```
/* Port types */
struct msg_port {
    int port_number;
    int transfert_type;
    int port_class;
    int port_protocol;
    struct msg_interface* intf;
};
struct msg_interface {
    int intf_number;
    unsigned int num_ports;
    struct msg_port* ports;
    int intf_class;
    int intf_protocol;
    struct msg_device* dev;
};
struct msg_device {
    char* name;
    struct device dev;
    struct msg_device_id id;
    unsigned int num_interfaces;
    struct msg_interface* interfaces;
    struct msg_low_level_driver* lld;
};
```

Listing 3 shows the data structure defining a driver. Drivers must define an ID table containing ID structures (identical as the ID structure contained in the device structure), it is used by the bus to match drivers to devices as follows. Each entry of its ID table is compared with the device's ID. If at least one of them matches, the driver will be bound with this device. When a driver is bound to a device, its *probe()* callback is invoked. From this call, a driver can analyze the interface of the device it has been bound with, and connect channels with device's ports. When a module is about to be unloaded, the driver is informed through its *disconnected()* callback. The *probe()* and *disconnect()* functions must never sleep as they are executed in an event handling context. For the explanation of the low-level driver concept, see Section 4.2.

Listing 3: Driver Data Structure

```
struct msg_driver {
    const char* name;
    struct device_driver driver;
    const struct msg_device_id* id_table;
    int (*probe)(struct msg_device*);
    void (*disconnect)(struct msg_device*);
};
```

Channels are communication pipes through which messages are traveling from and to drivers. A channel can be connected and disconnected to a device port. Channel structures, see Listing 4, belong to drivers. When a driver wants to connect a channel, it must allocate one (either dynamically or statically), and call the asynchronous *connect_channel()* function. A channel must be connected to at most one port and one port must not be connected to more than one channel. Channels ensure a First In First Out (FIFO) and lossless properties. When a driver wants to close a channel it calls the asynchronous function *close_channel()*.

Listing 4: Channel Data Structure

```
struct msg_buffer {
  struct msg_channel* channel;
  void* buffer;
  size_t length;
  size_t capacity;
  void (*release)(struct msg_buffer*);
};
struct msg_channel {
  int state;
  struct msg_port* port;
  struct msg_device* dev;
  void (*connected)(struct msg_channel*);
  void (*receive)(struct msg_buffer*);
  void (*closed)(struct msg_channel*);
};
```

All operations on channels are asynchronous operations—they never block, respecting our event-oriented programming model. Therefore, drivers need to set the following callbacks for each channel. The *connected()* callback, invoked when a channel is fully connected, allowing the driver to start sending and receiving messages through it. The *receive()* callback, invoked when a message is delivered from a channel. The *closed()* callback invoked when a channel is fully disconnected, allowing the driver to deallocate its channel data structure. All callbacks are called from an event handling context and must not sleep.

Drivers can perform two operations on a connected channel, given in Listing 5. The *channel_send()* operation is used to send a message to a device. It can be used with channels connected to either symmetric or

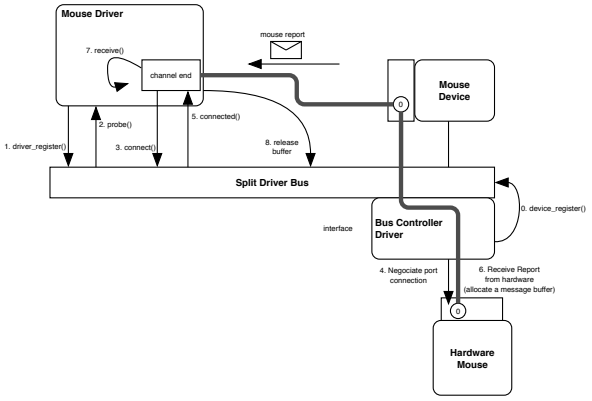


Figure 3: A Mouse Driver Example

asymmetrical ports. The *channel_ receive()* operation is used to request a message from a device. This operation can only be used with channels connected to asymmetric ports.

Listing 5: Channel Operations

```
int msg_connect_channel(struct msg_channel*);
int msg_close_channel(struct msg_channel*);
int msg_channel_send(struct msg_buffer*);
int msg_channel_receive(struct msg_buffer*);
```

Mouse Driver Example

A simple mouse driver illustrates the use of the above functions and data structures. It is an interesting example as it shows that our approach permits interrupt-based devices, something that the USB specification also supports, but via polling. To this end, the mouse device contains one interface holding a single symmetric port.

Figure 3 shows all the steps, from the perspective of the driver. First, the probe function is called, telling the driver that it has been matched to a device. The driver connects to that device, potentially exchanging initialization messages with that device. Whenever the mouse is moved or clicked by the end user, the device sends a report of the user activity as a message through the established channel with its driver. The reception of the message acts as an interrupt, telling the drive that the device needs attention. In this case, the message is self-contained, but a driver could send messages if necessary to inquiry or reconfigure the device at that point.

But it is also interesting to discuss the use of an asymmetrical port. Indeed, some mouse devices are designed as slave devices, which is typically the case for USB mouses. By a slave design, we mean that a mouse cannot send a message, like an interrupt, but it must be polled for a report. Remember that this is the standard USB behavior with *interrupt devices*; such devices are not using interrupts, they are polled regularly. This behavior is supported through asymmetrical ports in our programming model, via an explicit call of *msg_ channel_ receive()* when the device must be polled.

4.2 Internals

Implementing our proposal is mostly about implementing a Linux software bus. We will first recall briefly what it means to implement a Linux software bus for a hardware bus. Then, we will discuss how Linux buses can be used as aggregator for devices from across different buses.

Implementing a Linux Bus

All Linux bus internals are quite similar, leveraging a shared implementation provided as a core of functionalities by the Linux kernel. This shared implementation is essentially a working bus that can be tailored to one's needs. The idea is that an external agent will manage the bus, registering devices and drivers to the bus, the bus taking care of the logic of matching devices with drivers and then calling the drivers' probe functions for binding drivers to devices.

Our bus is a classical linux bus, but it belongs to the category of buses that act as brokers between drivers and devices. In our case, our bus provides message-oriented channels, very much like the USB bus. This means driver developers use the programming model we described earlier to use channels to send and receive messages. This in turn means that our implementation is responsible for the management of channels and the routing of messages.

Let's illustrate how this would be done over an I2C bus, a bus that natively supports messaging, but just sending or receiving uninterpreted payloads to or from a device, known by its address on the bus.

With an underlying I2C bus, our Bus Controller Driver would be a driver for the underlying I2C bus controller. Once the hardware controller of the bus is discovered and managed, it is business as usual for our bus, just sending and receiving messages across the I2C bus, like it would over any other networking technology like an Ethernet network for example. It is on the device side that the changes occur since devices need to understand our messaging format, using two possible designs. Some devices will have a small microcontroller and run a lightweight version of our bus software. Other devices will have a Serial Interface Engine à la USB, see Section 2.2, a small VHDL module along with a macrocell.

Implementing an Aggregator Bus

Many Linux software buses are just about managing one hardware bus, but other software buses are about aggregating functionally-related devices into one place, where relevant drivers could be registered. A perfect example of this is the HID software bus in Linux. It is a bus where all devices for human interactions must appear. For instance, all forms of pointing devices would appear there, such as mice, joysticks, or touch screens. From there, these different hardware devices could be aggregated in the concept of a single pointer that the window manager can use in moving the cursor on the screen.

We have the same challenge with our bus, we would like to aggregate the various devices that understand our split driver model, irrespective of the actual bus they are

plugged in. This suggests to introduce the concept of Low-Level Drivers (LLD) to our bus. The concept was invented by Linux developers for the HID bus for the same purpose. The purpose was to help reify an HID devices in the HID bus from other devices plugged in other buses. For instance, a mouse device on an I2C bus that would be reified as a HID mouse in the HID bus. The mean to do this is to add a driver to the I2C bus that will bind to the mouse. It is called a low-level driver because it is assumed that the HID bus is a parent of the I2C bus, in terms of the Linux bus hierarchy. That low-level driver will wrap the mouse device and present itself as an HID mouse device in the HID bus. The wrapping done by the low-level driver is bus-specific as it depends on what can be done with a device in the upper-level bus, in this case the HID bus.

In our case, the wrapping is about our message-based channels. Earlier, we have mentioned that each device has a corresponding low-level driver since the struct msg_device has a pointer to a struct msg_low_level_driver. The details of that struct are given Listing 6. The declared functions can be grouped in two groups: the downcall and upcall groups. The downcall group is about the functions that are implemented by the low-level driver and called by the bus implementation. The upcall group is about the functions that are implemented by the bus and called from the low-level driver.

Listing 6: Low-level Driver Definition

```
struct msg_ll_driver {
struct {
  int (*channel_connect)(msg_channel*);
  int (*channel_close)( msg_channel*);
  int (*channel_send)(msg_channel*,
                      msg_buffer*);
  int (*channel_receive)(msg_channel*,
                         msg_buffer*);
} downcalls;
struct {
  int (*channel_connected)(msg_channel*);
  int (*channel_sent)(msg_channel*,
                      msg_buffer*);
  void (*channel_received)(msg_channel*,
                           msg_buffer*);
  int (*channel_closed)(msg_channel*);
} upcalls;
};
```

These functions are the very functions that necessary to have an abstract implementation of our channels. Nothing fancy here, just a regular separation of concerns, allowing our bus to rely on low-level drivers for providing different implementations suited for different buses.

5 Validation

In order to validate our model, we implemented two proof-of-concept prototypes. The first was a Hardare/-Software implementation running on a Xilinx ZYBO board. The second was an integration of the USB bus.

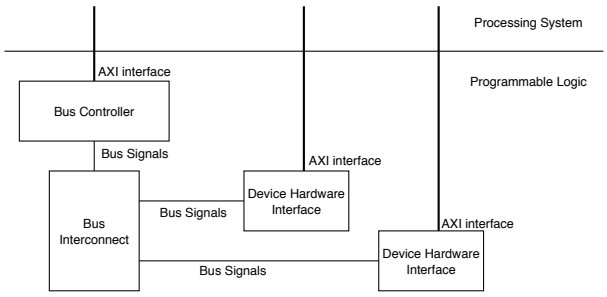


Figure 4: Hardware Bus Architecture

5.1 ZYBO implementation

In this experiment, we designed and implemented the whole split-driver model from hardware to software elements. The hardware part is a custom bus implemented into the Programmable Logic of the ZYBO ZYNQ System on Chip. It is split into the following pieces. The Bus Controller managing the whole bus, an interconnect, allowing to add multiple hardware devices on the bus, and hardware devices. Figure 4 show an overview of the hardware design. The bus controller contains two hardware FIFOs used to send and receive messages from the host perspective. When a message has to be sent to a device (called a descending transmission), the BCD put it into a memory buffer and ask the controller to start the transmission of that buffer. Using one of the two FIFOs, the controller execute the transmission of the data on the bus lines to the device, passing through the interconnect. The interconnect's primary role is to arbiter the bus, limiting the transmission of ascending messages from devices to host to only one device at a time. In our design, descending messages are broadcasted to all devices, meaning that they must check the destination of an incoming message before processing it. Future upgrades of the prototype would use a switch mechanism to avoid this if necessary (USB 2.0 also use broadcasting). Our devices are partially hardware implemented. Only the interface with the hardware bus is made out of pure hardware. The logic part is implemented in software that send and receive messages using the hardware part.

The software implementation consists of the following elements. The whole split-driver extension, introduced in the previous section, the Bus Controller Driver of our custom Bus Controller that reifies devices on the hardware bus into the split-driver bus, the logic part of our devices, and the front-end drivers.

To test this framework, we implemented a ramdisk device, which is a block device with its storage in memory. The disk device has only one port, used to receive read or write requests and send related responses. The hardware part of the device, receives requests, taken by the software logic part. These requests are then processed

by reading or writing into the memory storage (a memory buffer in the RAM) and sending back the response to the device's hardware interface that will in turns transmit them over the hardware bus. The disk buffer is allocated at device startup time, and used to store data. The buffer is divided into sectors of fixed size, like a real hardware disk. The BCD will reify that disk in the SDB, as a disk device with one interface and one port. The driver on the SDB will be a Linux block device driver, from the surrounding Linux perspective, that is, it exports a a regular block device interface. Consequently, end users can manipulate our disk as any regular storage device, using standard utilities such as *mkfs* and *mount* to create and mount a file system from a block device. This Linux block device interface is created by the driver when it probes the device and connects to device's port. Once connected, the driver can handle block requests from Linux file system by sending messages to the disk, either to read or write sectors.

Listing 7 shows the structure of the messages exchanged with the block device. A write request is processed by the disk by writing the incoming request payloads in the disk buffer, at the location stored in the request, then send a confirmation response. A read request is processed by reading into the disk buffer, and send a response with the requested payload after it. When a response is received, the driver uses the req_number to retrieve the related request. If the request was a read, the blocks to read, stored in the response payload, are written in the request buffer. The driver then calls the *blk_end_request_cur()* function, notifying the kernel that the request is completed.

Listing 7: Disk Request Structure

```
struct ramdisk_message {
    unsigned long req_number;
    unsigned long sector;
    unsigned long nsect;
    int operation; // READ or WRITE
    uint8_t payload[0];
};
```

Figure 5 gives the numbers of lines of code that were implemented. Thanks to extending the Linux bus framework, our implementation of the SDB is relatively small, at around 600 lines of code. The BCD for the hardware bus is naturally larger because it is directly interacting with the hardware, as the disk device logic. Note that a disk driver in a split-driver model is relatively small with only 400 lines of code. All the software here is implemented in the Linux kernel as kernel modules.

5.2 USB HID Integration

The next implementation integrates HID class USB devices in our model. It shows that USB devices can be supported, allowing to preserve the huge investments that went to the USB specifications, the USB host controllers, and of course in the USB devices. Wrapping USB devices is also interesting because it illustrates the use of class-generic low-level drivers. Indeed, we did not choose to wrap the USB host controller directly, taking control of the USB bus at the hardware level. Rather, we decided to exercise our low-level driver capability at the granularity of classes. This means that each USB device class is individually wrapped by one low-level driver.

Kernel Module	**Line of C**
Split-Driver Bus API	600
Bus Controller Driver	1300
Disk Driver module	400
Disk Device Logic	600
VHDL HW implementation	**Line of VHDL**
Bus Controller	4500
Bus Interconnect	120
Device Hardware Interface	4500

Figure 5: ZYBO Implementation Volumes

We chose to wrap a USB optical mouse, a device class that has an input interrupt endpoint, used by the mouse to send input reports containing data related to the state of mouse buttons and motion. Our low-level driver is therefore a USB driver, matching our optical mouse as an HID class. In fact, it is interesting to point out that the Linux bus for the USB reifies individual USB interfaces as devices and not the actual USB devices that could have multiple interfaces. For instance, this would mean that a device having both a HID interface and a mass storage interface, would be reified as two distinct devices by Linux. Rather than modifying the Linux USB core support, we designed our low-level driver to match USB interfaces rather USB devices. The approach shows the versatility of the low-level driver concept.

Figure 6 shows an overview of the implementation architecture. The low-level driver implements the operations defined on low-level drivers using USB Request Blocks (URBs). In fact, the operations of low-level drivers have a pretty straightforward mapping on URBs, thanks to the event-oriented nature of using URBs. The delicate part is the mapping from the USB HID mouse device to a device in our SDB. When the low-level driver, as a USB driver, is bound to a HID USB interface, it turns around and reifies a device in the SDB. To do that, the low-level driver fills the device ID structure, that happens to be very similar to the USB ID structure it obtains from the USB interface. This stretch of that mapping will depend on how much the specification of the original USB classes and interfaces will be preserved when adopted by the open source domain. In the case of the HID classes, the Linux adoption was pretty much a global adoption and therefore the translation is rather immediate.

The translation is of course concerned with the mapping of endpoints to our concept of device ports. In particular, we translate the well-known USB endpoint 0 into an official endpoint of the interface[2]. A HID mouse

[2]The USB specification as made a special case of the endpoint 0 that all devices must have and that does not belong

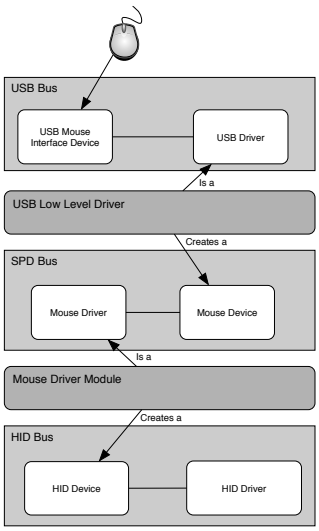


Figure 6: USB Mouse Integration Architecture

Module	Line of Code
USB Driver (SPD LLD)	300
Mouse Driver (HID LLD)	300

Figure 7: Implementation Volumes

defines four endpoints that are mapped to four distinct asymmetric ports. Using asymmetric ports means that the drive in our SDB will use the the *channel_ receive()* function to receive reports from the mouse. The driver can also use *channel_ send()* function to control the mouse configuration. This choices of asymmetric ports matches the slave design of a USB HID mouse.

Figure 7 shows the volume of the implemented programs and modules.

6 Conclusion

The work done so far is a proof of concept. Our main goal is to evangelize the idea that split drivers are a path to reduce the device driver struggle that has been plaguing the industry since it began. We demonstrated that the concept of split drivers can be successfully integrated within Linux bus framework for matching drivers to devices, independently from any specific bus or network technologies. We believe this first step opens up a world of interesting opportunities.

The best way to look at it is to consider it as open sourcing the USB concept of split drivers, based on the definition of message-based interfaces. Linux has done it already with the HID framework, starting from the HID specification defined by the USB standard and making it a Linux framework. The same could be done with all device classes defined by the USB standard. We strongly feel that it would be in the best interest of the industry that the USB standard body would spin off the device class standardization through a bus-independent standard body, following in the footstep of the SCSI standard.

As a consequence, the industry could develop class-generic drivers. These drivers are usually really simple, a few hundreds of C lines, with most of the complexity being the integration in the surrounding operating systems. This means that each operating system community could quickly develop such class-generic drivers. For the Linux community, this would probably start with porting the existing USB drivers to our framework. From there, it would be easy for the industry to leverage existing bus technologies to connect devices, using bus and network technologies as conduits for messages of uninterpreted payloads.

This represents a fundamental change of the interface between the software and hardware. It is about considering device interfaces as message-based interfaces rather than register-based interfaces. Of course, this would not apply to all devices, some devices that are the system bus would certainly not be impacted, like the memory controller, the programmable interrupt controller, or timers. But most devices are attached today on secondary buses, whom could become serial buses, suited for supporting a message-based interface. This hardware change could be mirrored in software with our split-driver approach, finally offering a path out of struggle for the availability of safe drivers.

Such an evolution would have great potential. It would help device manufacturers with reduce driver development costs, shorter time to market, and wider availability across operating systems and across hardware. It would help operating system teams to reduce the driver code base and the never-ending stream of related bugs. Open-source operating systems would especially benefit from this approach, since proprietary drivers have been a long standing issue. Here, the back-end drivers would remain proprietary while the front-end drivers would be open-source implementation and that easily portable.

The split-driver model is just safer and simpler than preserving the software-hardware interface across serial buses, like Thunderbolt is proposing. While we understand the desire to preserve existing drivers, preserving huge investments, we feel it is the wrong direction to take. Through serial buses, our hardware platforms are changing, they are evolving towards distributed systems. We all know that distributed systems are best approached via a message-based programming model. We believe that it is time to face forward and embrace this evolution towards distributed systems.

Fortunately, this does not have to be a brutal change, the evolution can be incremental, evolving bus technologies, devices, and drivers incrementally in the right direction. But research has to show the path, has to experiment both on the software and hardware evolutions, demonstrating their feasibility, and evangelizing the benefits. This work is a first step in that direction, a promising one.

to any interface.

References

[1] ATCA. Advanced Telecommunications Computing Architecture. https://en.wikipedia.org/wiki/Advanced_Telecommunications_Computing_Architecture.

[2] DMA Attack. https://en.wikipedia.org/wiki/DMA_attack.

[3] DDC. Display Data Channel. https://en.wikipedia.org/wiki/Display_Data_Channel.

[4] eCos. Free open source real-time operating system. http://ecos.sourceware.org.

[5] USB HID. USB Human Interface Device. https://en.wikipedia.org/wiki/USB_human_interface_device_class.

[6] Xen Hypervisor. http://www.xenproject.org.

[7] I2C. Inter-Integrated Circuit Specification. http://i2c.info/i2c-bus-specification.

[8] IPMI. Intelligent Platform Management Interface. https://en.wikipedia.org/wiki/Intelligent_Platform_Management_Interface.

[9] Asim Kadav and Michael M. Swift. Understanding modern device drivers. *SIGARCH Comput. Archit. News*, 40(1):87–98, March 2012.

[10] Linux Kernel. https://www.kernel.org.

[11] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):59–71, April 2006.

[12] PCI. Peripheral Component Interconnect Specification. https://pcisig.com/specifications.

[13] PCIe. Peripheral Component Interconnect Express. https://en.wikipedia.org/wiki/PCI_Express.

[14] PMBus. Power Management Bus. http://www.pmbus.org/Home.

[15] Display Port. https://en.wikipedia.org/wiki/DisplayPort.

[16] SCSI. Small Computer System Interface. https://en.wikipedia.org/wiki/SCSI.

[17] SMbus. System Management Bus. https://en.wikipedia.org/wiki/System_Management_Bus.

[18] Thunderbolt technology. https://developer.apple.com/library/mac/documentation/HardwareDrivers/Conceptual/ThunderboltDevGuide/Introduction/Introduction.html.

[19] uClinux. Embedded Linux/Microcontroller project. http://www.uclinux.org.

[20] udev Device Manager. https://en.wikipedia.org/wiki/Udev.

[21] USB. Universal Serial Bus 3.1 Specification. http://www.usb.org/developers/docs/.

[22] WindRiver. Free embedded operating system. http://rocket.windriver.com.

Relational Summaries for Interprocedural Analysis and Modular Verification of Synchronous Reactive Systems

Rémy BOUTONNET
VERIMAG - UGA
remy.boutonnet@imag.fr
Supervised by: Nicolas Halbwachs.

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

Abstract

Abstract interpretation is a theory of the safe approximation of the behavior of dynamic discrete systems. It is applied in static analysis to the automatic discovery of invariant properties on program variables. Linear Relation Analysis, based on abstract interpretation, discovers at each point of a sequential program a system of linear relations which are satisfied by the numerical variables in every execution. Despite being one of the most powerful relational analyses, it is rarely used in industrial static analysis tools due to its complexity. We propose a new approach, called *relational procedure summaries*, to enable the modular analysis of programs using Linear Relation Analysis.

1 Introduction

We are living in a world in which a growing number of critical systems are based on software components. Any software defect can lead to catastrophic events, massive injuries or deaths. The criticality of these systems is only matched by their pervasiveness and gives to software verification and static analysis a major importance.

Abstract interpretation [Cousot and Cousot, 1977] is a theory of the safe approximation of the behavior of dynamic discrete systems. It is especially applied in static analysis to the automatic discovery of invariant properties on program variables. The discovery of invariant properties can be used for proving safety properties, like the detection of arithmetic overflows and array bounds checking, for the optimization of control and memory requirements in compilation, and for the evaluation of the worst-case execution time (WCET) of programs by the discovery of loop bounds and unfeasible paths.

Linear Relation Analysis [Cousot and Halbwachs, 1978], based on abstract interpretation, discovers at each point of a sequential program a system of linear relations which are satisfied by the numerical variables in every execution. Although it is now classical, Linear Relation Analysis remains one of the most powerful analysis techniques for numerical variables. Furthermore, it is in particular one of the few existing *relational* analyses, which are able to discover relations among numerical variables.

However, it is rarely used in industrial static analysis tools, like Polyspace [MathWorks, 2016] or Astrée [Blanchet *et al.*, 2003], due to its cost. The algorithms that it depends on have an exponential worst-case complexity in the number of numerical variables. Moreover, industrial programs can have typically thousands of procedures, each one being analyzed by static analysis tools for each call context in very large variable environments.

Critical systems software, like in avionics and transportation systems, are often compiled from high-level declarative languages such as synchronous languages [Halbwachs, 1998] and their modular verification is hindered by this scalability problem.

The aim of this work is to address this scalability challenge, by exploiting the structure of programs to reduce analysis complexity.

Interprocedural analysis have been the subject of numerous studies since the early works [Allen, 1974] of Frances E. Allen in 1974. Several approaches have been proposed for the analysis of programs with procedures, we can classify them into four broad families.

Procedure inlining Procedure inlining replaces call sites by the body of the called procedure which is analyzed in the context of the caller. This results in very large contexts, containing both variables of the caller and variables of the called procedure. Inlining can potentially cause an exponential blow-up of the program size and it is not usable for the analysis of recursive procedures. It does not comply with our goal of taking advantage of programs structure in order to reduce analysis complexity.

Top-down analysis Top-down analyses reanalyze each procedure for every call site in the context of the caller. This is acceptable for procedures which are called once or very few times, but it is detrimental to analysis scalability for library procedures which are called a large number of times. Top-down analyses are also hindered

by their bad reuse of analysis results, if any. The functional approach described in [Sharir and Pnueli, 1978] is representative of this family.

Bottom-up analysis Bottom-up analyses analyze each procedure only once and produce a procedure summary independently of a particular call context. This summary is then adapted to the analysis of a given call site. The analyses described in [Ancourt *et al.*, 2010] and [Yorsh *et al.*, 2008] are representative of this family

Hybrid analysis Hybrid analyses are top-down analyses which are able to reuse analysis results of other procedure calls in similar contexts. This family is represented by [Zhang *et al.*, 2014].

We have designed a new bottom-up interprocedural analysis to automatically construct *relational procedure summaries* using Linear Relation Analysis. These summaries are termed *relational* because they are able to express linear relations between procedure parameters, between the values of the actual parameters before and after a procedure call. This paper describes the exploration, formalization and experimentation of our approach.

2 Contributions

We have made the following contributions

- We present a new program analysis technique for building *relational procedure summaries* using Linear Relation Analysis. We formally describe the construction of procedure summaries and their application to specific call contexts. Our presentation is grounded on the structural operational semantics of a simple imperative language, featuring all the programming constructs which are relevant to our approach.
- We describe an application of our approach to summaries of synchronous reactive programs.
- We experiment our approach on meaningful example programs to illustrate its key aspects all along this report.

The aliasing problem is a well-studied problem [Steensgaard, 1996] and we consider all the necessary analyses to be available [Shapiro and Horwitz, 1997], [Emami *et al.*, 1994].

Due to the time investment needed to design and implement a good interprocedural static analyzer for a real world language, we have left the proper implementation of our approach to a future work. All the examples given in this work have been computed using the facilities of existing static analyzers, such as PAGAI [Henry *et al.*, 2012] and Interproc [Jeannet, 2010].

3 Relational Procedure Summaries

We are interested in the automatic construction of relational procedure summaries. We want these summaries to be able to capture linear relations between the state of the parameters of a procedure before a call and their state after the call. Such a summary is intended at capturing an abstraction of the behavior of a procedure in an over-approximate fashion.

3.1 Introductory example

We will illustrate our approach on the **div** procedure which computes the euclidean division of a and b by the method of successive subtractions, as first presented by Euclid around 300 BC in [Euclid, c 300 BC]. The quotient is stored in q and the remainder in r.

```
-- require a >= 0 and b >= 1
procedure div(a, b, q, r)
begin
    r := a;
    q := 0;

    while r >= b
        r := r - b;
        q := q + 1;
    end;
end;
```

We give preconditions to procedures denoted by special **require** comments. The **div** procedure implements euclidean division only for positive integers with a non-zero divisor. Its precondition $\mathcal{A} \subseteq \mathcal{N}^4$ is such that

$$\mathcal{A} = \{(a,b,q,r) \in \mathcal{N}^4 \mid a \geq 0 \ \wedge \ b \geq 1\}$$

We are only interested in the effect of the **div** procedure on its parameters, its behavior regarding our considerations is

$$\begin{cases} q = 0 \ \wedge \ r = a & \text{if } a < b \\ q = 1 \ \wedge \ r = 0 & \text{if } a = b \\ q \geq 1 \ \wedge \ 0 \leq r \leq b-1 & \text{if } a > b \end{cases}$$

We accept to loose the property $a = bq + r$ which is non-linear, and which can't be represented in the convex polyhedra abstract domain.

We first analyze the **div** procedure with standard Linear Relation Analysis and only the information given by its precondition $\mathcal{A}$. The analysis is started on the initial abstract state $a \geq 0 \ \wedge \ b \geq 1$. The following invariant is obtained at the end of the **div** procedure.

$$0 \leq r \leq b-1 \ \wedge \ q \geq 0 \ \wedge \ a \geq 0 \ \wedge \ b \geq 1$$

This invariant is too coarse, it does not capture the behavior that we have previously described. It is characterized by a disjunctive property and classic Linear Relation Analysis is not able to discover such a property.

Procedures having a behavior characterized by a disjunctive property are very common. Thus our procedure summaries must be able to express disjunctive properties on procedures parameters.

Regarding the `div` procedure, we analyze separately the cases where $a < b$, $a = b$ and $a > b$. For $a < b$ with initial abstract state

$$a < b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1$$

We obtain the invariant

$$\mathbf{q} = \mathbf{0} \ \wedge \ \mathbf{r} = \mathbf{a} \ \wedge \ a < b \ \wedge \ a \geq 0 \ \wedge b \geq 1$$

For $a = b$ with initial abstract state

$$a = b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1$$

We obtain the invariant

$$\mathbf{q} = \mathbf{1} \ \wedge \ \mathbf{r} = \mathbf{0} \ \wedge \ a = b \ \wedge \ a \geq 0 \ \wedge b \geq 1$$

For $a > b$ with the initial abstract state

$$a > b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1$$

We obtain the invariant

$$\mathbf{q} \geq \mathbf{1} \ \wedge \ \mathbf{r} \geq \mathbf{0} \ \wedge \ \mathbf{r} \leq \mathbf{b} - \mathbf{1} \ \wedge \ a > b \ \wedge \ a \geq 0 \ \wedge b \geq 1$$

We can observe that by distinguishing different cases relative to the possible values of the parameters and by analyzing them separately, we obtain for each of them the property of the procedure behavior that we previously identified.

The concrete values of the parameters corresponding to the cases we have considered are a partition δ of the precondition $\mathcal{A} \subseteq \mathcal{N}^4$.

$$\delta = \begin{array}{l} \{(a,b,q,r) \in \mathcal{N}^4 \mid a < b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1\}, \\ \{(a,b,q,r) \in \mathcal{N}^4 \mid a = b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1\}, \\ \{(a,b,q,r) \in \mathcal{N}^4 \mid a > b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1\}\} \end{array}$$

It ensures that we have analyzed the procedure `div` for all its valid parameters. Thus, we can analyze a procedure more precisely, by considering a partition of its precondition and by analyzing the procedure separately for each member of this partition.

We denote by $(D^\sharp, \sqsubseteq)$ the convex polyhedra abstract domain and (α, γ) the Galois connection between the concrete domain $\mathcal{P}(\mathcal{N}^n)$ and the abstract domain $D^\sharp$.

We can design relational procedure summaries for a given partition δ of the precondition as functions $S : D^\sharp \to D^\sharp$, associating to an abstraction $d \in D^\sharp$ of a member of the partition, the invariant at the end of the procedure obtained as a result of the analysis with initial abstract state d.

An example of such a summary is $S_{div} : D^\sharp \to D^\sharp$ the summary of `div` which is

$$S_{div} = \left\{ \begin{array}{lcl} a < b \wedge a \geq 0 \wedge b \geq 1 & \mapsto & q = 0 \wedge r = a \\ a = b \wedge a \geq 0 \wedge b \geq 1 & \mapsto & q = 1 \wedge r = 0 \\ a > b \wedge a \geq 0 \wedge b \geq 1 & \mapsto & q \geq 1 \wedge 0 \leq r \leq b - 1 \end{array} \right.$$

Each member $x \in \delta$ of the partition δ is abstracted by a convex polyhedron $d \in D^\sharp$ such that $\alpha(x) = d$ and $S_{div}(d)$ is the invariant at the end of `div` which is obtained by analyzing `div` with the initial abstract state d using a standard Linear Relation Analysis.

Summarizing a procedure is a tradeoff between the precision of the summary, its size and the cost of analysis. We are not interested in an exact characterization of procedures but only in a safe approximation of their behavior.

3.2 Definition of procedure summaries

Before showing how the partition of the precondition can be chosen, we give first a definition of relational procedure summaries.

Definition 3.1 (Procedure summary) *Let $form_p \in Id^*$ be the list of the formal parameters of p, $\mathcal{N}^m$ the concrete domain restricted to the parameters of p, where $m =\mid form_p \mid$ is the length of $form_p$. The summary S_p of a procedure p with precondition $\mathcal{A} \subseteq N^m$ is a partial function $S_p : D^\sharp \to D^\sharp$ with a finite domain $Dom(S_p)$ such that $\delta = \{\gamma(x) \mid x \in Dom(S_p)\}$ is a partition of $\mathcal{A}$.*

$$\bigcup_{x \in Dom(S_p)} \gamma(x) = \mathcal{A}$$

$$\forall x_1, x_2 \in Dom(S_p), \ \gamma(x_1) \cap \gamma(x_2) = \emptyset$$

Definition 3.2 (Domain of a summary) *We denote by $Dom(S_p)$ the domain of the summary S_p of a procedure p such that*

$$Dom(S_p) = \{x \in D^\sharp \mid \exists y \in D^\sharp, S_p(x) = y\}$$

Each polyhedron P_i in the domain $Dom(S_p)$ can be seen as an hypothesis on the parameters of the procedure p.

Example 3.1 *We choose the domain φ_{div} for the summary S_{div} of `div` to be such that*

$$\varphi_{div} = \begin{array}{l} \{(a < b \ \wedge \ a \geq 0 \ \wedge b \geq 1), \\ (a = b \ \wedge \ a \geq 0 \ \wedge b \geq 1), \\ (a > b \ \wedge \ a \geq 0 \ \wedge b \geq 1)\} \end{array}$$

The domain of the summary S_{div} covers the precondition $\mathcal{A}$ of `div`.

3.3 Construction of procedure summaries

We describe in this section how to construct relational procedure summaries using Linear Relation Analysis. Several functions on identifiers that will be used for summary construction are defined.

We need a way to denote the initial values of procedure parameters before a call. The initial value of a parameter $x \in Id$ will be represented by x_0, the zero-indexed version of this parameter.

We denote by Id_0 an infinite set of identifiers disjoint from Id, in which every identifier is indexed by zero such as $x_0 \in Id_0$. The set of the finite lists of identifiers in Id_0 is denoted by Id_0^*.

The function $eq : Id^* \times Id_0^* \to D^\sharp$ defined below gives a polyhedron of the form $x_1 = y_0^1 \wedge ... \wedge \ x_n = y_0^n$ for two lists of identifiers $\langle x_1, ..., x_n \rangle \in Id^*$ and $\langle y_0^1, ..., y_0^n \rangle \in Id_0^*$.

Definition 3.3 (Function eq) *The function $eq : Id^* \times Id_0^* \to D^\sharp$ gives a polyhedron representing pairwise equalities between two lists of identifiers in Id^* and in Id_0^* such that*

$$eq(\epsilon, \epsilon) = \top$$

$$\forall x \in Id, \ \forall x_0 \in Id_0, \ \forall l_1 Id^*, \ \forall l_2 \in Id_0^*, \\ eq(x.l_1, x_0.l_2) = \{x = x_0\} \sqcap eq(l_1, l_2)$$

The function $z : Id \to Id_0$ is the one-to-one function associating to an identifier $x \in Id$ its zero-indexed version $x_0 \in Id_0$. The function $Z : Id^* \to Id_0^*$ associates to a list of formal parameters the list of the identifiers in Id_0 representing their initial values.

Definition 3.4 (Function Z) *The function $Z : Id^* \to Id_0^*$ associates to a finite list of identifiers in Id a list of identifiers in Id_0 having the same length.*

$$Z(\epsilon) = \epsilon$$

$$\forall x \in Id,\ \forall l \in Id^*,\ Z(x.l) = z(x).Z(l) \text{ where } z(x) = x_0$$

Definition 3.5 (Substitution on list of identifiers) *We define the function $l_1/l_2 : Id \to Id$ for two lists of identifiers $l_1 \in Id^*$ and $l_2 \in Id^*$ having the same length such that*

$$\forall v_1, v_2 \in Id, \forall x \in Id,$$

$$(v_1.\epsilon/v_2.\epsilon)(x) = \begin{cases} v_2 & \text{if } x = v_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\forall v_1, v_2 \in Id, \forall l_1, l_2 \in Id^*, \forall x \in Id,$$

$$(v_1.l_1/v_2.l_2)(x) = \begin{cases} v_2 & \text{if } x = v_1 \\ (l_1/l_2)(x) & \text{otherwise} \end{cases}$$

If $P \in D^\sharp$ is a convex polyhedron, $P[l_1/l_2]$ is the application of the substitution l_1/l_2 in the system of inequalities representing P.

Example 3.2 *Let $P = (x \leq n \ \wedge \ y = 100)$ be a convex polyhedron. The application $P[x \mapsto a, y \mapsto b]$ to P of the substitution $[x \mapsto a, y \mapsto b]$ is such that*

$$P[x \mapsto a, y \mapsto b] = (a \leq n \ \wedge \ b = 100)$$

We denote by $\Omega : Stmt \times D^\sharp \to D^\sharp$ the forward analysis function such that $\Omega(S, D)$ is the forward analysis of a statement S starting in an initial abstract state $d \in D^\sharp$. The function $F_S^\sharp : D^\sharp \to D^\sharp$ is the abstract semantics associated to statement S. We compute an ascending sequence (x_n) defined as

$$x_0 = d$$

$$x_{n+1} = x_n \nabla (x_n \sqcup F_S^\sharp(x_n))$$

and a descending sequence (y_n) starting on the limit $x^\sharp$ of the ascending sequence

$$y_0 = x^\sharp$$

$$y_{n+1} = F_S^\sharp(y_n)$$

Let $y^\sharp$ be the limit of the descending sequence. We define $\Omega(S, d)$ to be such that $\Omega(S, d) = y^\sharp$. For a statement $S \in Stmt$ and $d \in D^\sharp$, $\Omega(S, d)$ is the abstract state reachable after S when starting the analysis with abstract state d.

Definition 3.6 (Summary construction) *Let $\varphi = \{P_1, ..., P_n\}$ be a finite set of convex polyhedra on the formal parameters of a procedure p such that $\delta = \{\gamma(P_i) \mid P_i \in \varphi\}$ is a partition of $\mathcal{A}$. Let $B_p \in Stmt$ be the body of procedure p, $lv_p \in \mathcal{P}(Id)$ the set of the local variables of p and $\mathcal{A} \subseteq \mathcal{N}^m$ the precondition of p.*

The summary S_p of procedure p over the domain φ is

$$\forall P_i \in \varphi,\ S_p(P_i) = \ \Omega(B_p, P_i[form_p/Z(form_p)] \\ \sqcap\ eq(form_p, Z(form_p))) \downarrow lv_p$$

$Z(\text{form}_p)$ represent the initial values of the formal parameters of p.

$P_i[\text{form}_p/Z(\text{form}_p)]$ denotes the substitution in the convex polyhedron P of the formal parameters of p by their zero-indexed version.

Example 3.3 *If the formal parameters of p are $form_p = \langle x, y \rangle$ and $P_i = x \geq 0 \wedge y \geq x$, then $P_i[form_p/Z(form_p)] = x_0 \geq 0 \wedge y_0 \geq x_0$.*

$eq(\text{form}_p, Z(\text{form}_p))$ denotes the polyhedron defined by equality constraints between the formal parameters of the procedure p and the identifiers representing their initial values.

Example 3.4 *If the formal parameters of p are $form_p = \langle x, y \rangle$, then*

$$eq(form_p, Z(form_p)) = (x = x_0 \wedge y = y_0)$$

Example 3.5 *If the formal parameters of p are $form_p = \langle x, y \rangle$ and $P_i = x \geq 0 \wedge y \geq x$, then*

$$P_i[form_p/Z(form_p)] \sqcap eq(form_p, Z(form_p)) =$$

$$(x_0 \geq 0 \ \wedge \ y_0 \geq x_0 \ \wedge \ x = x_0 \ \wedge \ y = y_0)$$

$P_i[\text{form}_p/Z(\text{form}_p)] \sqcap eq(\text{form}_p, Z(\text{form}_p))$ is the transformation of the polyhedron P_i on the formal parameters of p into a polyhedron on the initial values of the formal parameters of p. The equality constraints between the identifiers in form_p and those in $Z(\text{form}_p)$ are added to the previous result by computing the intersection.

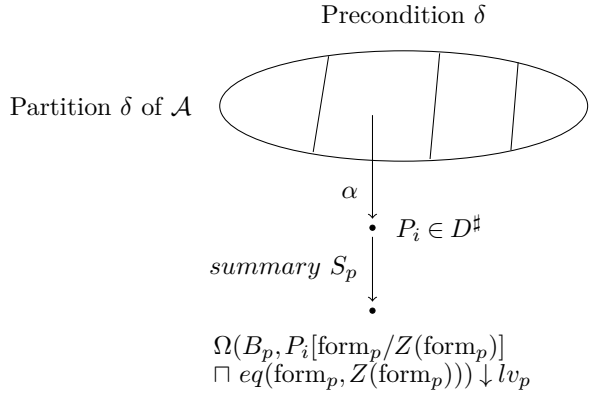


Figure 1: The construction of the summary S_p of a procedure p with a precondition $\mathcal{A}$.

As an illustration, we will construct the summary of the `div` procedure.

Example 3.6 (Summary of the div procedure) *We use the domain φ_{div} to construct the summary S_{div} of* `div`.

$$\varphi_{div} = \begin{array}{l}\{(a < b \ \wedge \ a \geq 0 \ \wedge b \geq 1),\\ (a = b \ \wedge \ a \geq 0 \ \wedge b \geq 1),\\ (a > b \ \wedge \ a \geq 0 \ \wedge b \geq 1)\}\end{array}$$

The formal parameters of `div` *are $form_{div} = \langle a, b, q, r\rangle$ and the identifiers denoting their initial values are $Z(form_{div}) = \langle a_0, b_0, q_0, r_0\rangle$. The* `div` *procedure has no local variables. We compute S_{div} for each element of the domain φ_{div}.*

$$\mathbf{P_1} = (\mathbf{a < b} \ \wedge \ \mathbf{a \geq 0} \ \wedge \mathbf{b \geq 1})$$

For $P_1 = (\mathbf{a < b} \ \wedge \ \mathbf{a \geq 0} \ \wedge \mathbf{b \geq 1})$, we compute $S_{div}(P_1)$ such that

$$\begin{array}{rcl} S_{div}(P_1) & = & \Omega(B_{div}, P_1[form_{div}/Z(form_{div})] \\ & & \sqcap \ eq(form_{div}, Z(form_{div}))) \downarrow lv_{div} \end{array}$$

$$\begin{array}{l} P_1[form_{div}/Z(form_{div})] = \\ (a < b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1)[form_{div}/Z(form_{div})] = \\ (a < b \ \wedge \ a \geq 0 \ \wedge \ b \geq 1)[a \mapsto a_0, b \mapsto b_0] = \\ (a_0 < b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1) \end{array}$$

We analyze B_{div} using Linear Relation Analysis with the initial abstract state I_1

$$\begin{array}{l} I_1 = (a_0 < b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1 \\ \quad \wedge \ a = a_0 \ \wedge \ b = b_0 \ \wedge \ q = q_0 \ \wedge \ r = r_0) \end{array}$$

$$\begin{array}{rcl} S_{div}(P_1) & = & \Omega(B_{div}, I_1) \\ & = & \mathbf{q = 0} \ \wedge \ \mathbf{r = a} \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ & & \wedge \ a_0 < b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1 \end{array}$$

$$\mathbf{P_2} = (\mathbf{a = b} \ \wedge \ \mathbf{a \geq 0} \ \wedge \mathbf{b \geq 1})$$

For $P_2 = (\mathbf{a = b} \ \wedge \ \mathbf{a \geq 0} \ \wedge \ \mathbf{b \geq 1})$, we compute $S_{div}(P_2)$

$$\begin{array}{rcl} S_{div}(P_2) & = & \Omega(B_{div}, P_2[form_{div}/Z(form_{div})] \\ & & \sqcap \ eq(form_{div}, Z(form_{div}))) \downarrow lv_{div} \end{array}$$

$$\begin{array}{l} P_2[form_{div}/Z(form_{div})] = \\ (a = b \ \wedge \ a \geq 0 \ \wedge b \geq 1)[form_{div}/Z(form_{div})] = \\ (a = b \ \wedge \ a \geq 0 \ \wedge b \geq 1)[a \mapsto a_0, b \mapsto b_0] = \\ (a_0 = b_0 \ \wedge \ a_0 \geq 0 \ \wedge b_0 \geq 1) \end{array}$$

We analyze B_{div} using Linear Relation Analysis with the initial abstract state I_2

$$\begin{array}{l} I_2 = (a_0 = b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1 \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ \quad \wedge \ q = q_0 \ \wedge \ r = r_0) \end{array}$$

$$\begin{array}{rcl} S_{div}(P_2) & = & \Omega(B_{div}, I_2) \\ & = & (\mathbf{q = a} \ \wedge \ \mathbf{r = 0} \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ & & \wedge \ a_0 = b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1) \end{array}$$

$$\mathbf{P_3} = (\mathbf{a > b} \ \wedge \ \mathbf{a \geq 0} \ \wedge \mathbf{b \ \geq 1})$$

For $P_3 = (\mathbf{a > b} \ \wedge \ \mathbf{a \geq 0} \ \wedge \mathbf{b \geq 1})$, we compute $S_{div}(P_3)$ such that

$$\begin{array}{rcl} S_{div}(P_3) & = & \Omega(B_{div}, P_3[form_{div}/Z(form_{div})] \\ & & \sqcap \ eq(form_{div}, Z(form_{div}))) \downarrow lv_{div} \end{array}$$

$$\begin{array}{l} P_3[form_{div}/Z(form_{div})] = \\ (a > b \ \wedge \ a \geq 0 \ \wedge b \geq 1)[form_{div}/Z(form_{div})] = \\ (a > b \ \wedge \ a \geq 0 \ \wedge b \geq 1)[a \mapsto a_0, b \mapsto b_0] = \\ (a_0 > b_0 \ \wedge \ a_0 \geq 0 \ \wedge b_0 \geq 1) \end{array}$$

We analyze B_{div} using Linear Relation Analysis with the initial abstract state I_3

$$\begin{array}{l} I_3 = (a_0 > b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1 \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ \quad \wedge \ q = q_0 \ \wedge \ r = r_0) \end{array}$$

$$\begin{array}{rcl} S_{div}(P_3) & = & \Omega(B_{div}, I_3) \\ & = & (\mathbf{q \geq 1} \ \wedge \ \mathbf{0 \leq r \leq b - 1} \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ & & \wedge \ a_0 > b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1) \end{array}$$

Finally, we obtain the following summary for `div`

$$\begin{array}{l} P_1 = (\mathbf{a < b} \ \wedge \ a \geq 0 \ \wedge b \ \geq 1) \\ P_2 = (\mathbf{a = b} \ \wedge \ a \geq 0 \ \wedge \ b \geq 1) \\ P_3 = (\mathbf{a > b} \ \wedge \ a \geq 0 \ \wedge \ b \geq 1) \\ S_{div}(P_1) = (\mathbf{q = 0} \ \wedge \ \mathbf{r = a} \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ \quad \wedge \ a_0 < b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1) \\ S_{div}(P_2) = (\mathbf{q = a} \ \wedge \ \mathbf{r = 0} \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ \quad \wedge \ a_0 = b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1) \\ S_{div}(P_3) = (\mathbf{q \geq 1} \ \wedge \ \mathbf{0 \leq r \leq b - 1} \ \wedge \ a = a_0 \ \wedge \ b = b_0 \\ \quad \wedge \ a_0 > b_0 \ \wedge \ a_0 \geq 0 \ \wedge \ b_0 \geq 1) \end{array}$$

It is nearly the same summary as the one we have given earlier, the initial values of the parameters have just been denoted by a zero-indexed identifier.

Example 3.7 (Counters) *The* `counter_update` *procedure comes from the field of reactive programming. It increments a counter represented by the variable cnt if an event occurs signaled by event = 1 and if cnt is lower than its maximum value max. The counter is set to zero when reset = 1 and keeps its previous value otherwise.*

```
-- require reset >= 0 and reset <= 1
-- require event >= 0 and event <= 1
-- require cnt >= 0 and max >= 0
procedure counter_update (reset, event, cnt, max)
begin
    if reset = 1 then
        cnt := 0;
    else
        if event = 1 then
            if cnt < max then
                cnt := cnt + 1;
            end;
        end;
    end;
end;
```

For the sake of simplicity, all the parameters of `counter_update` *are integers. The precondition $\mathcal{A}$ of the procedure is*

$$\begin{array}{l} \mathcal{A} = \{(reset, event, cnt, max) \in \mathcal{N}^4 \mid \\ 0 \leq reset \leq 1 \ \wedge \ 0 \leq event \leq 1 \ \wedge \ cnt \geq 0 \ \wedge \ max \geq 0\} \end{array}$$

We can construct different summaries of **counter_update** *by choosing different domains, bringing various levels of precision. The convex polyhedron abstracting the precondition is*

$$P = (0 \leq reset \leq 1 \wedge 0 \leq event \leq 1 \wedge cnt \geq 0 \wedge max \geq 0)$$

With a coarse domain $\varphi_1 = \{P\}$ *containing only* P*, we obtain the summary* S_1 *such that*

$$\begin{aligned} S_1(P) = (&\mathbf{0 \leq cnt \leq max + cnt_0} \ \wedge \ \mathbf{cnt \leq event + cnt_0} \\ &\wedge\ 0 \leq reset_0 \leq 1 \ \wedge\ 0 \leq event_0 \leq 1 \\ &\wedge\ max_0 \geq 0 \ \wedge\ reset = reset_0 \\ &\wedge\ event = event_0 \ \wedge\ max = max_0 \ \wedge\ cnt_0 \geq 0) \end{aligned}$$

We have discovered that cnt is bounded, as described by the constraint $0 \leq cnt \leq cnt_0 + max$*. However, when* $reset = 1$*, we do not have* $cnt = 0$*.*

We now use the domain φ_2 *which splits the parameters space by distinguishing two cases,* $reset = 0$ *and* $reset = 1$*, corresponding to the condition* $reset = 1$ *and its negation, taking into account the precondition in which we have* $0 \leq reset \leq 1$*.*

$$\varphi_2 = \{(P \wedge reset = 0), (P \wedge reset = 1)\}$$

$$\begin{aligned} S_2(P \wedge reset = 1) = (&\mathbf{cnt = 0} \ \wedge\ \mathbf{reset_0 = 1} \\ &\wedge\ 0 \leq event_0 \leq 1 \ \wedge\ max_0 \geq 0 \\ &\wedge\ reset = reset_0 \ \wedge\ event = event_0 \\ &\wedge\ max = max_0 \ \wedge\ cnt_0 \geq 0) \end{aligned}$$

$$\begin{aligned} S_2(P \wedge reset = 0) = (&\mathbf{cnt < cnt_0 + max} \\ &\wedge\ \mathbf{cnt < cnt_0 + event} \ \wedge\ \mathbf{cnt \geq cnt_0} \\ &\wedge\ 0 \leq event_0 \leq 1 \ \wedge\ max_0 \geq 0 \ \wedge\ cnt_0 \geq 0 \\ &\wedge\ event = event_0 \ \wedge\ max = max_0 \\ &\wedge\ reset = reset_0 \ \wedge\ reset = 0) \end{aligned}$$

Thus by splitting the parameters space according to the condition $reset = 1$ *we have obtained a summary* S_2 *which is more precise than* S_1*. It ensures that* $cnt = 0$ *when* $reset = 1$*.*

S_2 *can still be seen as an intermediate step on the road of precision. We want to refine* φ_2 *by splitting it according to the condition* $cnt < max$ *and its negation* $cnt \geq max$*. We will call this refined domain* φ_3 *and* S_3 *the obtained summary.*

$$\begin{aligned} \varphi_3 \quad = \quad &\{(P \wedge reset = 1), \\ &(P \wedge reset = 0 \wedge cnt < max), \\ &(P \wedge reset = 0 \wedge cnt \geq max)\} \end{aligned}$$

We do not need to split the case where $reset = 1$ *because* φ_3 *already gives a partition of the precondition, thus satisfying what we have required for the domains of procedure summaries.*

$$\begin{aligned} S_3(P \wedge reset = 1) = (&\mathbf{cnt = 0} \ \wedge\ \mathbf{reset_0 = 1} \\ &\wedge\ 0 \leq event_0 \leq 1 \ \wedge\ max_0 \geq 0 \\ &\wedge\ reset = reset_0 \ \wedge\ event = event_0 \\ &\wedge\ max = max_0 \ \wedge\ cnt_0 \geq 0) \end{aligned}$$

$$\begin{aligned} S_3(P \wedge reset = 0 \wedge cnt < max) = (&\mathbf{cnt = cnt_0 + event} \\ &\wedge\ cnt_0 < max \ \wedge\ reset_0 = 0 \\ &\wedge\ 0 \leq event_0 \leq 1 \ \wedge\ max_0 \geq 0 \\ &\wedge\ reset = reset_0 \ \wedge\ event = event_0 \\ &\wedge\ max = max_0 \ \wedge\ cnt_0 \geq 0) \end{aligned}$$

$$\begin{aligned} S_3(P \wedge reset = 0 \wedge cnt \geq max) = (&\mathbf{cnt = cnt_0} \\ &\wedge\ cnt_0 \geq max \ \wedge\ reset_0 = 0 \\ &\wedge\ 0 \leq event_0 \leq 1 \ \wedge\ max_0 \geq 0 \\ &\wedge\ reset = reset_0 \ \wedge\ event = event_0 \\ &\wedge\ max = max_0 \ \wedge\ cnt_0 \geq 0) \end{aligned}$$

S_3 *is the most precise summary of the* **counter_update** *procedure that we have computed. It ensures that* $cnt = 0$ *when* $reset = 1$ *which is a property we had in* S_2*, but now we have that* $cnt = cnt_0 + 1$ *when an event occurs and that* $event = 1$ *and* $cnt_0 < max$*.*

When no event occurs, $cnt = cnt_0$ *when* $event = 0$*.* S_3 *also ensures that* $cnt = cnt_0$ *when* $cnt_0 = max$ *and* $reset = 0$*. It ensures that the counter remains to its maximum value when it has reached it, until it is reset.*

We can therefore see some possible tradeoffs in our approach, from S_1 *to* S_3*, between the precision, the size of the summary and the cost of analysis.*

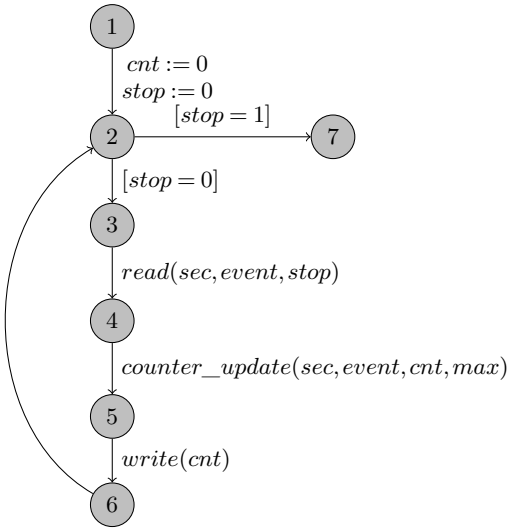
3.4 Application of procedure summaries

We focus our interest in this section on the application of procedure summaries, which is the transformation of a convex polyhedron on the actual parameters by a call statement using the summary of the called procedure.

Example 3.8 (Detector) *The* **detector** *procedure counts the number of events per second occurring in its environment until it reads a stop signal. When an event is read,* **event** *is set to* 1 *and otherwise to* 0*. The number of events per second is stored in* **cnt**. *When a second is read from the clock,* **sec** *is set to* 1 *by the* **read** *procedure. Similarly, the* **stop** *variable is set to* 1 *when the stop signal is received.*

```
-- require max >= 0
procedure detector (max : int)
   event, sec, cnt, stop : int;
begin
   cnt := 0; stop := 0;

   while stop = 0 do
       read(sec, event, stop);
       counter_update(sec, event, cnt, max);
       write(cnt);
   end;
end;
```



The precondition $\mathcal{A}$ of the **`detector`** *procedure is*

$$\mathcal{A} = \{max \in \mathcal{N} \mid max \geq 0\}$$

We compute the summary of the **`detector`** *procedure using the domain φ_1*

$$\varphi_1 = \{(max \geq 0)\}$$

We construct the summary of the **`procedure`** *using φ_1.*

$$X_1 = (max_0 \geq 0 \ \wedge \ max = max_0)$$

$$\begin{aligned} X_2 = \quad & (max_0 \geq 0 \ \wedge \ max = max_0 \ \wedge \ cnt = 0 \\ & \wedge \ stop = 0 \ \wedge \ event = 0 \ \wedge \ sec = 0) \end{aligned}$$

$$\begin{aligned} X_4 = \quad & (max_0 \geq 0 \ \wedge \ max = max_0 \ \wedge \ cnt = 0 \\ & \wedge \ 0 \leq stop \leq 1 \ \wedge \ 0 \leq event \leq 1 \\ & \wedge \ 0 \leq sec \leq 1) \end{aligned}$$

The current value of X_4 gives us a relation on the initial values of the parameters of **`counter_update`**. *We substitute the actual parameters of* **`counter_update`** *in X_4 by the zero-indexed version of its formal parameters.*

$$\begin{aligned} X_4' \quad & = X_4[sec \mapsto reset_0, \ event \mapsto event_0, \\ & \qquad cnt \mapsto cnt_0, \ max \mapsto max_0] \\ & = (max_0 \geq 0 \ \wedge \ cnt_0 = 0 \ \wedge \ 0 \leq stop \leq 1 \\ & \quad \wedge \ 0 \leq event_0 \leq 1 \ \wedge \ 0 \leq reset_0 \leq 1) \end{aligned}$$

We want to compute the value of X_5, which is the abstract state reachable at program point 5. We will use the summary S_{cu} of **`counter_update`** *that we have obtained previously. We perform an intersection between X_4' and each convex polyhedron of the summary S_{cu}.*

$$\begin{aligned} & X_4' \sqcap S_{cu}(P \wedge reset = 1) = \\ & cnt = 0 \ \wedge \ reset_0 = 1 \ \wedge \ 0 \leq event_0 \leq 1 \ \wedge \ max_0 \geq 0 \\ & \wedge \ reset = reset_0 \ \wedge \ event = event_0 \ \wedge \ max = max_0 \\ & \wedge \ cnt_0 = 0 \ \wedge \ 0 \leq stop \leq 1 \end{aligned}$$

$$\begin{aligned} & X_4' \sqcap S_{cu}(P \wedge reset = 0 \wedge cnt < max) = \\ & cnt = cnt_0 + event \ \wedge \ cnt_0 < max \ \wedge \ reset_0 = 0 \\ & \wedge \ reset = reset_0 \ \wedge \ event = event_0 \\ & \wedge \ max = max_0 \ \wedge \ cnt_0 = 0 \\ & \wedge \ 0 \leq stop \leq 1 \ \wedge \ 0 \leq event_0 \leq 1 \ \wedge \ max_0 \geq 0 \end{aligned}$$

$$\begin{aligned} & X_4' \sqcap S_{cu}(P \wedge reset = 0 \wedge cnt \geq max) = \\ & cnt = cnt_0 \ \wedge \ reset_0 = 0 \ \wedge \ 0 \leq event_0 \leq 1 \ \wedge \ max_0 \geq 0 \\ & \wedge \ reset = reset_0 \ \wedge \ event = event_0 \\ & \wedge \ max = max_0 \ \wedge \ cnt_0 = 0 \\ & \wedge \ 0 \leq stop \leq 1 \ \wedge \ cnt_0 \geq max \end{aligned}$$

We compute the convex hull of those 3 convex polyhedra and we denote by X_4'' this value.

$$\begin{aligned} X_4'' = (& cnt \leq event \ \wedge \ cnt + reset \leq 1 \\ & \wedge \ cnt \leq max \ \wedge \ cnt_0 = 0 \\ & \wedge \ event = event_0 \ \wedge \ max = max_0 \\ & \wedge \ reset = reset_0 \ \wedge \ event \leq 1 \\ & \wedge \ 0 \leq stop \leq 1 \ \wedge \ cnt \geq 0 \\ & \wedge \ reset \geq 0 \ \wedge \ max \geq 0) \end{aligned}$$

Our aim is to have the effect of the call to **`counter_update`** *on the variables of the caller, the* **`detector`** *procedure. We replace in X_4'' the formal parameters of* **`counter_update`** *by the corresponding actual parameters. This value is denoted by X_4'''.*

$$\begin{aligned} X_4''' = X_4''[& reset_0 \mapsto sec_0, \ event_0 \mapsto event_0, \\ & cnt_0 \mapsto cnt_0, \ max_0 \mapsto max_0 \\ & reset \mapsto sec, \ event \mapsto event, \\ & cnt \mapsto cnt, \ max \mapsto max] \end{aligned}$$

$$\begin{aligned} X_4''' = (& cnt \leq event \ \wedge \ cnt + sec \leq 1 \\ & \wedge \ cnt \leq max \ \wedge \ cnt_0 = 0 \\ & \wedge \ event = event_0 \ \wedge \ max = max_0 \\ & \wedge \ sec = sec_0 \ \wedge \ event \leq 1 \\ & \wedge \ 0 \leq stop \leq 1 \ \wedge \ cnt \geq 0 \\ & \wedge \ sec \geq 0 \ \wedge \ max \geq 0) \end{aligned}$$

X_5 is obtained by projecting out the zero-indexed versions of the actual parameters, denoting their state before the call to **`counter_update`**.

$$X_5 = X_4''' \downarrow \{sec_0, event_0, cnt_0\}$$

$$\begin{aligned} X_5 = (& cnt \leq event \ \wedge \ cnt + sec \leq 1 \ \wedge \ 0 \leq event \leq 1 \\ & \wedge \ 0 \leq stop \leq 1 \ \wedge \ cnt \geq 0 \\ & \wedge \ cnt \leq max \ \wedge \ sec \geq 0 \\ & \wedge \ max = max_0 \ \wedge \ max_0 \geq 0) \end{aligned}$$

We have $X_6 = X_5$ and we compute the current value of X_2 by applying widening.

$$\begin{aligned} X_2 = (& max_0 \geq 0 \ \wedge \ max = max_0 \ \wedge \ 0 \leq cnt \leq max \\ & \wedge \ 0 \leq stop \leq 1 \ \wedge \ 0 \leq event \leq 1 \ \wedge \ 0 \leq sec \leq 1) \end{aligned}$$

$X_4 = (max_0 \geq 0 \ \wedge \ max = max_0 \ \wedge \ 0 \leq cnt \leq max$
$\wedge \ 0 \leq stop \leq 1 \ \wedge \ 0 \leq event \leq 1 \ \wedge \ 0 \leq sec \leq 1)$

We compute similarly the current value of X_5.

$X'_4 = X_5[sec \mapsto reset_0, \ event \mapsto event_0,$
$cnt \mapsto cnt_0, \ max \mapsto max_0]$
$= (max_0 \geq 0 \ \wedge \ 0 \leq cnt_0 \leq max \ \wedge \ 0 \leq stop \leq 1$
$\wedge \ 0 \leq event_0 \leq 1 \ \wedge \ 0 \leq reset_0 \leq 1)$

We perform an intersection between X'_4 *and each convex polyhedron of the summary* S_{cu}.

$X'_4 \sqcap S_{cu}(P \wedge reset = 1) =$
$cnt = 0 \ \wedge \ reset_0 = 1 \ \wedge \ 0 \leq event_0 \leq 1 \ \wedge \ max_0 \geq 0$
$\wedge \ reset = reset_0 \ \wedge \ event = event_0 \ \wedge \ max = max_0$
$\wedge \ 0 \leq cnt_0 \leq max \ \wedge \ 0 \leq stop \leq 1$

$X'_4 \sqcap S_{cu}(P \wedge reset = 0 \wedge cnt < max) =$
$cnt = cnt_0 + event \ \wedge \ 0 \leq cnt_0 < max \ \wedge \ reset_0 = 0$
$\wedge \ reset = reset_0 \ \wedge \ event = event_0 \ \wedge \ max = max_0$
$\wedge \ 0 \leq stop \leq 1 \ \wedge \ 0 \leq event_0 \leq 1 \ \wedge \ max_0 \geq 0$

$X'_4 \sqcap S_{cu}(P \wedge reset = 0 \wedge cnt \geq max) =$
$cnt = cnt_0 \ \wedge \ reset_0 = 0 \ \wedge \ 0 \leq event_0 \leq 1$
$\wedge \ max_0 \geq 0$
$\wedge \ reset = reset_0 \ \wedge \ event = event_0 \ \wedge \ max = max_0$
$\wedge \ cnt_0 = max \ \wedge \ 0 \leq stop \leq 1$

We compute the convex hull of those 3 convex polyhedra and we denote by X''_4 *this value.*

$X''_4 = (0 \leq cnt \leq max \ \wedge \ cnt + reset_0 \leq cnt_0 + 1$
$\wedge \ cnt \leq cnt_0 + event \wedge \ 0 \leq cnt_0 \leq max$
$\wedge \ 0 \leq reset_0 \leq 1 \ \wedge \ 0 \leq event_0 \leq 1$
$\wedge \ 0 \leq stop \leq 1 \wedge \ max_0 \geq 0$
$\wedge \ reset = reset_0 \ \wedge \ event = event_0 \ \wedge \ max = max_0)$

$X'''_4 = (0 \leq cnt \leq max \ \wedge \ cnt + sec_0 \leq cnt_0 + 1$
$\wedge \ cnt \leq cnt_0 + event \wedge \ 0 \leq cnt_0 \leq max$
$\wedge \ 0 \leq sec_0 \leq 1 \ \wedge \ 0 \leq event_0 \leq 1$
$\wedge \ 0 \leq stop \leq 1 \wedge \ max_0 \geq 0$
$\wedge \ sec = sec_0 \ \wedge \ event = event_0 \ \wedge \ max = max_0)$

$X_5 = X'''_4 \downarrow \{sec_0, event_0, cnt_0\}$

$X_5 = (\mathbf{0 \leq cnt \leq max} \ \wedge \ \mathbf{cnt + sec \leq max + 1}$
$\wedge \ \mathbf{cnt \leq max + event} \wedge \ 0 \leq stop \leq 1$
$\wedge \ 0 \leq event \leq 1 \ \wedge \ 0 \leq sec \leq 1$
$\wedge \ max_0 \geq 0 \ \wedge \ max = max_0)$

We compute one term of the descending sequence and we obtain at program points 2

$X_2 = (\mathbf{0 \leq cnt \leq max} \ \wedge \ \mathbf{cnt + sec \leq max + 1}$
$\wedge \ \mathbf{cnt \leq max + event} \wedge \ 0 \leq stop \leq 1$
$\wedge \ 0 \leq event \leq 1 \ \wedge \ 0 \leq sec \leq 1$
$\wedge \ max_0 \geq 0 \ \wedge \ max = max_0)$

$X_8 = (\mathbf{0 \leq cnt \leq max} \ \wedge \ \mathbf{cnt + sec \leq max + 1}$
$\wedge \ \mathbf{cnt \leq max + event} \wedge \ stop = 1$
$\wedge \ 0 \leq event \leq 1 \ \wedge \ 0 \leq sec \leq 1$
$\wedge \ max_0 \geq 0 \ \wedge \ max = max_0)$

We have discovered in this program using the summary of `counter_update` *that the variable* `cnt`*, which counts the number of events per second, remains always between* 0 *and* `max` *at program point* 2.

Definition 3.7 (Summary application) *Let* f *be a procedure calling a procedure* g *in a call statement* $g(act_g)$ *where* $act_g \in Id^*$ *is the list of the actual parameters of* g *and* $R \in D^\sharp$ *is the abstract state reachable in* f *before the call statement to* g. *Let* $S_g : D^\sharp \to D^\sharp$ *be the summary of* g *and* $\mathcal{A} \subseteq \mathcal{N}^m$ *its precondition.*

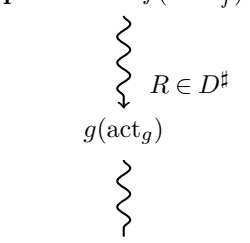
The abstract state reachable after the call statement to g *in* f *is*

if $\gamma(R[\boldsymbol{act_g}/\boldsymbol{form_g}]) \subseteq \mathcal{A}$

$$\bigsqcup_{P_i \in Dom(S_g)} R[act_g/Z(act_g)] \sqcap S_g(P_i)[Z(form_g)/Z(act_g)][form_g/act_g] \downarrow Z(act_g)$$

otherwise error

We denote by $app(S_p, R)$ *the application of the summary* $S_p : D^\sharp \to D^\sharp$ *of a procedure* p *in the abstract state* $R \in D^\sharp$.



The actual parameters of g in act_g are either formal parameters of f or local variables of f. The zero-indexed version $Z(\text{act}_g)$ of the actual parameters of g are variables representing the values of actual parameters before the call to g.

$R[\text{act}_g/Z(\text{act}_g)]$ is the substitution of the actual parameters act_g by their zero-indexed version $Z(\text{act}_g)$ in R, the abstract state reachable before the call. This gives a linear relation on the initial values of the actual parameters before the call to g.

For a convex polyhedron $P_i \in Dom(S_g)$, $S_g(P_i)[Z(\text{form}_g)/Z(\text{act}_g)]$ denotes parameter passing. It is the substitution in $S_g(P_i)$ of $Z(\text{form}_g)$ the zero-indexed formal parameters of g by $Z(\text{act}_g)$ the zero-indexed actual parameters of g.

$S_g(P_i)[Z(\text{form}_g)/Z(\text{act}_g)]$ is thus a linear relation between $Z(\text{act}_g)$ the zero-indexed actual parameters of g and form_g the formal parameters of g.

To capture the effect of the call statement, we want a linear relation between $Z(\text{act}_g)$ and act_g only, between the actual parameters of g and their zero-indexed version. Therefore we need to substitute the formal parameters form_g of g by the actual parameters act_g of g in the polyhedron obtained before. $S_g(P_i)[Z(\text{form}_g)/Z(\text{act}_g)][\text{form}_g/\text{act}_g]$ is then a linear relation between act_g the actual parameters of g and $Z(\text{act}_g)$ their zero-indexed version.

We then compute the intersection $R[\text{act}_g/Z(\text{act}_g)] \sqcap S_g(P_i)[Z(\text{form}_g)/Z(\text{act}_g)][\text{form}_g/\text{act}_g]$. The result is

still a linear relation between $Z(\text{act}_g)$ and act_g. Because we want the abstract state after the call, which must be a relation between act_g, lv_f and form_g, we eliminate $Z(\text{act}_g)$, the zero-indexed version of the actual parameters of g.

Finally, we compute the join for all the polyhedra P_i in the domain of the summary of g. This gives us the abstract state after the call to g.

4 Summary semantics

We discuss in this section the semantics and correctness of procedure summaries.

Definition 4.1 (Summary semantics) *The summary S_p of a procedure p represent the relation $\mathcal{R}_p$ defined as*

$$\mathcal{R}_p = \bigvee_{P \in Dom(S_p)} P(X_0) \wedge S_p(P)(X_0, X)$$

$$X_0 = Z(form_p) \qquad X = form_p$$

with $X_0 = Z(form_p)$ representing the initial values of the formal parameters of p and $X = form_p$ representing the final values of the formal parameters of p. We call the relation R_p the summary semantics of summary S_p.

$\mathcal{R}_p$ is a relation between the initial values of the parameters of p, represented by the variables in $X_0 = Z(\text{form}_p)$, and their final values, represented by the variables in $X = \text{form}_p$.

Proposition 4.1 (Summary correctness) *The summary S_p is a correct summary of a procedure p if and only if for every initial variable environment $\rho_0 \in Env$, every procedure environment $\pi \in PEnv$ and every initial memory $\sigma_0 \in Mem$, the execution of the body d_v S of procedure p produces a variable environment ρ and a memory σ giving a final value $\sigma(\rho(x))$ to every variable $x \in X$ and satisfying the relation R_p for every variable $x_0 \in X_0$ and $x \in X$ such that*

$$\forall \rho_0 \in Env,\ \forall \pi \in PEnv,\ \forall \sigma_0 \in Mem,$$
$$\pi, \rho_0, \sigma_0 \vdash d_v\ S : \rho, \sigma \Rightarrow \mathcal{R}_p[\forall x_0 \in X_0, x_0/\sigma_0(\rho_0(x_0))]\ [\forall x \in X, x/\sigma(\rho(x))]$$

5 Summaries of synchronous reactive programs

We describe in this section an application of our approach to the verification of synchronous reactive programs. Synchronous programming [Halbwachs *et al.*, 1991], [Halbwachs, 1998] has been proposed as a way to describe reactive systems such as automatic control systems. A synchronous program is assumed to react instantly and deterministically to events from its environment. Synchronous languages like Lustre [Halbwachs *et al.*, 1991] [Caspi *et al.*, 1987] have precise formal semantics and allow an especially eleguant programming style. They are used in industry for the development of critical systems, especially for avionics and nuclear control systems.

Lustre is a well-known data flow synchronous language, in which programs are directed graphs of components communicating using flows of values. Each component is defined by a system of equations in which each variable describes a flow, which is an infinite sequence of values. The **counter** component in Example 5.1 is a typical Lustre component.

Example 5.1 (The counter node)

```
node counter (reset, event : bool; max : int)
returns (cnt : int);
let
    cnt = 0 ->
        if reset then
            0
        else if event and pre(cnt) < max then
            pre(cnt) + 1
        else
            pre(cnt);
tel;
```

Lustre programs can be compiled into sequential code using an automata-based technique or by generating a single infinite loop and a step procedure for each component, implementing the inputs-to-outputs transformation performed during one cycle. The counter component of Example 5.1 is compiled into the **counter_step** procedure, shown in Example 5.2. The **main** procedure contains an infinite loop reading inputs, calling the **counter_step** procedure and writing outputs.

The *pre* operator in Lustre denotes the previous value of a flow. It is implemented using special *pre_id* variables, like *pre_cnt* in Example 5.2. Hence, the step procedure of a Lustre component takes not only parameters corresponding to input and output flows, but also parameters like *pre_cnt* used to hold the state of the component.

Existing approaches [Halbwachs *et al.*, 1997], [Jeannet, 2000] for the verification of synchronous data flow programs using Linear Relation Analysis are non-modular. The approach described in [Halbwachs *et al.*, 1997] is based on the automata representation of synchronous programs. In both approaches, the main component must be fully expanded in order to obtain a flat Lustre program.

We want to apply our *relational procedure summaries* approach to the verification of Lustre programs using Linear Relation Analysis in a modular fashion. This is only a preliminary assessment of our approach for the modular verification of synchronous programs. The summary of the step procedure of each component is computed only once, and applied in each call context, using summary application as previously defined.

Example 5.2

```
init : int;
procedure counter_step (reset, event : bool;
                        max, pre_cnt, cnt : int)
begin
    if init then
        cnt := 0;
    else
        if reset then
            cnt := 0;
        else
            if event and pre_cnt < max then
                cnt := pre_cnt + 1;
            else
                cnt := pre_cnt;
            end if;
        end if;
    end if;
end;

procedure main ()
    reset, event : bool;
    max, pre_cnt, cnt : int;
begin
    init := true;

    while true do
        read(reset, event, max);
        counter_step(reset, event, max,
                     pre_cnt, cnt);
        write(cnt);

        pre_cnt := cnt;
        init := false;
    done;
end;
```

We want to compute a summary of the **counter_step** procedure. We choose the domain φ of the summary such that

$$\varphi = \{P_1, P_2, P_3\}$$

The summary S of the **counter_step** procedure is such that $S = [P_1 \mapsto Q_1, P_2 \mapsto Q_2, P_3 \mapsto Q_3, P_4 \mapsto Q_4]$.

$$P_1 = (init \vee reset) \wedge n \geq 0$$

$$P_2 = \neg init \wedge \neg reset \wedge event \wedge pre\_cnt < n \wedge n \geq 0$$

$$P_3 = \neg init \wedge \neg reset \wedge \neg(event \wedge pre\_cnt < n) \wedge n \geq 0$$

$$Q_1 = (init \vee reset) \wedge n \geq 0 \wedge (cnt = 0)$$

$$Q_2 = \neg init \wedge \neg reset \wedge event \wedge pre\_cnt < n \wedge n \geq 0$$
$$\wedge\ cnt = pre\_cnt + 1 \wedge (pre(Q_1) \vee pre(Q_2) \vee pre(Q_3))$$

$$Q_3 = \neg init \wedge \neg reset \wedge \neg(event \wedge pre\_cnt < n)$$
$$\wedge\ n \geq 0 \wedge cnt = pre\_cnt \wedge (pre(Q_1) \vee pre(Q_2) \vee pre(Q_3))$$

We use an increasing and decreasing sequence in order to compute Q_1, Q_2, Q_3, defined as follows.

Initialization $Q_1^0 = \bot \quad Q_2^0 = \bot \quad Q_3^0 = \bot$

Iteration 1

$$Q_1^1 = (init \vee reset) \wedge n \geq 0 \wedge (cnt = 0)$$
$$Q_2^1 = \bot$$
$$Q_3^1 = \bot$$

Iteration 2

$$Q_2^2 = \neg init \wedge \neg reset \wedge event \wedge pre\_cnt < n$$
$$\wedge\ n \geq 0 \wedge cnt = pre\_cnt + 1 \wedge pre\_cnt = 0$$

$$Q_3^2 = \neg init \wedge \neg reset \wedge \neg(event \wedge pre\_cnt < n)$$
$$\wedge\ n \geq 0 \wedge cnt = pre\_cnt \wedge pre\_cnt = 0$$

Iteration 3

$$Q_2^3 = \neg init \wedge \neg reset \wedge event \wedge pre\_cnt < n$$
$$\wedge\ n \geq 0 \wedge cnt = pre\_cnt + 1 \wedge 0 \leq pre\_cnt \leq 1$$

$$\begin{aligned} Q_2'^3 &= Q_2^2 \nabla Q_2^3 \\ &= \neg init \wedge \neg reset \wedge event \wedge pre\_cnt < n \\ &\wedge\ n \geq 0 \wedge cnt = pre\_cnt + 1 \wedge 0 \leq cnt \leq n \end{aligned}$$

$$\begin{aligned} Q_3^3 &= \neg init \wedge \neg reset \wedge \neg(event \wedge pre\_cnt < n) \\ &\wedge\ n \geq 0 \wedge cnt = pre\_cnt \\ &\wedge (pre\_cnt = 0 \vee 0 \leq pre\_cnt \leq n) \\ &= \neg init \wedge \neg reset \wedge (\neg event \vee pre\_cnt = n) \wedge n \geq 0 \\ &\wedge\ cnt = pre\_cnt \wedge 0 \leq pre\_cnt \leq n \end{aligned}$$

$$\begin{aligned} Q_3'^3 &= Q_3^2 \nabla Q_3^3 \\ &= \neg init \wedge \neg reset \wedge (\neg event \vee pre\_cnt = n) \wedge n \geq 0 \\ &\wedge\ cnt = pre\_cnt \wedge 0 \leq pre\_cnt \leq n \end{aligned}$$

Results

$$Q_1 = (init \vee reset) \wedge n \geq 0 \wedge (cnt = 0)$$

$$Q_2 = \neg init \wedge \neg reset \wedge event \wedge pre\_cnt < n$$
$$\wedge\ n \geq 0 \wedge cnt = pre\_cnt + 1 \wedge 0 \leq cnt \leq n$$

$$Q_3 = \neg init \wedge \neg reset \wedge (\neg event \vee pre\_cnt = n) \wedge n \geq 0$$
$$\wedge\ cnt = pre\_cnt \wedge 0 \leq cnt \leq n$$

We have succeeded to find a summary of the step procedure associated to the **counter** component. This summary is constructed in order to take into account the cyclic behaviour of synchronous programs.

6 Conclusion

Linear Relation Analysis is one of the most powerful analysis based on abstract interpretation which is able to discover automatically linear relations between the variables of a program at each program point. Its complexity is the price to pay for its power, it hinders its scalability to large real-world software. This is why existing industrial static analysis tools usually implements weaker, but less complex static analysis techniques.

We proposed in this work a new program analysis to address this scalability challenge, by automatically constructing *relational* procedure summaries in a sound and safe fashion, using Linear Relation Analysis. We shown that summarizing procedures by a finite disjunction of linear relations between the initial and final values of the parameters is both feasible and expressive enough to enable a modular analysis. We discussed the correctness of procedure summaries and introduced summary semantics. We gave several heuristics enabling different tradeoffs regarding the precision and complexity of analysis. Finally, we described an extension of our approach, opening the way to the modular verification of synchronous reactive systems.

References

[Allen, 1974] Frances E. Allen. Interprocedural data flow analysis. In *IFIP Congress*, pages 398–402, 1974.

[Ancourt *et al.*, 2010] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267(1):3–16, October 2010.

[Blanchet *et al.*, 2003] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.

[Caspi *et al.*, 1987] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[Cousot and Halbwachs, 1978] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[Emami *et al.*, 1994] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Notices*, volume 29, pages 242–256. ACM, 1994.

[Euclid, c 300 BC] Euclid. *Euclid's elements, Book VII, Proposition 1.* c. 300 BC.

[Halbwachs *et al.*, 1991] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[Halbwachs *et al.*, 1997] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[Halbwachs, 1998] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16. Springer, 1998.

[Henry *et al.*, 2012] Julien Henry, David Monniaux, and Matthieu Moy. PAGAI: a path sensitive static analyzer. In *Tools for Automatic Program Analysis (TAPAS)*, 2012.

[Jeannet, 2000] Bertrand Jeannet. *Partitionnement dynamique dans l'Analyse de Relation Linéaire et application à la vérification de programmes synchrones.* PhD thesis, 2000.

[Jeannet, 2010] Bertrand Jeannet. Interproc analyzer for recursive programs with numerical variables. *INRIA, software and documentation are available at the following URL: http://pop-art. inrialpes. fr/interproc/interprocweb. cgi. Last accessed*, pages 06–11, 2010.

[MathWorks, 2016] MathWorks. Polyspace, 2016.

[Shapiro and Horwitz, 1997] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1997.

[Sharir and Pnueli, 1978] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[Steensgaard, 1996] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[Yorsh *et al.*, 2008] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *ACM SIGPLAN Notices*, volume 43, pages 221–234. ACM, 2008.

[Zhang *et al.*, 2014] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. *SIGPLAN Not.*, 49(6):249–258, June 2014.

Scat : Function prototypes retrieval inside stripped binaries

Christopher Ferreira
Magistère L3 - Supervised by Franck de Goër

1 Context

Some software systems either are or rely on closed-source code. This is problematic when security is an important concern because security analysis often needs high level information about the program which are mostly lost during the compilation process. At the extreme end, binaries called "stripped" only contain the bare minimum for the program to run.

Several approaches have been designed to address this problem, using static ([1; 3; 5]) or dynamic ([2; 6]) analysis and with different accuracy and execution time constraints depending on the use case. Scat is one of these systems, it uses a lightweight instrumentation with some heuristics to recover program informations. It is divided in several tools (arity, type, coupling, ...), each one of them achieving its purpose in one single instrumented execution.

Scat already provided accurate results for prototype detection, 93.8% accuracy for the arity detection tool and 89.4% for the type tool on a set of four programs. But there was still room for improvements.

Our work ranges from bug fixes and heuristics fine-tuning to removal of some implementation limitations. After our modifications the results were globally improved : 97.7% for arity and 95.9% for type using the same set of programs. Only some of these improvements will be detailed here, in order to highlight how scat manages to provide accurate results while remaining lightweight.

2 Call stack tracking

In order to detect the arity of functions and the type of parameters, scat requires a method to keep tracks of the call stack. Before our work, an ad-hoc solution was used which caused some functions to be simply ignored. But solving this problem was not as simple as using an automatically growing stack because that would mean a lot of work and memory in case of a substantial amount of recursive calls.

We devised a new call stack tracking system to solve this problem while staying true to scat lightweight goal. It is based on a simple assumption. Keeping track of all the call stack in the case of a recursion is unnecessary because the call stack contains mostly the same function and that we only need a reasonably sized sample of the function calls to deduced our results. The new system, that we named "hollow stack", only stores a fixed number of functions at the bottom (with a straightforward stack) and at the top (using a ring buffer) of the stack, thus forgetting the functions in the middle (replaced by an integer counter) in case of an overflow/recursion. This effectively guarantees, given a suitable fixed size, that the analysis will detect all functions before/at the end of the recursion and a sufficient number of the functions involved in the recursion.

3 Stack parameters detection

Another contribution of our work was to add support for stack parameters (scat was limited to the first 6 parameters before). As opposed to more involved analysis like the one proposed in [4], our solution is quite straightforward and only assumes the SystemV x86_64 call conventions[1][2].

Every time a function is called the stack pointer at this exact moment is stored. It delimits the portion of the stack pertaining to the calling function (before the pointer) and the portion pertaining to the called function (after the pointer). The stack parameters are stored in the last part of the caller stack (i.e: just before the pointer). Then, each time a memory read relative to the stack pointer (or the base/frame pointer) occurs, recognizing a stack parameter is only a matter of comparing the read address to the stack pointer stored for the current function.

4 Results

The following table sums up our results.

The Functions entry shows that the number of functions detected increases. This is mostly thanks to the added support for indirect calls which were not detected before and to a lesser extent the call stack tracking improvements.

The percentages give, for each entry, the number of functions or parameters detected by scat accordingly to the informations extracted from the source code. All results are improved and more consistent with a minimum of 92%.

The increase execution time for the detection of arity is really negligible given the new features. For the type detection, on the other hand, the increase is more significant but remains in accordance with our expectations.

[1] The first 6 parameters are passed using dedicated registers, remaining parameters are passed via the stack

[2] The register %sp always contains the stack top address

			bash		grep		mupdf		git	
Functions		#	53	**+8%**	77	**+7%**	569	**+19%**	476	**+21%**
Arity	Time	sec	3.6	**+12%**	3.8	**+8%**	22	**+22%**	60	**+13%**
	Params	%	94.3	**+24%**	94.8	**+17%**	95.0	**+0.8%**	98.8	**+8.2%**
	Return	%	100	**+2%**	98.7	**+0.1%**	99.5	**+1.8%**	98.1	**+2.4%**
Type	Time	sec	3.2	**+100%**	3.1	**+82%**	21	**+150%**	68	**+74%**
	Params	%	94.8	**+8.4%**	94.4	**+1%**	97.1	**+2.1%**	93.5	**+1.6%**
	Return	%	92.5	**+14%**	93.5	**+8.5%**	96.8	**+7.7%**	97.1	**+15.1%**

Table 1: Results after the improvements and comparison with previous results (omitted for brevity) for 4 programs

References

[1] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight : Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[2] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical Report 2, Electrical Engineering and Computer Sciences University of California at Berkeley, April–June 2009.

[3] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. Labeling library functions in stripped binaries. Technical report, Computer Sciences Department University of Wisconsin, 2011.

[4] Arun Lakhotia and Uday Eric. Stack shape analysis to detect obsfucated calls in binaries. In *Source Code Analysis and Manipulation*, 2004.

[5] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reversed engineering of types in binary programs. Technical report, Carnegie Mellong Uinversity, 2011.

[6] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures.

Formal proofs for an activation model of real-time systems

Lina Marsso
Spades, INRIA/Lig
marssolina@gmail.com

Supervised by: Sophie Quinton and Pascal Fradet

August 25, 2016

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

Abstract

Because real-time systems are often safety-critical, the analysis techniques used in this field should be sound and complete. However the real-time systems community has recently identified many unsound results due to human errors in their proofs. In particular, the well-known *general event load model* is ill-defined and leads to unsound analyses. This paper proposes formalized definitions and proofs related to this model using the Coq proof assistant. First, we revisit and formalize the definition of the general event load model. It mainly involves the identification of the needed hypotheses on traces in order to state theorems about this model. Then we generalize, formalize and prove the two fundamental theorems which allow switching between two representations of the general event load model. This work is entirely implemented and proved with Coq and will soon be integrated into an open-source library.

1 Introduction

The particularity of *real-time systems* is that they are constrained by timing requirements. Classical examples of such systems are aerospace, aircraft, automotive and medical systems. An example of timing requirement is a bound on the duration between the instant the pilot applies a command and the instant the aircraft changes directions. Real-time systems are often safety-critical systems: a malfunction in a real-time system may have serious consequences. As a glaring example, on June 4, 1996, the Ariane 5 spacecraft launch ended in a failure. Consequence includeded ecological and economical disasters. Clearly, such complex and safety-critical systems ask for rigorous and formal analysis.

Recently, the real-time systems analysis community has observed an important number of errata papers and unsound results, caused by a lack of formalization and handwritten proofs. To address this issue, there is a trend towards formalized and mechanized proofs using computer-verification tools. However, the few existing papers have each their own formalization, and use different computer-verification tools. This makes it impossible to compose results. Very recently, F. Cerqueira *et al.* [Cerqueira *et al.*, 2016] have proposed a unified formalization and proofs in an open source library called Prosa using the Coq proof assistant [The Coq development team, 2004]. Our final objective is to integrate the work presented here in Prosa. Real-time systems consist of software tasks executing on (possibly several) hardware resources. The activation frequency of each task is modeled by a so-called activation model. Several activation models exist. In this paper, we focus on one of them called the *general event load model*. This model describes scenarios of task activations by constraining the maximum number of activations in a given time duration, or dually the minimum duration containing a given number of task activations. Both representations are used by schedulability analyses. The general event load model is particularly useful for task activations which arrive in bursts, i.e., arrive by group of close activations. Conversion functions permit switching from the time-based view to the event-based view and back. The general event load model is an input of the analysis of the system. An incorrect input to the analysis would make the analysis unsound. The literature highlights contradictory definitions and a lack of proper formal proofs for general event load model. In this work, we provide a formal treatment of the event model definition and conversion functions.

The contributions of this paper are the following:

1. In order to get computable functions we revisit and formalize the general event load model definitions and theorems.
2. We identify the needed hypotheses and lemmas on traces in order to formalize and prove the theorems.
3. We prove the correctness of conversion functions between the two representations.

Our formalization and proofs are described using standard mathematical notions in this document. Some proofs written in Coq are illustrated in the appendix. The finality of this work is to contribute to the open source real-time system library Prosa [Cerqueira *et al.*, 2016].

The paper is organized as follows: Section 2 presents the background of this work. Section 3 presents the motivation and the problem statement of this thesis. In Section 4, we define formally the general event load model, and we discuss similarities and differences compared to the initial event load

model. Then, we explain how we proved and completed our specification in Section 5. Finally, we will presents the related work on mechanized proofs of real-time system analysis, suggests some research directions to explore and concludes.

2 Activation Model

The execution of a task is triggered by an input event called activation. An *activation trace* describes the set of *instants*[1] at which a given task is activated. Formally, an activation trace can be defined as a function mapping every instant to the number of activations occurring at that instant. An activation trace corresponds to a specific scenario of activations of a task.

An *activation model* is an abstraction of all possible traces. Each trace should respect the activation model. An activation model is used to define the input of the system and defines *when an instance of a task may arrive*. Its main application is the computation of the WCRT as proposed by Tindell [Tindell *et al.*, 1994]. Each task has an activation model, which defines the possible scenarios of activations. There exists several well-known activation models used in the literature. We focus on the *general event load* model [Henia *et al.*, 2005] (also known as *arbitrary event model*). The general event load model applies to activations but it can also be used for other events (e.g., start, finish, ...).

The general event load model describes the maximum and minimum number of event occurrences that may occur in a duration Δt, denoted $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ respectively. Alternatively, this model describes the functions $\delta^-(n)$ and $\delta^+(n)$ that represent the minimum and maximum duration where n events occur. Each one of these two representations can be computed from the other one thanks to *conversion functions*.

3 Motivation and Problem Statement

Real-time systems are often safety-critical systems. Therefore, the analysis techniques used in the field should be rigorous and correct. However, the real-time systems community has recently identified several unsound results.

As a glaring example, Ken Tindell *et al.* proposed in 1994, an extension of the response-time analysis of the Controller Area Network bus to static priority premptive scheduling [Tindell and Burns, 1994]. Only after 12 years did they notice that their work was partially incorrect [Davis *et al.*, 2007]. However, in the meantime the response-time analysis has been largely used in critical systems in the automotive industry and other commercial tools.

A second example relates to the general event-load model. There are contradictory definitions of the conversion functions, even in papers from the same team and on the same year [Axer *et al.*, 2013],[Neukirchner *et al.*, 2013]. For instance, there exists two distinct definitions of the same function:

$$\eta^+(\Delta t) = \max_{n \in \mathbb{N}^+} \{n \mid \delta^-(n) \leqslant \Delta t\}$$

[1]Throughout this document, we assume a discrete representation of time, where instants are natural numbers.

and

$$\eta^+(\Delta t) = \max_{n \in \mathbb{N}^+} \{n \mid \delta^-(n) < \Delta t\}$$

The first one in [Axer *et al.*, 2013] states that $\delta^-(n)$ should be only smaller than *or equal to* Δt, whereas $\delta^-(n)$ should strictly below Δt in the second one state that [Neukirchner *et al.*, 2013]. With high probability, at least one of these two papers is incorrect. In the state of the art, we found a lack of proper formal event model definition. For instance, in the thesis of S.Schliecker [Schliecker, 2010], the minimum event distance is defined informally by the sentence:

$\delta^-(n)$*, the size of the smallest time window within which n or more events may occur*,

whereas the function conversion is defined formally as

$$\delta^-(n) = \inf_{\Delta t \leqslant 0, \Delta t \in \mathbb{R}} \{\Delta t \mid \eta^+(\Delta t) \geqslant n\}$$

Its not possible to prove the correctness of a function partially formalized.

It appears that even the formal definition of the conversion is limited. Indeed, the function conversion supposes that respectively the minimum or maximum is reached, whereas with infinite traces a lower bound or upper bound seem to be more appropriate. Since the general event load model is an input of the system, an uncorrect event load model results in an erroneous schedulability analysis.

All these observations highlight a lack of rigor in a field whose applications are safety-critical. Even when the proposed techniques are guaranteed by pen-and-paper proofs, human errors show up. In this paper, we tackle this problem by formalizing the general event-load model and the associated conversion functions in the Coq proof assistant. The related work using a proof assistant for real-time analysis can be found in Section 6.

4 Timing Analysis with General Load Event Models

In this section, we present our revisited definitions for general event load models. We define the types and functions of the general event load models and discuss the differences between the initial models and ours.

Initially, the general load event model was defined in [Henia *et al.*, 2005]. Then a first definition based on traces and using two functions δ and η was proposed by [Quinton *et al.*, 2012]. These two functions simplify the definition of η^+ and δ^- as well as the conversion functions. The notion of trace was defined as an increasing function taking an event occurrence and returning its instant. Since functions are usually defined as function of time, we decided to follow the most common trace version which has the added advantage to be coherent with [Cerqueira *et al.*, 2016]. Inspired by this work, in this section we propose our general event model definitions.

4.1 Type Definitions

We consider discrete time as a succession of instants. The instant where an occurrence of an event takes place and time duration are represented by natural numbers. To improve the readability of our definitions, we use the following types:

(i) An INSTANT is the timestamp when some events may take place.

(ii) A DURATION expresses an interval of time between two instants.

(iii) NB_OCCURRENCES denotes a number of occurrences of an event at an instant or during a duration.

(iv) ID_OCCURRENCE identifies uniquely one occurrence of an event by its rank. The instant where an occurrence n takes place is lower or equal to the instant of the occurrence $(n+1)$.

The five previous types are represented by natural numbers.

4.2 Function Definitions

Trace

A trace is an infinite sequence of event occurrences. To simplify, we consider traces of occurrences of a single event. This definition can be generalized to more than one event. In practice, a trace is a file obtained by measuring event occurrences and the timestamps indicating at which instants they take place.

Definition 1. *A trace σ:type* INSTANT $\rightarrow$ NB_OCCURRENCES *is a function taking an instant and returning the number of occurrences of the event at this instant.*

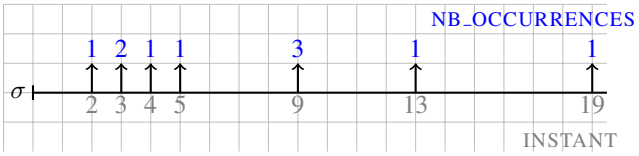


Figure 1: Example of a representation of a trace, the number of occurrences are in blue and instants are in gray.

For instance, in the trace σ represented in Figure 1, two occurrences (of the event under consideration) occur at instant 3 so $\sigma(3) = 2$.

All the following definitions are done relatively to a unique and implicit trace σ.

Event Distance Function (δ)

For a given trace σ, *the event distance function* δ defines for any occurrence n in σ, and a given number of occurrences k, the duration between the instant where the n-th occurrence takes place and the instant where the $(n+k)$-th occurrence takes place.

The event distance function δ definition is based on the intermediate function *iof*, which computes the instant when a given event occurrence takes place, *iof*: ID_OCCURRENCE $\rightarrow$ INSTANT and is defined as

$$iof(n) = iof'(n, 0)$$

where *iof'*: NB_OCCURRENCES $\rightarrow$ INSTANT $\rightarrow$ INSTANT is defined recursively as

$$iof'(n,t) = \begin{cases} t-1 & \text{if n = 0} \\ iof'((n-\sigma(t)),(t+1)) & \text{otherwise} \end{cases}$$

Definition 2. *The event distance function δ:* ID_OCCURRENCE $\rightarrow$ NB_OCCURRENCES $\rightarrow$ DURATION *is defined as*

$$\delta(n,k) = iof(n+k-1) - iof(n)$$

If $\delta(n,k) = x$ then the duration between the n-th and the $(n+k-1)$-th occurrence in the trace is x.

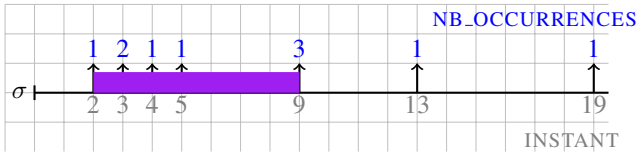


Figure 2: Example of a trace σ, with $\delta(2,8)$ in purple.

For instance, $d(n,k) = x$ means that the duration between the n-th occurrence and the $(n+k)$-th occurrence in the trace is x. Figure 2 illustrates the function on an example.

Minimum Event Distance Function (δ^-)

In real-time systems, the worst-case response time, that is, the maximum response time, must be evaluated to ensure that the system can meet its deadline. As we saw in Chapter **??**, the computation of the worst-case response time in a real-time system model is based on the minimum distance between two task activations.

A minimum event distance function δ^-: NB_OCCURRENCES $\rightarrow$ DURATION returns a lower bound of the distance between a number of occurrences in the trace σ.

Definition 3. *A minimum event distance function δ^- is a function that verifies the two following properties:*

- pseudo-super-additivity, *i.e,* $\forall a\, b,\ \delta^-(a+b+1) \geqslant \delta^-(a+1) + \delta^-(b+1)$
- δ-minimal, *i.e,* $\forall k\, n,\ \delta(n,k) \geqslant \delta^-(k)$

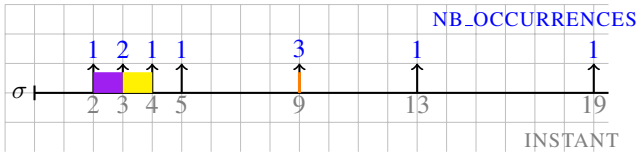


Figure 3: Example of a trace σ, with $\delta^-(3)$ in orange.

Event Load Function (η)

The *event load function* η returns for any occurrence n, the number of occurrences that appear in a given time window of duration Δt starting at the instant where occurrence n takes place.

For instance, $\eta(t, \Delta t) = x$ means that they are x occurrences of events in the half-open interval $[t,\ t+\Delta t[$. For example in Figure 2, there is one occurrence of event in the purple duration, i.e, $\eta(2,1) = 1$. Note that η does not count the two occurrences which occur at instant 3.

Maximum Event Load Function η^+
A maximum event load function η^+: DURATION $\rightarrow$ NB_OCCURRENCES returns an upper bound of the number of occurrences in a time window Δt.

Definition 4. *A maximum event load function η^+ is a function that verifies the two following properties:*

- sub-additivity, *i.e,* $\forall a\, b,\ \eta^+(a+b) \leqslant \eta^+(a) + \eta+(b)$
- η-maximal, *i.e,* $\forall \Delta t\, n,\ \eta(n, \Delta t) \leqslant \eta^+(\Delta t)$

Definition 5. *The event load function η:* INSTANT $\rightarrow$ DURATION $\rightarrow$ NB_OCCURRENCES *is defined recursively as*

$$\eta(t, \Delta t) = \begin{cases} 0 & if\ \Delta t = 0 \\ \sigma(t) + \eta(t+1, \Delta t - 1) & otherwise \end{cases}$$

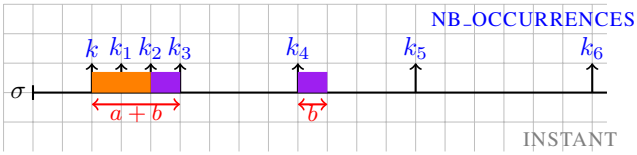


Figure 4: Example of a trace σ, with $\eta^+(a+b)$ is the $(k + k1 + k2 + k3)$ occurrences of the event that happened in the duration $a + b$.

Our definitions of the event minimum distance and the event maximum load function are different from the litterature [Henia *et al.*, 2005]. In the initial definitions, the event minimum distance function returns the minimum distance and the event maximum load function returns the maximum number of occurrences. The problem is that traces being infinite, this definition is not constructive: the functions δ^- and η^+ cannot be defined as fixpoints.

We therefore propose definitions that can be defined constructively where the minimum and the maximum may not be reached. The new definitions allow several possible event distance minimum functions or event load functions for the same trace σ. They only have to be respectively pseudo-supper-additive or sub-additive and to respect the δ-minimality (Definition 3) or η-maximality (Definition 4). In other words, our model is more general than the initial one.

4.3 Conversion Functions

The minimum distance function δ^- and the maximum event load function η^+ are strongly related. In this section, we provide conversion functions between δ^- and η^+. Their correctness are proved in the next chapter.

Minimum Event Distance to Maximum Load Conversion ($\delta^- \rightarrow \eta^+$)
The function η^+ is defined from δ^- by returning for each interval Δt the maximal number of occurrences k such that $\delta^-(k) < \Delta t$.

Theorem 1. *Let δ^- be a minimum event distance function (Definition 3) then the function η^+ defined as*

$$\eta^+(\Delta t) = \begin{cases} 0 & if\ \Delta t = 0 \\ max\{k \mid k \in \mathbb{N}, \delta^-(k) < \Delta t\} & otherwise \end{cases}$$

is a maximum load function.

In other words, Theorem 1 entails that $\eta^+(\Delta t)$ is sub-additive and η-maximal.

In Section 3, we saw two contradictory definitions of Theorem 1. Actually, we now know that the correct definition was given by M. Neukirchner *et al.* in [Neukirchner *et al.*, 2013]. Indeed, $\delta^-(k)$ must be strictly smaller than Δt because η^+ uses a half-open duration Δt. However, we should discern the case where the duration Δt is null, to avoid $\delta^-(k) < 0$.

Maximum Event Load to Distance Minimum Conversion ($\eta^+ \rightarrow \delta^-$)
The function δ^- is defined from η^+ by returning for each number of occurrences k the minimal duration Δt such that $\eta^+(\Delta t + 1) \geqslant k$.

Theorem 2. *Let η^+ be a maximum event load function (Definition 4) then the function δ^- defined as*

$$\delta^-(k) = min\{\Delta t \mid \Delta t \in \mathbb{N},\ \eta^+(\Delta t + 1) \geqslant k\}$$

is a minimum distance function.

In other words, Theorem 2 entails that $\delta^-(k)$ is pseudo-super-additive and δ-minimal (Definition 3).

In the standard general event load model the minimum and maximum are reached by the minimum event distance or maximum event functions respectively. The conversions between these two functions also reach the minimum or the maximum.

Since our definitions are slightly more general, we convert respectively sub-additive and η-maximal functions (Definition 4) into pseudo-super-additive and vice-versa.

5 Proofs of Correctness of the Conversion Functions

In this Section we provide the proofs for the main results related to the conversion of maximum event load and minimum distance functions stated in Section 4. However, before proving Theorems 1 and 2 we must adapt some definitions provided in Section 4 in order to make them computable.

The specification and proofs have been done within the Coq proof assistant. For convenience, we present them here using standard mathematical notions. (See the appendix for an illustration of the Coq implementation).

5.1 A Revisited Notion of Trace

In Chapter 4, the definition of the event distance function δ is based on the intermediate function *iof* (from Section 4.2), which computes the instant when a given event occurrence takes place.

$$iof(n) = iof'(n, 0)$$

where

$$iof'(n, t) = \begin{cases} t - 1 & \text{if n = 0} \\ iof'((n - \sigma(t)), (t + 1)) & \text{otherwise} \end{cases}$$

Unfortunately Coq refuses this definition of *iof* for two reasons.

The first issue with our definition is that *iof* is a partial function, which may not terminate on all inputs. For instance,

iof(2) is undefined for a trace containing only one event occurrence. This highlights the fact that Definition 1 is too general. In addition to being infinite in time, traces must also guarantee that events keep occurring. In other words, the trace function σ must not, even after a certain instant, be forever null. Formally, we only consider traces satisfying the following property:

Hypothesis 3. *For all instant t_1, there exists an instant t_2 such that*

$$t_2 > t_1 \wedge \sigma(t_2) > 0$$

Thanks to this hypothesis, *iof* is now a total function.

However, because Coq's "termination analysis" is not very sophisticated, it still refuses our new definition of *iof*. Indeed, *iof* is defined in terms of *iof'* whose termination is not obvious: to conclude that a recursive function is terminating, Coq needs a structurally decreasing argument. In consequence, from the definition of *iof'*, Coq must be able to infer that $n - \sigma(t) < n$, or equivalently, $\sigma(t) > 0$. However, Hypothesis 3 only guarantees that σ will be eventually non-null; therefore, $\sigma(t)$ may be temporarily null.

To provide an argument that is structurally decreasing, we must redefine *iof'* such that it only "iterates" on the instants where an event occurs. Hence, there will only be recursive calls to *iof'*(n, t) such that at least one event occurs at instant t. Thus, $\sigma(t)$ will always be strictly positive as required.

The standard library of Coq provides a module called *ConstructiveEpsilon* which from a property $P : \mathbb{N} \to$ *Prop*, allow defining a function finding a natural number that satisfies P. Based on Hypothesis 3, we can easily show that for all t_1, there exists a smaller t_2 such that $t_2 > t_1 \wedge \sigma(t_2) > 0$. We therefore have the property

$$P(t_1) = (\exists t_2, t_2 > t_1 \wedge \sigma(t_2) > 0) \wedge (\forall t_3.\, t_3 > t_1 \wedge t_3 < t_2 \implies \sigma(t_3) = 0)$$

This property allows the definition of the function $next(t)$: INSTANT $\to$ INSTANT which returns the first instant t' after a given instant t such that an event occurs at t':

$$next(t) = min\{t' \mid t' \in \mathbb{N}, t' \geqslant t \wedge \sigma(t') > 0\}$$

We can now redefine *iof'* in terms of *next*: *iof'*$(n, t) =$

$$\begin{cases} t - 1 & if\ n = 0 \\ \textit{iof'}(n - \sigma(\textit{next}(t)), \textit{next}(t) + 1) & \text{otherwise} \end{cases}$$

Because of the definition of *next*, $\sigma(\textit{next}(t))$ is strictly positive. Therefore, the first argument is strictly decreasing at each recursive call

$$n - \sigma(next(t)) < n$$

and Coq accepts the definition of *iof'* (and *iof*) as terminating.

Since, the number of occurrences happening at an instant is not a priori bounded, we can not write in Coq Theorem 1 defined as

$$\eta^+(\Delta t) = \begin{cases} 0 & if\ \Delta t = 0 \\ max\{k \mid k \in \mathbb{N}, \delta^-(k) < \Delta t\} & \text{otherwise} \end{cases}$$

Indeed, consider the trace such that $\sigma(t) = t$, there is no maximum number of occurences happening at an instant and η^+ is not computable. We consider only traces with a bounded number of occurrences and we make the assumption that such a maximum number of occurrences k_max exists.

Hypothesis 4. *For all instant t, $\sigma(t) < k_{max}$*

Thanks to this hypothesis, Theorem 1 is now well-defined.

5.2 Properties of the General Event Load Model

In order to prove the conversion theorems between the minimum distance and maximum event load functions, we first formalize and prove four lemmas between the four functions δ, η, δ^- and η^+. For conciseness we omit the proofs here which can be found in the appendix.

Since δ^- and η^+ are defined based on δ and η respectively, we describe the relation between δ and η; δ^- and η; and η^+ and δ in order to relate δ^- and η^+.

By definition, $\delta(n, k)$ returns the duration containing k event occurrences starting from the n-th event occurrence. Furthermore, $\eta(t, \Delta t)$ returns the number of event occurrences happening in the interval semi-open $[t, t + \Delta t[$.

A first property relating δ and η is that the duration containing $\eta(t, \Delta t)$ events starting from the first occurrence at or after t must be less than Δt.

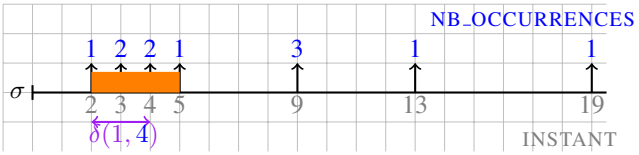


Figure 5: Example of a trace σ, with $\delta(first\_occ\_after(1), \eta(1, 2))$ in purple and Δt in orange.

Figure 5 illustrates this on an example for $t = 1$ and $\Delta t = 2$: The duration in purple is strictly smaller than the duration in orange. Formally,

Lemma 5. *For all instant t and non null duration Δt,*

$$\delta(\textit{first\_occ\_after}(t), \eta(t, \Delta t)) < \Delta t$$

where first_occ_after*:* INSTANT $\to$ ID_OCCURRENCE *is defined as follows:*

$$first\_occ\_after(t) = sob(t, 0, 0) + 1.$$

with sob*:* INSTANT $\to$ NB_OCCURRENCES $\to$ *instant defined as*

$$\textit{sob}(t, k, c) = \begin{cases} k + \sigma(c) & if\ t = 0 \\ \textit{sob}((t-1), (k + \sigma(c)), (c + 1)) & \textit{otherwise} \end{cases}$$

Proof By induction on k. see Appendix B.

A second property relating δ^- and η that the duration containing $\eta(t, \Delta t)$ events from the first occurrence after t must be less than Δt. Formally,

Lemma 6. *For all instant t and non-null duration Δt*

$$\delta^-(\eta(t, \Delta t)) < \Delta t$$

Proof trivial, by transitivity of δ-*minimality*. This follows easily from the fact that δ^- δ-minimal (Definition 3). Figure 5 illustrates this on an example for $t = 1$ and $\Delta t = 2$: where $\eta(1, 2) = 1$ and $\delta^-(\eta(1, 2)) = 0$.

Thirdly, we can bound the result of η in function of δ. By definition, $\eta(t, \Delta t)$ returns the exact number of event occurrences happening in the interval $[t, t + \Delta t[$. Furthermore, $\delta(n, k)$ returns the duration containing at least k event occurrences from the n-th event occurrence[2]. The number of occurrences happening in the duration $\delta(n, k) + 1$ from the instant of the n-th occurrence must be bigger or equal to k. Formally,

Lemma 7. *For all occurrences n, number of occurrences k and duration Δt,*

$$\eta(iof(n), \delta(n, k) + 1) \geqslant k$$

The proof structure is quite similar to the one proposed in Appendix B.

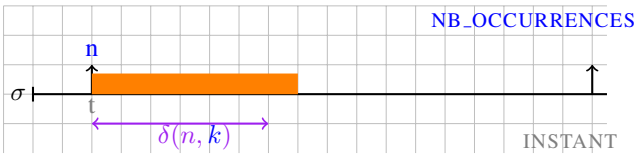


Figure 6: Example of a trace σ, with $(\delta(1, 2) + 1)$ in orange and $\delta(1, 2)$ in purple.

Figure 6 illustrates this relation with $\eta(iof(2), \delta(1, 2) + 1) = 4$ number of occurrences in the half-open orange duration.

Finally, because η^+ is η-maximal (Definition 4), it follows that the number of occurrences happening in the duration $\delta(n, k) + 1$ from the instant of n must be bigger than or equal to k. Formally,

Lemma 8. *For all occurrences n and number of occurrences k,*

$$\eta^+(\delta(n, k) + 1) \geqslant k$$

Proof trivial, by transitivity of *η-maximality*. Figure 6 illustrates this on an example for $n = 1$ and $k = 2$: where $\delta(1, 2) = 1$ and $\eta^+(\delta(1, 2) + 1) = 3$.

5.3 Correctness of the Conversions

In this section we first prove the correctness of the conversion from the event distance to the event load function. Then we prove the conversion from the event load to the event distance function.

Minimum Event Distance to Maximum Event Load ($\delta^- \to \eta^+$)

An event maximum load function is by definition a sub-additive η-maximal function. We want to prove that the function η^+ defined by Theorem 1 is a sub-additive η-maximal function (Definition 4).

We first prove the part of Theorem 1 restricted to sub-additivity:

[2]There may be several occurrences at the same instant as the k-th occurrence of the interval.

Lemma 9. *Let δ^- be a pseudo-super-additive function, then the function*

$$\eta^+(\Delta t) = \begin{cases} 0 & if\ \Delta t = 0 \\ max\{k \mid k \in \mathbb{N}, \delta^-(k) < \Delta t\} & otherwise \end{cases}$$

is sub_additive.

Proof. By cases.

Case $\Delta t = 0$. Trivial because $\eta^+(0) = 0$, thus $\eta^+(0) \leqslant \eta^+(0) + \eta^+(0)$.

Case $\Delta t > 0$. By definition of η^+ we have forall $x > 0$,

$$\eta^+(x) = max\{k \mid k \in \mathbb{N}, \delta^-(k) < x\}$$

and by definition of max

$$\delta^-(\eta^+(x) + 1) \geqslant x \ \wedge\ \delta^-(\eta^+(x)) < x.$$

Therefore, by taking $x = a$, $x = b$ and $x = a + b$ we have

$\delta^-(\eta^+(a)+1)+\delta^-(\eta^+(b)+1) \geqslant a+b$ and $\delta^-(\eta^+(a+b)) < a+b$

By transitivity,

$$\delta^-(\eta^+(a) + 1) + \delta^-(\eta^+(b) + 1) > \delta^-(\eta^+(a + b))$$

The pseudo-super-additivity of δ^- (Definition 3) entails

$$\delta^-(\eta^+(a) + \eta^+(b) + 1) \geqslant \delta^-(\eta^+(a) + 1) + \delta^-(\eta^+(b) + 1)$$

and by transitivity

$$\delta^-(\eta^+(a) + \eta^+(b) + 1) > \delta^-(\eta^+(a + b)).$$

Since δ^- is increasing (as a direct consequence of pseudo-super-additivity), we deduce

$$\eta^+(a) + \eta^+(b) + 1 > \eta^+(a + b)$$

and since η^+ is a function from $\mathbb{N}$ to $\mathbb{N}$

$$\eta^+(a) + \eta^+(b) \geqslant \eta^+(a + b)$$

□

To complete our proof, we now prove the part of Theorem 1 restricted to δ-minimal functions.

Lemma 10. *Let δ^- be a δ-minimal function, then the function*

$$\eta^+(\Delta t) = \begin{cases} 0 & if\ \Delta t = 0 \\ max\{k \mid k \in \mathbb{N}, \delta^-(k) < \Delta t\} & otherwise \end{cases}$$

is η-maximal

Proof. By cases.

Case $\Delta t = 0$. Trivial: $\forall t, \eta(t, \Delta t) = 0$ and $\eta^+(0) = 0$, then $\forall t,\ \eta(t, \Delta t) \leqslant \eta^+(0)$.

Case $\Delta t > 0$. By definition of η^+

$$\eta^+(\Delta t) = \begin{cases} 0 & if\ \Delta t = 0 \\ max\{k \mid k \in \mathbb{N}, \delta^-(k) < \Delta t\} & \text{otherwise} \end{cases}$$

we have for all $x \in \text{DURATION}^+$,

$$\eta^+(x) = max\{k \mid k \in \mathbb{N}, \delta^-(k) < x\}$$

and by definition of *max*

$$\delta^-(\eta^+(x)) < x \ \wedge \ \delta^-(\eta^+(x)+1) \geqslant x.$$

By Lemma 6 $\forall t\ \Delta t,\ \delta^-(\eta(t,\Delta t)) < \Delta t$

By transitivity $\forall t\ \Delta t,\ \delta^-(\eta(t,\Delta t)) < \delta^-(\eta^+(\Delta t)+1)$. The function δ^- is increasing so we can deduce

$$\forall t\ \Delta t, \eta(t,\Delta t) < \eta^+(\Delta t)+1.$$

and since η^+ is a function from $\mathbb{N} \to \mathbb{N}$

$$\forall t\ \Delta t,\ \eta(t,\Delta t) \ \leqslant \ \eta^+(\Delta t).$$

□

Since we proved the sub-additivity and the η-maximality, η^+ is a maximum event load function and Theorem 1 follows.

Maximum Event Load to Minimum Event Distance ($\eta^+ \to \delta^-$)

A event minimum distance function δ^- is by definition a pseudo-super-additive and δ-minimal functions. We want to prove that the function defined by Theorem 2 is pseudo-super-additive and δ-minimal (Definition 3).

We first prove Theorem 2 restricted to sub-addtive functions is

Lemma 11. *Let η^+ be a sub-additive function, then the function*

$$\delta^-(k) = min\{\Delta t \,|\, \Delta t \in \mathbb{N},\ \eta^+(\Delta t+1) \geqslant k\}$$

is pseudo-super-additive

Proof. By definition of δ^- we have for all $x \in$ NB_OCCURRENCES,

$$\delta^-(x) = min\{\Delta t | \Delta t \in \mathbb{N},\ \eta^+(\Delta x+1) \geqslant x\}$$

and by definition of the *max*

$$\eta^+(\delta^-(x)) < x \ \wedge \ \eta^+(\delta^-(x)+1) \geqslant x.$$

Therefore by taking, $x = a+b+1$, $x = a+b$ and $x = b+1$ we have

$$a+b+1 \leqslant \eta^+(\delta^-(a+b+1)+1)$$

and

$$\eta^+(\delta^-(a+1)) + \eta^+(\delta^-(b+1)) < a+b+1.$$

By transitivity

$$\eta^+(\delta^-(a+1)) + \eta^+(\delta^-(b+1)) < \eta^+(\delta^-(a+b+1)+1).$$

The sub-additivity of η^+ (Definition 4) entails

$$\eta^+(\delta^-(a+1)+\delta^-(b+1)) \leqslant \eta^+(\delta^-(a+1))+\eta^+(\delta^-(b+1)).$$

and by transitivity

$$\eta^+(\delta^-(a+1)+\delta^-(b+1)) < \eta^+(\delta^-(a+b+1)+1).$$

Since η^+ is increasing, we can deduce

$$\delta^-(a+1)+\delta^-(b+1) < \delta^-(a+b+1)+1.$$

and since δ^- is a function from $\mathbb{N}$ to $\mathbb{N}$

$$\delta^-(a+1)+\delta^-(b+1) \leqslant \delta^-(a+b+1)+1.$$

□

To complete our proof, we prove the Theorem 2 restricted to δ-minimal functions.

Lemma 12. *Let η^+ be a η-maximal function, then the function*

$$\delta^-(k) = min\{\Delta t | \Delta t \in \mathbb{N},\ \eta^+(\Delta t+1) \geqslant k\}$$

is δ-minimal.

Proof. By definition δ^- and by definition of *min*

$$\eta^+(\delta^-(k)) < k \ \wedge \ \eta^+(\delta^-(k)+1) \geqslant k.$$

By the Lemma 8

$$\forall n\ k \in \mathbb{N}, \eta^+(\delta(n,k)+1) \geqslant k.$$

By transitivity,

$$\forall n\ k,\ \eta^+(\delta(n,k)+1) > \eta^+(\delta^-(k)).$$

Since η^+ is increasing, we can deduce

$$\forall n\ k,\ \delta(n,k)+1 > \delta^-(k).$$

and since δ^- is a function from $\mathbb{N}$ to $\mathbb{N}$

$$\forall n\ k,\ \delta(n,k) \geqslant \delta^-(k).$$

□

Since we proved the pseudo-super-additivity and the δ-minimality, δ^- is a minimum distance function and Theorem 2 follows.

As a conclusion, it is interesting to note that the proofs of the main theorems are not very complex. The actual challenge was to find the additional lemmas and the missing hypotheses on the trace.

6 Related work: Formal Proofs for Real-Time

Our proofs have been completed using the proof assistant Coq. A proof assistant is a software tool assisting the development of formal proofs via human-machine collaboration through an interactive editor. The development environment provides a step-by-step verification of proofs. Some famous examples of proof assistants include Coq [The Coq development team, 2004], Prototype Verification System (PVS) [Owre *et al.*, 1992] and Isabelle/HOL [Nipkow *et al.*, 2002].

Mechanized proofs of real-time system analysis have already been proposed in the literature. The uniprocessor Priority Ceiling Protocol and the corresponding schedulability analysis has been proved correct by Dutertre using the PVS proof assistant [Dutertre, 1999]. Zhang *et al.* proved the correctness of the blocking bound for the Priority Inheritance Protocol with the Isabelle/HOL proof assistant [Zhang *et al.*, 2012]. The Network Calculus also has been formalized with Isabelle/HOL, with certification of bounds on the message delay in a toy network in [Mabille *et al.*, 2013a]. Using Coq, De Rauglaudre proved a schedulability condition for periodic tasks based on their phases and hyper period [De Rauglaudre, 2012]. Recently, Zhang *et al.* implemented Earliest Deadline First in a verification language based on Propositional Projection Temporal Logic and provided an optimality

proof using Coq [Zhang *et al.*, 2014]. The real-time community also employed other verification techniques to mechanize proofs such as model checking [Guan *et al.*, 2008; 2007] or abstract interpretation [Lv *et al.*, 2010; **?**]. Unfortunately, all these papers have their own formalization, which is not extendable and often not maintained.

Very recently, to avoid duplication of work, [Cerqueira *et al.*, 2016] proposes an extendable and maintainable open source library of formalisation and proofs. This library, Prosa, allows writing readable and formal proofs of schedulability analyses. Prosa uses the Coq proof assistant and the SSreflect extension library [Gonthier and Le, 2009]. It provides definitions and modular lemmas about arrival sequences, schedules, jobs, tasks... The usability of the framework is demonstrated through a case study, namely the multiprocessor response-time analysis of Bertogna and Cirinei for global fixed-priority and earliest-deadline first scheduling [Bertogna and Cirinei, 2007]. In this study, F. Cerqueira *et al.* successfully extend, formalize and prove Berogna and Critnei's analysis.

In the context of my thesis, we collaborate with the team of F. Cerqueira *et al.* and the team of Marc Boyer and project to integrate our work in Prosa. We develop this in the next section.

7 Future Work

Mechanized proofs for schedulability analysis is a complex task, which provides several possibilities of future extensions. In this section we present these possible developments by order of priority.

We have presented two conversion functions, between the minimum-event-distance function δ^- and maximum-event-load function η^+. These two functions are composable. And we expect the composition of the two conversions $F : \eta^+ \to \delta^-$ and $G : \delta^- \to \eta^+$ to satisfy

$$\delta^- = F(G(\delta^-)) \text{ and } \eta^+ = G(F(\eta^+))$$

These properties of the round-trip conversions remain to be proven.

In the context of our collaboration with F. Cerqueria *et al.*, we want to realise the three follows works.

1. To improve and validate the readability of my proofs in order to integrate them into Prosa.
2. Several activation models exist, the most popular is the PJd. We want to generalize the existing algorithms using the PJd model for sporadic and periodic tasks to the general event load model. F. Cerqueira *et al* [Cerqueira *et al.*, 2016] used the PJd model and a dedicated algorithm in their work. We are interested in the generalization of the algorithms that were proved and formalized in Prosa. Because their definitions are modular, we think that we can find easily where the PJd is used to replace it by the general event model.
3. The response-time bound for uniprocessor systems with arbitrary deadlines by Tindell *et al* uses the PJd model. One of the applications of the general event load model is to adapt this response-time bound to the general event model. The logical follow-up would be to formalize and prove this bound in a modular format in order to redo it with the initial response-time bound.

Our work is based on discrete time, whereas Marc Boyer *et al.* [Mabille *et al.*, 2013b] work on the network calculus which is based on continous time. An interesting research axis is to investigate the validity of our results in continous time.

8 Conclusion

The validation of critical real-time systems requires that to prove that they respect real-time constraints. To do so, the analysis techniques used in this field should be rigorous and sound. However, the real-time systems community has recently identified many unsound results. Even when the proposed techniques are guaranteed by pen-and-paper proofs, human errors show up. An example of a lack of rigor can be found in the general event-load model where we observe contradictory and informal definitions.

In this thesis, we have responded to this problem by first formalizing the general event load model and the associated conversion functions with the Coq proof assistant. We had to redefine a generic trace model. Then, we proposed a computer verified formal proof of conversion functions.

Computer verified formal proofs for real-time system can be found. Unfortunately, existing papers use their own formalization and are not composable. The objective of the open-source real-time system analysis library Prosa is to motivate the real-time system analysis community to collaborate on a single formalisation and write computer-verified proofs of well-known as well as novel schedulability analysis approaches. This thesis is a contribution to that ambitious effort.

Appendix

A General Event Model Definitions in Coq

The following listing contains the most relevant parts of our Coq implementation. The missing proofs are deliberately omitted for brevity.

```
(* ********************************************)
(** *       Types Definitions                  *)
(* ********************************************)
Definition instant := nat.

Definition duration := nat.

Definition nb_occurrences := nat.

Definition id_occurrence := nat.

Definition trace := instant → nb_occurrences.


(* ********************************************)
(** *       Assumptions                        *)
(* ********************************************)

(* One given trace *)
Variable sigma: trace.
```

```
(* Event at an instant bounded by *)
Variable k_max : nb_occurrences.

(* At any instant there is an event occuring later *)
Definition Later_event t1 t2 :=
t2 >= t1 ∧ sigma t2 > 0.

(* By hypotheis the trace sigma has that property
*)
Hypothesis AE_Events_occur :
forall t1, exists t2, Later_event t1 t2.

(* By hypothesis the number of occurrences
in an instant in the trace is bounded *)
Hypothesis E_bounded_event_at :
forall t, le (sigma t) k_max.

(* *********************************)
(** * The event load function *)
(* *********************************)

(** The event load function *)
Fixpoint eta (t : instant) (dt : duration) :
nb_occurrences :=
  match dt with
  | 0 ⇒ 0
  | S dt' ⇒ sigma t + eta (S t) dt'
  end.

Definition subadditive (f : nat → nat)
: Prop :=
  forall (x y : nat), f (x + y) <= f x + f y.

Definition max_eta_trace (f : duration →
nb_occurrences)
: Prop :=
  forall t dt, eta t dt <= f dt.

(** The event load maximum function *)
Definition eta_max (f : duration →
nb_occurrences)
: Prop :=
  max_eta_trace f ∧ subadditive f.

(* *************************************)
(** * The event distance function *)
(* *************************************)

(* compute the instant
after an number of occurrence *)
Program Fixpoint instant_of'
(remaining : nb_occurrences) (curr : instant)
{measure remaining} : instant :=
  match remaining with
  | 0 ⇒ curr − 1
  | _ ⇒ instant_of' (remaining − sigma (next curr))
          ((next curr) + 1)
  end.
Obligation 1.
set (N := next_prop curr).
destruct N as [? [? ?]].
omega.
Defined.

(* return the instant of an id of occurrence *)
Definition instant_of (n : id_occurrence)
: instant := instant_of' n 0.

(** The event distance function *)
Definition delta
(n : id_occurrence) (k : nb_occurrences)
: duration :=
instant_of (n + k − 1) − instant_of n.

Definition superadditive (f : nat → nat)
: Prop :=
  forall (x y : nat), f (x + y) >= f x + f y.

Definition pseudo_superadditive (f : nat → nat)
: Prop :=
  forall (x y : nat), f (x + (y + 1)) >= f (x + 1) + f (y + 1).

Definition min_delta_trace (f : nb_occurrences →
duration)
: Prop :=
  forall n k, delta n k >= f k.

(** The event distance minimum function *)
Definition delta_min (f : nb_occurrences →
duration)
: Prop :=
  min_delta_trace f ∧ pseudo_superadditive f.

(* *************************************)
(** *       Conversion Functions      *)
(* *************************************)

(* max {k | f(k) < dt} *)
Fixpoint max_k_in_dt (k: nb_occurrences)
(f: nb_occurrences → duration) (dt: duration)
: nb_occurrences :=
  match k with
  | 0 ⇒ k
  | S x ⇒ if ltb (f x) dt then x
                else max_k_in_dt (x−1) f dt
  end.

Definition max_nb_occ_in_dt :=
  max_k_in_dt k_max.


(* min {dt | g(dt + 1) >= k} *)
Program Fixpoint min_dt_with_k' (dt: duration)
(g: duration → nb_occurrences) (k: nb_occurrences)
{measure k }: duration :=
  match k with
  | 0 ⇒ dt
  | S x ⇒ if leb x (g (dt+1)) then dt
            else min_dt_with_k' (dt+1) g (k−1)
  end.
Obligation 1. omega.
Defined.

Definition min_dt_with_k :=
  min_dt_with_k' 0.

(* η⁺(dt) = max {k | f(k) < dt} *)
Definition conversion_delta_eta
```

```
(f: nb_occurrences → duration)
: duration → nb_occurrences :=
  fun (dt : duration) ⇒ max_nb_occ_in_dt f dt.

(* δ⁻(k) = min {dt | g(dt + 1) >= k} *)
Definition conversion_eta_delta
(g: duration → nb_occurrences)
: nb_occurrences → duration :=
  fun (k: nb_occurrences) ⇒ min_dt_with_k g k.

Definition conversion_round_trip
(f: nb_occurrences → duration)
: nb_occurrences → duration :=
  fun (k: nb_occurrences) ⇒
conversion_eta_delta (conversion_delta_eta f) k.

(* ****************************************)
(** *       Conversion Proofs             *)
(* ****************************************)

(* delta_eta_max respect
the max_eta_trace property *)
Property conversion_delta_eta_max_eta_trace :
forall (f : nb_occurrences → duration),
      delta_min f
      → max_eta_trace (conversion_delta_eta f).

Property conversion_delta_eta_sub_additive :
forall (f : nb_occurrences → duration),
      delta_min f
      → subadditive (conversion_delta_eta f).

(* delta_eta_max is an eta_max *)
Property conversion_delta_min_eta_max:
forall (f : nb_occurrences → duration),
      delta_min f
      → eta_max (conversion_delta_eta f).
intros.
unfold eta_max.
split.
  + apply conversion_delta_eta_max_eta_trace. easy.
  + apply conversion_delta_eta_sub_additive. easy.
Qed.

(* eta_delta_min respect
the min_delta_trace property *)
Property conversion_eta_delta_min_delta_trace :
forall (g: duration → nb_occurrences),
      eta_max g
      → min_delta_trace (conversion_eta_delta g).

(* eta_delta_min respect is
pseudo_superadditivity *)
Property conversion_eta_delta_min_superadditive:
forall (g: duration → nb_occurrences),
      eta_max g
      → pseudo_superadditive (conversion_eta_delta g).

(* eta_delta_min is an delta_min *)
Property conversion_eta_max_delta_min:
forall (g: duration → nb_occurrences),
      eta_max g
   → delta_min (conversion_eta_delta g).
Proof.
intros.
unfold eta_max.
split.
  + apply conversion_eta_delta_min_delta_trace. easy.
  + apply conversion_eta_delta_min_superadditive. easy.
Qed.
```

B Proof of the Lemma 5

We want to prove by induction on Δt the Lemma 5 defined as

$$\textit{for all instances of } t \textit{ and non null duration } \Delta t,$$
$$\delta(\textit{first\_occ\_after}(t), \eta(t, \Delta t)) < \Delta t$$

We first prove the Lemma 5 restricted to $\Delta t = 1$ is

Lemma 13.

$$\textit{for all instances of } t \textit{ and duration equal to } 1,$$
$$\delta(\textit{first\_occ\_after}(t), \eta(t, 1)) < 1$$

Proof. Case $\Delta t = 1$.
By definition of η

$$\eta(t, \Delta t) = \begin{cases} 0 & if\ \Delta t = 0 \\ \sigma(t) + \eta(t+1, \Delta t - 1) & \text{otherwise} \end{cases}$$

we can deduce that $\eta(t, 1) = \sigma(t)$.
Then either $\sigma(t) = 0$ or $\sigma(t) > 0$,

Case $\sigma(t) = 0$.
We can deduce that

$$\delta(\delta(\textit{ first\_occ\_after}(t), \eta(t, 1))),\ \ \delta(\delta(\textit{ first\_occ\_after}(t), \sigma(t)))$$

and

$$\delta(\delta(\textit{ first\_occ\_after}(t), 0))$$

By definition of δ (Definition 2) and $A = \textit{first\_occ\_after}(t)$

$$\delta(A, 0) = \textit{instant\_of}(A + 0 - 1) - \textit{instant\_of(A)}$$

and since δ^- is a function from $\mathbb{N}$ to $\mathbb{N}$

$$(\delta(\textit{first\_occ\_after}(t), \eta(t, 1)) = 0) < 1$$

Case $\sigma(t) > 0$.
We can deduce by the definition of δ and $A = \textit{first\_occ\_after}(t)$ that

$$\delta(A, \sigma(t)) = \textit{instant\_of}(A + \sigma(t) - 1) - \textit{instant\_of}(A)$$

Because $\sigma(t) > 0$ we can deduce

$$\textit{instant\_of(first\_occ\_after}(t)) = t$$

and because $\sigma(t)$ occurrences at the instant t

$$\textit{instant\_of(first\_occ\_after}(t)) = \textit{instant\_of(first\_occ\_after}(t+\sigma(t)-1))$$

we deduce with $A = \textit{instant\_of(first\_occ\_after}(t))$

$$\delta(A, \sigma(t)) = \textit{instant\_of}(A + \sigma(t) - 1) - \textit{instant\_of}(A) = 0$$

and

$$\delta(\textit{instant\_of(first\_occ\_after}(t)), \eta(t, 1)) = 0$$

then for $\sigma(t) > 0$

$$(\delta(\textit{first\_occ\_after}(t), \eta(t, 1)) = 0) < 1$$

□

We proved the base case, let us suppose the following induction hypothesis

Hypothesis 14.

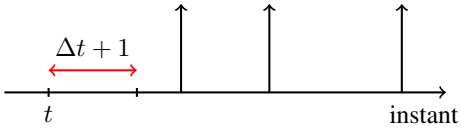
$$\textit{for all instances of } t \textit{ and non null duration } \Delta t,$$
$$\delta(\textit{first\_occ\_after}(t), \eta(t, \Delta t)) < \Delta t$$

Then we prove the induction step, Lemma 5 restricted to $\Delta t + 1$:

Lemma 15.

$$\textit{for all instances of } t \textit{ and duration equal to } \Delta t + 1,$$
$$\delta(\textit{first\_occ\_after}(t), \Delta t + 1) < \Delta t + 1$$

We prove by 3 cases, the first one consist on $\sigma(t)+\sigma(\Delta t+1) = 0$



Proof. Case 1
We can deduce from $\sigma(t) + \sigma(\Delta t + 1) = 0$

$$\textit{instant\_of}(\textit{first\_occ\_after}(t)) > (t + \Delta t + 1)$$

and $\eta(t, \Delta t + 1) = 0$
By defintion of δ (Definition 2), $\forall n, \delta(n, 0) = 0$ then

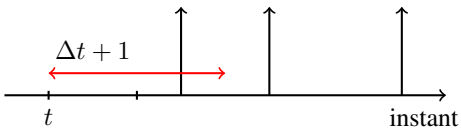
$\delta(\textit{first\_occ\_after}(t), \eta(t, \Delta t+1)) = \delta(\textit{first\_occ\_after}(t), 0) = 0$

Because $\Delta t + 1 > 0$, we can conclude

$$\delta(\textit{first\_occ\_after}(t), \eta(t, \Delta t + 1)) < \Delta t + 1$$

□

Then we prove the case 2, which consist on $\sigma(t) = 0$



Proof. Case 2
We can deduce from $\sigma(t) = 0$

$$\textit{instant\_of}(\textit{first\_occ\_after}(t)) > t$$

By definition of η (Definition 5)

$$\eta(t, \Delta t + 1) = \sigma(t) + \eta(t + 1, \Delta t) = 0 + \eta(t + 1, \Delta t)$$

then can deduce

$$\eta(t, \Delta t + 1) = \eta(t + 1, \Delta t)$$

By induction hypothesis for $t = t + 1$ (Hypothesis 14)

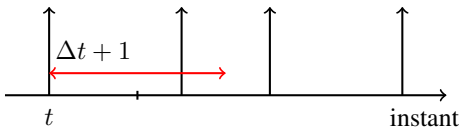
$$\delta(\textit{first\_occ\_after}(t), \eta(t + 1, \Delta t + 1)) < \Delta t$$

Since $\Delta t + 1 > \Delta t$ we conclude

$$\delta(\textit{first\_occ\_after}(t), \eta(t + 1, \Delta t + 1)) < \Delta t + 1$$

□

Finnaly, we prove the case 3, which consist on $\sigma(t) > 0$



Proof. Case 3.
By definition of η (Definition 5)

$$\eta(t, \Delta t + 1) = \sigma(t) + \eta(t + 1, \Delta t)$$

By induction hypothesis for $t = t + 1$ (Hypothesis 14)

$$\delta(\textit{first\_occ\_after}(t + 1), \eta(t + 1, \Delta t)) < \Delta t \quad (1)$$

and by base case

$$\delta(\textit{first\_occ\_after}(t), \eta(t + 1, 1)) < 1 \quad (2)$$

and also

$$\delta(\textit{first\_occ\_after}(t), \sigma(t)) < 1$$

By transitivity (1 + 2)

$$\delta(\textit{first\_occ\_after}(t + 1), \eta(t + 1, \Delta t))+$$
$$\delta(\textit{first\_occ\_after}(t), \eta(t + 1, 1)) < \Delta t + 1$$

By definition $\eta(t, 1) = \sigma(t)$ then $\delta(\textit{first\_occ\_after}(t + 1), \eta(t, 1)) = 0$,
$\eta(t + 1, \Delta t) > 0$, $\eta(t + 1, 1) > 0$ and $\textit{first\_occ\_after}(t + 1) > \textit{first\_occ\_after}(t)$.
By Lemma 16 with $F() = \textit{first\_occ\_after}()$

$\delta(F(t), \eta(t, 1)+\eta(t+1, \Delta t)) < \delta(F(t+1), \eta(t+1, \Delta t))+\delta(F(t), \eta(t, 1))$

By definition of η we have $\eta(t, \Delta t + 1) = \eta(t, 1) + \eta(t + 1, \Delta t)$,
we can deduce

$$\delta(\textit{first\_occ\_after}(t), \eta(t, \Delta t + 1)) < \Delta t + 1$$

□

Let us prove the following intermediary lemma:

Lemma 16. *For all $n \in \mathbb{N}$ and for all $k, k' \in \mathbb{N}^+$, if $\delta(n, k) = 0$ and $\textit{instant\_of}(n + k - 1), \textit{instant\_of}(n + k)$ then*

$$\delta(n, k + k') < \delta(n + k, k') + \delta(n, k)$$

Proof. By definition of Lemma 16

$$\delta(n, k) = \textit{instant\_of}(n + k - 1) - \textit{instant\_of}(n) = 0$$

and since $k > 0$

$$\textit{instant\_of}(n + k - 1) = \textit{instant\_of}(n) \quad (3)$$

By definition of δ

$\delta(n+k, k') = \textit{instant\_of}(n+k+k'-1) - \textit{instant\_of}(n+k)$

and

$$\delta(n, k+k') = \textit{instant\_of}(n+k+k'-1) - \textit{instant\_of}(n) \quad (4)$$

since we work with a discret time

$$\textit{instant\_of}(n+k-1)+1 = \textit{instant\_of}(n+k)$$

We rewrite (4) by using (3)

$$\delta(n, k+k') = \textit{instant\_of}(n+k+k'-1) - \textit{instant\_of}(n+k-1)$$

By transitivity

$$\begin{aligned}&\textit{instant\_of}(n+k+k'-1) - \textit{instant\_of}(n+k-1)+1\\ &= \textit{instant\_of}(n+k+k'-1) - \textit{instant\_of}(n+k)\end{aligned}$$

that corresponds to

$$\delta(n, k+k')+1 = \delta(n+k, k')$$

Since $\delta(n,k) = 0$ we have

$$\delta(n, k+k')+1 = \delta(n+k, k') + \delta(n, k)$$

and since δ^- is a function from $\mathbb{N}$ to $\mathbb{N}$

$$\delta(n, k+k') < \delta(n+k, k') + \delta(n, k)$$

□

References

[Axer *et al.*, 2013] Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Dobel Bjorn, and Hartig Herman. Response-time analysis of parallel fork-join workloads with real-time constraints. pages 215–224, 2013.

[Bertogna and Cirinei, 2007] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2007), 3-6 December 2007, Tucson, Arizona, USA*, pages 149–160, 2007.

[Cerqueira *et al.*, 2016] Felipe Cerqueira, Felix M Stutz, and Björn B Brandenburg. Prosa: A case for readable mechanized schedulability analysis. 2016.

[Davis *et al.*, 2007] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[De Rauglaudre, 2012] Daniel De Rauglaudre. Vérification formelle de conditions d'ordonnancabilité de tâches temps réel périodiques strictes. In *JFLA-Journées Francophones des Langages Applicatifs*, 2012.

[Dutertre, 1999] Bruno Dutertre. The priority ceiling protocol: formalization and analysis using PVS. In *Proc. of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS)*, pages 151–160, 1999.

[Gonthier and Le, 2009] Georges Gonthier and Roux Stéphane Le. An ssreflect tutorial. 2009.

[Guan *et al.*, 2007] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *Software Technologies for Embedded and Ubiquitous Systems, 5th IFIP WG 10.2 International Workshop, SEUS 2007, Santorini Island, Greece, May 2007. Revised Papers*, pages 263–272, 2007.

[Guan *et al.*, 2008] Nan Guan, Zonghua Gu, Mingsong Lv, Qingxu Deng, and Ge Yu. Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking. In *Object Oriented Real-Time Distributed Computing (ISORC)*, pages 556–560. IEEE, 2008.

[Henia *et al.*, 2005] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis-the symta/s approach. In *Computers and Digital Techniques, IEE Proceedings-*, volume 152, pages 148–166. IET, 2005.

[Lv *et al.*, 2010] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 339–349. IEEE, 2010.

[Mabille *et al.*, 2013a] Etienne Mabille, Marc Boyer, Loic Féjoz, and Stephan Merz. Certifying network calculus in a proof assistant. In *EUCASS-5th European Conference for Aeronautics and Space Sciences*, 2013.

[Mabille *et al.*, 2013b] Etienne Mabille, Marc Boyer, Loïc Fejoz, and Stephan Merz. Certifying network calculus in a proof assistant. In *Proceedings of the 5th European Conference for Aeronautics and Space Sciences (EUCASS)*, 2013.

[The Coq development team, 2004] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[Neukirchner *et al.*, 2013] Moritz Neukirchner, Sophie Quinton, Tobias Michaels, Philip Axer, and Rolf Ernst. Sensitivity analysis for arbitrary activation patterns in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 135–140. EDA Consortium, 2013.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[Owre *et al.*, 1992] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, pages 748–752, 1992.

[Quinton *et al.*, 2012] Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 515–520. IEEE, 2012.

[Schliecker, 2010] Simon Schliecker. Performance analysis of multiprocessor real-time systems with shared resources. 2010.

[Tindell and Burns, 1994] Ken Tindell and Alan Burns. Guaranteeing message latencies on control area network (can). In *Proceedings of the 1st International CAN Conference*. Citeseer, 1994.

[Tindell *et al.*, 1994] Ken W Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[Zhang *et al.*, 2012] Xingyuan Zhang, Christian Urban, and Chunhan Wu. Priority inheritance protocol proved correct. In *Interactive Theorem Proving*, pages 217–232. Springer, 2012.

[Zhang *et al.*, 2014] Nan Zhang, Zhenhua Duan, Cong Tian, and Ding-Zhu Du. A formal proof of the deadline driven scheduler in PPTL axiomatic system. *Theor. Comput. Sci.*, 554:229–253, 2014.

Factorization for sat-solving

Delise Antoine, Maréchal Alexandre, Périn Michaël

24 Août 2016

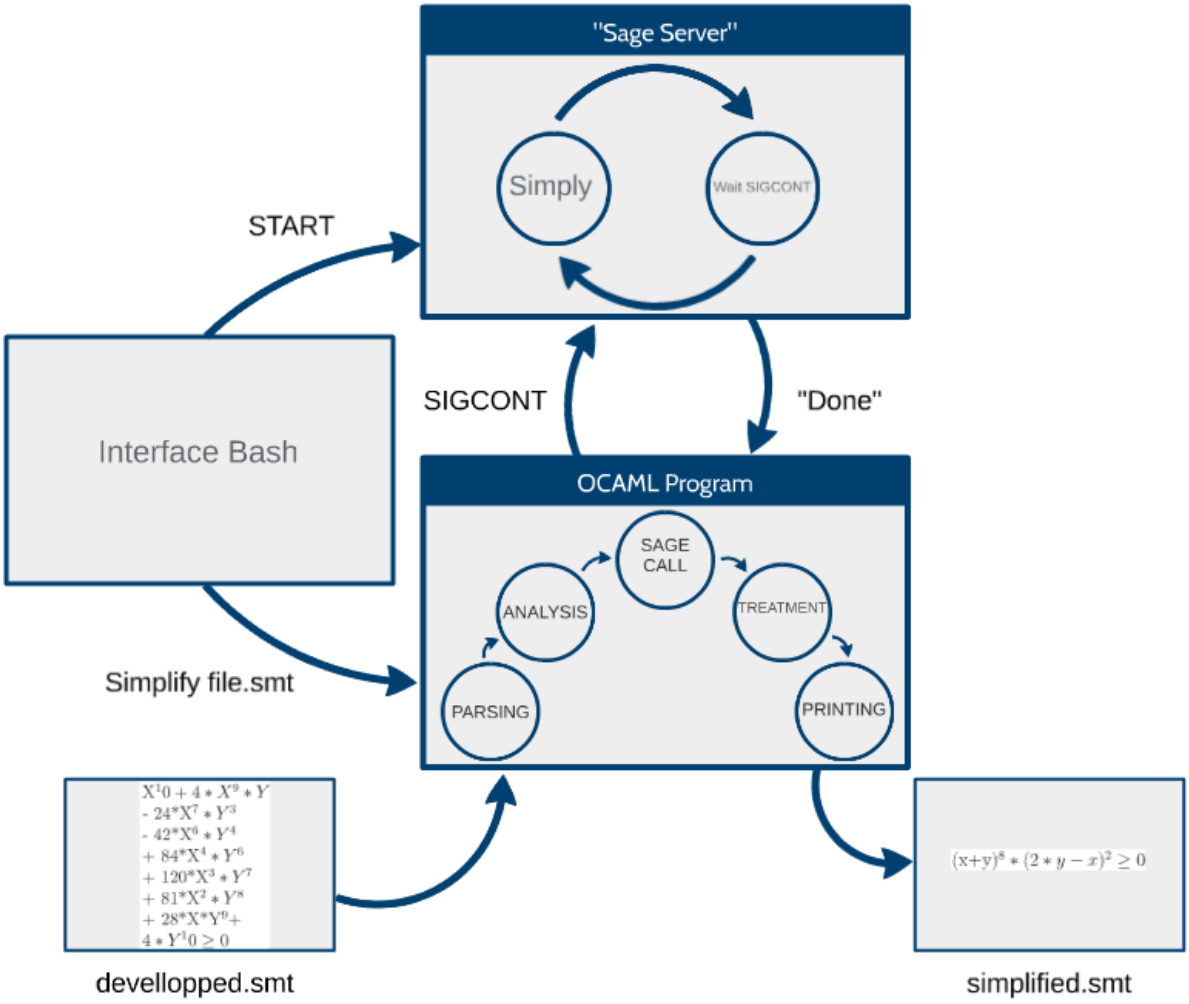
1 Polynomial constraint in SMT-solving

The second goal of a solver after finding the solution is being the faster.
To achieve that a solver can invoke many methods. for the purpose of prospecting new methods which make solver faster, we are proposing to study the ability of solver to take advantages of factorization.

Intuitively a polynomial in-equation should be simpler, and faster to solve, once factorized.
For instance take the following in-equation:
$-X^{10} - 4 * X^9 * Y + 24 * X^7 * Y^3 + 42 * X^6 * Y^4 - 84 * X^4 * Y^6 - 120 * X^3 * Y^7 - 81 * X^2 * Y^8 - 28 * X * Y^9 - 4 * Y^{10} > 0$
It is be a bit harder to decide the existence (or not) of its solutions in comparison to the factorized form:
$-(x + y)^8 * (2 * y - x)^2 > 0$
The constraint has trivially no solution since it is the negation of a product of two squares polynomials.

We develop an algorithm that factorizes a given list of polynomial and replace all occurrences of a factor by a reference to a uniquely defined polynomial. This algorithm reveals the structure of the problem and the question is: can the current solvers exploit that structure to be more efficient ?
For instance the original list:
$P_0 = (x + 1)^3 * (y - 2) * (x + 3)$
$P_1 = (x + 1)^3 * (y - 2) * (z - 1)$
Will be rephrased as:
$P_0 = P_2 * (x + 3)$
$P_1 = P_2 * (z - 1)$
$P_2 = (x + 1)^3 * (y - 2)$

Figure 1: Tool's connections



2 Experimentation

For testing the efficiency of factorization on solver we developed a platform that analyzes a smt problem, retrieves some structure (like in the previous example) and generates a equivalent and simpler smt problem. Once the collections of problems have been rephrased we can compare the time taken by a sat-solver to resolve it for the original problem and the rephrased one.

The interface of the platform is a bash script, it make up the working environment and initialize the other tools.
The main program consists in a Ocaml parser and printer of smt problem in the smt2 format. It exchange polynomial list with the "sage server" to simplify them and build the new equivalent problem.

The factorisation algorithm is developed in python and it is builds on top of a existing algorithm of the SageMath library for factorisation of multivariate polynomials. The Sage part is a server running a infinite loop reading a list of polynomial in a file, writing the rephrased smt-problem in a file, and then suspending itself.

Like the python machine, and the SageMath library take a long time to start (a few seconds if python is already started), the python script have been turn in a "server" kind one. Like

describe in the previous paragraph, waking up the sage program by sending a SIGCONT is instantly and allow us to plot only the true execution time with the ocaml program.

3 Results

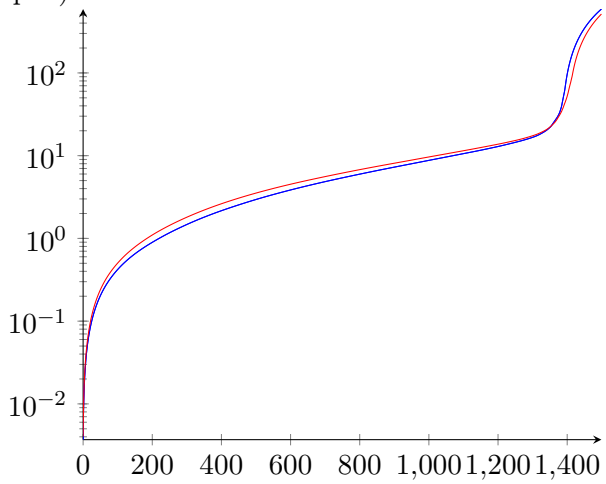
The figure is a cactus plot representing the time taken by z3 solver to solve a control assay (in Blue color) and the "simplied" one by the method explained in Context part (in Red).
We can see both curves are the same. However, according the theory, the test with factorization should be faster than the normal one. This result

reveal that a SMT-solver like Z3 — one of the most efficient ones — do not take advantages of simplification like factorization. As a consequence, there is room for methods exploiting factorization.

4 Perspectives

We developed a modular platform which can be reused for further experiments. Each part are dedicated to specific task:
The BASH interface allow to manage test sets and offer parameter to fragment them and producing control assay. More it build up the working environment by making different tools of the platform communicate.
The ocaml program is the core of the platform, it parse, analyse, modify, build and print smt problem and invoke the dedicated script of the sage server.
The solver is a external tool that is ran on a smt-file. Thus, it is possible to change a parameter for testing with any other sat-solver.
The SageMath script make some transformation on and mathematical operations (like factorization) on the given list of polynomial. So it's possible to experiment others decomposition algorithm. For instance, when a polynomial P is irreducible, the euclidian division by the polynomials of the problem can reveal relations such as $P = P_1 * P_2 + P_3$ which is another exploitable structure.

Figure 2: Solving time (factorized and developed)



References

[VMCAI]L Bernardin. Efficient Multivariate Factorization Over Finite Field. Algorithm used by SageMaths
[SMT-lib standard]http://smtlib.cs.uiowa.edu/language.shtml
[SageMath]http://www.sagemath.org
[Z3]SAT-Solver used for the plot figure 2: http://research.microsoft.com/en-us/um/people/leonardo/strategy.pdf

Is the factorization helpful for current sat-solvers ?

Delise Antoine, Maréchal Alexandre, Périn Michaël

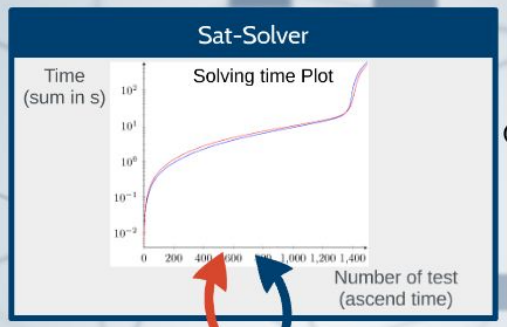
Factorization provide information about the sign of a polynomial expression

$$-9x^4-4y^4+6x^3y-5xy^3+12x^2y^2-18x^3+8y^3+17xy^2-6x^2y-4y^2-9x^2-12xy > 0$$

$$<=> -(x-y+1)^2 * (2y+3x)^2 > 0$$

Extracting theses information should help to solve the sign of an expression and find a solution of sat problems involving polynomial

So we build tools to find factorization in sat problem to determined if current sat-solver are able to benefit of them.

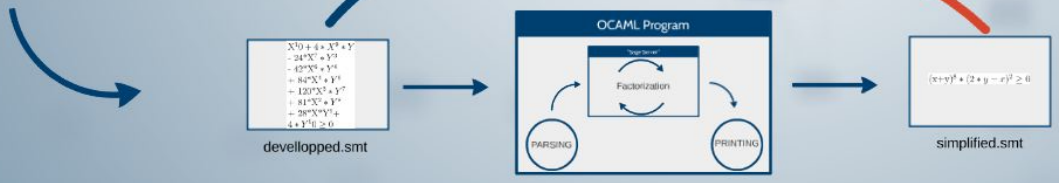


There is no differences when polynomial of the smt problems are factorized.
Current sat-solvers still can't take advantages of factorizations

Splitting methods are promising to improve the solving efficiency of problem involving several polynomial in-equations but tool allowing sat-solver benefiting of them need to be developed

Tool developed for the research:

- An analyzer of smt problem in ocaml
- A modular system of script of factorization based on SageMath
- A test platform to manage test sets



The modularity allow the user replacing easily the factorization script by an other simplification to test the efficiency of that one.

Magistère Internship Report

Boyan MILANOV
Université Grenoble Alpes
Grenoble, FRANCE
boyan.milanov@imag.fr

Supervised by: Josselin FEIST.

Abstract

Manually building an ROP exploit is a heavy and time-consuming task, of which every attacker would like to get rid of. Several tools based on symbolic execution and SMT-Solvers have been developed so as to help building payloads, or to do it automatically. However, their efficiency is challenged by two main issues caused by the exclusive use of SMT format to model gagdets. In the following paper we expose those issues and propose a slightly different approach that might enable to bypass those difficulties.

1 Introduction : The Chaining Challenge

When exploiting a security flaw against a system whose security have been increased, building a ROP exploit is a common thing to do. This type of attack is difficult to set up and a tool helping it permits a precious gain of time. Nevertheless, automation of gadgets chaining is far for being trivial[1]. Such a process leans on multiple strategies, of whom the most important is the ability to make a 'semantic request' for a gadget. That is to say, the tool must be able to seek for a gadget that would perform certain actions on the processor and/or the memory, among all the available gadgets.
To do so, the tool must dispose of a semantic representation of the gadgets. This way, it can compute whether each gadget suits a request or not, and build a 'bank' of available actions.
The question of the semantic representation of a gadget have so far been solved almost only by using a SMT-LIB model of the piece of code. Running a SMT-Solver and adding constraints enables indeed to make semantic requests and compute the possible effects of a gadget. However, in practice, this representation encounters limitations that prevents the development of a truly efficient tool.
Consequently, the need for a tool using a not necessarily completely different but alternative representation have become apparent.

2 SMT-Solvers limitations

The use of SMT-LIB format to model the gadgets have several drawbacks, in terms of relevance and efficiency. They are listed in this section.

1. Running a SMT-Solver is not made at a negligible cost. Insofar as the number of gadgets is often very high (several thousands), and given the tremendous amount of possible requests, using a solver directly will result in way too numerous calculations, which can not be performed in acceptable time[2].
2. SMT-Solvers are seeking for models which satisfies formulas. Those models are a numeric assignation of concrete values to registers and memory : they do not permit to use abstract values to represent registers and memory. As a result, the treatment of semantic queries is often biased by "false positives" due to models that correspond to particular, fancy, and unlikely states of the processor/memory.

3 Proposed solution

The use of a different way to represent the gadget's semantic leads to more precise and flexible informations about it's potential effects. This representation have been implemented during the internship in a tool named *ROPGenerator*

To start with, an enhanced implementation of Data-Dependence-Graphs enables to compute the effects of a given gadget. For each register and each octet in memory, it is possible to give their "final values" (values after the gadget was executed), depending on the initial values, and with almost no use of SMT-Solvers.

Moreover, the dependencies computed by the tool are conditional : a gadget can have multiple effects, depending on the initial state of the computer. Those effects are all calculated with their corresponding conditions. This process gives the opportunity to enlarge the range of usable gadgets, and opens the door towards automatic chaining of gadgets.

4 Further work

Some enhancements can still be made on the tool. First of all, detecting of identical/similar gadgets could reduce the computation time. Then, using a small database of gadget's semantics could help the dependency computation. Eventually, developing dedicated algorithms and heuristics would enable automatic generation of a ROP-Chain.

References

[1] Nguyen Anh Quynh. Optirop : the art of hunting rop gadgets. 2013.

[2] Edward SCHWARTZ, Thanassis AVGERINOS, and David BRUMLEY. Q : Exploit hardening made easy. 2011.